Revised version: originally published in Proceedings of the 5th International Conference on Software Quality, Austin, Texas, 23 to 26 October 1995.

Toward the Design Quality Evaluation of Object-Oriented Software Systems

Fernando Brito e Abreu (INESC/ISEG)
Miguel Goulão, Rita Esteves (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

This paper presents some advances towards the quantitative evaluation of design attributes of object-oriented software systems. We believe that these attributes can express the quality of internal structure, thus being strongly correlated with quality characteristics like *analyzability*, *changeability*, *stability* and *testabilility*, which are important to software developers and maintainers.
An OO design metrics set is reviewed, along with its rationale. An experiment for collection and analysis of those metrics is described and several suppositions regarding the design are evaluated. A considerable number of class taxonomies written in the C++ language were used as a sample. A tool to collect those metrics was built and used for that purpose. Statistical analysis was performed to evaluate the collected data. Results show that some design heuristics can be derived and used to help guide the design process. It was also clear that a number of follow-up topics deserve further research.

1. INTRODUCTION

The backbone of any software system is its design. It is the skeleton where the flesh (code) will be supported. A defective skeleton will not allow harmonious growth and will not easily accommodate change without amputations or cumbersome prothesis with all kinds of side effects. Because requirements analysis is most times incomplete, we must be able to build software designs which are easily understandable, alterable, testable and preferably stable (with small propagation of modifications). The Object-Oriented (OO) paradigm includes a set of mechanisms[1] such as *inheritance, encapsulation, polymorphism and message-passing* that are believed to allow the construction of designs where those features are enforced. However, a designer must be able to use those mechanisms in a "convenient" way. Long before the OO languages became widespread, it was possible to build software with an OO "flavor", using conventional 3rd generation languages. Conversely, 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. Being a "creative" activity, where multiple alternatives are often available for the same partition of the system being modeled, design would greatly benefit if some heuristics could help choose the way. Design metrics are being used for this purpose.

Several research works in the OO design metrics arena were produced in recent years [Dumke95, Sellers95, Campanai94, Cant94, Chidamber94, Hopkins94, Abreu93]. However, there is a lack of experimental validation. Worse than that, there is scarce information on how the proposed metrics should be used. Facing the available metrics literature, software practitioners are often left with the unpleasant feeling that "not everything that counts can be counted, and not everything that can be counted counts"[2]. A better scenario can be found on the field of OO reuse metrics, where experimental studies like [Melo95, Lewis91] are shedding some light.

An earlier paper [Abreu94] proposed the MOOD[3] set of metrics. These metrics allow the use of the main mechanisms of the Object-Oriented paradigm to be evaluated and are reviewed here. They are supposed to help establish comparisons and derive conclusions among heterogeneous systems (different size, complexity, application domain and/or OO implementation language), thus allowing cumulative knowledge to be achieved. Although the language heterogeneity is not yet addressed in this paper, an experiment is described where the sample (OO systems from which the MOOD metrics were collected) is a good representation of all the other differences.

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 the detailed review of the MOOD set along with its rationale. A simple case study in C++ is used to illustrate the basic concepts. The following section describes an experiment of systematic collection of the MOOD metrics,

[1] - Some of those are a natural evolution of concepts and constructs present in structured programming and founded on abstract data type theory.

[2] - Albert Einstein

[3] - Metrics for Object Oriented Design

including the tool used, the target sample and the results achieved. Section 5 discusses the experimental results and proposes some design heuristics based on the MOOD set. Some new research directions emerging from this study are mentioned in section 6. The final section presents a retrospective overview.

2. RESEARCH GOALS AND STRATEGY

2.1 Main goals

The research being carried out in this area by the Software Engineering Group at INESC[4] in cooperation with the Lisbon Technical University has two main goals:

Goal 1 - Improve the OO design process to achieve better maintainability

Maintenance is (and will surely continue to be) the major resource waster in the whole software life-cycle. Maintainability can be evaluated through several quality sub-characteristics like *analyzability*, *changeability*, *stability* and *testabilility* [ISO9126]. Object-orientation is believed to reduce the referred effort waste, if its basic mechanisms are used conveniently. We believe and expect to prove that, at the system level, there are patterns for the extent of use of encapsulation, inheritance, polymorphism or cooperation among classes which are closely correlated with those quality characteristics. By finding those patterns, through thorough experimental validation, we do not expect to identify a good design when we see one, but rather to say that a certain design is more maintainable than another. This is particularly useful for inexperienced designers, often faced with a combinatorial explosion of arbitrary design decisions.

Goal 2 - Improve the OO estimation process to achieve better resource allocation

Producing effort and schedule estimates for OO software development requires evaluating the size and complexity of the system to be built. The percentage that is going to be built from scratch and the percent that is going to be reused with minor or major adaptations (from existing component libraries) must also be evaluated, along with corresponding efforts. Selecting and adapting components, for instance, may demand considerable effort. A complete model for OO projects resource estimation that accommodates these concerns is our other goal.

2.2 Strategy

The strategy toward the stated goals can be expressed in a stepwise way. Some steps can be "climbed" concurrently.

Steps for Goal 1:

i) MOOD metrics set proposal (first introduced in [Abreu94]);
ii) practical validation of the underlying rationale of the proposed set, by means of a comparative evaluation of several "*supposed-to-be-well-designed*" OO systems (partly included in this paper);
iii) construction and public distribution of a tool for automatic collection of MOOD metrics from OO languages source code (scheduled for this summer); support for industrial and academic wide experimentation[5] and statistical validation of results;
iv) theoretical validation of the MOOD metrics (using Measurement Theory);
v) MOOD set refinement based on iii) and iv) results (MOOD V2 proposal);
vi) embedding MOOD V2 metrics on a OO CASE tool;
vii) assessment of correlation between MOOD metrics and maintainability sub-characteristics.

Steps for Goal 2:

i) proposal of a generic OO software system complexity metric (under way);
ii) theoretical validation of the complexity metric (using a set of *desiderata* defined in [Weyuker88]);
iii) develop a model for the effect of reuse in productivity, validated with published reports data;
iv) proposal of a resource estimation model[6];
v) construction and public distribution of a tool to support the estimation process, based on the proposed approach;
vi) public validation of the model with real-world OO projects;
vii) model calibration and refinement.

3. THE MOOD METRICS SET

3.1 Introduction

The MOOD (Metrics for Object Oriented Design) set includes[7] the following metrics:

- *Method Hiding Factor (MHF)*
- *Attribute Hiding Factor (AHF)*
- *Method Inheritance Factor (MIF)*
- *Attribute Inheritance Factor (AIF)*
- *Polymorphism Factor (PF)*
- *Coupling Factor (CF)*

4 - A private Portuguese non-profit R&D organization

5 - Potential MOODKIT users will be asked to disclose the metrics collected with it (anonymity of origin will be guaranteed on request), thus helping to enlarge the data set and calibrating the heuristics. Suggestions for tool improvement will be welcome..

6 - To be named MOORED (Model for Object Oriented Resource Estimation Determination)

7 - Although not the complete V1 set (which also includes the Clustering and Reuse Factors), these were the ones found relevant within the scope of this paper.

Each of those metrics refers to a basic structural mechanism of the object-oriented paradigm as *encapsulation* (MHF and AHF), *inheritance* (MIF and AIF), *polymorphism* (PF) and *message-passing* (CF) and are expressed as quotients. 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 on the same design (i.e. considering the same number of classes and inheritance relations). As a consequence, these metrics:

1. are *expressed as percentages*, ranging from 0% (no use) to 100% (maximum use);
2. are *dimensionless*, which avoids the often misleading, subjective or "artificial" units that pervaded the metrics literature with its often “esoteric” flavor.

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 measuring the same systems.
These metrics are also expected to be system *size independent.* A partial demonstration of this assertion is included below. Size independence allows inter-project comparison, thus fostering cumulative knowledge.
The MOOD metrics definitions make no reference to specific language constructs. However, since each language has its own constructs that allow for implementation of OO mechanisms in more or less detail, a binding for the case studies language (C++) is included ahead[8]. Similar bindings will be proposed for other OO languages in the near future. A validation experiment with Eiffel [Meyer92] is currently under way. This expected, but yet to be proved, *language independence* will broaden the applicability of this metric set by allowing comparison of heterogeneous system implementations.

3.2 A supporting example

To help clarify the metrics determination process, the following C++ code, adapted from [Young92], will be used in next sections. It is a subset of a class taxonomy where Application and Clock inherit from UIComponent. This last class inherits from BasicComponent (the base class).

```
class BasicComponent {

 protected:
        char    *_name;
        Widget   _w;
        BasicComponent( const char *); //Constructor
 public:
        virtual ~BasicComponent(); // Destructor
        virtual void manage();
        virtual void unmanage();
        const Widget baseWidget() { return _w; }
};

class UIComponent : public BasicComponent {

 private:
        static void widgetDestroyedCallback
                ( Widget, XtPointer, XtPointer );
 protected:
        UIComponent ( const char * ); //Constructor

        void installDestroyHandler();
        virtual void widgetDestroyed();
        void setDefaultResources
                ( const Widget , const String *);
        void getResources
                ( const XtResourceList, const int );
 public:
        virtual ~UIComponent();   // Destructor
        virtual void manage();
        virtual const char *const className()
                { return "UIComponent"; }
};

class Application : public UIComponent {

// Allow main and MainWindow to access protected member functions

#if (XlibSpecificationRelease>=5)
        friend void main ( int, char ** );
#else
        friend void main ( unsigned int, char ** );
#endif

friend class MainWindow;

 private:
        void registerWindow ( MainWindow * );
        void unregisterWindow ( MainWindow * );
 protected:
        Display     *_display;
        XtAppContext _appContext;

        // Functions to handle Xt interface
        #if (XlibSpecificationRelease>=5)
         virtual void initialize ( int *, char ** );
        #else
         virtual void initialize(unsigned int *,char **);
        #endif
        virtual void handleEvents();
```

[8] - There are still some pending issues related to how templates and exception handling (throw, try, catch, ...) should be considered.

```
        char            *_applicationClass;
        MainWindow      **_windows;
        int             _numWindows;
 public:
        Application ( char * );     // Constructor
        virtual ~Application();     // Destructor
        void manage();
        void unmanage();
        void iconify();
        Display   *display()
                { return _display; }
        XtAppContext   appContext()
                { return _appContext; }
        const char   *applicationClass()
                { return _applicationClass; }
        virtual const char *const className()
                { return "Application"; }
};

class Clock : public UIComponent {

 private:
        int       _delta;   // The time between ticks
        XtIntervalId _id;  // Xt Timeout identifier

        virtual void timeout();       // Called every delta
        virtual void speedChanged ( int );

        static void timeoutCallback
                ( XtPointer, XtIntervalId * );
        static void speedChangedCallback
                ( Widget, XtPointer, XtPointer );
 protected:
        virtual void tick()= 0;
 public:
        Clock ( Widget, char *,
                int         // Minimum speed
                int );      // Maximum speed
        ~Clock ();          // Destructor

        void stop();        // Stop the clock
        void pulse();       // Make the clock tick once
        void start();       // Start or restart the clock

        virtual const char *const className()
                { return ( "Clock" ); }
};
```

3.3 Metrics definition and language bindings

3.3.1 Method Hiding Factor:

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

where:

TC = total number of classes in the system under consideration

$M_d(C_i) = M_v(C_i) + M_h(C_i)$ =methods defined in C_i

$M_v(C_i)$ = visible methods in class C_i

$M_h(C_i)$ = hidden methods in class C_i

MOOD/C++ bindings

- methods - constructors, destructors, function members (including virtual[9] ones) and operator definitions
- visible methods - methods in public clause
- hidden methods - methods in private[10] and protected[11] clauses

Notes:

- inherited methods are not considered here;
- function members with the same identifier (*"function-name overloading")* but with different signatures (distinct formal parameter list) are considered as distinct methods.

Examples:

- $M_h(BasicComponent)$ = 1 { constructor }
- $M_v(BasicComponent)$ = 4 { destructor, manage, unmanage, baseWidget }
- $M_d(BasicComponent)$ = 1 + 4 = 5

- $M_h(UIComponent)$ = 6 { widgetDestroyedCallback, constructor, installDestroyHandler, widgetDestroyed, setDefaultResources, getResources }
- $M_v(UIComponent)$ = 3 { destructor, manage, className }
- $M_d(UIComponent)$ = 6 + 3 = 9

- $M_h(Application)$ = 5 { main, registerWindow, unregisterWindow, initialize, handleEvents }
- $M_v(Application)$ = 9 { constructor, destructor, manage, unmanage, iconify, display, appContext, applicationClass, className }
- $M_d(Application)$ = 5 + 9 = 14

- $M_h(Clock)$ = 5 { timeout, speedChanged, timeoutCallback, speedChangedCallback, tick }
- $M_v(Clock)$ = 6 { constructor, destructor, stop, pulse, start, className }
- $M_d(Clock)$ = 5 + 6 = 11

---

9 - Also called deferred

10 - These methods are only reachable within the scope of the class to which they belong (i.e. they are not even inherited by subclasses).

11 - These methods are only reachable within the scope of the class to which they belong and their derived classes (descendants).

12 - similar to the "generics" construct in the ADA language

3.3.2 Attribute Hiding Factor :

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

where:
$A_d(C_i) = A_v(C_i) + A_h(C_i)$ = attributes defined in C_i
$A_v(C_i)$ = visible attributes in class C_i
$A_h(C_i)$ = hidden attributes in class C_i

MOOD/C++ bindings
- attributes - data members
- visible attributes - data members in public clause
- hidden attributes - data members in private[13] and protected[14] clauses

Examples:
- $A_h(BasicComponent)$ = 2 { _name, _w }
- $A_v(BasicComponent)$ = 0
- $A_d(BasicComponent)$ = 2 + 0 = 2

- $A_h(UIComponent)$ = 0
- $A_v(UIComponent)$ = 0
- $A_d(UIComponent)$ = 0 + 0 = 0

- $A_h(Application)$ = 5 { _display, _appContext, _applicationClass, _windows, _numWindows }
- $A_v(Application)$ = 0
- $A_d(Application)$ = 5 + 0 = 5

- $A_h(Clock)$ = 2 { _delta, _id }
- $A_v(Clock)$ = 0
- $A_d(Clock)$ = 2 + 0 = 2

3.3.3 Method Inheritance Factor:

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

where:
$M_a(C_i) = M_d(C_i) + M_i(C_i)$= available methods in C_i
$M_d(C_i) = M_n(C_i) + M_o(C_i)$=methods defined in C_i

$M_n(C_i)$ = new methods in C_i
$M_o(C_i)$ = overriding methods in C_i
$M_i(C_i)$ = methods inherited in C_i

MOOD/C++ bindings
- methods defined - those declared within C_i
- new methods - those declared within C_i that do not override inherited ones
- overriding methods - those declared within C_i that override (redefine) inherited ones
- methods inherited - those inherited (and not overridden) in C_i
- available methods - those that can be invoked in association with C_i

Examples:
- $M_n(BasicComponent)$ = 5 { constructor, destructor, manage, unmanage, baseWidget }
- $M_o(BasicComponent)$ = 0
- $M_i(BasicComponent)$ = 0
- $M_d(BasicComponent)$ = 5 + 0 = 5
- $M_a(BasicComponent)$ = 5 + 0 = 5

- $M_n(UIComponent)$ = 8 { widgetDestroyedCallback, constructor, installDestroyHandler, widgetDestroyed, setDefaultResources, getResources, destructor, className }
- $M_o(UIComponent)$ = 1 { manage }
- $M_i(UIComponent)$ = 4 { BasicComponent, ~BasicComponent, unmanage, baseWidget }
- $M_d(UIComponent)$ = 8 + 1 = 9
- $M_a(UIComponent)$ = 9 + 4 = 13

- $M_n(Application)$ = 11 { main, registerWindow, unregisterWindow, initialize, handleEvents, constructor, destructor, iconify, display, appContext, applicationClass }
- $M_o(Application)$ = 3 { manage, unmanage, className }
- $M_i(Application)$ = 9 { BasicComponent, ~BasicComponent, baseWidget, UIComponent, installDestroyHandler, widgetDestroyed, setDefaultResources, getResources, ~UIComponent }
- $M_d(Application)$ = 11 + 3 = 14
- $M_a(Application)$ = 14 + 9 = 23

- $M_n(Clock)$ = 10 { timeout, speedChanged, timeoutCallback, speedChangedCallback, tick, constructor, destructor, stop, pulse, start }
- $M_o(Clock)$ = 1 { className }
- $M_i(Clock)$ = 11 { BasicComponent, ~BasicComponent, unmanage, baseWidget, UICom-

---

[13] - These attributes are only reachable within the scope of the class to which they belong (i.e. they are not even inherited by subclasses).
[14] - These attributes are only reachable within the scope of the class to which they belong and their descendants.
[15] - not included in any class

ponent, installDestroyHandler, widgetDestroyed, setDefaultResources, getResources, ~UIComponent, manage }

- $M_d(Clock)$ = 10 + 1 = 11
- $M_a(Clock)$ = 11 + 11 = 22

3.3.4 Attribute Inheritance Factor :

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

where:
$A_a(C_i) = A_d(C_i) + A_i(C_i)$ = attributes available in C_i
$A_d(C_i) = A_n(C_i) + A_o(C_i)$ = attributes defined in C_i
$A_n(C_i)$ = new attributes in class C_i
$A_o(C_i)$ = overriding attributes in class C_i
$A_i(C_i)$ = attributes inherited in class C_i

MOOD/C++ bindings

- attributes defined - those declared within C_i
- new attributes defined - those declared within C_i that do not override inherited ones
- overriding attributes - those declared within C_i that override (redefine) inherited ones
- attributes inherited - those inherited (and not overridden) in C_i
- available attributes - those that can be manipulated in association with C_i

Examples:

- $A_n(BasicComponent)$ = 2 { _name, _w }
- $A_o(BasicComponent)$ = 0
- $A_i(BasicComponent)$ = 0
- $A_d(BasicComponent)$ = 2 + 0 = 2
- $A_a(BasicComponent)$ = 2 + 0 = 2

- $A_n(UIComponent)$ = 0
- $A_o(UIComponent)$ = 0
- $A_i(UIComponent)$ = 2 { _name, _w }
- $A_d(UIComponent)$ = 0 + 0 = 0
- $A_a(UIComponent)$ = 0 + 2 = 2

- $A_n(Application)$ = 5 { _display, _appContext, _applicationClass, _windows, _numWindows }
- $A_o(Application)$ = 0
- $A_i(Application)$ = 2 { _name, _w }
- $A_d(Application)$ = 5 + 0 = 5
- $A_a(Application)$ = 5 + 2 = 7

- $A_n(Clock)$ = 2 { _delta, _id }
- $A_o(Clock)$ = 0
- $A_i(Clock)$ = 2 { _name, _w }
- $A_d(Clock)$ = 2 + 0 = 2
- $A_a(Clock)$ = 2 + 2 = 4

3.3.5 Polymorphism Factor:

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

where:
$M_o(C_i)$ = overriding methods in class C_i
$M_n(C_i)$ = new methods in class C_i
$DC(C_i)$ = number of descendants of class C_i (derived classes)

The numerator represents the *actual number of possible different polymorphic situations.* Indeed, a given message sent to class C_i can be bound (statically or dynamically) to a named method implementation, which can have as many shapes ("morphos" in ancient greek) as the number of times this same method is overridden (in C_i descendants).
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 derived classes.

Note: this metric definition was somewhat modified, when compared with its initial proposal [Abreu94]; this was a result of our better understanding, due to the experimental validation described in this paper.

Examples:

- $M_o(BasicComponent)$ = 0
- $M_n(BasicComponent)$ = 5 { constructor, destructor, manage, unmanage, baseWidget }
- $DC(BasicComponent)$ = 3 { UIComponent, Application, Clock }

- $M_o(UIComponent)$ = 1 { manage }
- $M_n(UIComponent)$ = 8 { widgetDestroyedCallback, constructor, installDestroyHandler, widgetDestroyed, setDefaultResources, getResources, destructor, className }
- $DC(UIComponent)$ = 2 { Application, Clock }

- $M_o(Application)$ = 3 { manage, unmanage, className }
- $M_n(Application)$ = 11 { main, registerWindow, unregisterWindow, initialize, handleEvents, cons-

tructor, destructor, iconify, display, appContext, applicationClass }

- $DC(Application)$ = 0

- $M_o(Clock)$ = 1 { className }
- $M_n(Clock)$ = 10 { timeout, speedChanged, timeoutCallback, speedChangedCallback, tick, constructor, destructor, stop, pulse, start }
- $DC(Clock)$ = 0

3.3.6 Coupling Factor:

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

where:

$TC^2 - TC$ = maximum number of couplings in a system with TC classes

$2\times\sum_{i=1}^{TC} DC(C_i)$ = maximum number of couplings due to inheritance

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

The client-server relation ($C_c \Rightarrow C_s$) means that C_c (*client* class) contains at least one reference to a feature (method or attribute) of class C_s (*supplier* class). $C_c \rightarrow C_s$ represents an inheritance relation (C_c inheriting from C_s). Because we want to evaluate *non-inheritance* coupling, C_c and C_s should not be tied by any inheritance relationship (direct or indirect). The numerator then represents the *actual number of couplings not imputable to inheritance.* The denominator stands for the *maximum possible number of non-inheritance couplings* in a system with TC classes.

Note: this metric definition was also refined, when compared with its initial proposal [Abreu94].

MOOD/C++ bindings

Client-server relations can have several shapes:

- regular message passing - a given class invokes another class interface function member;
- "forced" message passing - a given class invokes any type (visible or hidden) of function member from another class by means of a *friend* clause;
- object allocation and deallocation - a class invokes another class constructor or destructor;
- reference (in the client class) to a server class as an attribute or within a method formal parameter list - these references are due to semantic associations[16] among classes;

Example:

- *UIComponent* is client of class *MainWindow*

4. A FIELD TRIAL

4.1 The Tool

MOODKIT, a simplified tool for metrics extraction from source code, was developed and tested with the sample described in the next section. Version 1.1 supports the collection on C++ code of all MOOD metrics mentioned in this paper. It was built using ANSI C and scripts with standard UNIX commands (awk, grep, find, etc).

It is worth mentioning that the first attempt to collect the MOOD metrics (on the MFC library, described ahead) was done manually. It took an effort of about two man.weeks (two persons during a full week) and it became clear that the collection process is really a repetitive, tedious, boring, time-consuming and expensive task for humans! By using MOODKIT, the effort to do the same job was cut to a half man.day, something around 5% of the manual collection effort.

4.2 The Sample

The population to be modeled is not the whole set of software systems built using the OO paradigm. Instead, we want a representation of a population of "well designed" ones. We believe that the chosen set meets this goal, either because their elements have been in the public domain or commercial use for some time (with a great number of users trying them and suggesting upgrades) or because they were made by trustworthy teams with proven experience from academia and/or industry.

The sample measured with MOODKIT V1.1 was a collection of class libraries written in the C++ programming language. For each of those libraries a small synopsis is included below.

- **Microsoft Foundation Classes (MFC)**

Origin: Microsoft Corporation

Brief description: application framework designed for use in developing applications for Microsoft Windows; works in concert with the entire Microsoft Visual C++ development system, including the Visual WorkBench, AppStudio, AppWizard and ClassWizard; includes general-purpose classes for time, date, data-structures and file I/O, architectural classes to support commands, documents, views, printing and help, high level abstractions including toolbars, status bars, edit views, form views, print previews, scrolling views and

[16] - with a given arity (1:1, 1:n or n:m)

splitter windows and supports standard OLE user-interfaces and shared DLLs.

- **GNU glib++ (GNU)**

Origin: Free Software Foundation / Cygnus Support

Brief description: part of GNU's public domain programming environment; contains general purpose classes for manipulating strings, lists, files, etc.

- **ET++ library (ET++)**

Origin: UBILAB / Union des Banques Suisses (Switzerland)

Brief description: homogeneous library integrating user interfacing building blocks, basic data structures and support for object input/output with high level application framework components; it uses the terminology of Smalltalk-80 collection libraries.

- **NewMat library (NMAT)**

Origin: Robert B. Davies (robert.davies@vuw.ac.nz - Victoria University - New Zealand)

Brief description: package intended for scientists and engineers who need to manipulate a variety of types of matrices using standard matrix operations; emphasis is on the kind of operations needed in statistical calculations such as least squares, linear equation solving and eigenvalues.

- **MotifApp library (MOTIF)**

Origin: Douglas A. Young [Young92]

Brief description: public domain library providing a set of C++ classes on top of OSF/MOTIF for manipulation of menus, dialogs, windows and other widgets; it allows to use the OSF/MOTIF library in a OO style. Because its main creation purpose was academic (example included in a book), it is the smallest library and perhaps the least used (and reengineered) in "real-life" of all included in the sample. It is likely to generate some outliers.

For a better perspective over the sample, Table 1 includes some size metrics for each library. Rows refer to the total number of declared classes, total number of declared methods, total number of declared attributes (data members) and the total number of lines of code.

	MFC	**GNU**	**ET++**	**NMat**	**Motif**	TOTAL
Classes	128	84	262	86	35	595
Methods	3080	1478	4812	848	199	10417
Attrib.	608	151	980	125	76	1940
LOC	74895	15960	55022	12795	4884	163556

Table 1 - Some indicators of sample size

4.3 The Results

Table 2 and figure 1 summarize the results obtained through application of MOODKIT on the above sample. Even though the size of the sample is not as large as we would have liked, we consider it sufficiently meaningful for the purpose of the remaining study. The statistical analysis included in the next sections is expected to partially corroborate this assumption.

	MFC	***GNU***	***ET++***	***NMat***	***Motif***
MHF	24,6%	13,3%	9,6%	11,1%	39,2%
AHF	68,4%	84,1%	69,4%	76,8%	100,0%
MIF	83,2%	63,1%	83,9%	73,1%	64,3%
AIF	59,6%	62,6%	51,8%	56,6%	50,3%
COF	9,0%	2,8%	7,7%	27,1%	7,6%
PF	2,7%	3,5%	4,5%	12,2%	9,8%

Table 2 - Resulting metrics for the sample

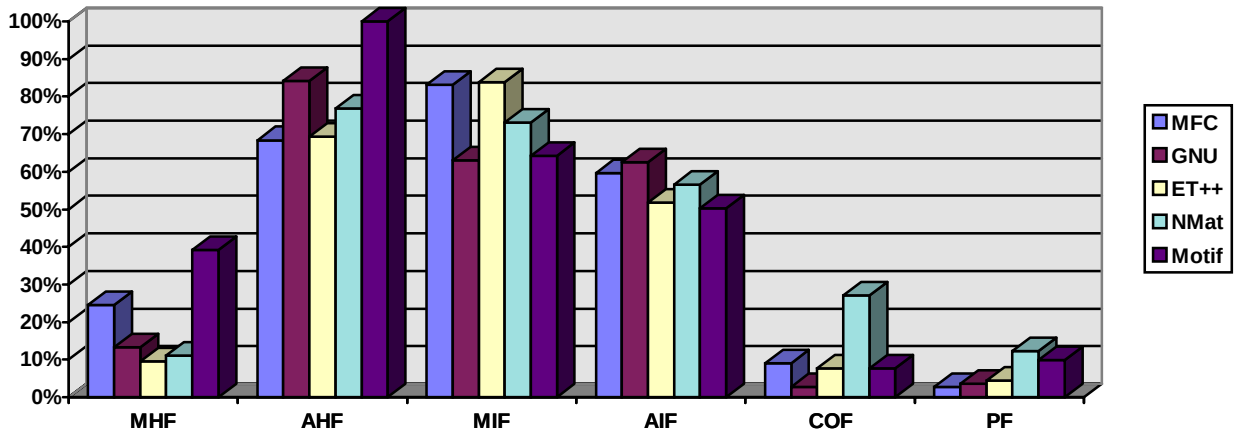

Figure 1 - MOOD metrics

5. DISCUSSION

5.1 Which shape for design heuristics?

The MOOD metric set enables expression of some recommendations for designers. This section explains the corresponding rationale. We will pick an Electronic Engineering analogy for representing our design heuristics. Let it be called "the filters metaphor".

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 more perfect, 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 an 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[17] 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 corresponding 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, for implementing that 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 classes interface should be a stepwise decomposition process, where more and more details are added. This decomposition will use hidden methods, thus getting 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 then the one of a *band-pass filter*.

MIF and AIF are measures of *inheritance*. This is 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.

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*.

Resulting *polymorphism* potential is measured through the *PF* 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[18]. 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.

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.

---

17 - Such is the case in the Motif library.

18 - 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.

	Minimum	*Mean*	*Maximum*	*Shape*
MHF	10,4%	*19,6%*	28,7%	*BP*
AHF	70,2%	*79,7%*	*89,3%*	*HP*
MIF	66,2%	*73,5%*	80,8%	*BP*
AIF	52,4%	*56,2%*	60,0%	*BP*
COF	3,9%	*10,8%*	17,7%	*BP*
PF	3,5%	*6,5%*	9,6%	*BP*

Table 3 - 90% Confidence interval for the sample

5.2 Confidence intervals

The confidence interval is a range on either side of the mean. If the sample has a considerable size (which is not yet the case), we can say with a particular level of confidence (90% in this case) that all the population metric values will lie in the specified intervals. Being more specific, if we assume that:
a) the population has a normal distribution
b) the sample is a good representative of the population
then the probability that further randomly sampled metrics lie inside the corresponding intervals is 90%. The accuracy of the intervals (i.e. their range reduction) is proportional to the square root of the sample size. We can then expect to disclose more accurate ranges and/or a bigger confidence (e.g. 95%) depending on our sample growth.

The first hypothesis (normal distribution) is a usual starting point for statistical analysis. Again, with a bigger sample, we may find that another type of distribution is a better representative. The second hypothesis (sample representativeness) has already been agreed upon in section 4.

Table 3 represents the 90% confidence interval for the sample mean of each MOOD metric. 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 values in the shaded zones. For instance, if the Coupling Factor exceeds 17,7% 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 "normal" boundaries of good practice and that the consequences may be the ones already referred. Besides this outlier identification, the MOOD metrics can also help decide among alternative design implementations by helping to rank them.

5.3 Size independence

In this section we will analyze the hypothesis formulated about the size independence of each MOOD metric. For that purpose we correlated each of the sizes included in Table 1, with the corresponding values of each metric (in Table 2). The resulting correlations follow:

	Classes	*Methods*	*Attrib.*	*LOC*
MHF	-0,59	-0,48	-0,37	-0,19
AHF	-0,74	-0,79	-0,74	-0,81
MIF	0,79	0,84	0,88	0,89
AIF	-0,19	-0,05	-0,21	0,14
COF	-0,14	-0,28	-0,25	-0,21
PF	-0,47	-0,66	-0,58	-0,70

Table 4 - Correlation of MOOD with some size metrics

Examining Table 4 we can find that all metrics except AHF and MIF (shaded zone) are fairly size independent as they show low correlations[19] with all size metrics. These anomalies might indicate one of two possibilities:

a) AHF and MIF are ill-defined as to what constitutes the desired size-independence;
b) the sample is somehow biased due to its small size;

We believe that hypothesis b) is more likely to be true, mainly because AHF and MIF have similar definitions to MHF and AIF, respectively, which show no significant correlation with any measure of size. Therefore, AHF and MIF size-dependence can not be conclusive until a bigger sample is available and analyzed. As our sample grows, we will wait until the correlations stabilize and then infer a more definite conclusion.

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. Besides, we think it is possible to interpret MOOD metrics as probabilities[20] in the sense that they quantify the presence or absence of a certain feature. This kind of interpretation allows the application of statistical theory to the MOOD metrics. If their statistical independent is proven, they can be combined (e.g. multiplied) so that the result can still be interpreted as a probability. With these reasons in mind, we evaluated the correlation among the sample value series for the defined metrics. Table 4 summarizes the results achieved.

	MHF	**AHF**	**MIF**	**AIF**	**COF**	**PF**
MHF		0,68	-0,34	-0,38	-0,28	0,17
AHF			-0,87	-0,31	-0,20	0,46
MIF				-0,13	0,15	-0,35
AIF					-0,09	-0,46
COF						0,75
PF						

Table 5 - Correlation among the MOOD metrics

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

[20] - A similar approach was used in the ESPRIT REBOOT project [Stalhane92].

Considering Table 5 we can conclude that most metric pairs (except the two shaded) exhibit a low correlation value.
The COF-PF correlation is due to the COF value for the NewMat library (27,1%) which is clearly an outlier. If its value was, for instance, equal to the other 4 libraries COF average (6,8%), then the COF-PF correlation would be 0,11!
Such an easy explanation for the AHF-MIF correlation was not found. We believe it is a coincidental situation, again stemming from a small sample size[21].

6. FUTURE WORK

6.1 MOODKIT evolution and CASE tools

A beta-test version (V1.2) of the MOODKIT tool, that will support the collection, storage and analysis of the MOOD metrics on C++ source code, is under construction and will soon be disclosed for public domain.
A completely reengineered version 2 is being designed. Its core (metrics definition dictionary, metrics storage, human-machine interface) will be based on a language independent central repository with storage, retrieval and graphical capabilities. It will use specific "stubs"[22] for metrics capture from distinct OO language source code. An Eiffel stub is being built and a Smalltalk one is also planned for the coming year. MOOD bindings for those languages will also be published in following papers.

There is an increasing interest from OO CASE tool makers in design metrics. Output from the ROSE tool[23], for instance, is being used at Rational [Fay94] to derive object-oriented metrics. Following this trend we are currently working with OBLOG Software[24], to extend their OBLOG CASE tool (supporting the OBLOG - OBject LOGic method [Sernadas91] with design metrics and heuristics based on MOOD.

6.2 Experimental validation

Using MOODKIT V1.0, we started an extensive evaluation of available systems (C++ class libraries) and tried to derive and refine some design heuristics based on the results (metrics) obtained. The public availability of a next version of this toolkit, will allow people either from industry or academia to replicate the experiment herein described with other OO systems. The increase of the sample size will allow us to divide it in percentiles, discard outliers and achieve better confidence intervals. The refined heuristics will again be published.

As already mentioned, we also expect to prove that the MOOD metrics are fairly implementation language independent. Size independence and statistical independence among metrics will also be further and further assessed by means of correlation analysis.

6.3 Analysis metrics

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 were committed. Looking at the analysis instead of design would then be a step forward towards 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[25], 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.

7. FINAL OVERVIEW

The adoption of the Object-Oriented paradigm is expected to help produce better and cheaper software. The main structural mechanisms of this paradigm, namely, *inheritance*, *encapsulation*, *information hiding or polymorphism*, are the keys to foster reuse and achieve easier maintainability. However, the use of language constructs that support those mechanisms can be more or less intensive, depending mostly on the designer ability. We can then expect rather different quality products to emerge, as well as different productivity gains. Advances in quality and productivity need to be correlated with the use of those constructs. We then need to evaluate this use quantitatively to guide OO design. A validation experiment of a metric set named MOOD, suited for this purpose was presented in this paper. This set allows comparison of different systems or different implementations of the same system. Some design heuristics based on a filter metaphor were introduced. Their automatic attainment was achieved

21 - Note that these metrics (AHF and MIF) are the same that have shown anomalies on the previous section.

22 - Based on language parsers

23 - supporting the Booch method

24 - A private Portuguese R&D company owned by Espírito Santo Bank

25 - [Darscht94] reports the intention to build an OO metrics collection prototype integrated with this tool.

and discussed on a sample of C++ libraries. It is hoped that those heuristics will be of some help to novice designers.

Object-orientation is not “the” silver bullet [Brooks86], but it seems to be the best bullet available today to face the pervasive software crisis. Keeping on the evolution track means we must be able to quantify our improvements. Metrics will help us to achieve this goal.

REFERENCES

[Abreu93] Abreu, F. Brito and Carapuça, Rogério, "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 the 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.

[Brooks86] Brooks, Frederick P. Jr., "Essence and Accidents of Software Engineering", Proceedings of *Information Processing 86*, H.-J. Kugler (ed.), Elsevier Science Publishers B. V. (North Holland), IFIP 86, also published in IEEE Computer, April 1987.

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

[Cant94]S.N. Cant, B. Henderson-Sellers, and D.R. Jeffery, "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.

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

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

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

[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.

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

[Lewis91] Lewis, John A. and Henry, Sallie M. and Kafura, Dennis G. : "An Empirical Study of the Object-Oriented Paradigm and Software Reuse" Proc. of *ACM SIGPLAN Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'91)*, pp. 184-196, 1991.

[Melo95] Melo Walcélio L., Briand Lionel and Basili Victor 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 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.

[Sellers95] Henderson-Sellers, B., "Identifying internal and external characteristics of classes likely to be useful as structural complexity metrics”, *Proceedings of 1994 Intern. Conf. 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 Modelling”, *Data and Knowledge Engineering*, vol.6, n.6, pp.479-508, 1991.

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

[Young92] Young, D.A., *Object-Oriented Programming with C++ and OSF/MOTIF*, Prentice-Hall, 1992