Originally published in Proceedings of TOOLS’96 USA,
Santa Barbara, California, July 1996.

The Design of Eiffel Programs: Quantitative Evaluation Using the MOOD Metrics

Fernando Brito e Abreu (INESC / ISEG)

Rita Esteves, Miguel Goulão (INESC / IST)

INESC, Rua Alves Redol, 9, Apartado 13069, 1000 Lisboa, PORTUGAL

phone: +351-1-3100226 / fax: +351-1-3145843 / email: fba@inesc.pt

ABSTRACT

The main mechanisms of the Object-Oriented paradigm are supposed to help produce better and cheaper software. As with other things, their usage is more or less intensive, depending mostly on the designer knowledge, experience and ability. This paper reviews a set of metrics called MOOD, suited for evaluating the use of those mechanisms. A MOOD-to-Eiffel binding is introduced. Some code fragments are presented to illustrate the concepts and to clarify the measurement process. A sample of Eiffel libraries is used to collect these metrics. Statistical analysis is performed on the sample and some hypotheses are drawn and discussed. Some preliminary heuristics that can be used during the design process are then derived. Those heuristics can be of some help to designers, mainly if embedded in CASE tools.

1. INTRODUCTION

The Object-Oriented (OO) paradigm includes a set of mechanisms such as *inheritance, encapsulation, polymorphism and message-passing* that are believed to allow the construction of more robust and easily extendible software [Meyer88]. However, by simply using an OO language that supports those mechanisms, we are not automatically favored with an increase in software quality and development productivity, because its effective use relies on the designer’s ability. In fact, the OO design activity is not only about mapping the requirements specification but also on how to use those mechanisms to maximize their potential benefits.

The learning curve from novice to experienced designers is steep, mainly for those going through a paradigm shift[1]. Design is a “creative” activity where multiple alternatives are often available, for the same partition of the system being modeled. A great benefit would be obtained if some heuristics could help in choosing the way, namely by identifying avoidable or at least uncommon practices. Design metrics can be used for this purpose, as will be seen in this paper.

Several research works in the OO design metrics arena were produced in recent years [Dumke95] [Sellers95] [Campanai94] [Cant94] [Chidamber94] [Hopkins94] [Abreu93]. However, there is scarce information on how the proposed metrics should be used, as well as a lack of experimental validations[2]. This paper is hoped to shorten this gap.

A former paper [Abreu94] proposed the MOOD (Metrics for Object Oriented Design) set of metrics. These metrics, which will be reviewed here, allow an evaluation of the use of the mechanisms of the Object-Oriented paradigm. They are aimed at helping to establish comparisons and derive conclusions among heterogeneous systems (different size, complexity, application domain and/or OO implementation language). A metrics collection experiment,

[1] - It was recognised [Conner94] that, on average, students with some degree of proficiency in another paradigm (e.g. imperative) had more difficulties learning how to design the OO way, than those doing it from the start (i.e. with no previous software development knowledge whatsoever). The explanation was that students had to unlearn habits.

[2] - In other related areas such as OO reuse metrics, some interesting experimental validation studies were conducted [Melo95], [Lewis91].

that will be described here, is based on a sample of OO libraries that is reasonably representative of some of that heterogeneity.

This paper is organized as follows: the next section introduces the main goals and strategy of the current research work from which this paper originated. Section 3 includes a brief review of the MOOD set along with a detailed binding for the Eiffel language. Section 4 describes a MOOD metrics collection experiment, including the tool used, the sample identification and the results achieved. Section 5 discusses the results of statistical data analysis, proposes some design heuristics based on the MOOD set and provides some preliminary critiques of the results. Some follow-up research directions are mentioned in section 6.

2. RESEARCH GOALS AND STRATEGY

The research being carried out in this area by the Software Engineering Group at INESC, a private Portuguese non-profit R&D organization, in cooperation with the Lisbon Technical University, is focused on two main goals:

- ***Improving the OO design process to achieve better maintainability and reliability***
- ***Improving the OO estimation process to achieve better resource allocation***

The results of an experiment, where the impact of object-oriented design on resulting software quality attributes such as reliability (defect density) and maintainability (normalized rework effort) was empirically evaluated, were presented in [Abreu96]. The results achieved so far show that design alternatives may have a strong influence on resulting quality. Quantifying this influence can help to train novice designers by means of heuristics [Abreu95] embedded in design tools. On the other hand, being able to predict the resulting reliability and maintainability is very important for project managers during the resource allocation (planning) process.

We are also concerned about producing effort and schedule estimates for OO software development. Among other things, this requires evaluating the size and complexity of the software system to be built. A part of this system is usually built from scratch and is expected to be the one where most effort is spent. The remaining part will be reused, with minor or major adaptations, from existing component libraries. Selecting, understanding and adapting components may demand considerable effort. The activity of producing reusable components requires an intense abstraction effort and does not pay in the short term. A complete model for OO projects resource estimation, denominated MOORED (Model for Object-Oriented Resource Estimates Determination) that accommodates these concerns, is under study.

3. THE MOOD METRICS SET

3.1 Introduction

The MOOD set includes the *Method Hiding Factor (MHF), Attribute Hiding Factor (AHF), Method Inheritance Factor (MIF), Attribute Inheritance Factor (AIF), Polymorphism Factor (POF) and Coupling Factor (COF).* These metrics are defined at the system or subsystem[3] level while in other approaches, such as the well know set proposed in [Chidamber94], the metrics are defined at the class level.

Each MOOD metric is associated with such basic structural mechanisms of the object-oriented paradigm as *encapsulation* (MHF and AHF), *inheritance* (MIF and AIF), *polymorphism* (POF) or *message-passing* and *association* (COF). The mathematical definition of each MOOD metric will be introduced after the underlying basic concepts are made clear. Each metric is expressed as a quotient where the numerator represents the actual use of one of those mechanisms for a given design. The denominator, acting as a normalizer, represents the hypothetical maximum achievable use for the same mechanism within the same universe of discourse that is, considering the same classes and inheritance relations. As a consequence, these metrics are expressed as percentages, ranging from 0% (no use) to 100% (maximum use) and thus are *dimensionless.* This avoids the misleading, subjective or "artificial" units that are often found in the metrics literature.

Being *formally defined,* the MOOD metrics avoid subjectivity of measurement and thus allow replicability. In other words, different people at different times or places can yield the same values when

[3] - Collection of classes organized in some way to offer a given functionality as a whole.

measuring the same systems.

These metrics are also expected to be system *size independent.* A partial demonstration of this assertion is included in a following section. Size independence allows inter-project comparison, thus fostering cumulative knowledge.

The MOOD metrics definitions make no reference to specific language constructs. The *language (in)dependence* will broaden the applicability of this set of metrics by allowing comparison of heterogeneous system implementations.

3.2 MOOD to Eiffel binding

Each OO programming language has its own constructs that allow for implementation of OO mechanisms in somewhat distinct fashions. A mapping, hereafter referred as binding, of MOOD concepts and terminology to the adopted implementation language is thus needed. A binding for the Eiffel language [Meyer92] is included in this section. A binding for the C++ language was introduced in [Abreu95]. Bindings for other OO languages are expected to be defined in the near future.

This section will present a stepwise introduction of each relevant concept in MOOD, immediately followed by the corresponding counterpart in the Eiffel language. Context swapping (MOOD to Eiffel or vice-versa) will be identified by the following *icons* at the left margin:

CONTEXT	***MOOD***	***Eiffel***
ICON	📜	🗼

3.2.1 Methods

The MOOD concept of **method** is that of a wrapped piece of procedural code (the body) whose execution as a whole is triggered by some agent. This is done through an interface that is identified by a unique name (within a certain range) and which may contain some mechanism of interchange with the calling agent (such as parameters or returned values). Methods are used to perform operations of several kinds such as obtaining or modifying the status of objects.

In Eiffel we count as ***methods*** the following features:

- class *features* that have an implementation (body) within the **do ... end** clause**,** even if declared **obsolete**[4]
- class features with a **deferred ... end** clause [5]
- external functions with the **external ... end** clause [6]
- class **constants** with the **once ... end** clause [7]

Several methods can share the same implementation[8]. This is not just an alias (synonym) mechanism, because these methods can be independently redefined in any subclasses. Therefore, for the purpose of MOOD measures, they were considered as distinct methods.

Some *method* examples follow:

```
feature {NONE}

cartesian(a, b: REAL) is
-- "normal" method
        do
                x := a;
                y := b;
        end;

a, b: REAL is
-- 2 methods with the same implementation
        do
                Result := y;
        end;
```

[4] - Eiffel allows to declare any *feature* (not only routines) as **obsolete**. This allows to denote that the obsolete feature was not discarded only for the sake of compatibility with older versions and, as such, should not be used in new code.

[5] - One method is said to be deferred when its body (implementation) is not defined in the current class but instead in one (or more) of its subclasses.

[6] - These external functions can be implemented in other languages (e.g. C, Pascal or even assembly code).

[7] - These constants have a body similar to the one declared in a **do ... end** clause, which is only executed once (first call). Subsequent calls return the same value (the one calculated in the first call).

[8] - by simply mentioning a list of identifiers instead of a single one.

```
arity: INTEGER is
-- deferred method
          deferred
          end;

file_owner(uid: INTEGER) : STRING is
-- external method
          external
                    "C"
          end;

search_substring(other: STRING; start: INTEGER):
INTEGER is
-- obsolete method
obsolete "Use '' substring_index '' instead"
          do
                 Result:=substring_index(other, start);
          end;

i: Complex is
-- constant method
          once
                    !! Result.cartesian (0, 1)
          end;
```

3.2.2 Attributes

The MOOD concept of **attribute** is one of an independently identified data structure, either static or dynamic, transient or persistent, atomic or structured (e.g. record or array), which is used to store constants or variables. Attributes are used, among other things, to represent the status of each object in the system.

In Eiffel we count as ***attributes*** the following features:

- class *features* without an implementation (body);
- simple constants - attributes that keep the same value for all class instances; they can have several types (*integer, real, double, boolean, character, string, bit*).

Some examples of *attributes* are:

```
feature
          -- "normal" attributes
          width : INTEGER;.
          a, b : REAL;

          -- constant attribute
          True_constant: STRING is "true";
```

3.2.3 Methods and Attributes Visibility

The MOOD concept of **visibility**, associated with what is often referred to as the *range* or *scope* of an identifier, is related to the use of information hiding mechanisms. Each feature (method M or attribute A) is either visible or hidden from a given class C. If a feature is visible to a class C, then C can use that feature. Therefore, we can define the following logic function:

$$is\_visible(C_i.M_\alpha, C_j) = \begin{cases} 1 & iff \begin{cases} j \neq i \\ C_j \ may\ call\ M_\alpha \end{cases} \\ 0 & otherwise \end{cases}$$

$$is\_visible(C_i.A_\delta, C_j) = \begin{cases} 1 & iff \begin{cases} j \neq i \\ C_j \ may\ reference\ A_\delta \end{cases} \\ 0 & otherwise \end{cases}$$

We can specify whether an Eiffel feature is visible by explicit reference to a set of classes (list within brackets) after the **feature** keyword, for locally defined features, or in the **export** clause, for inherited features. In the latter clause we can specify visibilities separately for features or lists of features. The pre-defined symbol **ALL** can be used to refer to all features inherited.

If a feature is visible for a class C (whether this is the one where the feature was defined or one declared within the above mentioned set) then, by default, it is visible to all descendants of class C.

If the class set is omitted then, by default, the declared features are visible to all classes. This can be made explicit, after the **feature** or the **export** keywords, by indicating in that set the **ANY** class (a pre-defined superclass of all root classes of defined inheritance hierarchy trees). Then the features could be used by **ANY** and all its descendants, that is, by all classes in the system. This type of visibility is equivalent to the one obtained through the **public** clause in C++.

If the **NONE** class (a pre-defined descendant of all existing classes) is declared in the set, then the corresponding features can only be used within the range of the class where they were declared, as well as of their subclasses. The same effect can also be obtained by indicating an empty set ({}). This is equivalent to the **protected** [9] clause in C++.

From the preceding discussion, it is evident that a *feature* can be visible to some classes and hidden from the rest. Some examples follow:

```
feature
-- visible feature to all classes
        capacity: INTEGER is
                do
                        Result := area.count
                end;

feature {ANY}
-- explicitly visible feature to all classes
...


feature {NONE}
-- visible to current class and its descendants
...

feature {}
-- also visible to current class and its descendants
...

feature {HISTORY_LIST}
-- visible to current class and descendants, as well
as to HISTORY_LIST class and its descendants
...
```

The *visibility of a feature* is defined as the percentage of the system classes, *other than the one where it was defined*, for which that feature is visible. Supposing TC is the total number of classes in the system under consideration, then the visibilities of method M_α and attribute A_δ, both defined in class C_i, are given by:

$$V(C_i.M_\alpha) = \frac{\sum_{j=1}^{TC} is\_visible(C_i.M_\alpha, C_j)}{TC-1}$$

$$V(C_i.A_\delta) = \frac{\sum_{j=1}^{TC} is\_visible(C_i.A_\delta, C_j)}{TC-1}$$

The denominator is the number of all classes except the one where the feature is defined. Function V may range from zero (the feature is hidden from all classes) to one (the feature is visible to all classes).

For the purpose of MOOD measurement, changes in the visibility of inherited features are accounted for in the class where they were initially defined. In other words, changes of visibility in any descendent class will eventually increase the number of classes that can potentially use the feature. A similar situation arises when we have feature name clashing in **multiple inheritance.** The resulting visibility of a feature inherited from two or more classes, which had different visibilities in each ascending class, will be the union of the corresponding visibilities.

Some illustrating examples follow:

```
class A
        feature {NONE}
                g: INTEGER
-- g is visible to A and its descendants

        feature {J, K}
                z, w: BOOLEAN
-- z and w are visible to A, J, K and their descen-
dants


class B
        feature
                x, y: INTEGER is
                        do
                                ...
                        end
-- x and y are visible to all classes

        feature {D}
                f, g, h : INTEGER
-- f, g and h are visible to B, D and their descen-
dants


class C
-- this class inherits from A and B and changes the
visibility of inherited features in the export clauses

        inherit
                A
                        export
                                {B, F, H} ALL;
                                {H, F, I} g;
                        end
-- all features inherited from A, except g, become
```

[9] - There is no equivalent in Eiffel to the **private** clause of C++.

visible for classes C, B, F, H and respective descendants;

B

export

{F, G} x, h;

-- features x and h become visible to C, F, G and their descendants;

{ANY} f;

-- feature f becomes visible for all classes;

{H, I, J} g;

-- feature g (inherited from both A and B) becomes visible to the set of classes that is the union of the classes declared in the two export clauses (C, H, F, I, J and their descendants).

end

end

Since, as previously stated, changes in the visibility of inherited features are, for MOOD measurements, only considered in the class where they were initially defined, we obtain the visibility table[10] represented below (Table 1). An "X" indicates that the feature in the left is visible to the class above and their descendants (e.g. feature z of class A is visible to classes B, C, F, H, J, K and respective descendants, including the ones of A). Shaded zones represent situations where visibility is not defined.

	A	**B**	**C**	**D**	**F**	**G**	**H**	**I**	**J**	**K**
A.w	▨	X	X		X		X		X	X
A.z	▨	X	X		X		X		X	X
A.g	▨		X	X	X		X	X	X	
B.x	X	▨	X	X	X	X	X	X	X	X
B.y	X	▨	X	X	X	X	X	X	X	X
B.f	X	▨	X	X	X	X	X	X	X	X
B.g		▨	X	X	X		X	X	X	
B.h		▨	X	X	X	X				

Table 1 - Visibility table for A and B features

As an example suppose that:

i) C is the only descendant of B;

ii) classes D, F and G belong to disjoined inheritance hierarchies (thus with no common descendants);

iii) $DC(C_i)$ is the number of descendants of C_i;

then:

$$V(B.h)=\frac{4+DC(C)+DC(D)+DC(F)+DC(G)}{TC-1}$$

where the constant 4 accounts for classes C, D, F and G.

3.2.4 Defined Features

Features **defined** in a class are the ones whose declaration lies within that class. That includes the ones that are not implemented (deferred or external features). We then define the following functions:

$M_d(C_i)$ = methods defined in class C_i

$A_d(C_i)$ = attributes defined in class C_i

We are now able to introduce the Method Hiding Factor (MHF) and Attribute Hiding Factor (AHF) as:

$$\boxed{MHF=\frac{\sum_{i=1}^{TC}\sum_{m=1}^{M_d(C_i)}(1-V(M_{mi}))}{\sum_{i=1}^{TC}M_d(C_i)}}$$

$$\boxed{AHF=\frac{\sum_{i=1}^{TC}\sum_{m=1}^{A_d(C_i)}(1-V(A_{mi}))}{\sum_{i=1}^{TC}A_d(C_i)}}$$

3.2.5 New and Overriding Features

A defined feature can be either a **new** or an **overriding** version of an inherited one. **New** features are the ones whose name is different from any inherited feature and thus do not override them. **Overriding** features are the ones that change the definition of inherited features. The following functions are then defined:

$M_d(C_i)=M_n(C_i)+M_o(C_i)$ = methods defined in class C_i

$A_d(C_i)=A_n(C_i)+A_o(C_i)$ = attributes defined in class C_i

where:

[10] - Assuming that no more inheritance relations hold.

$M_n(C_i)$ = new methods in class C_i

$M_o(C_i)$ = overriding methods in class C_i

$A_n(C_i)$ = new attributes in class C_i

$A_o(C_i)$ = overriding attributes in class C_i

In Eiffel we consider the following overriding situations:

- feature contained in a **redefine** clause - explicit modification of an inherited implementation; this modification can be located in several places:
  - i) in the *signature* (interface) - arguments or result types; the number of parameters cannot be altered and the arguments and result types must conform with those of the inherited *feature*;
  - ii) in the *implementation*, by transforming a function without parameters into an attribute (the opposite is not allowed) or by changing a method body or even by changing local declarations;
  - iii) in the *specification*, by changing pre or post-conditions.
- *effectivation* of a feature declared as deferred in one ascendant - the feature gets an implementation;
- feature in an **undefine** clause - it becomes deferred (in a broad sense the implementation is also changed).

Notes:

1. We can change the name of a method defined in a superclass by using a **rename** clause. The former method name can then be used by other methods. This detail should be given attention since the same identifier, within the same class hierarchy, can represent distinct features, even without a **redefine** statement.
2. Methods declared as **frozen,** which means they cannot be redefined, are counted just as the others.

Some examples about redefinitions follow:

```
class ARRAY
inherit
        INDEXABLE

        -- redefinition through undefinition
                undefine
                        consistent
                end;

        -- redefinition through redefinition
                redefine
                        copy, is_equal
                end;


class HISTORY_L_W
inherit
        FORM_D
                rename
                        make as form_d_make
                end;

        -- redefinition of a renamed feature
                redefine
                        form_d_make
                end;

        feature
                frozen make (n: INTEGER) is
        -- new frozen feature (note that "old"
        make feature was renamed)
                require
                        non_negative_size:n>= 0
                do
                        make_area (n);
                ensure
                        empty_string:count = 0;
                end;

class STRING
        feature
                arity: INTEGER is
                        deferred
                        end;

class BOOLEAN
        inherit
                STRING
        end;

        feature
                arity: INTEGER is
        -- redefinition through effectivation
        do
                Result := area.count
        end;
```

3.2.6 Polymorphic Features

An important characteristic of the object-oriented paradigm is polymorphism[11]. This characteristic is such that a given message sent to class C_i can be bound (statically or dynamically) to a named method implementation in C_i or one of its descendants. Thus, the message recipient can have as many distinct implementations as the number of times this same method is overridden in C_i descendants. We then define the Polymorphism Factor (POF) as:

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

The numerator represents the *actual number of possible different polymorphic situations*. The denominator represents the *maximum number of possible distinct polymorphic situations* for class C_i. This would be the case where all new methods defined in C_i would be overridden in all its descendants.

3.2.7 Inherited Features

Inherited features in a class C_i are those which are inherited and not overridden in that class. An inheritance relation, for instance C_d inheriting from C_a, is represented by $C_d \rightarrow C_a$. We then define the following functions:

$M_i(C_i)$ = methods inherited in class C_i

$A_i(C_i)$ = attributes inherited in class C_i

In multiple inheritance situations, features inherited from distinct classes are joined (merged in just one feature) if they have the same signature (interface)[12] and are deferred[13]. The automatic join can be avoided by using the **rename** clause. An example of this situation is illustrated below:

```
class INDEXABLE
        feature
                copy is
                        do
                                ...
                        end;
                is_equal is
                        do
                                ...
                        end;

class RESIZABLE
        feature
                copy is
                        do
                                ...
                        end;

class STRING
        -- automatic join of copy
        inherit
                INDEXABLE
                RESIZABLE

class STRING_1
        -- avoiding the automatic join
        inherit
                INDEXABLE
                        rename
                                copy as copy_index
                        end;
                RESIZABLE
```

3.2.8 Available Features

Available features (methods or attributes) in a class C are the ones that can be used in association with C. Available features are the defined plus the inherited ones. We then define the following functions:

$M_a(C_i) = M_d(C_i) + M_i(C_i)$ = available methods in class C_i

[11] - From the ancient Greek “poly” (several) and “morphos” (shapes).

[12] - The **rename** clause can be used to “force” this identity.

[13] - The **undefine** clause can be used to discard existing implementations.

$A_a(C_i) = A_d(C_i) + A_i(C_i)$ = available attributes in class C_i

Now we can introduce the Method Inheritance Factor (MIF) and the Attribute Inheritance Factor (AIF):

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

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

3.2.9 Coupling

Coupling is due to the representation of associations between classes (static coupling) and message exchanges between their instances (dynamic coupling). It can be identified by the existence of several kinds of references. In MOOD a class C_c is said to be a *client* of another class C_s (the *supplier*) and is represented by $C_c \Rightarrow C_s$ if C_c contains at least one non-inheritance reference to C_s. A reference can be made in an attribute or method argument type, a local method type (returned value) or even a call to a method belonging to the supplier class.

Clientele is represented by the *is_client* logic function. For the sake of simplicity, clientele shape and strength (number of references made to the client class) are not considered. Therefore we have:

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

Some coupling examples are represented below:

```
class S              -- the supplier
        ...
class C              -- the client
        feature {NONE}
                cartesian (a, b : S) is
-- clientele through a method argument type
                        local
                                x, y : S
-- clientele through a local attribute
                        do
                                x := a;
                                y := b;
                        end;
```

Notes:

1. A usual reference is the one made to the constructor of the supplier class (*features in* ***creation*** *clause). There is no explicit* deallocation (destructor feature) in Eiffel since there is an automatic garbage collector.
2. Generic classes are configurable through specific parameters. A typical example is that of a class that implements a list or a stack of some undefined type. A generic parameter indicates the type. Because of late binding (run-time) we can not calculate which is going to be the supplier of the generic class.
3. Expanded classes[14] are counted just as "regular" ones.

The Coupling Factor (COF) is then defined as:

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

The numerator represents the *actual number of couplings not imputable to inheritance.* The denominator is the *maximum possible number of non-inheritance couplings* in a system with TC classes.

4. METRICS COLLECTION EXPERIMENT

4.1 The tool

Manual collection of any metrics is generally a repetitive (thus tedious) and time-consuming (thus expensive) task. To avoid it, a tool for metrics extraction from source code, named MOODKIT, was developed at INESC. Version 1.1 supported the collection on C++ code of all MOOD metrics

14 - Usually a feature of the class type C is a *reference* to that class. If C is expanded (**expanded** keyword is declared in the class header) then the feature is an *instance* (object) of class C.

described in this paper. It was built using ANSI C and scripts with standard UNIX commands (awk, grep, find, etc) and was disclosed as shareware.

A completely re-engineered version 2.0 supporting both Eiffel and C++ was used in this study. Its core (metrics definition dictionary, metrics storage, human-machine interface) is based on a language-independent central repository with storage, retrieval and graphical capabilities. It uses specific "stubs", based on language parsers, for metrics capture from source code. Stubs for other languages are also envisaged. In order to do so, MOOD bindings for those languages must be defined beforehand.

4.2 The sample

A collection of several class libraries written in the Eiffel programming language was measured with MOODKIT. The first four are included in the ISE standard development environment (version 3). The whole sample consists of:

- **EiffelBase library** - contains reusable components covering the needs of many applications in the area of fundamental data structures and algorithms;
- **EiffelVision library** - includes toolkit-independent GUI facilities, suitable for building interactive graphical applications able to run on different platforms (maps a single source text to various GUI toolkits);
- **EiffelLex library** - lexical analysis library;
- **EiffelParse library** - parsing library;
- **Structure library**[15] - data structure library based on circular-linked-lists; characteristics like used memory space and security were optimized; feature names are intuitive for users of other Eiffel libraries.
- **Yoocc (Yes! an Object-Oriented Compiler Compiler)** - compiler compiler that uses an extended parse library that evolved from the ISE EiffelParse library. It is a tool for automatically generating a processor framework (Eiffel source code) from a grammar [Avotins95a].
- **Trooper (Truly Reusable OO Parser for Eiffel Re-engineering)** - encapsulates the complexities of lexical analysis and parsing of Eiffel texts using an extended parse library that evolved from the ISE EiffelParse library. Semantic actions can be added to the parsing constructs to develop different tools such as pretty-printers, compilers, etc [Avotins95b].

This sample is expected to be a good representation of a population of reasonably well designed software systems written in the Eiffel language. More than half (EiffelBase, EiffelVision, EiffelLex and EiffelParse) have been in commercial use for some time[16] and were developed by the creators of the Eiffel Language. The others were also produced by trustworthy teams.

For a better perspective on the sample, Table 2 includes some size metrics for each library. Columns refer to the total number of declared classes, total number of declared methods, total number of declared attributes and total number of lines of code.

	Classes	*Methods*	*Attributes*	*LOC*
EiffelBase	151	2165	480	28658
EiffelVision	550	4477	1547	68162
EiffelLex	30	215	79	5302
EiffelParse	13	88	15	1434
Structure	15	121	18	2300
Trooper	387	409	43	19516
Yoocc	69	230	54	5978
TOTAL	1215	7705	2236	131350

Table 2 - Some indicators of sample size

4.3 The results

4.3.1 Preliminary statistics

Table 3 summarizes the results obtained through

[15] - This library was developed and is maintained by Nicolas Waquier (nicolas@esclare.fdn.org). Analyzed version was 2.5 (October 1995).

[16] - Thus with a considerable number of users finding bugs, suggesting upgrades, etc.

application of MOODKIT on the above sample. These results will be the basis for the statistical analysis described in this and the following sections.

	MHF	*AHF*	*MIF*	*AIF*	*COF*	*POF*
EiffelBase	17,68%	21,81%	81,86%	54,50%	1,87%	8,88%
EiffelVision	19,91%	72,15%	90,52%	91,67%	0,51%	2,94%
EiffelLex	35,04%	45,35%	61,40%	71,68%	4,45%	10,38%
EiffelParse	47,92%	29,44%	67,88%	75,00%	7,14%	11,21%
Structure	8,56%	23,41%	20,39%	58,14%	8,78%	5,24%
Trooper	6,47%	4,65%	58,39%	90,53%	0,21%	3,08%
Yoocc	53,80%	39,54%	53,16%	65,16%	0,86%	14,54%

Table 3 - MOOD metrics values for the sample

MOOD metrics can be considered as *continuous random variables*:

- *continuous* (as opposed to discrete) because in the interval where they are defined ([0, 100%]) they can take any real value; in other words, if we take any sub-interval of that interval, MOOD metrics can take an infinite number of distinct values within it;
- *random* because their precise values for a given system are not known until a data collection experiment is performed.

Some questions thus arise regarding the above sample, from a statistical point of view:

- Is there a central tendency or location of the values observed for each MOOD metric?

- How spread (dispersed) are the data for each metric?

Before starting the statistical description of our sample we need to identify and discard possible outliers. These are spurious values that can bias our analysis. To do so we include in Figure 1 the box plots for each MOOD metric.

Boxplots are formed from "boxes", which contain the 50% of values falling between the 25th and 75th percentiles. The line across the box is the median. The "tails" are lines that extend from the box to the highest and lowest values, excluding *outliers*. The latter are the ones with values between 1,5 and 3 box lengths from the upper and lower edge of the box. Sometimes these plots also include **extremes** that are values more than 3 box-lengths apart from the upper or lower edge of the box.

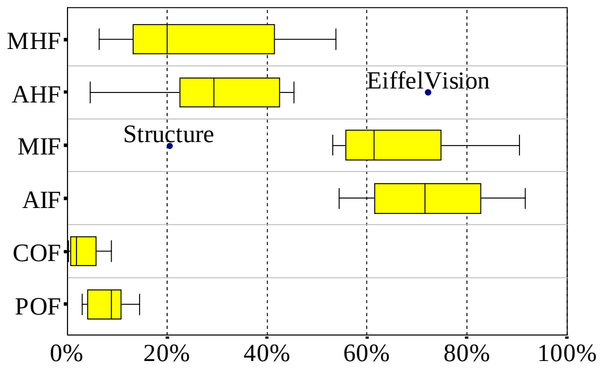

Figure 1 - Box plots for each MOOD metric

The sample described has no extremes, but has two outliers (AHF for EiffelVision and MIF for Structure) which are marked with black dots. These were removed from our sample and substituted by the mean of all other observed values of the corresponding metric. We are now in position to present some descriptive statistics (Table 4), which include:

- the sample mean;
- the variance - a measure of the dispersion around the mean, equal to the sum of the squared deviations from the mean divided by one less than the sample size;
- the standard deviation - the square root of the variance; has the advantage over variance of being expressed in the same units of measure as the random variable under study;
- the kurtosis - a measure of the extent to which the distribution is “tail-heavy”, compared to a normal distribution. In other words, it characterizes its relative peakedness or flatness. Positive Kurtosis indicates more cases in the extreme tails than in a normal distribution with the same variance, which corresponds to a relatively peaked distribution. Negative Kurtosis indicates less cases in the extreme tails than in a normal distribution with the same variance, which denotes a relatively flat distribution;
- the skewness - a measure of the asymmetry of a distribution around its mean. Positive skewness indicates a distribution with an asymmetric tail extending toward more positive values (the more

extreme values are greater than the mean). Negative skewness indicates a distribution with an asymmetric tail extending towards more negative values (the more extreme values are less than the mean).

Variable	*Mean*	*Variance*	*Std.Dev*	*Kurtosis*	*Skewness*
MHF	27,05%	0,0353	18,8%	-1,61	0,44
AHF	27,37%	0,0173	13,2%	0,74	-0,44
MIF	68,87%	0,0175	13,2%	-0,47	0,69
AIF	72,38%	0,0214	14,6%	-1,39	0,31
COF	3,40%	0,0012	3,4%	-1,26	0,75
POF	8,04%	0,0019	4,4%	-1,39	0,12

Table 4 - Some descriptive statistics for the sample of MOOD metrics

4.3.2 Finding a Probability Distribution

From Figure 1 and Table 4 we observe that all metrics seem to have moderate flat distributions, except AHF that seems to have a moderate peaked distribution. However, the low kurtosis absolute values allow to infer that their distributions are not very far from being normally distributed. Regarding skewness, all metrics except AHF appear to have moderate right skewed distributions. Again, the low absolute values found for skewness support the hypothesis of normal distributions.

If any assumptions are to be made based on normality, we need to be more precise and test the observed sample for goodness of fit with that kind of distribution.

For that purpose we will use the One-Sample Kolmogorov-Smirnov Test that is used to test the null hypothesis that a sample comes from a population in which the variable has a given distribution. Thus we have the following null (H0) and alternative (H1) hypotheses:

H0: the metric is normally distributed

H1: the metric is not normally distributed

Table 5 includes the results of the test. From it we can conclude that H1 can be rejected and thus we accept H0. This does not necessarily means that H0 is true, but simply that the available data does not allow it to be rejected.

	K-S Z	*2-Tailed P*
MHF	0,5808	0,8887
AHF	0,5120	0,9558
MIF	0,5668	0,9049
AIF	0,4719	0,9791
COF	0,6431	0,8025
POF	0,4379	0,9908

Table 5 - One-Sample Kolmogorov-Smirnov Test parameters

4.3.3 Confidence Intervals

In a repeated sampling context we expect a large percentage of the various sample confidence intervals to contain the (unknown) true population mean. In other words, if we repeatedly draw random samples of a given size N, from a normally distributed population and compute the confidence interval estimates for each, then 1-α percent of them will contain the true population mean (usually identified as β). α is called the level of statistical significance (usually 10%, 5% or 1%) and represents the probability of a type I error[17]. The confidence intervals are then characterized by 1-α and thus are said to be, for instance, 90%, 95% or 99%. As we make that percentage closer to 100% the interval becomes larger, as expected. Bigger sample sizes lead to smaller intervals. In fact, interval

[17] - If we consider two hypotheses:

H_0: the confidence interval contains β

H_1: the confidence interval does not contain β

we can identify two types of error:

Action:	**Possible real values for H_0:** true	false
Reject H_0	Type I error	Correct decision
Do not reject H_0	Correct decision	Type II error

Note: rejecting a hypothesis means that the sample information does not support it; failing to reject it does not necessarily mean it is true!

spread is inversely proportional to the square root of the sample size.

Since all metrics seem to be normally distributed, we can derive the following confidence intervals:

	Lower limit	*Upper limit*
MHF	15.4%	38.7%
AHF	19.2%	35.5%
MIF	60.6%	77.1%
AIF	63.3%	81.5%
COF	1.3%	5.5%
POF	5.3%	10.8%

Table 6 - 90% Confidence Intervals for mean of MOOD metrics

5. DISCUSSION

5.1 Design heuristics shape

This section presents a discussion on how the MOOD metrics can be used to express some recommendations for designers. Those will be based on a “filters' metaphor”, an Electronic Engineering analogy whose rationale follows.

Theoretically, a high-pass filter is not expected to affect signal frequencies above a certain value (the cutoff frequency). Below that value, the filter acts as a hindrance for frequency. By analogy, a high-pass heuristic is the one that suggests that there is a lower limit for a given metric. Going below that limit is a hindrance to resulting software quality. For those who do not like thresholds, we may say that the analogy is even better, if we realize that “real” filters do not have them. Indeed their shape is not a step but a curve with a bigger slope at the cutoff zone. Resulting software quality characteristics are also expected to be strongly attenuated (or increased, depending on the direction) as we approach the cutoff values. The reasoning for a band-pass heuristic is similar, except that we have two cutoff zones (a lower and a higher one).

AHF and MHF are a measure of the use of the information hiding concept that is supported by the encapsulation mechanism. Information hiding allows, among other things, to: (i) cope with complexity by looking at complex components such as “black boxes”, (ii) reduce "side-effects" provoked by implementation refinement, (iii) support a top-down approach, (iv) test and integrate systems incrementally.

For attributes (AHF) we want this mechanism to be used as much as possible. Ideally all attributes would be hidden, thus being only accessed by the corresponding class methods. Very low values for AHF should trigger the designers’ attention. The associated design heuristic shape is that of a *high-pass filter*.

The number of visible methods is a measure of the class functionality. Increasing the overall functionality will then reduce MHF. However, to implement this functionality we must adopt a top-down approach, where the abstract interface (visible methods) should only be the tip of the iceberg. In other words, the implementation of the class interface should be a stepwise decomposition process, where more and more details are added. This decomposition will use hidden methods, thus obtaining the above mentioned information-hiding benefits and favoring a MHF increase. This apparent contradiction is reconciled if we consider MHF to have values within an interval. A very low MHF would then indicate an insufficiently abstracted implementation. Conversely, a high MHF would indicate very little functionality. The design heuristic shape for MHF is thus the same as for a *band-pass filter*.

MIF and AIF are measures of inheritance, a mechanism for expressing similarity among classes that allows the portrayal of generalization and specialization relations and a simplification of the definition of inheriting classes, by means of reuse. At first sight we might be tempted to think that inheritance should be used extensively. However, the composition of several inheritance relations builds a directed acyclic graph (inheritance hierarchy tree), whose depth and width make understandability and testability quickly fade away. A *band-pass filter* shape seems appropriate for the corresponding heuristics.

Resulting polymorphism potential is measured through the POF metric. Polymorphism arises from inheritance and its use has pros and cons. Allowing binding (usually at run time) of a common message call to one of several classes (in the same hierarchy) is supposed to reduce complexity and to allow refinement of the class hierarchy without side-effects. On the other hand, if we need to debug such a hierarchy, by tracing the control flow, this same polymorphism will make the job harder. This is particularly true if we compare this situation with the procedural counterpart, where for a similar functionality we usually have a series of decision statements for triggering the required operation. We can then state that polymorphism ought to be bounded within a certain range. Naturally, a *band-pass filter* is the corresponding shape for the respective design heuristic.

The COF metric is a measure of coupling between classes. Coupling can be due to message-passing among class instances (dynamic coupling) or to semantic association links (static coupling). It has been noted [Meyer88] that it is desirable that classes communicate with as few others as possible and even then, that they exchange as little information as possible. Coupling relations increase complexity, reduce encapsulation and potential reuse, and limit understandability and maintainability. Thus, it seems that we should avoid it as much as possible. Very high values of COF should be avoided by designers. However, for a given application, classes must cooperate somehow to deliver some kind of functionality. Therefore, COF is expected to be lower bounded. Accordingly, the design heuristic shape will be the one of a *band-pass filter*.

As a conclusion we may say that the design heuristics can exhibit two shapes: high-pass (HP) and band-pass (BP), depending on the metric considered, as shown in the next table.

Metric	MHF	AHF	MIF	AIF	COF	POF
Shape	BP	HP	BP	BP	BP	BP

Table 7 - Heuristic shapes

5.2 Reasoning with Heuristics

Taking into account the considerations made regarding the heuristics shape made in the previous section, we can take as initial thresholds for triggering the designer attention, the lower and upper limits of the 90% confidence intervals (contained in Table 6). For instance, if the Coupling Factor exceeds 5,5% the designer could be warned somehow (supposing that he is using a design tool with embedded metrics capture). He would then realize that his design lies outside the common good practices boundaries and that the consequences may be the ones already referred. Besides this outlier identification, the MOOD metrics can also help choose between alternative design implementations by helping to rank them.

According to heuristics in Table 7, AHF is not upper bounded. Thus, although EiffelVision is a sample outlier, it should not be seen as a spurious one. On the contrary, the authors of this library used the data hiding mechanism in a more appropriate fashion than what was found in the other ones!

The MIF value for the Structure library (20,39%) is well below the lower limit of the confidence interval (60.6%). Thus it seems that the author of this library *has not taken significant advantage of the inheritance mechanism.*

5.3 Size independence

This section includes the analysis of the size independence hypothesis for each MOOD metric. For that purpose Table 8 shows the correlations of each of the sizes included in Table 2, with the corresponding values of each metric (Table 3).

	Classes	***Methods***	***Attributes***	***LOC***
MHF	-0,47	-0,28	-0,23	-0,36
AHF	-0,49	-0,11	-0,04	-0,23
MIF	0,50	0,88	0,84	0,79
AIF	0,76	0,36	0,43	0,51
COF	-0,70	-0,49	-0,45	-0,59
POF	-0,73	-0,48	-0,48	-0,58

Table 8 - Correlation of MOOD with some size metrics

Examining Table 8 we see that all metrics except MIF (shaded zone) are fairly size independent as they show low correlations[18] with most size metrics. Some metrics have a moderate size dependence on the number of classes. These few discrepancies might indicate one of two possibilities:

a) the metrics are ill-defined regarding size-independence;

b) the sample is somehow biased due to its small

[18] - Considering a 70% threshold, which seems appropriate.

size and thus the correlation is merely coincidental.

Since AIF and MIF have similar definitions, but do not show significant correlation with the same size metrics, we tend to accept hypothesis b). Therefore the hypothetical size dependencies can not be conclusive until a bigger sample is available and analyzed.

5.4 Statistical independence

Each MOOD metric should quantify a distinct feature of an OO system. To achieve this goal they need to be independent from each other. A measure of this independence is the linear correlation. Correlation one between two random variables means that one variable can be expressed as an exact linear function of the other variable. Correlation zero means that there is no linear association between the random variables. Statistically independent variables thus have zero correlation[19]. The sign of the correlation indicates whether their association is direct (positive sign) or inverse (negative sign).

We then evaluated the linear correlation among the sample value series for the defined metrics, two by two. Table 9 summarizes the results achieved.

Variable	***MHF***	***AHF***	***MIF***	***AIF***	***COF***
AHF	0,7451				
MIF	0,3761	0,1548			
AIF	0,1361	0,3045	0,0958		
COF	0,0505	0,2035	0,0754	0,4477	
POF	0,8862	0,7084	0,4857	0,5261	0,1315

Table 9 - Correlation among the MOOD metrics

Considering Table 9 we can conclude that most metric series pairs (except the three shaded) exhibit a low correlation value. There are no easy explanations for the unexpected correlations, that are believed to be coincidental, except that they probably stem from a small sample.

19 - Notice that the opposite is not necessarily true since the variables can be associated somehow non-linearly.

5.5 Criticizing the Results

The results presented in previous sections were based on assumptions such as sample significance, normality of distributions, etc. Several identifiable problems are worth mentioning, which we are aware may limit the validity of our conclusions. Among them are:

- small sample size - thus insufficiently meaningful for the purpose of our study;
- sampling variation - the very act of selecting a sample of software systems from a larger population introduces randomness in the form of uncontrolled and unpredictable variation;
- inappropriate probability distribution - normal distributions may not apply; however, by the Central Limit Theorem [Griffiths93] if the sample size is sufficiently large, which is not the case, then the normal hypothesis will stand.

In spite of the eventual errors induced, we think the sample is a good representation of the population of reasonably well-designed software systems using the Eiffel Language. The accuracy (i.e. their range reduction) of the confidence intervals for all metrics will increase with bigger samples.

6. FUTURE WORK

6.1 Experimental validation

Although the described experiment fulfilled the goal of extending our perception on the use of the basic structural mechanisms in object-oriented designs, we intend to conduct further experiments on larger samples to assess the consistency and reproducibility of achieved results. The public availability of the MOODKIT tool will allow people either from industry or academia to replicate the experiment herein described with these or other OO systems. The increase in sample size will also lead to better (smaller) confidence intervals (and thus refined heuristics) and will hopefully confirm some criteria underlying the metrics definition (size independence and statistical independence among metrics themselves).

An experiment conducted at the University of Maryland (USA) and reported in [Abreu96] evaluated the impact of object-oriented design (expressed by the MOOD metrics) on resulting software quality attributes (defect density and rework). The results achieved so far allow to infer that design alternatives have a strong influence on resulting quality. Being able to predict the resulting reliability and maintainability is very important to project managers during the resource allocation (planning) process. Another MOOD validation effort has recently started at University of Southampton (United Kingdom). Further validation experiments with a larger sample of projects are sought. The impact on other quality attributes like efficiency, portability, usability and functionality must also be assessed.

Other experiments should also be conducted to verify application domain dependencies, whose effect on corresponding design architectures is expected to be somehow reflected in the MOOD metrics. We may come across application-domain specific design heuristics.

6.2 MOOD bindings to other OO languages

Bindings for C++ [Abreu95] and Eiffel [self] were already produced and embedded in the MOODKIT tool. To enlarge the applicability of the MOOD metrics we expect to produce bindings for other OO languages such as Smalltalk, Java and OOPascal. These bindings will include (1) a mapping of concepts and terminology between MOOD and the language under consideration and (2) a description of how basic measures needed to compute MOOD metrics can be performed in that language. Stubs for the adopted languages, based on parsers built with lex and yacc, can then be integrated in the MOODKIT architecture. MOOD independence regarding implementation languages will also be further assessed and eventually refined.

6.3 Extending MOOD metrics to the analysis phase

Metrics should be collected and used to identify possible flaws as early as possible in the life-cycle, before too much work is spent based on them. It is a well-known fact that the effort of correcting and recovering from those defects increases non-linearly with elapsed project progress since they are committed. Looking at the analysis instead of design would then be a step forward toward cost-effectiveness. The object-oriented paradigm is supposed, at least theoretically, to allow a seamless analysis-design-coding transition. Many analysis and design methods have emerged [Champeaux92] in the past few years, with their own diagrammatic representations of differently named abstractions representing not-so-different basic concepts. This plethora gave birth to tools, such as ParadigmPlus or ObjectMaker, supporting multiple analysis and design methods. These tools map the information extracted from the distinct diagrams used by those different methods into a common repository, thus allowing diagrammatic conversions. Therefore, despite the apparent diversity of OO analysis models, we think it is possible to define a common set of metrics for analysis, a “natural” evolution of the MOOD set.

6.4 Embedding MOOD metrics in a CASE tool

There is an increasing interest from OO CASE tool makers in design metrics. Output from the ROSE tool (which supports the Booch method), for instance, is being used at Rational [Fay94] to derive object-oriented metrics. [Darscht94] reports the intention to build an OO metrics collection prototype integrated with the ObjectMaker tool. Oblog Software, a private Portuguese R&D company, also intends to extend its OBLOG CASE tool, that supports the OBLOG - OBject LOGic method [Sernadas91], with design metrics and heuristics based on MOOD.

6.5 Metrics for Object Oriented Design Patterns

Object-oriented design patterns [Gamma95] are currently a very active research field. They seem to be one yellow brick road to the promised reuse-land. Substantial increases in quality and productivity are expected to happen if software developers really start using these new “bricks”. However, the patterns’ adoption greatly depends on their complexity, adaptability, functionality and reliability. All those characteristics must be quantitatively evaluated in order to define acceptance criteria, assess reuse potential and risk or compare different pattern implementations for similar functionalities.

If a pattern has a high complexity, potential users will not understand it and its adoption will be jeopardized. Measuring and establishing reasonable limits for a pattern’s complexity seems to be a must. A generic OO complexity metric is expected to be built upon a combination of the MOOD metrics.

The functionality offered by a pattern represents its power to solve a certain category of problems. Some patterns have a much wider coverage than others in the sense that they can solve a given problem in many different contexts. This functionality should also be evaluated quantitatively.

Patterns are not supposed to be used “as is” (verbatim reuse). Instead, they are supposed to be somehow configured or adapted (leverage reuse) to solve a particular problem of the system under construction. Therefore, their degree of adaptability should also be quantified. A reduced configuration capability would degrade the pattern’s desired generality. Too much flexibility, on the other hand, would surely depend on several compromises that would sacrifice efficiency and memory usage optimization, provoke inadmissible increase in complexity and eventually produce undesirable side-effects.

The unreliability of a system that is built using an adopted pattern can have its origins in the pattern itself or apart from it. Testing different systems with embedded patterns and selecting only the faults that depend on the pattern's inclusion, should allow us to correlate them with the pattern’s complexity. From there we can build predictive models for reliability or/and redesign patterns for an increased reliability.

Several avenues for future work have been outlined. There is a vast ground for research in the OO metrics field. An increasing number of researchers [Whitty95] are out there to prove it.

REFERENCES

[Abreu93] Abreu, F. Brito and Carapuça, R., "Candidate Metrics for Object-Oriented Software within a Taxonomy Framework", Proceedings of AQUIS'93 (Achieving QUality In Software), Venice, Italy, October 1993; selected for reprint in Journal of Systems and Software, Vol. 23 (1), pp.87-96, July 1994.

[Abreu94] Abreu, F. Brito and Carapuça R., "Object-Oriented Software Engineering: Measuring and Controlling the Development Process", Proceedings of the 4th International Conference on Software Quality, ASQC, McLean, VA, USA, October 1994.

[Abreu95] Abreu, F. Brito; Goulão, Miguel and Esteves, Rita, “Toward the Design Quality Evaluation of Object-Oriented Software Systems”. Proceedings of the 5th International Conference on Software Quality, Austin, Texas, USA, October 1995.

[Abreu96] Abreu, F. Brito and Melo, Walcelio, “Evaluating the Impact of Object-Oriented Design on Software Quality”. Proceedings of the Third International Software Metrics Symposium, Berlin, March 1996.

[Avotins95a] Avotins, Jon; Mingins, Christine and Schmidt, Heinz, “YOOCC: Yes! An Object-oriented Compiler Compiler”, Proceedings of TOOLS USA'95, Santa Barbara, Prentice Hall, 1995, or Technical Report, Department of Software Development, Monash University (http://insect.sd.monash.edu.au:/), 1995.

[Avotins95b] Avotins, Jon; Maughan, Glenn and Mingins, Christine, “TROOPER”, Technical Report, Department of Software Development, Monash University (http://insect.sd.monash.edu.au:/), 1995.

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

[Cant94] Cant, S. N.; Henderson-Sellers, B. and Jeffery, D. R., "Application of cognitive complexity metrics to object-oriented programs", Journal of Object-Oriented Programming, pp. 52-63, July-August 1994.

[Champeaux92] Champeaux, Dennis De and Faure, Penelope, "A Comparative Study of Object-Oriented Analysis Methods", Journal of Object-Oriented Programming, vol. 4, n. 10, pp. 21-33, March / April 1992.

[Chidamber94] Chidamber, S. and Kemerer, C., "A Metrics Suite for Object Oriented Design", IEEE Transactions on Software Engineering, vol.20, n.6, pp. 476-493, June 1994.

[Conner94] Conner, D. Brookshire; Niguidula, David and Van Dam, Andries, “Object-Oriented Programming: Getting It Right at the Start”, proceedings of Educators’ Symposium, OOPSLA’94, Portland, USA, October 1994.

[Darscht94] Darscht, Pablo, “Assessing Objects Along the Development Process” Workshop on Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, submission 2, OOPSLA’94, Portland, USA, October 1994.

[Dumke95] Dumke, Reiner R., "A Measurement Framework for Object-Oriented Software Development", Annals of Software Engineering, Vol. 1, 1995.

[Fay94] Fay, Bill; Hamilton, Jim and Ohnjec, Viktor, “Position / Experience Report” Workshop on Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, submission 3, OOPSLA’94, Portland, USA, October 1994.

[Gamma95] Gamma, E.; Helm, R.; Johnson, R. and Vlissides, J., “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1995.

[Griffiths93] Griffiths, W.; Hill, R. and Judge, G., “Learning and Practicing Econometrics”, John Wiley & Sons, 1993.

[Hopkins94] Hopkins, Trevor P., "Complexity Metrics for Quality Assessment of Object-Oriented Design", SQM´94, Edinburgh, July 1994, proceedings published as Software Quality Management II, vol. 2: Building Quality into Software, pp. 467-481, Computational Mechanics Press, 1994.

[Lewis91] Lewis, John A., Henry, Sallie M. and Kafura, Dennis G. : "An Empirical Study of the Object-Oriented Paradigm and Software Reuse", proceedings of OOPSLA'91, pp. 184-196, 1991.

[Melo95] Melo, W., Briand, L. and Basili, V. R., “Measuring the Impact of Reuse on Quality and Productivity in Object-Oriented Systems”, technical report CS-TR-3395, University of Maryland, Dep. of Computer Science, January 1995.

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

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

[Sellers95] Henderson-Sellers, B., "Identifying Internal and External Characteristics of Classes Likely to be Useful as Structural Complexity Metrics”, Proceedings of 1994 International Conference on Object Oriented Information Systems OOIS'94, London, December 1994, Springer-Verlag, pp. 227-230, London, 1995.

[Sernadas91] Sernadas, C. and Fiadeiro, J., “Towards Object-Oriented Conceptual Modeling”, Data and Knowledge Engineering, vol. 6, n. 6, pp. 479-508, 1991.

[Whitty95] Whitty R., “Object-Oriented Metrics: People and Publications”, technical report, PROMISE project no. IED4/1/9308, South Bank University, 1995.