Originally published in Proceedings of the 3rd International
Software Metrics Symposium (METRICS'96),
IEEE, Berlin, Germany, March 1996.

Evaluating the Impact of Object-Oriented Design on Software Quality

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

Walcélio Melo
University of Maryland /
Inst.Adv.Comp.Stud.
A.V.Williams Bldg., College Park
MD 20742, USA
melo@cs.umd.edu

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 were used to assess the referred impact. Data obtained in this experiment show how OO design mechanisms such as inheritance, polymorphism, information hiding and coupling, can influence quality characteristics like reliability or maintainability. Some predictive models based on OO design metrics are also presented.

1. Introduction

The contribution of metrics to the overall objective of software quality is understood and fully recognized by the software engineering community in general [24,27] and particularly emphasized by the software quality community [11,13]. Several standards like [15,16,17,18] were produced where their need is endorsed.

Process and product metrics can help both *managing* activities, such as scheduling, costing, staffing and controlling, and *engineering* activities such as analysing, designing, coding, documenting and testing.

One of the most influencing factors of software systems quality, where metrics can play an important role, is the structure of software 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. In other words, cumulative knowledge plays a very important part in the design phase. Novice designers are therefore exposed to a myriad of design decisions that surely affect the final outcome. The intensive use of patterns, frameworks and other reusable components is expected to ease this problem, but current practice does not include yet their widespread adoption.

Being able to predict some software quality characteristics based on the design, is one of our great motivations. This ability will allow the designing process to be guided, for instance, by means of heuristics. One well-known heuristic for object-oriented design is expressed by the *Law of Demeter* [20]. 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, this has yet to be done.

Since the early days of computer science many approaches to quantify the internal structure of procedural software systems have emerged [31]. Some of those "traditional" metrics can still be used with the object-oriented paradigm, especially at the method level [2]. However, the need to quantify the distinctive features of this paradigm gave birth, in recent years, to new metric sets. Most of those sets have yet to be experimentally validated. This validation step usually consists of correlation studies between internal (design) and external (quality) attributes. A brief review of some known validation efforts follows.

The MOOSE metrics, a set of OO design metrics [9], were validated using the same project data as those in this paper [7]. Besides discussing the metrics' advantages and drawbacks, the authors claim that several of them appear to be adequate for predicting class failure-proneness during the early phases of the life-cycle. Nevertheless, some critics on the MOOSE metrics' imprecise and ambiguous definition (lack of language bindings) were raised [10].

In [19] the authors used an extension of the MOOSE set to build a regression model that is said to be adequate for predicting changeability (effort of

correcting or enhancing classes). This model was validated with data from two systems built with an object-oriented dialect of Ada.

A metric derived from the design information captured in class definitions, for measuring the number and strength of the object interactions, was proposed in [1]. The authors claim the metric's usefulness for predicting experts' design preferences. To validate this allegation they used 9 sets of distinct 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 [25]. A validation experiment based on a system with approximately one million lines of Ada[1] code is described. Results showed that those metrics were able to "discriminate between packages that are, or are not, likely to undergo significant changes". On the other hand, the authors recognize that the same experiment showed no linear correlation between their information-hiding metric and change.

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

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 means of experimental validation. In order to measure the OO design characteristics, a suite of metrics called MOOD [3] was adopted. Motivations behind the MOOD set definition include:

(1) coverage of basic structural mechanisms of the object-oriented paradigm such 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;

(4) language independence to broaden 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 statistical analyses on the collected data, discusses the use of the adopted metrics set and proposes and validates software quality predictive models; finally, section 5 concludes the paper by presenting lessons learned and future work.

[1] According to [29] Ada may be considered as *object-based* but not *object-oriented* because its objects (packages) do not have a class (type).

2. 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, we need a mapping of concepts and terminology, hereafter called *binding*, between MOOD and the adopted language.

2.2. Metrics definitions and language bindings

This section contains an overview of the MOOD metrics along with abstracted bindings for two OO languages, C++ [28] and Eiffel [23]. Fully detailed bindings for C++ and Eiffel can be found in [4] and [5], respectively. Readers familiarized with MOOD metrics can skip to section 3.

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:

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

and:

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

The MHF numerator is the sum of the invisibilities of all methods defined in all classes. The invisibility of a method is the percentage of the total classes from which this method is not visible.

The MHF denominator is the total number of methods defined in the system under consideration.

MOOD		*C++*	*Eiffel*
TC	***total classes***	*total number of classes*	*same as for C++*
	methods	*constructors; destructors; function members[2]; operator definitions*	*class features with implementation (**do** clause) or without it (**deferred** clause); **external** functions; constants with **once** clause*
$M_d(C_i)$	***methods defined*** *(not inherited)*	*all methods declared in the class including **virtual** (deferred) ones*	*all methods declared in the class, even if declared **obsolete**;*
$V(M_{mi})$	***visibility*** *- % of the total classes from which the method M_{mi} is visible*	*= 1 for methods in **public** clauses; = 0 for those in **private** clauses; = $DC(C_i)/(TC-1)$ for methods in **protected** clauses note: $DC(C_i)$= descendants of C_i*	*= 1 by omission or if **ANY** is mentioned; = $DC(C_i)/(TC-1)$ if **NONE** or empty brackets **{ }** are mentioned; else[3]= #{descendants of $C_i \cup$ classes within brackets **{...}** and their descendants $\cup$ exports[4] } / (TC-1)*

2.2.2. Attribute Hiding Factor :

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

where:

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

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

The AHF numerator is the sum of the invisibilities of all attributes defined in all classes. The invisibility of an attribute is the percentage of the total classes from which this attribute is not visible.

The AHF denominator is the total number of attributes defined in the system under consideration.

MOOD		*C++*	*Eiffel*
$A_d(C_i)$	***attributes defined*** *(not inherited)*	*data members*	*class features without implementation; simple typed constants (integer, boolean, character)*
$V(A_{mi})$	***visibility*** *- % of the total classes from which A_{mi} is visible*	*= 1 for attributes in **public** clauses; = 0 for attributes in **private** clause; = $DC(C_i)/(TC-1)$ for attributes in **protected** clauses note: $DC(C_i)$= descendants of C_i;*	*= 1 by omission or if **ANY** is mentioned; = $DC(C_i)/(TC-1)$ if **NONE** or empty brackets **{ }** are mentioned; else=#{descendants of $C_i \cup$ classes within brackets **{...}** and their descendants $\cup$ exports } / (TC-1)*

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

The MIF numerator is the sum of inherited methods in all classes of the system under consideration.

The MIF denominator is the total number of available methods (locally defined plus inherited) for all classes.

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

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

[3] "# " is the cardinality operator (set size). The Visibility is the cardinality of a union of sets so that repetitions are eliminated.

[4] Classes specified in the "export" clause of feature M_{mi} in descendants of class C_i.

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

The AIF numerator is the sum of inherited attributes in all classes of the system under consideration.

The AIF denominator is the total number of available attributes (locally defined plus inherited) for all classes.

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

The POF numerator represents the *actual number of possible different polymorphic situations*. Indeed, a given message sent to class $\boldsymbol{C_i}$ can be bound, statically or dynamically, to a named method implementation. The latter can have as many shapes ("morphos" in ancient Greek) as the number of times this same method is overridden (in $\boldsymbol{C_i}$ descendants).

The POF denominator represents the *maximum number of possible distinct polymorphic situations* for class $\boldsymbol{C_i}$. This would be the case where all new methods defined in $\boldsymbol{C_i}$ would be overridden in all of their derived classes.

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

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

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

The COF denominator stands for the *maximum possible number of couplings* in a system with *TC* classes.

The client-supplier relation (represented by $\boldsymbol{C_c} \Rightarrow \boldsymbol{C_s}$) means that $\boldsymbol{C_c}$ (*client* class) contains *at least one* non-inheritance reference to a feature (method or attribute) of class $\boldsymbol{C_s}$ (*supplier* class). The COF numerator then represents the *actual number of couplings not imputable to inheritance.*

Client-supplier relations can have several shapes:

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

3. Controlled Data Collection Experiment

The impact of OO design on software quality will be evaluated in this paper, by examining the degree to which MOOD metrics allow to predict *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 [21] were used. Section 3.1 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

The population under study was a graduate and senior level class offered by the Department of Computer Science at the University of Maryland, between September and December 1994. 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, defect 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. Defects found in these two first phases were reported to the students. This ensured 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. This procedure avoided the common situation where many defects are "informally" fixed (not reported), when testing is performed by the developers themselves.

During the repair phase, the students were asked to correct their systems based on the defects found by the independent test group.

The development environment and technology used were consistent with current practice in industry and academia. Sun Sparc stations were used as implementation platforms. OMT [26], 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. The following libraries were provided to the students:

- MotifApp - this public domain library [30] 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 the use of libraries was not mandatory, each design team adopted different reuse options.

3.2. Collected data

In this experiment both product and process data were gathered. 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 [21].

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 is distributed freely for those who want 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 and standard deviation).

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%
Mean	14.7%	72.3%	32.6%	20.6%	4.2%	8.0%
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

Using standard terminology [14] we say that *errors* are inaccuracies in the human thinking process committed while trying to understand given information, solving problems or using methods and tools. These errors cause the introduction of *defects*, also known as "faults" or "bugs", in the software deliverables such as documents or source code. In this experiment, defects were detected by white-box testing and reported in appropriate forms. *Failures* are concrete manifestations of defects within the software. These were exposed in this experiment by black-box testing. Notice that one defect may cause different failures, although distinct defects may cause similar failures.

To collect *rework effort*, expressed by man.hours spent on correcting defects found, other forms were filled out by the developers.

Table 2 shows the project ID, project size [SLOC] (Source Lines Of Code), defect density [defects found / KSLOC], failure density [detected failures / KSLOC], and normalized rework [man.hours / KSLOC], along with some descriptive statistics, for all projects considered in the experiment.

Project	*Size*	*Defect Density*	*Failure Density*	*Normalized Rework*
1	13981	1.72	3.08	3.65
2	5068	6.51	8.29	14.01
3	9735	4.31	4.52	9.45
4	8543	3.86	4.80	8.43
5	8173	3.18	8.20	7.22
6	6368	3.93	4.40	8.01
7	6571	2.28	2.43	4.72
8	5068	8.68	14.80	18.35
Mean	7938	4.29	6.31	9.23
Std. Dev.	2951	2.28	4.04	4.84

Table 2: Process metrics for each project

4. The Impact of OO Design Properties

4.1. Clarifying the scope

The software quality community is a strong believer that the organization of the software process is the main driver of achieved quality [6,13]. Since we entirely agree with this premise, all teams in the experiment repeatedly adopted the same well-defined development process as described in section 3.1. We could then extirpate this discriminative factor (type of process used) from our controlled experiment. Furthermore, since all projects in the experiment were based in the same universe of discourse (a video rental business), they were good representatives of distinct design alternatives, whose effect on quality we wanted to quantify. Thus we believe that quantitative differences in achieved software quality characteristics are exactly due to the corresponding design options.

One of the most important design options is the degree of incorporation of available library components. This is often referred as *external reuse* as opposed to *internal reuse* (reuse through inheritance) [8]. Reusable components tend to be better designed than ordinary program code, either because their conception is more careful, or because its repeated use brings out quickly any flaws in its design or implementation. Therefore the amount of reuse is expected to produce a positive impact on overall system design and, consequently, in resulting software quality characteristics [21]. The projects analysed in this paper had different degrees of reuse adoption. The analysis of how the reuse strategies brought about distinct design properties is beyond the scope of this paper.

Next section includes an attempt to explain the individual impact of the design properties, represented by each MOOD metric, on the recorded software quality measures. Section 4.3 introduces and validates some linear regression models that allow to predict the cumulative impact of all MOOD metrics on resulting software quality characteristics.

4.2. Individual impact

The MOOD metrics are believed to quantify independent aspects of the design and therefore their effect on quality can be assessed individually. To provide some evidence about the relationship between OO design and software project quality, the correlations between the MOOD metrics and the quality measures of defect density (DD), failure density (FD) and normalized rework (NR) were determined. The resulting coefficients of correlation are shown in Table 3.

	MHF	*AHF*	*MIF*	*AIF*	*POF*	*COF*
DD	-0.565	-0.127	-0.781	-0.558	-0.683	0.914
FD	-0.629	-0.126	-0.635	-0.373	-0.691	0.913
NR	-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 (defect and failure densities) and rework. This means that once MHF increases, the defect density and the effort spent to fix defects 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, class implementation should be a stepwise

process, where more and more details (hidden methods) are added. Therefore, the stepwise top-down implementation favors 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[5]. In fact, *information hiding*, supported by *encapsulation* mechanisms, allows to cope with complexity by turning complex components in "black boxes", thus reducing "side-effects". Ideally, all attributes would be hidden and only accessed by the corresponding class methods (AHF=100%).
- Methods Inheritance Factor (MIF) has a moderate negative correlation with failure density and a high negative correlation with both defect density and normalized rework measure. This means that once MIF increases the defect density and the effort spent to fix defects 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 defect density and rework, when used sparingly. Very high values of MIF (above the 70% to 80% range [4]) are believed to reverse this beneficial effect, but this assumption still lacks experimental validation[6].
- Attributes Inheritance Factor (AIF) has a low negative correlation with failure density and a moderate negative correlation with both defect density and normalized rework measure. For now this result does not allow any strongly supported conclusions to be drawn. Next section will bring new insights on the impact of AIF.
- Polymorphism Factor (POF) has a moderate to high negative correlation with defect and failure 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 these 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 are also expected to increase. This result shows that coupling in software systems has a strong negative impact on software quality and therefore should be kept to the minimum required during design. It is desirable that classes communicate with as few others as possible [22] because coupling relations increase complexity, reduce encapsulation and potential reuse, and limit understandability and maintainability.

[5] The cumulative impact of MOOD metrics, to be discussed in next section, will show a more elucidative impact of AHF.

[6] Increased depth and width of the inheritance hierarchy trees make understandability and testability fade away. However, as seen in Table 1, MIF values were very low in most projects.

4.3. Cumulative impact and prediction

4.3.1. Model hypothesis

Predictive models can be developed to quantify the impact of OO design on software quality. In our case study we want to explain how some variables such as defect density (DD), failure density (FD) and normalized rework effort (NR) depend upon the MOOD metrics.

Assuming that each MOOD metric is not redundant, or by other words, has some additional information content not embodied in the other metrics, then it cannot be expressed as a linear function of the others. This implies the absence of exact multicollinearity.

Under these conditions we can state the hypothesis that the following *multiple regression models* are valid:

$$DD = \mu_{DD} + \alpha_{MHF}.MHF + \alpha_{AHF}.AHF + \alpha_{MIF}.MIF + \alpha_{AIF}.AIF + \alpha_{POF}.POF + \alpha_{COF}.COF + \varepsilon_{DD}$$

$$FD = \mu_{FD} + \beta_{MHF}.MHF + \beta_{AHF}.AHF + \beta_{MIF}.MIF + \beta_{AIF}.AIF + \beta_{POF}.POF + \beta_{COF}.COF + \varepsilon_{FD}$$

$$NR = \mu_{NR} + \Upsilon_{MHF}.MHF + \Upsilon_{AHF}.AHF + \Upsilon_{MIF}.MIF + \Upsilon_{AIF}.AIF + \Upsilon_{POF}.POF + \Upsilon_{COF}.COF + \varepsilon_{NR}$$

The MOOD metrics are the *independent* variables in these models, also called *explanatory variables*, *regressor variables* or, more simply, *regressors*. DD, FD and NR are called the *dependent* or *outcome variables*.

The α, β and Υ parameters are called *response coefficients*. They quantify the change in the outcome variable produced by a unit increase in the corresponding explanatory variable, when all other independent variables are held constant. For instance, α_{MIF} is the change in defect density when the MIF metric is increased in one unit (e.g. 1% increase) and all other MOOD variables are held constant. In other words, α_{MIF} is the response of defect density to a change in MIF. The magnitude of the response coefficients denotes the extent of changes due to the corresponding metric by itself. They can be represented as partial derivatives:

$$\alpha_{MHF} = \frac{\partial DD}{\partial MHF} \qquad \beta_{MHF} = \frac{\partial FD}{\partial MHF} \qquad \Upsilon_{MHF} = \frac{\partial NR}{\partial MHF}$$

$$\alpha_{AHF} = \frac{\partial DD}{\partial AHF} \qquad \beta_{AHF} = \frac{\partial FD}{\partial AHF} \qquad \Upsilon_{AHF} = \frac{\partial NR}{\partial AHF}$$

$$\alpha_{MIF} = \frac{\partial DD}{\partial MIF} \qquad \beta_{MIF} = \frac{\partial FD}{\partial MIF} \qquad \Upsilon_{MIF} = \frac{\partial NR}{\partial MIF}$$

$$\alpha_{AIF}=\frac{\partial DD}{\partial AIF} \qquad \beta_{AIF}=\frac{\partial FD}{\partial AIF} \qquad \Upsilon_{AIF}=\frac{\partial NR}{\partial AIF}$$

$$\alpha_{POF}=\frac{\partial DD}{\partial POF} \qquad \beta_{POF}=\frac{\partial FD}{\partial POF} \qquad \Upsilon_{POF}=\frac{\partial NR}{\partial POF}$$

$$\alpha_{COF}=\frac{\partial DD}{\partial COF} \qquad \beta_{COF}=\frac{\partial FD}{\partial COF} \qquad \Upsilon_{COF}=\frac{\partial NR}{\partial COF}$$

The μ parameters are called *intercept parameters*. The strict mathematical interpretation is the value of the outcome variable when all explanatory variables are zero. Since the situation where all MOOD metrics are zero is unrealistic, our models are not supposed to be a good approximation of reality in that zone.

In a real sample, such as the one introduced in this paper, all the observations of the independent and dependent variables will not coincide exactly with the linear relationships expressed in the previous equations. Therefore we need to add random error terms (ε_{DD} ,ε_{FD} *or* ε_{NR}). We assume that these errors have equal probabilities of being either positive or negative. Over a large sample of observations they will average out to zero. Thus the expected or mean values for the errors will be:

$$E[\varepsilon_{DD}] = E[\varepsilon_{FD}] = E[\varepsilon_{NR}] = 0$$

We also assume that some errors are not more likely to be bigger than others. Thus all observations will have the same (unknown) finite variances given by:

$$var(\varepsilon_{DD}) = E[\varepsilon^2_{DD}] = \sigma^2_{DD}$$
$$var(\varepsilon_{FD}) = E[\varepsilon^2_{FD}] = \sigma^2_{DD}$$
$$var(\varepsilon_{NR}) = E[\varepsilon^2_{NR}] = \sigma^2_{NR}$$

Under the previous suppositions we can assume that the errors are normally distributed with zero mean and σ^2 variance:

$$\varepsilon_{DD} \sim N(0, \sigma^2_{DD})$$
$$\varepsilon_{FD} \sim N(0, \sigma^2_{FD})$$
$$\varepsilon_{NR} \sim N(0, \sigma^2_{NR})$$

Since the observations of the outcome variables depend on the corresponding random error term, then the dependent variables can be viewed as random variables whose statistical properties follow those of the errors:

$$DD \sim N(\mu_{DD} + \alpha_{MHF}.MHF + \alpha_{AHF}.AHF + \alpha_{MIF}.MIF + \alpha_{AIF}.AIF + \alpha_{POF}.POF + \alpha_{COF}.COF , \sigma^2_{DD})$$

$$FD \sim N(\mu_{FD} + \beta_{MHF}.MHF + \beta_{AHF}.AHF + \beta_{MIF}.MIF + \beta_{AIF}.AIF + \beta_{POF}.POF + \beta_{COF}.COF , \sigma^2_{FD})$$

$$NR \sim N(\mu_{NR} + \Upsilon_{MHF}.MHF + \Upsilon_{AHF}.AHF + \Upsilon_{MIF}.MIF + \Upsilon_{AIF}.AIF + \Upsilon_{POF}.POF + \Upsilon_{COF}.COF , \sigma^2_{NR})$$

Under the suppositions about the statistical distributions of DD, FD and NR we get the following *estimators*:

$$\hat{DD} = \mu_{DD} + \alpha_{MHF}.MHF + \alpha_{AHF}.AHF + \alpha_{MIF}.MIF + \alpha_{AIF}.AIF + \alpha_{POF}.POF + \alpha_{COF}.COF$$

$$\hat{FD} = \mu_{FD} + \beta_{MHF}.MHF + \beta_{AHF}.AHF + \beta_{MIF}.MIF + \beta_{AIF}.AIF + \beta_{POF}.POF + \beta_{COF}.COF$$

$$\hat{NR} = \mu_{NR} + \Upsilon_{MHF}.MHF + \Upsilon_{AHF}.AHF + \Upsilon_{MIF}.MIF + \Upsilon_{AIF}.AIF + \Upsilon_{POF}.POF + \Upsilon_{COF}.COF$$

Based on the available data sample we calculated the μ, α, β *and* Υ parameters represented in Table 4.

	α	β	Υ
MHF	10.958	0.743	23.770
AHF	-0.649	5.065	-1.574
MIF	2.194	25.557	5.664
AIF	-7.564	-14.753	-16.941
POF	-24.194	-68.134	-57.937
COF	29.959	65.354	62.070
μ	2.643	-5.088	6.011

Table 4: Response coefficients and intercept parameters

4.3.2. Model misspecification

The specification of a multiple regression model brings with it uncertainty concerning whether we have chosen the correct set of regressors. We can either over- or underspecify its number. Overspecifying means we have included irrelevant regressors (extraneous variables) that have no influence on the outcomes for the dependant variable. Underspecifying means we have omitted relevant attributes that do influence the dependant variable. The misspecifications in the regressors set are called *variable specification errors*. There is a bias-variance trade-off due to these errors. If we *overspecify*, our models will be *inefficient* since the achieved variance for the least square estimator will be greater. If we *underspecify*, the least square estimators will have minimum variance but will be *biased*.

We will use the *tolerance* statistic to verify how much each MOOD metric contributes to our models. The tolerance for an explanatory variable is the proportion of its variance not accounted for by other independent variables in the equation. Lower tolerance means fewer information contributions to the regression model. The sample tolerances for MOOD are included in Table 5.

	DD	***FD***	***NR***
MHF	24.02%	24.0%	24.02%
AHF	36.98%	37.0%	36.98%
MIF	3.17%	3.2%	3.17%
AIF	6.81%	6.8%	6.81%
POF	15.07%	15.1%	15.07%
COF	26.65%	26.6%	26.65%

Table 5: Tolerances for MOOD metrics

From Table 5 we conclude that AHF and COF are bigger contributors, while MIF and AIF have lower influence. However, even the latter metrics have a sufficient impact on the model not to be considered as extraneous. Furthermore, we get unbiased estimators at the risk of a slightly greater variance.

4.3.3. Model validation

The validity of our models depends on: (i) evidence about the linearity assumption, (ii) how well they fit the sample and (iii) their predictive ability beyond the sample. For that purpose we calculated the statistics included in Table 6.

	DD	*FD*	*NR*
Multiple R	99.909%	98.326%	99.998%
R Square	99.818%	96.680%	99.996%
Adjusted R Square	98.728%	76.763%	99.969%

Table 6: R Statistics

Multiple R is the correlation coefficient between the observed and predicted values of the dependent variable. As seen in Table 6 the three models show an almost perfect linear relationship between the dependent variables and the independent variables (MOOD metrics). This validates the linearity assumption in which the models were based.

The square of the multiple R, *R Squared*, which is also called *coefficient of determination*, is a measure of the proportion of variation in the dependent variable that is explained by variation in the explanatory variables. This coefficient is a measure of how well the estimated regression fits the data, usually called *goodness of fit*. Table 6 shows a very good fit for all models. Thus we deduct that the three models allow to calculate DD, FD or NR with minor errors within the sampled universe. As an example, using the estimator equations with the coefficients extracted from Table 4 and the corresponding MOOD metrics from Table 1, we get:

system 3:

$\hat{DD}$ *=4.38 defects / KSLOC (+1.4% of the actual value*[7]*)*

system 5:

$\hat{FD}$ *=7.68 failures / KSLOC (-6.3% of the actual value)*

system 7:

$\hat{RE}$ *= 4.73 man.hours / KSLOC (+0.2% of actual value)*

[7] included in Table 2.

The applicability of a predictive model bears on its ability to explain as fully as possible the variation of the outcome variables based on the corresponding explanatory variables beyond the sample data. A model estimated from a sample, fits the sample better than it will fit the population. The sample R squared thus tends to overestimate the goodness of fit of the model in the population. Thus we used the *Adjusted R Square* that is an estimate of how well the model will fit the population. Adjusted R squared corrects the optimistic bias of the sample R squared by taking sample size and the number of regressors into account. We then expect that our models will be able to produce estimates where only around 1.27% of the defect density, 23.24% of the failure density and 0.03% of the normalized rework effort will be left unexplained. These percentages, that correspond to *1 - Adjusted R Squared*, are due to the variation in error terms or to the variation in other missing variables that implicitly form part of the error terms.

5. 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 in order to measure the characteristics of OO design. The results achieved so far allow us 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 [4] 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 Ada9X large-scale projects developed at the Software Engineering Laboratory (NASA Goddard Space Center) is expected to be done in the near future. The impact on other quality attributes such as 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 Ada9X in order to conduct new experiments and assess whether adopted languages affect quality characteristics differently.

We also intend to launch a research line on the complexity of design patterns [12]. These seem to be a natural road to the “promised reuse-land”. Substantial

increases in quality and productivity are expected if software developers *really* start using these new "bricks". The adoption of patterns greatly depends on their understandability, smooth integration (lack of side effects), functionality and reliability. All of these characteristics must be quantitatively evaluated in order to define acceptance criteria and compare different pattern implementations for similar functionalities.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] F. B. Abreu; R. Esteves; M. Goulão. *"The Design of Eiffel Programs: Quantitative Evaluation Using the MOOD Metrics"*. INESC Technical report, January 1996.

[6] L. Arthur. "Improving Software Quality: An Insider's Guide to TQM", John Wiley, New York, 1993.

[7] 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.

[8] J. M. Bieman; J. X. Zhao. "*Reuse Through Inheritance: a Quantitative Study of C++ Software*". In S. Samadzadeh and M. Zand (editors), *Proc. of the ACM SIGSOFT Symposium on Software Reusability*, Seattle, WA, April 28-30, 1995.

[9] 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.

[10] 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.

[11] N. Fenton (editor). "Software Metrics: A Rigorous Approach", Chapman & Hall (UK) or Van Nostrand Reinhold (USA), 1991.

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

[13] W. Humphrey. "Managing the Software Process", SEI Series in Software Engineering, Addison-Wesley, 1989.

[14] Institute of Electrical and Electronic Engineers Standards Board. "*ANSI/IEEE Standard 982 - Standard Dictionary of Measures to Produce Reliable Software"*, 1988.

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

[16] 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, 1995.

[17] 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).

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

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

[20] K. J. Lieberherr; I. M. Holland. *"Assuring Good Style for Object-Oriented Programs", IEEE Software*, pp. 38-48, September 1989.

[21] 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.

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

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

[24] R. Pressman. "*Software Engineering: a Practitioner's Approach*" (3rd edition), McGraw-Hill, 1992.

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

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

[27] I. Sommerville. "*Software Engineering*" (4th edition), Addison-Wesley, 1992.

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

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

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

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