Technical Report, Univ. of Maryland, Dep. of Computer Science, College Park, MD, 20742 USA. April 1995.

A VALIDATION OF OBJECT-ORIENTED DESIGN METRICS

Victor R. Basili*, Lionel Briand and Walcélio L. Melo***

***University of Maryland**
Dep. of Computer Sciences
Institute for Advanced Computer Studies
College Park, MD, 20770 USA
{basili|melo}@cs.umd.edu

****CRIM**
1801 McGill College Av.
Montréal (Quebec)
H3A 2N4, Canada
lbriand@crim.ca

Abstract

This paper presents the results of a study conducted at the University of Maryland in which we experimentally investigated the suite of Object-Oriented (OO) design metrics introduced by [Chidamber&Kemerer, 1994]. In order to do this, we assessed these metrics as predictors of fault-prone classes. This study is complementary to [Lie&Henry, 1993] where the same suite of metrics had been used to assess frequencies of maintenance changes to classes. To perform our validation accurately, we collected data on the development of eight medium-sized information management systems based on identical requirements. All eight projects were developed using a sequential life cycle model, a well-known OO analysis/design method and the C++ programming language. Based on experimental results, the advantages and drawbacks of these OO metrics are discussed and suggestions for improvement are provided. Several of Chidamber&Kemerer's OO metrics appear to be adequate to predict class fault-proneness during the early phases of the life-cycle. We also showed that they are, on our data set, better predictors than "traditional" code metrics, which can only be collected at a later phase of the software development processes.

Key-words: Object-Oriented Design Metrics; Error Prediction Model; Object-Oriented Software Development; C++ Programming Language.

During this work the authors were, in part, supported by the following grants: Basili by NASA grant NSG-5123, Briand by the IGLOO project and NFS grant 01-5-24845, and Melo by a Westinghouse grant and UMIACS.

Published as a Technical Report, Univ. of Maryland, Dep. of Computer Science, College Park, MD, 20742. April 1995. CS-TR-3443.

1. Introduction

1.1 Motivation

The development of a large software system is a time- and resource-consuming activity. Even with the increasing automation of software development activities, resources are still scarce. Therefore, we need to be able to provide accurate information and guidelines to managers to help them make decisions, plan and schedule activities, and allocate resources for the different software activities that take place during software evolution. Software metrics are thus necessary to identify where the resource issues are; they are a crucial source of information for decision-making [Harrison, 1994].

Testing of large systems is an example of a resource- and time-consuming activity. Applying equal testing and verification effort to all parts of a software system has become cost-prohibitive. Therefore, one needs to be able to identify fault-prone classes so that testing/verification effort can be concentrated on these classes [Harrison, 1988]. The availability of adequate product design metrics for characterizing error-prone classes is thus vital.

Dozens of product metrics have been proposed [Fenton, 1991], used, and, sometimes, experimentally validated in academia [Basili&Hutchens, 1982] and industry, e.g., number of lines of code, MacCabe complexity metric, etc. In fact, many companies have built their own cost, quality and resource prediction models based on product metrics. TRW [Boehm, 1981], the Software Engineering Laboratory (SEL) [McGarry *et. al.*, 1994] and Hewlett Packard [Grady, 1994] are examples of software organizations that have been using product metrics to build their cost, resource, defect, and productivity models.

1.2 Issues

In the last decade, many companies have started to introduce Object-Oriented (OO) technology into their software development environments. OO analysis/design methods, OO languages, and OO development environments are currently popular worldwide in both small and large software organizations. The insertion of OO technology in the software industry, however, has created new challenges for companies which use product metrics as a tool for monitoring, controlling and improving the way they develop and maintain software. Therefore, metrics which reflect the specificities of the OO paradigm must be defined and validated in order to be used in industry. Some studies have concluded that "traditional" product metrics are not sufficient for characterizing, assessing and predicting the quality of OO software systems. For example, based on a study at Texas Instruments, [Brooks, 1993] has reported that McCabe cyclomatic complexity appeared to be an inadequate metric for use in software development based on OO technology.

To address this issue, OO metrics have recently been proposed in the literature [Abreu&Carapuça, 1994; Chidamber&Kemerer, 1994]. However, most of them have not undergone a thorough and comprehensive experimental validation. [Briand *et.al.*, 1994] and [Lie&Henry, 1993] are rare exceptions in this respect. The work described in this paper is an additional step toward a thorough experimental validation of the OO metric suite defined in [Chidamber&Kemerer, 1994]. This paper presents the results of a study conducted at the University of Maryland in which we performed an experimental validation of that suite of OO metrics with regard to their ability to identify fault-prone classes. Data were collected during the development of eight medium-sized management information systems based on identical requirements. All eight projects were developed using a sequential life cycle model, a well-known Object-Oriented analysis/design method [Rumbaugh et al, 1991], and the C++ programming language [Stroustrup, 1991]. In fact, we used an experiment framework that should be representative of currently used technology in industrial settings. This study discusses the strengths and weaknesses of the validated OO metrics with respect to predicting faults across classes.

1.3. Outline

This paper is organized as follows. Section 2 presents the suite of OO metrics proposed by Chidamber&Kemerer (1994), and the methodology we used for experimental validation. Section 3 presents the data collected together with the statistical analysis of the data. Section 4 compares our study with other works on the subject. Finally, section 5 concludes the paper by presenting lessons learned and future work.

2. Description of the Study

2.1. Experiment goal

The goal of this study was to analyze experimentally the OO design metrics proposed in [Chidamber&Kemerer, 1994] for the purpose of evaluating whether or not these metrics are suitable for predicting the probability of detecting faulty classes. From [Chidamber&Kemerer, 1994], [Chidamber&Kemerer, 1995] and [Churcher&Shepperd, 1995], it is clear that the definitions of these metrics are not language independent. As a consequence, we had to slightly adjust some of Chidamber&Kemerer's metrics in order to reflect the specificities of C++. These metrics are as follows:

- Weighted Methods per Class (WMC). WMC measures the complexity of an individual class. Based on [Chidamber&Kemerer, 1994], if we consider all methods of a class to be equally complex, then WMC is simply the number of methods defined in each class. In this study, we adopted this approach for the sake of simplicity and because the choice of a complexity metric would be somewhat arbitrary since it is not fully specified in the metric suite. Thus, WMC is defined as being the number of all member functions and operators defined in each class. However, "friend" operators (C++ specific construct) are not counted. Member functions and operators inherited from the ancestors of a class are also not counted. This definition is identical the one described in [Chidamber&Kemerer, 1995]. The assumption behind this metric is that a class with significantly more member functions than its peers is more complex, and by consequence tends to be more fault-prone.

  Churcher&Shepperd (1995) have argued that WMC can be measured in different ways depending on how member functions and operations defined in a C++ class are counted. We believe that the different counting rules proposed by [Churcher&Shepperd, 1995] correspond to different metrics, similar to the WMC metric, and which must be experimentally validated as well. A validation of Churcher&Shepperd's WMC-like metrics is, however, beyond the scope of this paper.

- Depth of Inheritance Tree of a class (DIT) – DIT is defined as the maximum depth of the inheritance graph of each class. C++ allows multiple inheritance and therefore classes can be organized into a directed acyclic graph instead of trees. DIT, in our case, measures the number of ancestors of a class. The assumption behind this metric is that well-designed OO systems are those structured as forests of classes, rather than as one very large inheritance lattice. In other words, a class located deeper in a class inheritance lattice is supposed to be more fault-prone because the class inherits a large number of definitions from its ancestors.
- Number Of Children of a Class (NOC) – This is the number of direct descendants for each class. Classes with large number of children are difficult to modify and usually require more testing because the class potentially affects all of its children. Thus, a class with numerous children has to provide services in a larger number of contexts and must be more flexible. We expect this to introduce more complexity into the class design.
- Coupling Between Object classes (CBO) – A class is coupled to another one if it uses its member functions and/or instance variables. CBO provides the number of classes to which a given class is coupled. The assumption behind this metric is that highly coupled classes are more fault-prone than weakly coupled classes. So coupling between classes should be identified in order to concentrate testing and/or inspections on such classes.
- Response For a Class (RFC) – This is the number of methods that can potentially be executed in response to a message received by an object of that class. In our study, RFC is the number of functions directly invoked by member functions or operators of a class. The assumption here is that the larger the response set of a class, the higher the complexity of the class, and the more fault-prone and difficult to modify.

- Lack of Cohesion on Methods (LCOM) – This is the number of pairs of member functions without shared instance variables, minus the number of pairs of member functions with shared instance variables. However, the metric is set to 0 whenever the above subtraction is negative. A class with low cohesion among its methods suggests an inappropriate design, (i.e., the encapsulation of unrelated program objects and member functions that should not be together), which is likely to be fault-prone.

Readers acquainted with C++ can see that many particularities of C++ are not taken into account by Chidamber&Kemerer's metrics, e.g., C++ templates, friend classes, etc. In fact, additional work is necessary in order to extend the proposed OO metric set with metrics specifically tailored to C++.

2.2 Experimental framework

In order to experimentally validate the OO metrics proposed in [Chidamber&Kemerer, 1994] with regard to their capabilities to predict fault probability, we ran a controlled study over four months (from September to December, 1994). The population under study was a graduate level class offered by the Department of Computer Science at the University of Maryland. The students were not required to have previous experience or training in the application domain or OO methods. All students had some experience with C or C++ programming and relational databases and therefore had the basic skills necessary for such an experiment.

The students were randomly grouped into 8 teams. Each team developed a medium-sized management information system that supports the rental/return process of a hypothetical video rental business, and maintains 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. At the end of each phase, a document was delivered: Analysis document, design document, code, error report, and finally, modified code, respectively.

Requirement specifications and design documents were checked in order to verify that they matched the system requirements. Errors found in these first two phases were reported to the students. This maximized the chances 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 system based on the errors found by the independent test group.

OMT, an OO Analysis/Design method, was used during the analysis and design phases [Rumbaugh *et. al.*, 1991]. The C++ programming language, the GNU software development environment, and OSF/MOTIF were used during the implementation. Sparc Sun stations were used as the implementation platform. Therefore, the development environment and technology we used are representative of what is currently used in industry and academia.

The following libraries were provided to the students:

a) *MotifApp.* This public domain library provides a set of C++ classes on top of OSF/MOTIF for manipulation of windows, dialogs, menus, etc. [Young, 1992]. The MotifApp library provides a way to use the OSF/Motif widgets in an OO programming/design style.

b) *GNU library.* This public domain library is provided in the GNU C++ programming environment. It contains functions for manipulation of string, files, lists, etc.

c) *C++ database library.* This library provides a C++ implementation of multi-indexed B-Trees.

No special training was provided for the students in order to teach them how to use these libraries. However, a tutorial describing how to implement OSF/Motif applications was given to the students. In addition, a C++ programmer, familiar with OSF/Motif applications, was available to answer questions about the use of OSF/Motif widgets and the libraries. A hundred small programs exemplifying how to use OSF/Motif widgets were also provided. Finally, the code sources and the complete documentation of the libraries were made available. It is important to note that the students were not required to use the libraries and, depending on the particular design they adopted, different reuse choices were expected.

We also provided a specific domain application library in order to make our experiment more representative of the "real world". This library implemented the graphical user interface for insertion/removal of customers and was implemented in such a way that the main resources of the OSF/Motif widgets and MotifApp library were used. Therefore, this library contained a small part of the implementation required for the development of the rental system.

2.3. Data Collection

We collected: (1) the source code of the C++ programs delivered at the end of the implementation phase, (2) data about these programs, (3) data about errors found during the testing phase and fixes during the repair phase, and (4) the repaired source code of the C++ programs delivered at the end of the life cycle. GEN++ [Devanbu, 1992] was used to extract Chidamber&Kemerer's OO design metrics directly from the source code of the programs delivered at the end of the implementation phase. To collect items (2) and (3) , we used the following forms, which have been tailored from those used by the Software Engineering Laboratory [Heller et. al, 1992]:

- Defect Report Form.
- Component Origination Form.

In the following sections, we comment on the purpose of the Component Origination and Defect Report forms used in our experiment and the data they helped collect.

2.3.1 Defect Report Form

This form was used to gather data about (1) the defects found during the testing phase, (2) classes changed to correct such defects, and (3) the effort in correcting them. The latter includes:

- how long it took to determine precisely what change was needed. This includes the effort required for understanding the change or finding the cause of the error, locating where the change was to be made, and determining that all effects of the change were accounted for.
- how much time it took to implement the correction. This includes design changes, code modification, regression testing, and updates to documentation.

2.3.2 *Component Origination Form*

This form is used to record information that characterizes each class under development in the project at the time it goes into configuration management. Firstly, this form is used to capture whether the class has been developed from scratch or has been developed from a reused class. In the latter case, we collected the amount of modification (none, small or large) that was needed to meet the system requirements and design as well as the name of the reused class. By small/large, we mean that less/more than 25% of the original code had been modified, respectively. However, this kind of data was difficult to obtain because we do not have appropriate tools to collect this data automatically. As a simplification, we asked the developers to tell us if more or less than 25% of a class had been changed. In the former case, the class was labeled: *Extensively modified* and in the latter case: *Slightly modified*. Classes reused without modification were labeled: *verbatim reused.*

In addition, the name of the sub-system to which the class belonged was also collected. In our study, we had three types of sub-systems: graphical user interface (GUI), textual user interface (TUI), and database processing (DB).

3. Analysis of Experimental Results

In this section, we will attempt to assess experimentally whether the OO design metrics defined in [Chidamber&Kemerer, 1994] are suitable predictors of fault-prone classes. This will help us assess these metrics as quality indicators and how they compare to common code metrics. Thus, we intend to provide the type of empirical validation that we think is necessary before any attempt to use such metrics as objective and early indicators of quality. Section 3.1 shows the descriptive distributions of the OO metrics in the studied sample whereas Section 3.2 provides the results of univariate and multivariate analyses of the relationships between OO metrics and fault-proneness.

3.1. Analysis of Distributions

Figure 1 shows the distributions of the analyzed OO metrics based on 180 classes present in the studied systems. Table 1 provides common descriptive statistics of the metric distributions. These results indicate that inheritance hierarchies are somewhat flat (DIT) and that classes have, in general, few children (NOC). In addition, most classes show a lack of cohesion (LCOM) near 0. This latter metric does not seem to differentiate classes well and this stems from its definition which prevents any negative measure. This issue will be discussed further in Section 3.2.

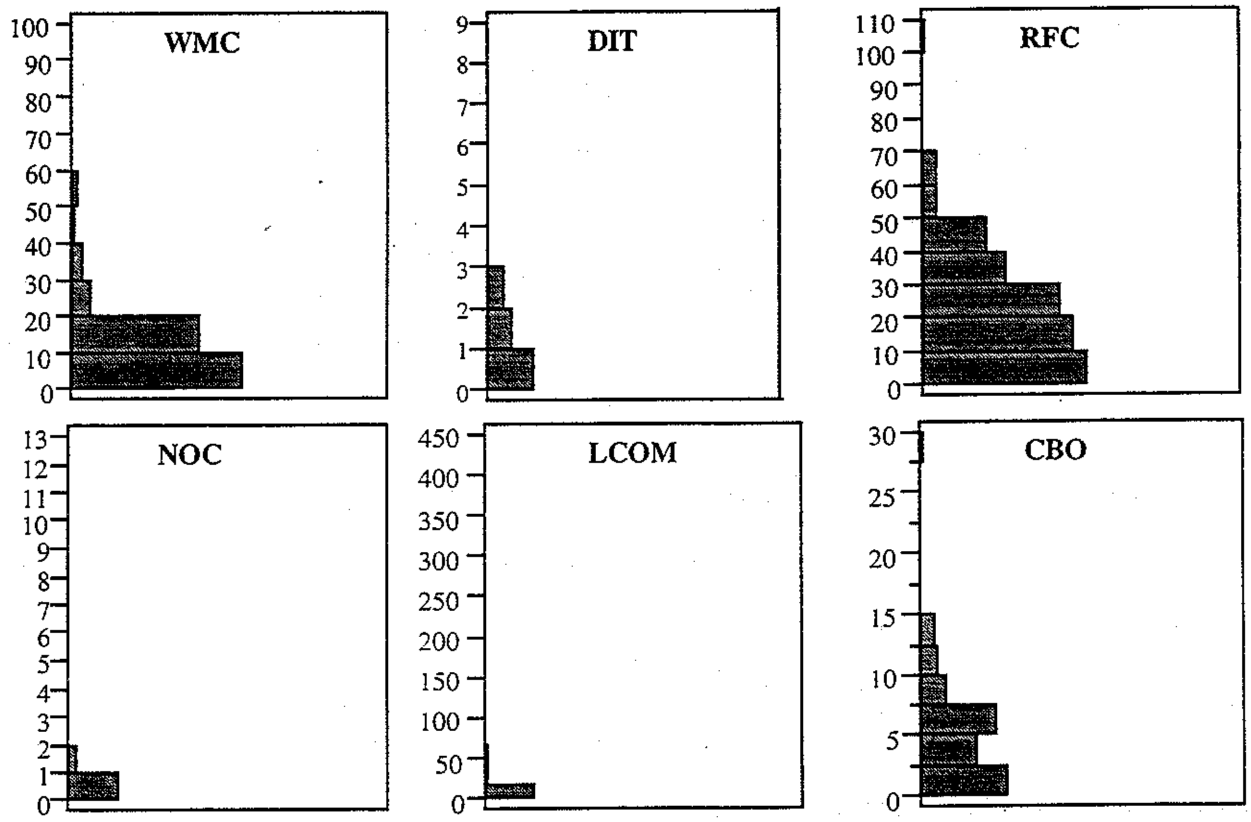

Figure 1: Distribution of the analyzed OO metrics

	WMC	DIT	RFC	NOC	LCOM	CBO
maximum	99.000	9.0000	105.00	13.000	426.00	30.00
minimum	1.0000	0.0000	0.0000	0.0000	0.0000	0.000
median	9.5000	0.0000	19.5000	0.0000	0.0000	5.000
Mean	13.3897	1.3179	33.9141	0.2308	9.7077	6.7962
Std Dev	14.9052	1.9896	33.3703	1.5377	63.7766	7.5614

Table 1: Descriptive statistics of the analyzed OO metrics.

Descriptive statistics will be useful to help us interpret the results of the analysis in the remainder of this section. In addition, they will facilitate comparisons of results from future similar studies.

3.2 The Relationships between Fault Probability and OO Metrics

3.2.1 Analysis Methodology

The response variable we use to validate the OO design metrics is binary, i.e., was a fault detected in a class during testing phases? We used logistic regression to analyze the relationship between metrics and the fault-proneness of classes. Logistic regression is a classification technique [Hosmer&Lemeshow, 1989] used in many experimental sciences based on maximum likelihood estimation. In this case, a careful outlier analysis must be performed in order to make sure that the observed trend is not the result of a few observations [Dillon&Goldstein, 1984], even though logistic regression is deemed to be more robust for outliers than least-square regression.

In particular, we first used univariate logistic regression, to evaluate the relationship of each of the metrics in isolation and fault-proneness. Then, we performed multivariate logistic regression, to evaluate the predictive capability of those metrics that had been assessed sufficiently significant in the univariate analysis (e.g., $\alpha < 0.10$ is a reasonable heuristic). This modeling process is further described in [Hosmer&Lemeshow, 1989].

A multivariate logistic regression model is based on the following relationship equation (the univariate logistic regression model is a special case of this, where only one variable appears):

$$\log\left(\frac{p}{1 - p}\right) = C_0 + C_1X_1 + C_2X_2 + \cdots + C_nX_n \qquad (1)$$

where p is the probability that a fault will be found in a class during the validation phase, and the X_i's are the OO metrics included as predictors in the model (called *covariates* of the logistic regression equation). In the two extreme cases, i.e., when a variable is either non-significant or entirely differentiates fault-prone classes, the curve (between p and any single X_i, i.e., assuming that all other X_j's are constant) approximates a horizontal line and a vertical line respectively. In between, the curve takes a flexible S shape. However, since p is unknown, the coefficients C_i will be estimated through a likelihood function optimization [Hosmer&Lemeshow, 1989]. This procedure assumes that all observations are statistically independent. When building the regression equations, each observation was weighted according to the number of faults detected in each class. The rationale is that each detection of a fault is considered as an independent event: Classes where no faults were detected were weighted 1.

Tables 2 and 3 contain the results we obtained through, respectively, univariate and multivariate logistic regression on all of the 180 classes. We report those related to the metrics that turned out to be the most significant across all eight development projects. For each metric, we provide the following statistics:

- Coefficient (appearing in Tables 2 and 3), the estimated regression coefficient. The larger the coefficient in absolute value, the stronger the impact of the explanatory variable on the probability p of a fault to be detected in a class.
- $\Delta\psi$ (appearing in Table 2 only), which is based on the notion of odd ratio [Hosmer&Lemeshow, 1989], and provides an evaluation of the impact of the metric on the response variable. More specifically, the odds ratio $\psi(X)$ represents the ratio between the probability of having a fault and the probability of not having a fault when the value of the metric is X. As an example, if, for a given value X, $\psi(X)$ is 2, then it is twice as likely that the

class does contain a fault than that it does not contain a fault. The value of $\Delta\psi$ is computed by means of the following formula:

$$\Delta\psi = \frac{\psi(X+1)}{\psi(X)} \qquad (2)$$

Therefore, $\Delta\psi$ represents the reduction/increase in the odd ratio when the value X increases by 1 unit. This provides a more intuitive insight than regression coefficients into the impact of explanatory variables.

- The level of significance (α, appearing in Tables 2 and 3) provides an insight into the accuracy of the coefficient estimates. It tells the reader about the probability of the coefficient being different from zero by chance. Usually, a level of significance of $\alpha = 0.05$ (i.e., 5% probability) is used as a threshold to determine whether an explanatory variable is a significant predictor. However, the choice of a particular level of significance is ultimately a subjective decision and other levels such as $\alpha = 0.01$ or 0.1 are common. Also, the larger the level of significance, the larger the standard deviation of the estimated coefficients, and the less believable the calculated impact of the explanatory variables. The significance test is based on a likelihood ratio test [Hosmer&Lemeshow, 1989] commonly used in the framework of logistic regression.

Based on equation (1), the likelihood function of a data set of size *D* is:

$$L = \prod_{i=1}^{D} \pi(x_i) \qquad (3)$$

where:

$$\pi(x_i) = \frac{e^{(C_o + C_1 \bullet X_{i1} + \ldots + C_n \bullet X_{in}) \bullet Y_i}}{1 + e^{(C_o + C_1 \bullet X_{i1} + \ldots + C_n \bullet X_{in})}} \qquad (4)$$

where Y_i is assigned the value 1 if the class does not contain any fault, 0 otherwise. The n-dimensional vectors X_i contain the OO design metrics characterizing each of the D observations. Also, $\pi(X_i)$ represents the estimated probability for a class to contain (or not, depending on which is the case) a fault. The coefficients that will maximize the likelihood function will be the regression coefficient estimates. For mathematical convenience, $l = Ln[L]$, the *log-likelihood*, is usually maximized.

One of the global measure of goodness of fit we will use for logistic regression models is R^2, a statistic defined as:

$$R^2 = \frac{(l_0 - l_n)}{(l_0 - l_s)}$$

where

- l_0 is the log-likelihood function without using any covariate (just the intercept),
- l_n is the log-likelihood of the model including the n selected design metrics as covariates,
- l_s is the log-likelihood of the *saturated model*, i.e., where Y_i, (0 or 1) is substituted for each probability $\pi(X_i)$ in l. The log-likelihood l_s is the maximum value that can be assigned to l.

The higher the R^2, the more accurate the model. However, as opposed to the R^2 of least-square regression, high R^2's are rare for logistic regression because l_n rarely approaches the value of l_s since the computed $\pi(X_i)$'s in l_n rarely approach 1. The interested reader may refer to [Hosmer&Lemeshow, 1989] for a detailed introduction to logistic regression. Finally, R^2 may be described as a measure of the *proportion of total uncertainty* that is attributed to the model fit.

3.2.2 Univariate Analysis

In this section, we analyze the six OO metrics introduced in [Chidamber&Kemerer, 1994] (though slightly adapted to our context) with regard to the probability of fault detection in a class during test phases. In our case, it is equivalent for the logistic model to calculate the probability of a single fault to be detected in a class.

- Weighted Methods per Class (WMC) was shown to be somewhat significant ($\alpha = 0.06$) overall. For new and extensively modified classes and for UI (Graphical and Textual User Interface) classes, the results are much better: $\alpha = 0.0003$ and $\alpha = 0.001$, respectively. As expected, the larger the WMC, the larger the probability of fault detection. These results can be explained by the fact that the internal complexity does not have a strong impact if the class is reused verbatim or with very slight modifications. In that case, the class interface properties will have the most significant impact.
- Depth of Inheritance Tree of a class (DIT) was shown to be very significant ($\alpha = 0.0000$) overall. As expected, the larger the DIT, the larger the probability of defect detection. Again, results improve (Logistic R^2 goes from 0.06 to 0.13) when only new and extensively modified classes are considered.
- Response For a Class (RFC) was shown to be very significant overall ($\alpha = 0.0000$). Predictably, the larger the RFC, the larger the probability of defect detection. However, the logistic R^2 improved significantly for new and extensively modified classes and UI classes (from 0.06 to 0.24 and 0.36, respectively). Reasons are believed to be the same as for WMC for extensively modified classes. In addition, UI classes show a distribution which is significantly different from that of DB classes: the mean and median are significantly higher. This, as a result, may strengthen the impact of RFC when performing the analysis.
- Number Of Children of a Class (NOC) appeared to be very significant (except in the case of UI classes) but the observed trend is contrary to what was expected. The larger the NOC, the lower the probability of defect detection. This surprising trend can be explained by the combined facts that most classes do not have more than one child and that verbatim reused classes are somewhat associated with a large NOC. Since we have observed that reuse was a

significant factor in fault density [Melo *et. al.*, 1995], this explains why large NOC classes are less fault-prone. Moreover, there is some instability across class subsets with respect to the impact of NOC on the probability of detecting a fault in a class (see $\Delta\psi$'s in Table 2). This may be explained in part by the lack of variability on this measurement scale (see distributions in Figure 1).

- Lack of Cohesion on Methods (LCOM) was shown to be insignificant in all cases (this is why the results are not shown in Table 2) and this should be expected since the distribution of LCOM shows a lack of variability and a few very large outliers. This stems in part from the definition of LCOM where the metric is set to 0 when the number of class pairs sharing variable instances is larger than that of the ones not sharing any instances. This definition is definitely not appropriate in our case since it sets cohesion to 0 for classes with very different cohesions and keeps us from analyzing the actual impact of cohesion based on our data sample.
- Coupling Between Object classes (CBO) is significant and more particularly so for UI classes ($\alpha = 0.0000$ and $R^2 = 0.17$). No satisfactory explanation could be found for differences in pattern between UI and DB classes.

It is important to remember, when looking at the results in Table 2, that the various metrics have different units. Some of these units represent "big steps" on each respective measurement scale while others represent "smaller steps". As a consequence, some coefficients show a very small impact (i.e., $\Delta\psi$'s) when compared to others. This is not, however, a valid criterion to evaluate the predictive usefulness of such metrics.

Most importantly, besides NOC, all metrics appear to have a very stable impact across various categories of classes (i.e., DB, UI, New-Ext, etc.). This is somewhat encouraging since it tells us that, in that respect, the various types of components are comparable. If we were considering different types of faults separately, results might be different. Such a refinement is, however, part of our future research plans.

Metrics	Coefficient	$\Delta\psi$	α	R^2	Classes
WMC (1)	-0.022	98%	0.0607	0.007	ALL
WMC (2)	-0.086	92%	0.00035	0.024	New-Ext
WMC (3)	-0.027	103%	0.0656	0.0154	DB
WMC (4)	-0.0944	91%	0.0019	0.0467	UI
DIT (1)	-0.485	62%	0.0000	0.0648	ALL
DIT (2)	-0.868	42%	0.0000	0.1314	New-Ext
DIT (3)	-0.475	62%	0.043	0.0187	DB
DIT (4)	-0.29	75%	0.024	0.017	UI
RFC (1)	-0.085	92%	0.0000	0.0648	ALL
RFC (2)	-0.087	92%	0.0000	0.2477	New-Ext
RFC (3)	-0.077	93%	0.0000	0.188	DB
RFC (4)	-0.108	90%	0.0000	0.3624	UI
NOC (1)	3.3848	3000%	0.0000	0.1426	ALL
NOC (2)	3.62	3734%	0.0011	3.6235	New-Ext
NOC (3)	2.05	777%	0.0000	0.0826	DB
CBO (1)	-0.142	87%	0.0000	0.068	ALL
CBO (2)	-0.079	92%	0.017	0.02	New-Ext
CBO (3)	-0.086	92%	0.006	0.034	DB
CBO (4)	-0.284	75%	0.0000	0.17	UI

Table 2: Univariate Analysis - Summary of experimental results.

3.2.3 Multivariate Analysis

The OO design metrics presented in the previous section can be used early in the life cycle to build a predictive model of fault-prone classes. In order to obtain an optimal model, we included these metrics into a multivariate logistic regression model. However, only the metrics that significantly improve the predictive power of the multivariate model were included through a stepwise selection process. Another significant predictor of fault-proneness is the level of reuse of the class (called "origin" in Table 3). This information is available at the end of the design phase when reuse candidates have been identified in available libraries and the required amount of change can be estimated. Table 3 describes the computed multivariate model. Using such a model for classification, the results shown in Table 4 are obtained by using a classification threshold of p(Fault detection) = 0.5 for the probability of detecting a single defect in a given class, i.e., when $p > 0.5$, the class is classified as faulty and otherwise as non-faulty. As expected, classes predicted as faulty contain a large number of faults (250 faults on 48 classes) because those classes tend to show a better classification accuracy.

We now assess the impact of using such a prediction model by assuming, in order to simplify computations, that inspections of classes are 100% effective in finding faults. In that case, 80 classes (predicted as faulty) out of 180 would be inspected and 48 faulty classes out of 58 would be identified before testing. If we now take into account individual faults, 250 faults out of 258 would be detected during inspection. As mentioned above, such a good result stems from the fact that the prediction model is more accurate for multiple-faults classes.

	Coefficient	α
Intercept	3.13	0.0000
DIT	-0.50	0.0004
RFC	-0.11	0.0000
NOC	2.01	0.0178
RFC	-0.13	0.0072
CBO	-0.238	0.0001
Origin	-1.84	0.0000

Table 3: Multivariate Analysis with OO design metrics

Predicted Actual	**No fault**	**Fault**
No Fault	90	32
Fault	10 (18)	48 (250)

Table 4: Classification Results with OO Design Metrics. The figures before parentheses in the right column are the number of classes classified as faulty. The figures between the parentheses are the faults contained in those classes.

In order to evaluate the predictive accuracy of these OO design metrics, it would be interesting to compare their predictive capability with the one of the usual code metrics, that can only be obtained later in the development life cycle. Three code metrics, among the ones provided by the Amadeus tool [Amadeus, 1994], were selected through a stepwise regression procedure. Table 5 shows the resulting parameter estimations of the multivariate logistic regression model where: *MaxStatNext* is the maximum level of statement nesting in a class, *FunctDef* is the number of function declarations, and *FunctCall* is the number of function calls. However, based on the whole set of metrics provided by Amadeus, other multivariate models yield results of similar accuracy. This model happens to be, however, the model resulting from the use of a standard, stepwise logistic regression analysis procedure.

	Coefficient	α
Intercept	0.39	0.0384
MaxStatNest	-0.286	0.0252
FunctDef	0.166	0.0010
FunctCall	-0.0277	0.0000

Table 5: Multivariate Analysis with Code Metrics

In addition to being collectable only later in the process, code metrics appear to be somewhat poorer as predictors of class fault-proneness (see Table 6). In this case, 112 classes (predicted as faulty) out of 180 would be inspected and 51 faulty classes out of 58 would be detected. If we now take into account individual faults, 231 faults out of 268 would be detected during inspection. Three more faulty classes would be corrected (51 versus 48) but 32 more classes would have to be inspected (112 versus 80). Moreover, the OO design metrics are better predictors of classes containing large numbers of faults since 19 more faults (250 versus 231) would be detected in that case. Therefore, predictions based on code metrics appear to be poorer. Table 7 confirms that result by showing the values of correctness (percentage of classes correctly predicted as faulty) and completeness (percentage of faulty classes detected). Values between parentheses present predictions' correctness and completeness values when classes are weighted according to the number of faults they contain (classes with no fault are weighted 1).

Predicted **Actual**	**No fault**	**Fault**
No Fault	61	61
Fault	7 (37)	51 (231)

Table 6: Classification Results based on code metrics shown in Table 5

Model **Accuracy**	**OO** **metrics**	**Code** **metrics**
Completeness	88% (93%)	83% (86%)
Correctness	60% (92%)	45.5% (86%)

Table 7: Classification Accuracies based on OO and code metrics shown in Table 3 and Table 5

4. Related Work

As far as we know, the only studies attempting to experimentally validate OO metrics are [Lie&Henry, 1993] and [Briand et. al., 1994]. In [Briand et. al. ,1994], metrics for measuring abstract data type (ADT) cohesion and coupling are proposed and are experimentally validated as predictors of faulty ADT's. Further work will consist of verifying that the metrics proposed by [Briand et. al. ,1994] are also applicable to C++ programs, in a context of inheritance.

To the knowledge of the authors, [Lie&Henry, 1993] is the only study which can really be compared to the work we describe in this paper. Li and Henry have proposed a suite of OO design metrics. They validated this suite of metrics by studying the number of changes performed in two commercial systems implemented with an OO dialect of Ada. The suite of OO design metrics used by Li and Henry extends Chidamber&Kemerer's OO metrics with two additional metrics:

- Message Passing Coupling (MPC) which is calculated as the number of send statements defined in a class.
- Data Abstraction Coupling (DAC) which is calculated as the number of abstract data types used in the measured class and defined in another class of the system.

They combined the six Chidamber&Kemerer's OO metrics with these last two metrics in a single least-square regression model. According to the authors, their model was adequate in predicting the size of changes in classes during the maintenance phase. They did not, however, look at the time spent changing a class nor the cause of changes (e.g., corrections, enhancement, etc.). In addition, they assumed that the number of modifications in a class is proportional to the effort spent to change it, which is not necessarily true. Also, we do not believe that the number of changes can be considered as a measure of maintainability since it is not dependent on the modifiability of a class but on the correctness and functional stability of the class.

In this study, we did not consider DAC and MPC because they could not be directly applied in our experimental context (C++ does not provide send statements). Based on the way DAC was defined by Lie&Henry, it cannot be directly used for C++. DAC could, however, be redefined/tailored to our needs, providing another way to calculate coupling across C++ classes. This is, however, beyond of the scope of this paper.

An important difference in our work is that we have used the occurrence of faults in a class to verify whether Chidamber&Kemerer's OO metrics were adequate quality predictors. Of course, many other quality measures of interest could be used in this context, e.g., change productivity. Last, the modeling technique we used (i.e., logistic regression) to predict fault-prone classes is different because of the nature of the dependent variable which is binary in our case. This has led us to use a classification technique.

5. Conclusions and further work

In this experiment, we collected data about defects found in Object-Oriented classes. Based on these data we verified experimentally how much fault-proneness is influenced by internal (e.g., size, cohesion) and external (e.g., coupling) design characteristics of OO classes. From the results presented above, several of Chidamber&Kemerer's OO metrics appear to be adequate to predict class fault-proneness during the early phases of the life-cycle. We also showed that Chidamber&Kemerer's OO metrics are better predictors than "traditional" code metrics on our data set, which, in addition, can only be collected at a later phase of the software development processes.

Our future work includes:

- replicating this study in an industrial setting: a sample of large-scale projects developed in C++ and Ada95 in the framework of the NASA Goddard Flight Dynamics Division (Software Engineering Laboratory). This work should help us better understand the prediction capabilities of the suite of OO metrics described in this paper. By doing that, we intend to:

  - build models and provide guidance to improve the allocation of resources with respect to test and verification efforts,
  - gain a better understanding of the impact of OO design strategies (e.g., simple versus multiple inheritance) on defect density and rework. In this study, because of an inadequate data collection process, we were unable to analyze the capability of OO design metrics to predict rework. We believe that this drawback could be overcome by refining our data collection process in order to capture how much effort was spent on each class individually.

- analyzing OO libraries in order to identify "good" and "bad" OO design patterns. Design patterns have been claimed to be a way to improve reuse and quality of OO software systems [Gamma et. al, 1995]. We intend to use the approach described in this paper to assess organization-specific design patterns, thus providing guidelines about what OO design patterns should be encouraged and which ones should be avoided due to their fault-proneness or their lack of maintainability.
- studying the variations, in terms of metric definitions and experimental results, between different OO programming languages. The fault-proneness prediction capabilities of the suite of OO metrics discussed in this paper can be different depending on the used programming language. Work must be undertaken to validate this suite of OO design metrics across different OO languages, e.g., Ada95, Smalltalk, Eifeil, C++, etc.
- extending the experimental investigation to other OO metrics proposed in the literature (e.g., [Abreu&Carapuça, 1994]) and develop new metrics, e.g., more language specific.

Acknowledgements

This work would not have been possible without the help of Dr. Prem Devanbu from AT&T Labs. We also want to thank Dr. Gianluigi Caldiera for co-teaching the OMT method to the students who participated in this study, and Carolyn Seaman, Barbara Swain and Roseanne Tesoriero for their suggestions that helped improve both the content and the form of this paper.

References

F. B. Abreu and R. Carapuça (1994). "Candidate metrics for object-oriented software within a taxonomy framework". *Journal of System and Software*, 26(1):87–96.

Amadeus Software Research, Inc. (1994). "*Getting Started with Amadeus*." Amadeus Measurement System.

V. Basili; G. Caldiera; F. McGarry; R. Pajerski; G. Page (1992). "The Software Engineering Laboratory: An Operational Software Experience Factory". In *Proc. of the 14th Int'l Conf. on Software Engineering.*

V. Basili and D. Hutchens (1982). "Analyzing a syntactic family of complexity metrics". *IEEE Trans. on Software Engineering*, SE-9(6):664–673.

L. Briand; S. Morasca; V. Basili (1994). *"Goal-Driven Definition on Product Metrics Based on Properties"*. CS-TR-3346, University of Maryland, Dep. of CS, College Park, MD, 20742.

I. Brooks (1993). "Object-oriented metrics collection and evaluation with a software process". Presented at *OOPSLA'93 Workshop on Processes and Metrics for Object-Oriented Software Development*, Washington, DC.

S. R. Chidamber and C. F. Kemerer (1994). "A metrics suite for object-oriented design". *IEEE Trans. on Software Engineering*, 20(6):476–493.

S. R. Chidamber and C. F. Kemerer (1995). "Authors Reply". *IEEE Trans. on Software Engineering, IEEE Trans. on Software Engineering*, 21(3):265.

N. I. Churcher and M. J. Shepperd (1995). "Comments on 'A Metrics Suite for Object-Oriented Design' ". *IEEE Trans. on Software Engineering*, 21(3):263–265.

P. Devanbu (1992). "GENOA/GENII – a customizable language and front-end independent code analyzer". In *Proc. of the 14th Int'l Conf. on Software Engineering*, Melbourne, Australia.

J. Devore (1991). *"Probability and Statistics for Engineering and Sciences."* Brooks/Cole Publishing Company.

W. Dillon and M. Goldstein (1984). *"Multivariate Analysis: Methods and Applications."* Wiley.

W. Harrison (1988). "Using software metrics to allocate testing resources." *Journal of Management Information Systems*. 4(4):93–105.

W. Harrison (1994). "Software measurement: a decision-process approach". *Advances in Computers*, vol 39, pp. 51–105.

G. Heller; J. Valett; M. Wild (1992). *Data Collection Procedure for the Software Engineering Laboratory (SEL) Database.* SEL Series, SEL-92-002.

D. Hosmer and S. Lemeshow (1989). *"Applied Logistic Regression."* Wiley-Interscience.

N. E. Fenton (1991). "*Software Metrics: A Rigorous Approach*". Chapman&Hall.

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

W. Lie and S. Henry (1993). "Object-oriented metrics that predict maintainability". *Journal of Systems and Software*. 23(2):111–122.

F. McGarry; R. Pajersk; G. Page; S. Waligora; V. Basili; M. Zelkowitz (1994). "*Software Process Improvement in the NASA Software Engineering Laboratory*". Carnegie Mellon University, Software Engineering Institute, Technical Report, Dec. 1994. CMU/SEI-95-TR-22.

W. Melo; L. Briand; V. Basili (1995). "*Measuring the Impact of Reuse on Quality and Productivity in Object-Oriented Systems.*" Technical Report, University of Maryland, Dep. of Computer Science, Jan. 1995, CS-TR-3395.

J. Rumbaugh; M. Blaha; W. Premerlani; F. Eddy; W. Lorensen (1991). "*Object-Oriented Modeling and Design.*" Prentice-Hall.

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

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