Evaluating the Impact of Object-Oriented Design on Software Quality

Fernando Brito e Abreu
ISEG / Lisbon Technical University and INESC
Rua Alves Redol, 9, Apartado 13069
1000 Lisbon, Portugal
fba@inesc.pt

Walcélio Melo
University of Maryland
Institute for Advanced Computer Studies
A.V.Williams Bldg., College Park
MD 20742, USA
melo@cs.umd.edu

Accepted for Publication
3rd Int'l S/W Metrics Symposium, March 1996, Berlin, Germany.

Abstract

This paper describes the results of a study where the impact of Object-Oriented design on software quality characteristics is experimentally evaluated. A suite of metrics for OO design called MOOD was adopted to measure the use of OO design mechanisms. Data collected on the development of eight small-sized information management systems based on identical requirements was used to assess the referred impact. The results obtained in this experiment show how OO design mechanisms like inheritance, polymorphism, information hiding and coupling influence quality characteristics such as defect density and rework. Predictive models based on OO design metrics built in this study are also presented.

***Key-words**: Metrics for Object-Oriented Design; Software Quality Characteristics; Error Prediction Model; Effort Prediction Model; Rework on Object-Oriented Software Development.*

1. Introduction

Software metrics help managers in several activities of the development life cycle such as scheduling, costing, staffing and controlling and therefore contribute to the overall objective of software quality. Their need is fully recognized by the software engineering community and included in standards like [IEEE1061, 1990], [ISO9000/3, 1991], [ISO9126, 1991] and [ISO14598, 1995].

Besides process factors, also product factors are expected to influence the resulting software quality. Among them is the design. The analysis to design transition is an activity where a skeleton for a computable implementation supporting the defined system requirements is defined. This transition often offers several degrees of liberty. Decisions on best alternatives are usually fuzzy and mostly based on expert judgment. By other words, cumulative knowledge plays a very important part in the design phase. The intensive use of patterns, frameworks and other reusable

components is expected to ease this open problem, but current practice is still far from widespread adoption.

Novice designers are therefore exposed to a myriad of design decisions that surely affect the final outcome. Being able to predict some of its features is one of our great motivations. This availability will allow to guide the designing process for instance by means of heuristics. Perhaps the most well known heuristic for object-oriented design is the Law of Demeter [Lieberherr et al., 1989]. This "law" restricts the message sending structure of methods in order to organize and reduce dependencies between classes. The authors say "...We believe that the Law of Demeter promotes maintainability and comprehensibility, but to prove this in absolute terms would require a large experiment with a statistical evaluation. ...". Unfortunately, to the extent of our knowledge, that hasn't been done yet.

The main goal of this paper is to evaluate the impact of OO design on software quality characteristics such as *defect density and rework* by mean of experimental validation. In order to measure the OO design characteristics, a suite of metrics called MOOD [Abreu et al., 1994] was adopted. Motivations behind the MOOD set definition were: *(1) coverage* of the basic structural mechanisms of the object-oriented paradigm as *encapsulation*, *inheritance*, *polymorphism* and *message-passing, (2) formal definition* to avoid subjectivity of measurement and thus allow replicability, *(3) size independence* to allow inter-project comparison, thus fostering cumulative knowledge and *(4) language independence* to broad the applicability of this metric set by allowing comparison of heterogeneous system implementations.

The outline of this paper is the following: section 2 presents the MOOD metrics suite for OO design; section 3 describes an experiment where process and product metrics were collected; section 4 includes the statistical analysis on the collected data, discusses the validation of the adopted metrics set and finally proposes derived predictive models; section 5 includes an overview of related research works; finally, section 6 concludes the paper by presenting lessons learned and future work.

2. The suite of metrics for Object-Oriented Design

2.1 Introduction

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

- Method Hiding Factor (MHF)
- Attribute Hiding Factor (AHF)
- Method Inheritance Factor (MIF)
- Attribute Inheritance Factor (AIF)
- Polymorphism Factor (POF)
- Coupling Factor (COF)

Each of these metrics refers to a basic structural mechanism of the object-oriented paradigm as *encapsulation* (MHF and AHF), *inheritance* (MIF and AIF), *polymorphism* (POF) and *message-passing* (COF). 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, an abstracted binding for two OO language**s**, C++ [Stroustrup,

1991] and Eiffel [Meyer, 1992], is included ahead1. In the remainder of this section an overview of those metrics will be provided. Readers familiarized with MOOD can skip this section.

2.2 Metrics definition and language bindings

2.2.1 Method Hiding Factor:

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

where:

$$V(M_{mi}) = \frac{\sum_{j=1}^{TC} is_visible(M_{mi}, C_j)}{TC}$$

$$is_visible(M_{mi}, C_j) = \begin{cases} 1 & iff \quad C_j \quad can \quad call \quad M_{mi} \\ 0 & otherwise \end{cases}$$

MOOD		C++	Eiffel
TC	***total classes***	*total number of classes*	*same as for C++*
	methods	*constructors; destructors; function members2; operator definitions*	*class features with implementation (**do** clause) or without it (**deferred** clause); **external** functions; constants with **once** clause*
Md(Ci)	***methods defined*** *(not inherited)*	*all methods declared in the class including **virtual** (deferred) ones*	*all methods declared in the class, even if declared **obsolete**;*
V(Mmi)	***visibility*** *- % of the total classes from which the method Mmi is visible*	*=1 for methods in **public** clause; =1/TC for methods in **private** clause; =(1+DC(Ci))/TC for methods in **protected** clause (note: DC(Ci)descendants of Ci*	*= 1 by omission or if **ANY** is mentioned; = (1 + DC(Ci))/TC if **NONE** or empty brackets **{ }** are mentioned; else = (1+ DC(Ci)+ number of classes between brackets **{...}** + their descendants + exports) / TC*

2.2.2 Attribute Hiding Factor :

---

1 - See [Abreu et al., 1995] for more details on C++ binding.

2 - Function members with the same identifier ("function-name overloading") but with different signatures (distinct formal parameter list) are considered as distinct methods.

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

where:

$$V(A_{mi}) = \frac{\sum_{j=1}^{TC} is_visible(A_{mi}, C_j)}{TC}$$

$$is_visible(A_{mi}, C_j) = \begin{cases} 1 & iff \quad C_j \quad can \quad reference \quad A_{mi} \\ 0 & otherwise \end{cases}$$

MOOD		C++	Eiffel
Ad(Ci)	***attributes defined*** *(not inherited)*	*data members*	*class features without implementation; simple typed constants (integer, boolean, character)*
V(Ami)	***visibility*** *- % of the total classes from which Ami is visible*	*= 1 for attributes in* ***public*** *clause;* *= 1/TC for attributes in* ***private*** *clause;* *=(1+DC(Ci))/TC for attributes in* ***protected*** *clause* *note: DC(Ci)= descendants of Ci;*	*= 1 by omission or if* ***ANY*** *is mentioned;* *= (1 + DC(Ci))/TC if* ***NONE*** *or empty brackets* ***{ }*** *are mentioned;* *else =(1+DC(Ci)+ number of classes between brackets* ***{...}*** *+ their descendants + exports) / TC*

2.2.3 Method Inheritance Factor:

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

where:

$$M_a(C_i) = M_d(C_i) + M_i(C_i)$$

MOOD		C++	Eiffel
$M_a(C_i)$	***available methods***	*function members that can be invoked in association with Ci*	*features that can be invoked in association with Ci*
$M_d(C_i)$	***methods defined***	*function members declared within Ci*	*features declared within Ci*
$M_i(C_i)$	***inherited methods***	*function members inherited (and not overridden) in Ci*	*features inherited in Ci and not in* ***redefine*** *or* ***undefine*** *clauses*

2.2.4 Attribute Inheritance Factor :

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

where:

$$A_a(C_i) = A_d(C_i) + A_i(C_i)$$

MOOD		C++	Eiffel
$A_a(C_i)$	**available attributes**	*data members that can be invoked associated with Ci*	*similar to* $M_a(C_i)$
$A_d(C_i)$	**attributes defined**	*data members declared within Ci*	*similar to* $M_d(C_i)$
$A_i(C_i)$	**inherited attributes**	*data members inherited (and not overridden) in Ci*	*similar to* $M_i(C_i)$

2.2.5 Polymorphism Factor:

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

where:

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

MOOD		C++	Eiffel
$DC(Ci)$	**descendants count**	*number of classes descending from Ci*	*number of classes descending from Ci*
$M_n(C_i)$	**new methods**	*function members declared within Ci that do not override inherited ones*	*features declared within Ci that do not override inherited ones*
$M_o(C_i)$	**overriding methods**	*function members declared within Ci that override (redefine) inherited ones*	*features in **redefine** and **undefine** clauses; deferred features which were inherited and implemented in Ci*

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 their all derived classes.

2.2.6 Coupling Factor:

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

where:

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

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

The client-server relation ($C_c \Rightarrow C_s$) means that C_c (*client* class) contains *at least one* non-inheritance reference to a feature (method or attribute) of class C_s (*supplier* class). The numerator then represents the *actual number of couplings not imputable to inheritance.* The denominator stands for the *maximum possible number of couplings* in a system with TC classes. Client-server relations can have several shapes:

Client-Server shapes	***C++***	***Eiffel***
regular message passing	*call to the interface of a function member in another class;*	*call to a feature in the client class*
"forced" message passing	*call to a visible or hidden function member in another class by means of a* ***friend*** *clause;*	*doesn't apply*
object allocation and deallocation	*call to class constructor or destructor;*	*call to features in* ***creation*** *clause; there is no explicit deallocation3*
semantic associations among classes with a certain arity (e.g. 1:1, 1:n or n:m);	*reference to a server class as a data member or as a formal parameter in a function member interface*	*reference to a server class as a formal parameter in a feature interface; formal parameter of a* ***generic*** *class; reference to a server class as a local typed feature*

3. Controlled Data Collection Experiment

The impact of OO design on software quality will be evaluated by examining the degree to which MOOD metrics correlate with *defect density* (a ***reliability*** measure) and *normalized rework* (corrective maintenance effort, a ***maintainability*** measure). Data gathered in a controlled experiment performed at the University of Maryland [Melo et al, 1995] was used. Section 3.1

3 - Eiffel has an automatic garbage collection mechanism.

provides further details about this experiment and section 3.2 describes the product and process measures that were collected in it.

3.1 Description of the experiment

This experiment was carried out from September to December, 1994. The population under study was a graduate and senior level class offered by the Department of Computer Science at the University of Maryland. All students had some experience with C or C++ programming and relational databases.

The students were randomly grouped into teams. Each team developed a medium-size management information system that supported the rental/return process of a hypothetical video rental business and maintained customer and video databases.

The development process was performed according to a sequential software engineering life-cycle model derived from the Waterfall model. This model includes the following phases: Analysis, Design, Implementation, Testing, and Repair. By the end of each phase a document was delivered: requirements specification, design document, code, error report and modified code, respectively. Requirements specification and design documents were reviewed by an expert in order to verify if they matched the system requirements. Errors found in these two first phases were reported to the students. This guaranteed that the implementation began with a correct OO analysis/design. The testing phase was accomplished by an independent group composed of experienced software professionals. This group tested all systems according to similar test plans and using functional testing techniques. During the repair phase, the students were asked to correct their systems based on the errors found by the independent test group.

The development environment and technology used were consistent with current practice in industry and academia. OMT [Rumbaugh et al, 1991], an OO Analysis and Design method, was used during the analysis and design phases. The C++ programming language, the GNU software development environment and OSF/MOTIF were used during the implementation. Sun Sparc stations were used as the implementation platform.

The following libraries were provided to the students:

- MotifApp - this public domain library [Young, 1992] provides a set of C++ classes on top of OSF/MOTIF for manipulation of windows, dialogs, menus, etc. The MotifApp library provides a way to use the OSF/Motif library in an OO programming/design style.
- GNU library - this library is a public domain library provided in the GNU C++ programming environment. It contains functions for manipulation of strings, files, lists, etc.
- C++ database library - this library provides a C++ implementation of multi-indexed B-Trees.

Libraries' source code and complete documentation were made available as well as a hundred small programs exemplifying how to use OSF/Motif widgets. A domain specific application library was also provided in order to make the experiment more representative of the ``real world". This library implemented the graphical user interface for insertion and removal of customers and was implemented in such a way that the main resources of the OSF/Motif and MotifApp libraries were used. Because students were not mandated to use the libraries, each design team adopted different reuse options.

3.2 Collected data

Both product and process data were gathered as a part of this experiment. Only the relevant data that helped the MOOD metrics validation process will be described here. For further details about how these data were gathered and validated see [Melo et al, 1995].

3.2.1 Product design data

MOODKIT, a tool to extract MOOD metrics from C++ or Eiffel source code was built and is being maintained at INESC. MOODKIT V2 runs on a UNIX platform with Motif interface and its usage is free for those who are available to share the collected data. MOODKIT V1.3 (an older version) was used in this experiment to analyze the 8 projects. Table 1 shows the MOOD metrics for each project as well as other descriptive statistics (mean, standard deviation, maximum and minimum).

Project	MHF	AHF	MIF	AIF	POF	COF
1	15,5%	67,4%	47,8%	18,8%	11,9%	3,9%
2	0,0%	0,0%	23,1%	18,8%	0,0%	15,9%
3	36,4%	95,7%	38,2%	22,4%	6,0%	2,3%
4	3,6%	93,9%	10,8%	0,0%	0,0%	3,9%
5	15,1%	50,9%	53,0%	44,8%	1,3%	6,1%
6	22,4%	74,4%	46,6%	26,1%	7,4%	6,2%
7	24,9%	97,7%	41,5%	34,1%	6,9%	2,8%
8	0,0%	98,7%	0,0%	0,0%	0,0%	22,4%
Minimum	0,0%	0,0%	0,0%	0,0%	0,0%	2,3%
Mean	14,7%	72,3%	32,6%	20,6%	4,2%	8,0%
Maximum	36,4%	98,7%	53,0%	44,8%	11,9%	22,4%
Std. Dev.	13,0%	33,9%	19,2%	15,4%	4,5%	7,3%

Table 1: MOOD metrics for each project

3.2.2 Process data

To collect *rework* effort, forms were filled out by the developers to track person-hours spent on isolating errors, as well as correcting them. The number of *defects found* was also collected. *Defect* is a generic designation that refers to either an *error* or a *fault*. Errors and faults are two pertinent ways to count defects, and so they were both considered in this experiment. *Errors* are defects in the human thinking process made while trying to understand given information, solving problems or using methods and tools. *Faults* are concrete manifestations of errors within the software. One error may cause different faults, although distinct errors may cause similar faults. In this experiment an error is assumed to be represented by a single error report form; a fault is represented by a physical change to a component.

Table 2 shows the project ID, project size ([SLOC] - Source Lines Of Code), number of errors and faults found, error and fault density ([KSLOC-1]), rework ([person.hours]) and normalized rework ([person.hours.KSLOC-1]) for all projects considered in the experiment.

Project	Size (SLOC)	Error Density	Fault Density	Rework	Normalized Rework
1	13981	1,72	3,08	51	3,65
2	5068	6,51	8,29	71	14,01
3	9735	4,31	4,52	92	9,45
4	8543	3,86	4,80	72	8,43
5	8173	3,18	8,20	59	7,22
6	6368	3,93	4,40	51	8,01
7	6571	2,28	2,43	31	4,72
8	5068	8,68	14,80	93	18,35

Table 2: Process metrics for each project

4. Discussion of Results

4.1 The impact of OO properties on software project quality

To provide some evidence about the relationship between OO design and software project quality, the correlation between the MOOD metrics and the quality measures of defect density (fault and error density) and normalized rework (effort spent on repairing errors) was determined. The resulting coefficients of correlation are shown in Table 3.

	MHF	AHF	MIF	AIF	POF	COF
Error Density	-0,565	-0,127	-0,781	-0,558	-0,683	0,914
Fault Density	-0,629	-0,126	-0,635	-0,373	-0,691	0,913
Normalized Rework	-0,569	-0,143	-0,780	-0,549	-0,707	0,907

Table 3: Pearson r correlation coefficients of MOOD and quality measures

Based on the data provided in Table 3, the following conclusions can be drawn:

- Methods Hiding Factor (MHF) has a moderate negative correlation with defect density (error and fault densities) and rework. This means that once MHF increases the defect density and the effort spent to fix errors will be expected to decrease. As expected, the procedural abstraction that supports the top-down development approach is an appropriate technique to increase software quality. In fact the implementation of a class interface should be a stepwise decomposition process, where more and more details (hidden methods) are added. This decomposition will then favor a MHF increase along with the mentioned quality benefits.

- Attributes Hiding Factor (AHF) did not show any significant correlation. This was a bit of a surprise because it was expected that data encapsulation would have a bigger impact on software quality. In fact *information hiding,* supported by the *encapsulation* mechanism, allows to cope with complexity by looking at complex components as “black boxes” and thus reducing "side-effects". Ideally, all attributes would be hidden and only accessed by the corresponding class’ methods (AHF=100%). The availability of other (bigger) samples will allow more conclusive assertions on the AHF to quality impact.

- Methods Inheritance Factor (MIF) has a moderate negative correlation with fault density and a high negative correlation with both error density and normalized rework measure. This means that once MIF increases the defect density and the effort spent to fix errors will be expected to decrease. These results show how inheritance, one of the most controversial concepts in OO design, appears to be an appropriate technique to reduce error density and rework, when used sparingly. Very high values of MIF (above the 70% to 80% range [Abreu et al, 1995]) are supposed to reverse this benefit effect, but this assumption still lacks experimental validation4.

- Attributes Inheritance Factor (AIF) has a low negative correlation with fault density and a moderate negative correlation with both error density and normalized rework measure. This result does not allow to draw any strongly supported conclusions.

---

4 - An increased depth and width of the inheritance hierarchy trees makes understandability and testability fade away. However, as seen in table 1, MIF values were very low in all projects.

- Polymorphism Factor (POF) has a moderate to high negative correlation with error and fault densities as well as with rework. This means that an appropriate use of polymorphism in OO project designs should decrease the defect density as well as rework. However, very high values of POF (well above 10%, which is not the case in this sample) are expected to reduce this quality benefits. In fact, to understand and debug a highly polymorphical hierarchy, for instance by tracing the control flow, will be much harder than the procedural counterpart, where for a similar functionality we usually have a series of decision statements for triggering the required operation.

- Coupling Factor (COF) has a very high positive correlation with all quality measures. Therefore, as coupling among classes increases, the defect density and normalized rework is also expected to increase. This result shows that coupling in software systems has a strong negative impact on software quality and then should be avoided during design. In fact, many authors like [Meyer, 1988] have noted that it is desirable that classes communicate with as few others as possible because coupling relations increase complexity, reduce encapsulation and potential reuse, and limit understandability and maintainability.

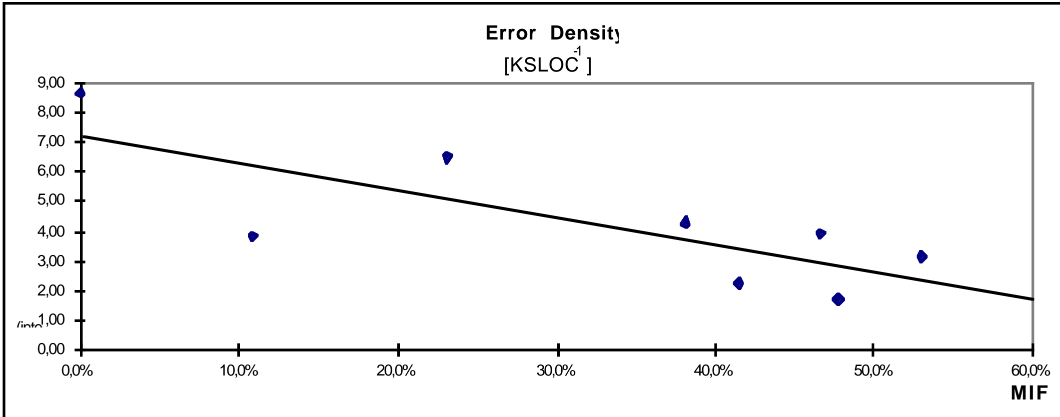
4.2 Predicting defect density and rework

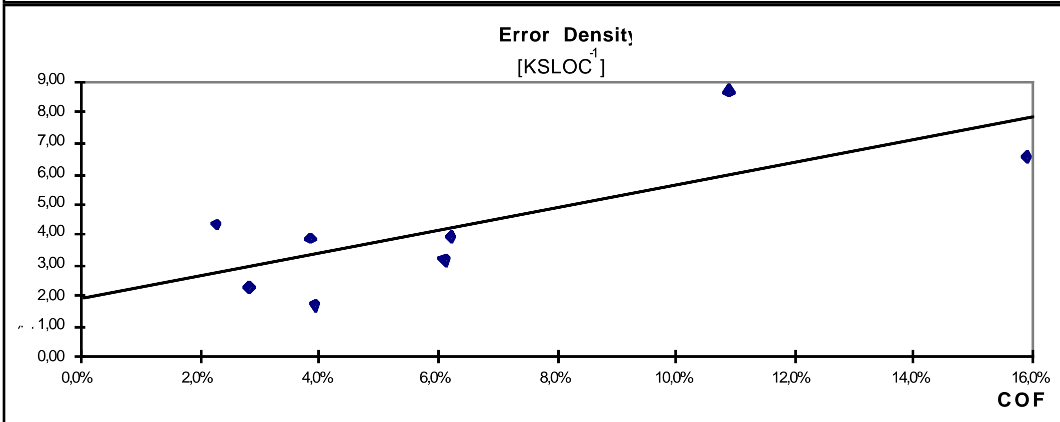
Predictive models can be developed to quantify the impact of OO design on software quality. Three MOOD metrics (MIF, COF and POF) that proved to better correlate with defect density and normalized rework were chosen. Table 4 shows the predictive models built using univariate linear regression between the statistically significant MOOD metrics and defect density and rework.

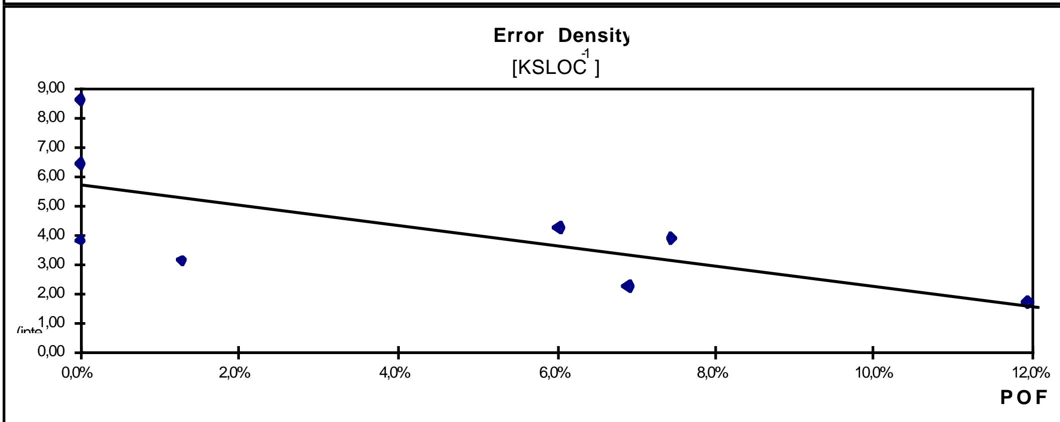
		MIF	***COF***	***POF***
Error Density	Intercept	7,34	1,94	5,76
	Slope	-9,27	36,43	-34,50
Fault Density	Intercept	10,67	2,49	8,91
	Slope	-13,34	58,74	-61,80
Normalized Rework	Intercept	15,64	4,20	12,41
	Slope	-19,64	77,31	-75,76

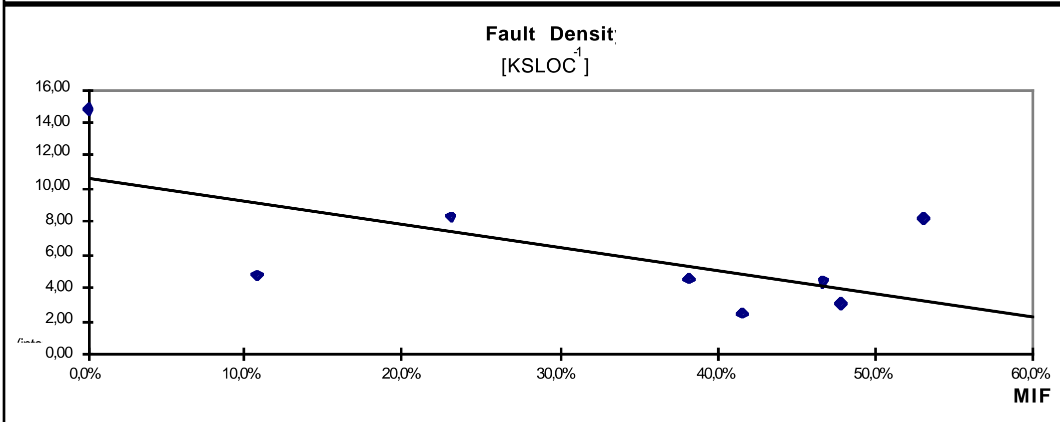
Table 4: Predictive models for defect density and rework

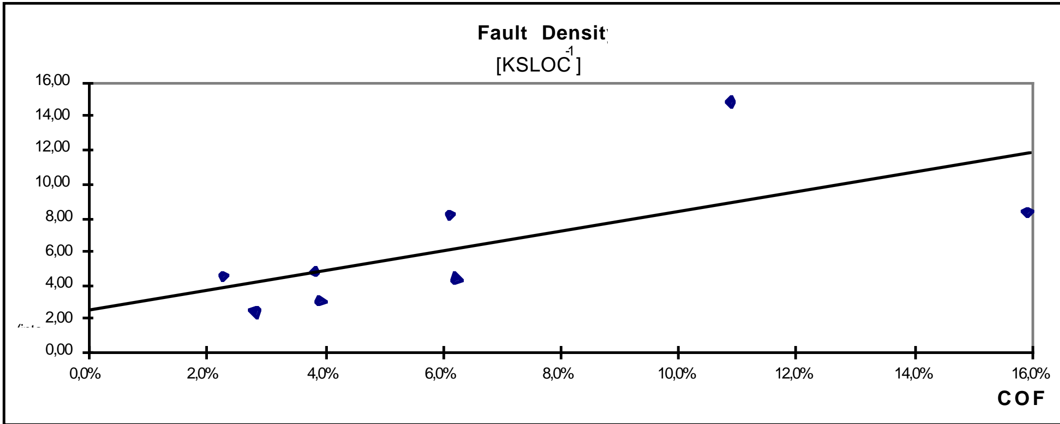
Next 9 figures represent the predictive models graphically. Estimates on errors and faults to be found, as well as rework effort, are obtainable straightforwardly if MOOD metrics (independent variables) are available.

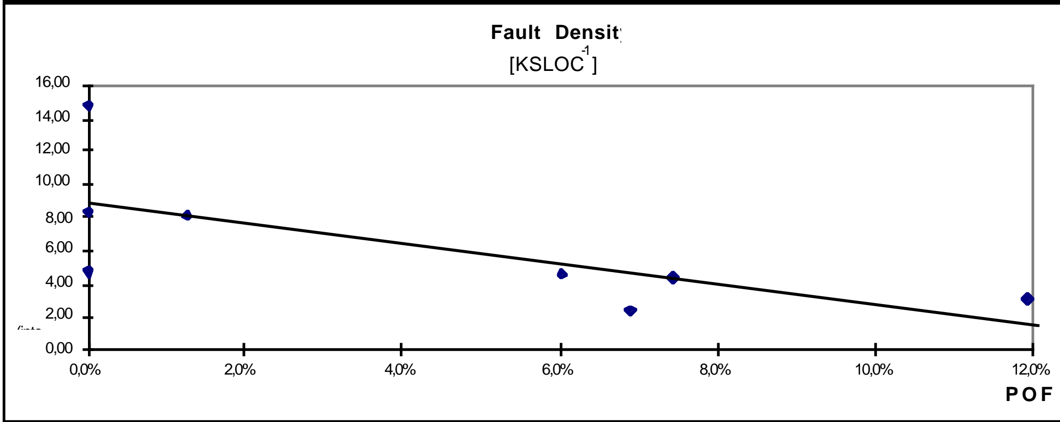

Error Density
[KSLOC-1]
9,00
8,00
7,00
6,00
5,00
4,00
3,00
2,00
1,00
0,00
0,0%
10,0%
20,0%
30,0%
40,0%
50,0%
60,0%
MIF

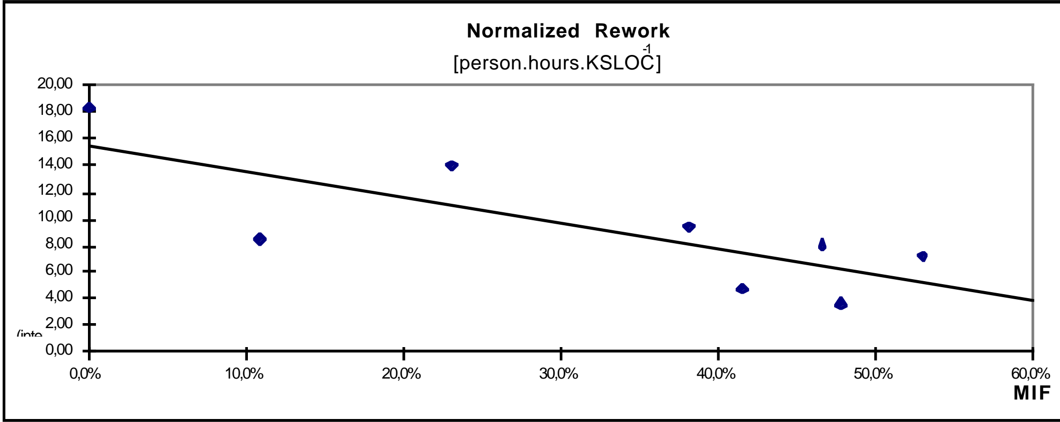

Error Density
[KSLOC-1]
9,00
8,00
7,00
6,00
5,00
4,00
3,00
2,00
1,00
0,00
0,0%
2,0%
4,0%
6,0%
8,0%
10,0%
12,0%
14,0%
16,0%
COF

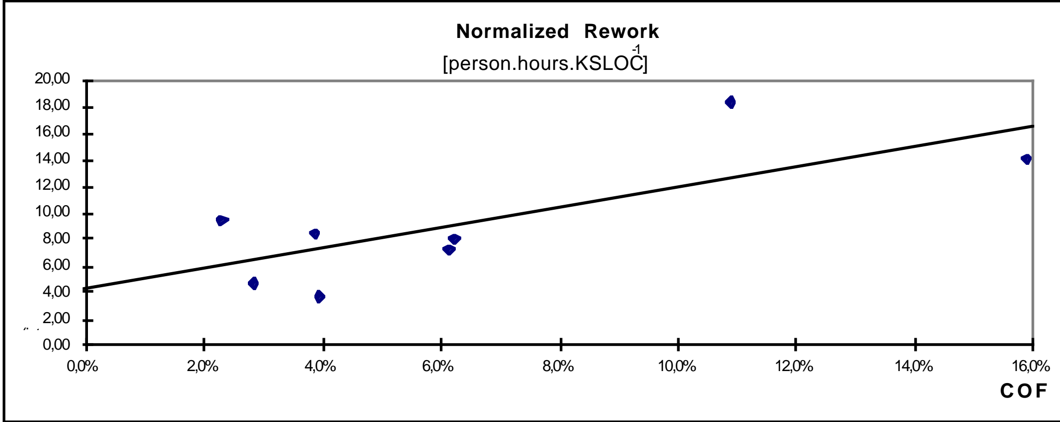

Error Density
[KSLOC-1]
9,00
8,00
7,00
6,00
5,00
4,00
3,00
2,00
1,00
0,00
0,0%
2,0%
4,0%
6,0%
8,0%
10,0%
12,0%
POF

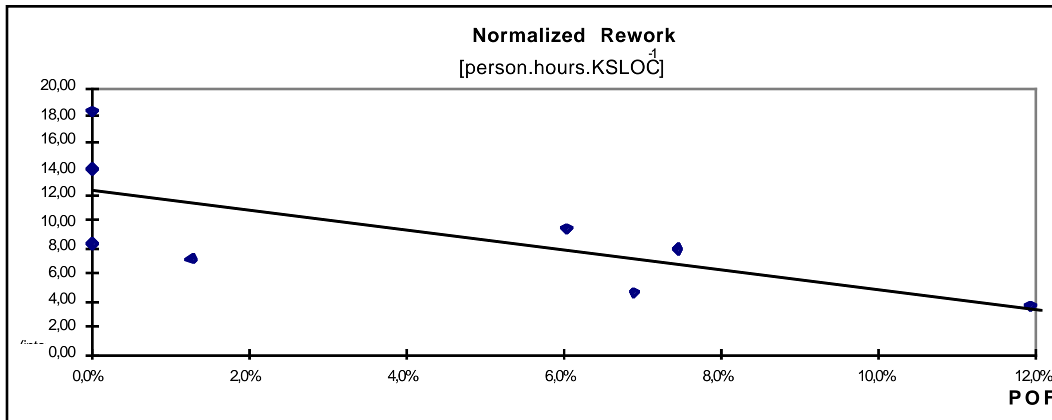

Fault Density
[KSLOC-1]
16,00
14,00
12,00
10,00
8,00
6,00
4,00
2,00
0,00
0,0%
10,0%
20,0%
30,0%
40,0%
50,0%
60,0%
MIF


Fault Densit
[KSLOC-1]
16,00
14,00
12,00
10,00
8,00
6,00
4,00
2,00
0,00
0,0%
2,0%
4,0%
6,0%
8,0%
10,0%
12,0%
14,0%
16,0%
COF


Fault Densit
[KSLOC-1]
16,00
14,00
12,00
10,00
8,00
6,00
4,00
2,00
0,00
0,0%
2,0%
4,0%
6,0%
8,0%
10,0%
12,0%
POF


Normalized Rework
[person.hours.KSLOC-1]
20,00
18,00
16,00
14,00
12,00
10,00
8,00
6,00
4,00
2,00
0,00
0,0%
10,0%
20,0%
30,0%
40,0%
50,0%
60,0%
MIF


Normalized Rework
[person.hours.KSLOC-1]
20,00
18,00
16,00
14,00
12,00
10,00
8,00
6,00
4,00
2,00
0,00
0,0%
2,0%
4,0%
6,0%
8,0%
10,0%
12,0%
14,0%
16,0%
COF



The MOOD metrics are supposed to traduce independent aspects of the design and therefore their effect on quality can be assessed individually as done in this study. The cumulative effect of all those design features on software quality, where individual influences are identified, will be the target of a future study.

5. Related Work

Since the early days of computer science many approaches to quantify the internal structure of procedural software systems have emerged [Zuse, 1991]. Some of those "traditional" metrics can still be used with the object-oriented paradigm, specially at the method level [Abreu et al., 1993]. However, the need to quantify the distinctive features of this paradigm gave birth to new metric sets in recent years. Most of those metrics haven't been experimentally validated yet. This validation step usually consists of correlation studies between internal (design) attributes and external (quality) attributes. A brief survey of known validation efforts follows.

The MOOSE metrics proposed in [Chidamber et al., 1994] were validated in [Basili et al., 1995] using the same project data as the one in this paper. Besides discussing the metrics' advantages and drawbacks, the authors claim that several of them appear to be adequate to predict class fault-proneness during the early phases of the life-cycle. Nevertheless, some severe critics on the MOOSE metrics' imprecise and ambiguous definition (lack of language bindings) were raised in [Churcher et al., 1995].

In [Li et al., 1994] the authors used an extension of the MOOSE set to build a regression model that is said to be adequate to predict changeability (effort of correcting or enhancing classes). The model was validated with data from two systems built with an object-oriented dialect of Ada.

[Abbott et al., 1994] proposed a metric derived from the design information captured in class definitions for measuring the number and strength of the object interactions and claim it is useful for predicting experts' design preferences. To validate the metric they used 9 sets of 2 or 3 design alternatives and compared the evaluations suggested by both the proposed metric and a panel of object-oriented design experts. They found out that the preferred alternatives were coincident in 80% of the cases.

Module and system level metrics for information hiding are described in [Rising et al., 1994]. A validation experiment based on a system with approximately one million lines of Ada5 code is described. Results showed that those metrics were able to "discriminate between packages that are

5 - According to [Wegner, 1987] Ada may be considered as *object-based* but not *object-oriented* because its objects (packages) do not have a class (type).

or are not likely to undergo significant changes". On the other hand the authors recognize that the same experiment showed that there is no linear correlation between their information-hiding metric and change.

Although the above survey is not exhaustive, there is an obvious lack of conclusive studies in this field and further research is required.

6. Conclusions and further work

This paper presented the results of an experiment where the impact of object-oriented design on resulting software quality attributes (defect density and rework) was empirically evaluated. The MOOD set of metrics was adopted to measure the characteristics of OO design. The results achieved so far allow to infer that in fact the design alternatives may have a strong influence on resulting quality. Quantifying this influence can help to train novice designers by means of heuristics [Abreu et al, 1995] embedded in design tools. Being able to predict the resulting reliability and maintainability is very important to project managers during the resource allocation (planning) process.

This work is a small step toward the understanding of how software designs affect resulting quality. Further validation experiments with a larger sample of projects is expected to be carried out. A replication of this experiment with a sample of C++ and Ada95 large-scale projects developed at the Software Engineering Laboratory (NASA Goddard Space Center) is expected to be done next year. The impact on other quality attributes like efficiency, portability, usability and functionality must also be assessed. The public availability of a tool to collect the adopted design metrics is expected to foster further experiments throughout the academic and industrial communities.

Among our priorities is the definition of MOOD bindings for Smalltalk and Ada95 in order to conduct new experiments and assess if adopted languages affect quality characteristics differently.

The authors intend to launch a research line on the complexity of design patterns [Gamma et al, 1995]. These seem to be a natural 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". The patterns adoption greatly depends on their understandability, smooth integration (lack of side effects), functionality and reliability. All those characteristics must be quantitatively evaluated in order to define acceptance criteria and compare different pattern implementations for similar functionalities.

References

[Abbott et al, 1994] D. H. Abbott.; T. D. Korson; J. D. McGregor. *"A proposed design complexity measure for object-oriented development"*. Clemson University, TR 94-105, April 1994.

[Abreu et al, 1993] F. B. Abreu; R. Carapuça. *"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.

[Abreu et al, 1994] F. B. Abreu; R. Carapuça. *"Object-Oriented Software Engineering: Measuring and Controlling the Development Process"*. Proceedings of the *4th International Conference on Software Quality,* McLean, Virginia, USA, October 1994.

[Abreu et al, 1995] F. B. Abreu; M. Goulão; R. Esteves. *"Toward the Design Quality Evaluation of Object-Oriented Software Systems"*. Proceedings of the *5th International Conference on Software Quality,* Austin, Texas, USA, October 1995.

[Basili et al, 1995] V. Basili; L. Briand; W. Melo. *"A Validation of Object-Oriented Design Metrics"*. Technical Report CS-TR-3343, University of Maryland, Department of Computer Science, May. 1995.

[Chidamber et al, 1994] S. Chidamber; C. Kemmerer. *"A metrics suite for object oriented design"*. Center of Information Systems Research (MIT), WP No. 249, July 1993 ; also published in *IEEE Transactions on Software Engineering*, Vol. 20 (6), pp.476-493, June 1994.

[Churcher et al., 1995] N. I. Churcher; M. J. Shepperd. *"Comments on 'A metrics suite for object oriented design' ". IEEE Transactions on Software Engineering*, Vol. 21 (3), pp.263-265, 1995.

[Gamma et al, 1995] E. Gamma; R. Helm; R. Johnson; J. Vlissides. *"Design Patterns: Elements of Reusable Object-Oriented Software"*. Addison-Wesley, 1995.

[IEEE1061, 1990] Institute of Electrical and Electronic Engineers. *"ANSI/IEEE P-1061/D21 - Standard for a Software Quality Metrics Methodology*". 1990.

[ISO9000/3, 1991] International Organization for Standardization. *"ISO/IEC 9000 / Part 3 - Guidelines for the Application of ISO 9001 to the Development, Supply and Maintenance of Software"*. ISO JTC1/SC7, 1991 (currently under revision).

[ISO9126, 1991] International Organization for Standardization. *"ISO/IEC 9126 - Information Technology - Software Product Evaluation - Quality Characteristics and Guidelines for their use"*. ISO JTC1/SC7, 1991 (currently under revision).

[ISO14598, 1995] International Organization for Standardization. *"ISO/IEC 14598 - Information Technology - Software Product Evaluation"*. ISO JTC1/SC7, 1995 (currently in CD stage).

[Li et al, 1994] W. Li; S. Henry. *"Object-oriented metrics that predict maintainability". Journal of Systems and Software,* Vol. 23 (2), pp.111-122, 1994.

[Rising et al, 1994] L. Rising; F. Calliss. "An information hiding metric". *Journal of Systems and Software,* Vol. 26, pp. 211-220, 1994.

[Melo et al, 1995] W. Melo; L. Briand; V. Basili. "*Measuring the Impact of Reuse on Quality and Productivity in Object-Oriented Systems*". Technical Report CS-TR-3395, University of Maryland, Department of Computer Science, January 1995.

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

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

[Rumbaugh et al, 1991] J. Rumbaugh; M. Blaha; W. Premerlani; F. Eddy; W. Lorensen. *"Object-Oriented Modelling and Design"*. Prentice-Hall, 1991.

[Stroustrup, 1991] B. Stroustrup. *"The C++ Programming Language"*. Addison-Wesley Series in Computer Science, 1991 (2nd edition).

[Wegner, 1987] P. Wegner. *"Dimensions of Object-Oriented Design"*. Proceedings of the *OOPSLA'87 Conference,* pp.168-182, October 1987.

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

[Zuse, 1991] H. Zuse. *"Software Complexity: Measures and Methods"*. Walter de Gruyer (New York), 1991.