

A Validation of Object-Oriented Design Metrics as Quality Indicators

Victor R. Basili, *Fellow, IEEE*, Lionel C. Briand,
and Walcélio L. Melo, *Member, IEEE Computer Society*

Abstract—This paper presents the results of a study in which we empirically investigated the suite of object-oriented (OO) design metrics introduced in [13]. More specifically, our goal is to assess these metrics as predictors of fault-prone classes and, therefore, determine whether they can be used as early quality indicators. This study is complementary to the work described in [30] 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 empirical and quantitative analysis, the advantages and drawbacks of these OO metrics are discussed. Several of Chidamber and Kemerer's OO metrics appear to be useful to predict class fault-proneness during the early phases of the life-cycle. Also, on our data set, they are better predictors than "traditional" code metrics, which can only be collected at a later phase of the software development processes.

Index Terms—Object-oriented design metrics, error prediction model, object-oriented software development, C++ programming language.

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 development. Software metrics are, thus, necessary to identify where the resources are needed; they are a crucial source of information for decision-making [22].

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 modules so that testing/verification effort can be concentrated on these modules [21]. The availability of adequate product design metrics for characterizing error-prone modules is, thus, vital.

Many product metrics have been proposed [16], [26], used, and, sometimes, empirically validated [3], [4], [19], [30], e.g., number of lines of code, McCabe complexity metric, etc. In fact, many companies have built their own cost, quality, and resource prediction models based on product metrics. TRW [7], the Software Engineering Laboratory (SEL) [31], and Hewlett Packard [20] 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, in [12] it was 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 [1], [6], [13]. However, with a few exceptions [10], [30], most of them have not undergone an

- V.R. Basili is with the University of Maryland, Institute for Advanced Computer Studies and Computer Science Dept., A.V. Williams Bldg., College Park, MD 20742. E-mail: basili@cs.umd.edu.
- L.C. Briand is with Fraunhofer-Institute for Experimental Software Engineering, Technologiepark 11, Sauerwiesen 6, D-67661, Kaiserslautern, Germany. E-mail: briand@iese.ftg.de.
- W.L. Melo is with the Centre de Recherche Informatique de Montréal, 1801 McGill College Ave., Montréal, Québec, H3A 2N4, Canada. E-mail: wmelo@crim.ca.

Manuscript received Sept. 7, 1995; accepted Aug. 30, 1996.

Recommended for acceptance by H. Muller.

For information on obtaining reprints of this article, please send e-mail to: transse@computer.org, and reference IEEECS Log Number S95819.

empirical validation (see [9] and [35] for further discussion of the empirical validation of measures). Empirical validation aims at demonstrating the usefulness of a measure in practice and is, therefore, a crucial activity to establish the overall validity of a measure. A measure may be correct from a measurement theory perspective (i.e., be consistent with the agreed upon empirical relational system) but be of no practical relevance to the problem at hand. On the other hand, a measure may not be entirely satisfactory from a theoretical perspective but can be a good enough approximation and work fine in practice.

In this paper, we present the results of a study in which we performed an empirical validation of the OO metric suite defined in [13] with regard to their ability to identify fault-prone classes. However, the theoretical validation of these metrics is not addressed here and, as a complement to this paper, the reader may refer to a discussion about the mathematical properties of Chidamber and Kemerer's metrics in [11], [24].

Data were collected during 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 Object-Oriented analysis/design method [33], and the C++ programming language [36]. Despite the fact that these projects were run in a university setting, we set up a framework that was representative of currently used technology in industrial settings.

1.3 Outline

This paper is organized as follows. Section 2 presents the suite of OO metrics proposed by Chidamber and Kemerer [13], offers the experimental hypotheses to be tested, and then shows a case study from which process and product data were collected allowing a quantitative validation of this suite of metrics. Section 3 presents the actual data collected together with the statistical analysis of the data. Section 4 compares our study with other works on the subject. Section 5 concludes the paper by presenting lessons learned and future work.

2 DESIGN OF THE EMPIRICAL STUDY

2.1 Dependent and Independent Variables

The goal of this study was to analyze empirically the OO design metrics proposed in [13] for the purpose of evaluating whether or not these metrics are useful for predicting the probability of detecting faulty classes. Assuming testing was performed properly and thoroughly, the probability of fault detection in a class during acceptance testing should be a good indicator of its probability of containing a fault and, therefore, a relevant measure of fault-proneness. The construct validity¹ of our dependent variable can, thus, be demonstrated.

Other measures such as class fault density could have been used. However, the variability in terms of number of

faults in our data set is small: Faults were detected only in 36 percent of the classes and 84 percent of the classes contain less than three faults. Therefore, using a dependent variable with low variability would have affected our ability to identify significant relationships between OO design metrics and this dependent variable.

In addition, it was difficult to decide what was the best way to measure the size of classes given the large number of alternatives (e.g., LOC, SLOC, number of methods, number of attributes, etc.). The probability of fault detection was, therefore, the most straightforward and practical measure of fault-proneness and, therefore, a suitable dependent variable for our study. Based on [13], [14], and [15], it is clear that the definitions of these metrics are not language independent. As a consequence, we had to slightly adjust some of Chidamber and 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 [13], 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 to the one described in [14].
- In [15], Churcher and Shepperd 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 in [15] correspond to different metrics, similar to the WMC metric, and which must be empirically validated as well. A validation of Churcher and 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.
- **Number Of Children of a Class (NOC)**—This is the number of direct descendants for each class.
- **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.
- **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 C++ functions directly invoked by member functions or operators of a C++ class.

1. Construct validity is discussed further in [27]. It is defined as the extent to which the theoretical construct of interest (e.g., our dependent variable: fault-proneness) is measured successfully, i.e., do we really measure what we purport to measure?

- **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 zero whenever the above subtraction is negative.

Readers acquainted with C++ can see that some particularities of C++ are not taken into account by Chidamber and 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 Hypotheses

In order to validate the above metrics as quality indicators, their expected relationship with fault-proneness (or rather the measure we selected for this attribute: probability of fault detection) must be validated. The experimental hypotheses to be statistically tested are, for each metric, as follows:

- **H-WMC:** A class with significantly more member functions than its peers is more complex and, by consequence, tends to be more fault-prone.
- **H-DIT:** 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. In addition, deep hierarchies often imply problems of conceptual integrity, i.e., it becomes unclear which class to specialize from in order to include a subclass in the inheritance hierarchy [17].
- **H-NOC:** Classes with large number of children (i.e., subclasses) are difficult to modify and usually require more testing because the class potentially affects all of its children. Furthermore, a class with numerous children may have 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 and, therefore, we expect classes with large number of children to be more fault-prone.
- **H-CBO:** Highly coupled classes are more fault-prone than weakly coupled classes because they depend more heavily on methods and objects defined in other classes.
- **H-RFC:** Classes with larger response sets implement more complex functionalities and are, therefore, more fault-prone.
- **H-LCOM:** Classes 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 more fault-prone.

2.3 Study Participants

In order to validate the hypotheses stated in the previous section, we ran an empirical study over four months (from September to December 1994). The study participants were the students of an upper division undergraduate/graduate

level course offered by the Department of Computer Science at the University of Maryland. The objective of this class was to teach OO software analysis and design. 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 a study.

In order to control for differences in skills and experience among students, the students were randomly grouped into eight teams of three students. Furthermore, in order to ensure the groups were comparable with respect to the ability of their members, the following procedure (i.e., known as "blocking" [27]) was used to assign students to groups:

- First, the level of experience of each student was characterized at the beginning of the study. We used questionnaires and performed interviews. We asked the students information regarding their previous working experience, their student status (part-time, full-time student), their computer science degree (BS, MSc, PhD), their previous experiences with analysis/design methods, and their skill regarding various programming languages.
- Second, each of the eight most experienced students was randomly assigned to a different group (i.e., team). Students considered most experienced were computer science PhD candidates who had already implemented large (≥ 10 thousands source lines of code, KSLOC) C or C++ programs and those with industrial experience greater than two years in C programming. None of the students had significant experience in Object-Oriented software analysis and design methods. Similarly, each of the eight next most experienced students were randomly assigned to different groups and this was repeated for the remaining eight students.

2.4 The Development Process

Each team was asked to develop a medium-sized management information system that supports the rental/return process of a hypothetical video rental business, and maintains customer and video databases. Such an application domain had the advantage of being easily comprehensible and, therefore, we could make sure that system requirements could be easily interpreted by students regardless of their educational background.

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 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. Acceptance testing was performed by an independent group (see Section 2.5). 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 [33]. 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. Our results are, thus, more likely to be generalizable to other development environments (external validity).

The following libraries were provided to the students:

- 1) *MotifApp*. This public domain library provides a set of C++ classes on top of OSF/MOTIF for manipulation of windows, dialogues, menus, etc. [37]. The MotifApp library provides a way to use the OSF/Motif widgets in an OO programming/design style.
- 2) *GNU library*. This public domain library is provided in the GNU C++ programming environment. It contains functions for manipulation of string, files, lists, etc.
- 3) *C++ database library*. This library provides a C++ implementation of multi-indexed B-Trees.

We also provided a specific domain application library in order to make our study more representative of industrial conditions. 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.

No special training was provided for the students 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. In addition, the source code and the complete documentation of the libraries were made available. Finally, it is important to note the students were not required to use the libraries and, depending on the particular design they adopted, different reuse choices were expected.

2.5 Testing

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, spending eight hours testing each system.

2.6 Nature of the Study

Our empirical study is not what could be called formally a *controlled* experiment since the independent variables (i.e., OO design metrics) are not controlled for and not as-

signed randomly to classes. Such a design would not be implementable. Rather, our study is more observational in nature. However, it is important to note that we have tried to make the results of our study as generalizable as possible (i.e., maximizing external validity) by a careful selection of the study participants, the study material, and the development process. Nevertheless, there is a greater danger that the study be exposed to confounding variables and all significant relationships should be carefully interpreted.

2.7 Data Collection Procedures and Measurement Instruments

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++ [18] was used to extract Chidamber and 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 [23]:

- Fault Report Form.
- Component Origination Form.

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

2.7.1 Data Collection Forms

A fault report form was used to gather data about

- 1) the faults found during the testing phase,
- 2) classes changed to correct such faults, and
- 3) the effort in correcting them.

The latter was not used in this study. Further details can be found in [5].

A component origination form was used to record information that characterizes each class under development in the project at the time it goes into configuration management. First, this form was 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 needed to meet the system requirements and design: none, slight (less than 25 percent of code changed), or extensive (more than 25 percent of code change) as well as the name of the reused class. 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 two types of sub-systems: user interface (UI) and database processing (DB).

2.7.2 Data Collected

Chidamber and Kemerer's OO design metrics were collected for each of the 180 classes across the eight systems under study. In addition, all faults detected during testing

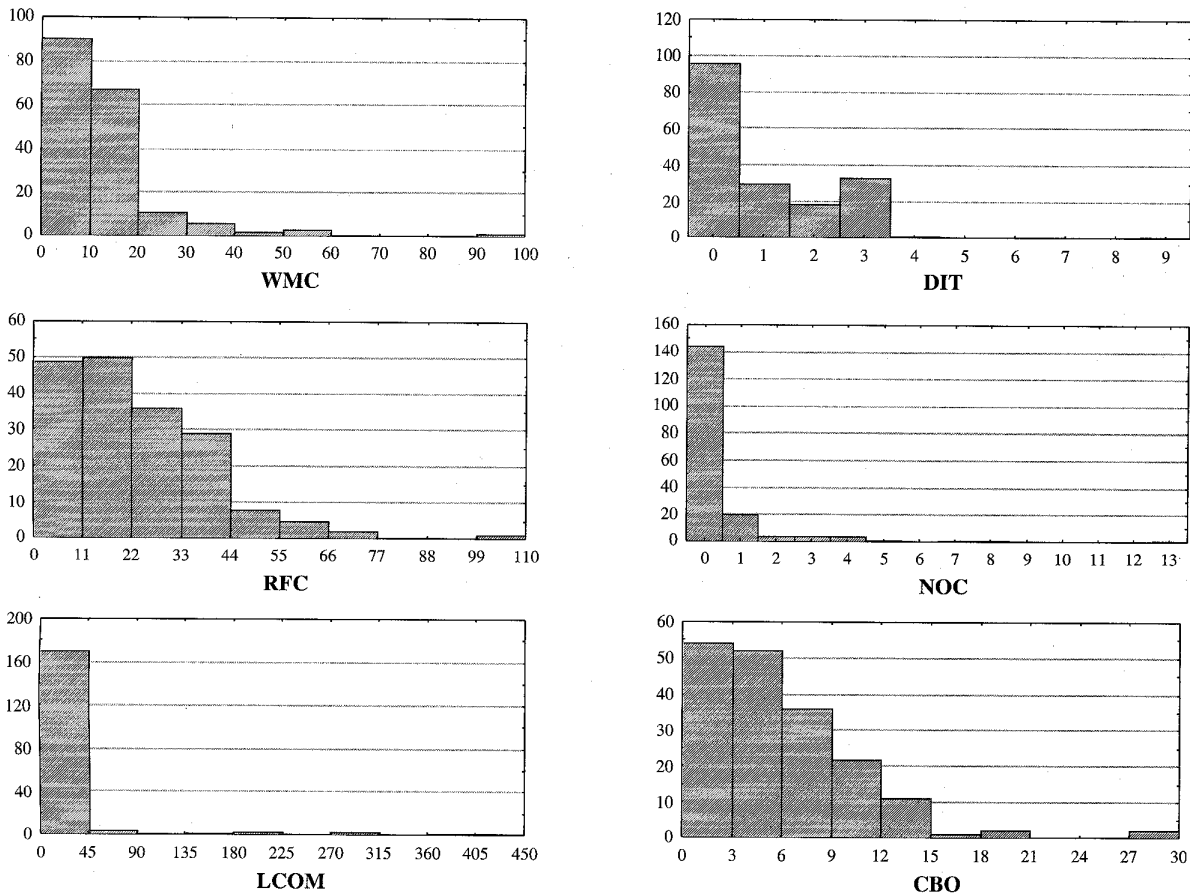


Fig. 1. Distribution of the analyzed OO metrics. The X axes represents

the values of the metric. The Y axes represents the number of class.

activities were located in the systems and, therefore, associated with one or several of their classes.

3 DATA ANALYSIS

In this section, we will assess empirically whether the OO design metrics defined in [13] are useful predictors of fault-prone classes. This will help us assess these metrics as quality indicators and how they compare to common code metrics. 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 is made [9]. 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 Distribution and Correlation Analyses

Fig. 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) (this result is similar to what was found in [13]). In addition, most classes show a lack of

cohesion (LCOM) near zero. This latter metric does not seem to differentiate classes well and this may stem from its definition which prevents any negative measure. This issue will be discussed further in Section 3.2.

TABLE 1
DESCRIPTIVE STATISTICS OF THE 180 STUDIED C++ CLASSES

	WMC	DIT	RFC	NOC	LCOM	CBO
Maximum	99.00	9.00	105.00	13.00	426.00	30.00
Minimum	1.00	0.00	0.00	0.00	0.00	0.00
Median	9.50	0.00	19.50	0.00	0.00	5.00
Mean	13.40	1.32	33.91	0.23	9.70	6.80
Std Dev	14.90	1.99	33.37	1.54	63.77	7.56

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.

TABLE 2
CORRELATION ANALYSIS

	r^2 Values					
	WMC	DIT	RFC	NOC	LCOM	CBO
WMC	1	0.02	0.24	0	0.38	0.13
DIT		1	0	0	0.01	0
RFC			1	0	0.09	0.31
NOC				1	0	0
LCOM					1	0.01

Table 2 shows very clearly that linear Pearson's correlations (R^2 : Coefficient of determination) between the studied OO metrics are, in general, very weak. Three coefficients of determination appear somewhat more significant (bold coefficients in Table 2). However, when looking at the scatterplots, only the relationship between CBO and RFC seems not to be due to outliers. We conclude that these metrics are mostly statistically independent and, therefore, do not capture a great deal of redundant information.

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, a standard technique based on maximum likelihood estimation, to analyze the relationships between metrics and the fault-proneness of classes. Currently, logistic regression is a standard classification technique [25] used in experimental sciences. It has already been used in software engineering to predict error-prone components [8], [29], [32].

Other classification techniques such as classification trees [34], Optimized Set Reduction [8], or neural networks [28] could have been used. However, our goal here is not to compare multivariate analysis techniques (see [8] for a comparison study) but, based on a suitable and standard technique, to validate empirically a set of metrics.

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. This modeling process is further described in [25].

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):

$$\pi(X_1, X_2, \dots, X_n) = \frac{e^{(C_0 + C_1 \cdot X_{i1} + \dots + C_n \cdot X_{in}) \cdot Y_i}}{1 + e^{(C_0 + C_1 \cdot X_{i1} + \dots + C_n \cdot X_{in})}} \quad (*)$$

where π is the probability that a fault was found in a class during the validation phase, and the X_i s are the design metrics included as explanatory variables in the model (called *covariates* of the logistic regression equation). The curve between π and any single X_i —i.e., assuming that all other X_j s are constant—takes a flexible S shape which ranges between two extreme cases:

- 1) when a variable is not significant, then the curve approximates a horizontal line, i.e., π does not depend on X_i and
- 2) when a variable entirely differentiates error-prone software parts, then the curve approximates a step function.

Such a S shape is perfectly suitable as long as the relationship between X_i s and π is monotonic, an assumption consistent with the empirical hypotheses to be tested in this study. Otherwise, higher degree terms have to be introduced in equation (*).

The coefficients C_i s will be estimated through the maximization of a likelihood function, built in the usual fashion, i.e., as the product of the probabilities of the single observations, which are functions of the covariates (whose values are known in the observations) and the coefficients (which are the unknowns). For mathematical convenience, $l = \ln[L]$, the loglikelihood, is usually the function to be maximized. This procedure assumes that all observations are statistically independent. In our context, an observation is the (non)detection of a fault in a C++ class. Each (non) detection of a fault is assumed to be an event independent from other fault (non)detections. Each data vector in the data set describes an observation and has the following components: An event category (fault, no fault) and a set of OO design metrics (described in Section 2.1) characterizing either the class where the fault was detected or a class where no fault was detected.

The global measure of goodness of fit we will use for such a model is assessed via R^2 —not to be confused with the least-square regression R^2 —they are built upon very different formulae, even though they both range between zero and one and are similar from an intuitive perspective. The higher R^2 , the higher the effect of the model's explanatory variables, 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. For this reason, the reader should not interpret logistic regression R^2 s using the usual heuristics for least-square regression R^2 s. (The interested reader may refer to [21] for a detailed discussion of this issue.). Logistic regression R^2 is defined by the following ratio:

$$R^2 = \frac{LL_S - LL}{LL_S}$$

where

- ◆ LL is the loglikelihood obtained by Maximum Likelihood Estimation of the model described in formula (*)
- ◆ LL_S is the loglikelihood obtained by Maximum Likelihood Estimation of a model without any variables, i.e., with only C_0 . By carrying out all the calculations, it can be shown that LL_S is given by

$$LL_S = m_0 \ln\left(\frac{m_0}{m_0 + m_1}\right) + m_1 \ln\left(\frac{m_1}{m_0 + m_1}\right)$$

where m_0 (resp., m_1) represents the number of observations for which there are no faults (resp., there is a fault). Looking at the above formula, $LL_S/(m_0 + m_1)$ may be interpreted as the uncertainty associated with the distribution of the dependent variable Y , according to Information Theory concepts. It is the uncertainty left when the variable-less model is used. Likewise, $LL/(m_0 + m_1)$ may be interpreted as the uncertainty left when the model with the covariates is used. As a consequence, $(LL_S - LL)/(m_0 + m_1)$ may be interpreted as the part of uncertainty that is explained by the model. Therefore, the ratio $(LL_S - LL)/LL_S$ may be interpreted as the proportion of uncertainty explained by the model.

Tables 3 and 4 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 3 and 4), the estimated regression coefficient. The larger the coefficient in absolute value, the stronger the impact (positive or negative, according to the sign of the coefficient) of the explanatory variable on the probability p of a fault to be detected in a class.

TABLE 3
UNIVARIATE ANALYSIS—SUMMARY OF RESULTS

Metrics	Coefficient	$\Delta\psi$	p-value	R ²	Classes
WMC (1)	0.022	2%	0.0607	0.007	ALL
WMC (2)	0.086	9%	0.0003	0.024	New-Ext
WMC (3)	0.027	3%	0.0656	0.015	DB
WMC (4)	0.094	10%	0.0019	0.047	UI
DIT (1)	0.485	62%	0.0000	0.065	ALL
DIT (2)	0.868	138%	0.0000	0.131	New-Ext
DIT (3)	0.475	60%	0.043	0.019	DB
DIT (4)	0.29	34%	0.024	0.017	UI
RFC (1)	0.085	9%	0.0000	0.065	ALL
RFC (2)	0.087	8%	0.0000	0.248	New-Ext
RFC (3)	0.077	8%	0.0000	0.188	DB
RFC (4)	0.108	11%	0.0000	0.362	UI
NOC (1)	-3.3848	-96%	0.0000	0.143	ALL
NOC (2)	-3.62	-97%	0.0011	0.362	New-Ext
NOC (3)	-2.05	-77%	0.0000	0.083	DB
CBO (1)	0.142	15%	0.0000	0.068	ALL
CBO (2)	0.079	8%	0.017	0.020	New-Ext
CBO (3)	0.086	9%	0.006	0.034	DB
CBO (4)	0.284	33%	0.0000	0.170	UI

ALL means all the classes. New-Ext stands for classes which have been created from scratch or extensively modified. DB labels classes implementing database manipulations. UI labels classes implementing user interface functions.

TABLE 4
MULTIVARIATE ANALYSIS WITH OO DESIGN METRICS

	Coefficient	p-value
Intercept	3.13	0.0000
DIT	0.50	0.0004
RFC	0.11	0.0000
NOC	-2.01	0.0178
CBO	0.13	0.0072
Class Origin	1.84	0.0000

- $\Delta\psi$ (appearing in Table 3 only), which is based on the notion of odd ratio [25], 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 two, 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:

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

Therefore, $\Delta\psi$ represents the reduction/increase in the odds ratio (expressed as a percentage in Table 3) when the value X increases by one unit. This is designed to provide an intuitive insight into the impact of explanatory variables.

- The statistical significance (p-value, appearing in Tables 3 and 4) 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. Historically, a significance threshold of $\alpha = 0.05$ (i.e., 5 percent probability) has often been used to determine whether an explanatory variable was 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 [25] commonly used in the framework of logistic regression.

3.2.2 Univariate Analysis

In this section, we analyze the relationships between six OO metrics introduced in [13] (though slightly adapted to our context) and the probability of fault detection in a class during test phases. Thus, we intend to test the hypotheses stated in Section 2.2.

- Weighted Methods per Class (WMC) was shown to be somewhat significant (p-value = 0.06) overall. For new and extensively modified classes and for UI (Graphical and Textual User Interface) classes, the results are more significant: p-value = 0.0003 and p-value = 0.001, respectively. Therefore, the H-WMC hypothesis is supported by these results: 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 (p-value = 0.0000) overall. The H-DIT hypothesis is supported by the results: The larger the DIT, the larger the probability of fault detection. Again, the strength of the relationship increases (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 (p-value = 0.0000). The H-RFC hypothesis is supported by the results: The larger the RFC, the larger the probability of fault detection. Again, 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 stated by

the H-NOC hypothesis: The larger the NOC, the lower the probability of fault 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 negative factor on fault density [5], 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 3). This may be explained in part by the lack of variability on the NOC measurement scale (see descriptive analysis in Table 1 and distribution in Fig. 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 3) 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 zero 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 zero 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 (p-value = 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 3, 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, however, is not a valid criterion to evaluate the predictive usefulness of such metrics.

Most importantly, aside from 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, the results might be different. Such a refinement is, however, part of our future research plans.

3.2.3 Multivariate Analysis

The OO design metrics presented in the previous section can be used early in the life cycle (high- or low-level design) 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 "Class origin" in Table 4). This information is available at the end of the design phase when reuse candidates have been iden-

tified in available libraries and the amount of change required can be estimated. Table 4 describes the computed multivariate model. Using such a model for classification, the results shown in Table 5 are obtained by using a classification threshold of $\pi(\text{Fault detection}) = 0.5$, i.e., when $\pi > 0.5$, the class is classified as faulty and, otherwise, as nonfaulty. 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.

TABLE 5
CLASSIFICATION RESULTS WITH OO DESIGN METRICS

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

The figures before parentheses in the right column are the number of classes classified as faulty. The figures within the parentheses are the faults contained in those classes.

We now assess the impact of using such a prediction model by assuming, in order to simplify computations, that inspections of classes are 100 percent 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. To summarize, results show that the studied OO metrics are useful predictors of fault-proneness.

In order to evaluate the predictive accuracy of these OO design metrics, it would be interesting to compare their predictive capability and that of usual code metrics even though they can only be obtained later in the development life cycle. Three code metrics, from the set provided by the Amadeus tool² [2], were selected through a stepwise logistic regression procedure. Table 6 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. It should be noted that other multivariate models can be generated using different metrics provided by Amadeus and yield results of similar accuracy. The model in Table 6 happens to be, however, the one resulting from the use of a standard, stepwise logistic regression analysis procedure.

TABLE 6
MULTIVARIATE ANALYSIS WITH CODE METRICS

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

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 7). In this case, 112 classes

2. The Amadeus tool provides 35 code metrics, e.g., lines of code with and without blank, executable statements, declaration statements, function declaration, function definitions, function calls, cyclomatic complexity, loop statements, maximum class depth and width in a file, number of method declarations, definitions, and average number of methods.

(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) resulting in a significant extra effort. 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
CLASSIFICATION RESULTS BASED ON CODE METRICS
SHOWN IN TABLE 6

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

Table 8 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 one).

TABLE 8
CLASSIFICATION ACCURACIES BASED ON OO AND CODE
METRICS SHOWN IN TABLE 3 AND TABLE 6

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

3.2.4 Threats to Validity

Several threats to the external validity of our study may limit the generalizability of our results:

- The programs developed lie between five KSLOC and 14 KSLOC. Those programs are small as compared to large industry systems. The relationships between the studied OO design metrics and the fault introduction probability are the results of a complex psychological phenomenon and they may look very different in larger programs.
- The conceptual complexity of these systems was rather limited. Again, many different problems may arise in more complex systems.
- It is likely that the study participants were not as well trained and as experienced as average professional programmers. However, this was partially addressed as discussed in Section 2.4.

4 RELATED WORK

In [10], metrics for measuring abstract data type (ADT) cohesion and coupling are proposed and are validated as predictors of faulty ADTs. The main differences and similarities between the work here and [10] are as follows (see Table 9). They did not empirically validate their metrics on OO programs in a context of inheritance but they used a similar validation approach. In both cases, statistical model

were built to predict component (i.e., ADTs and classes, respectively) fault-proneness (i.e., probability of fault detection) by using multiple logistic regression.

In [30], a validation of Chidamber and Kemerer's OO metrics studying the number of changes performed in two commercial systems implemented with an OO dialect of Ada was conducted. They show that Chidamber and Kemerer's OO metrics appeared to be adequate in predicting the frequency of changes across classes during the maintenance phase. They provided a model to predict the number of modifications in a class, which they assume is proportional to change effort and is representative of class maintainability.

The work described in [30] is comparable to our work in the following ways (see Table 9). Li and Henry [30] used the same suite of OO metrics we used. They also used data from products implemented in an OO language which provides multiple inheritance, overloading, and polymorphism. On the other hand, we used the probability of fault detection as the dependent variable of our statistical model. Thus, our goal was to assess whether Chidamber and Kemerer's OO metrics were useful predictors of fault-prone classes. In addition, in [30] (multivariate) least-square linear regression was used to build a predictive model whereas we used logistic regression (i.e., a classification technique for binary dependent variables). The nature of our dependent variable (i.e., (non)occurrence of fault detection) has led us to use logistic regression [25].

TABLE 9
SOME DIFFERENCES AND SIMILARITIES BETWEEN
[10], [30], AND OUR WORK

CRITERIA	VALIDATION WORK		
	Briand et al. [10]	Li and Henry [30]	Our work
Suite of Metrics	ADT Cohesion and Coupling	CK metrics	CK metrics
Type of products	Ada	OO dialect of Ada	C++
Dependent variable	fault occurrence in Ada packages	number of changes in component's	fault occurrence in C++ classes
Statistical technique	logistic regression	least-square regression	logistic regression

5 CONCLUSIONS AND FURTHER WORK

In this study, we collected data about faults found in object-oriented classes. Based on these data, we verified 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, five out of the six Chidamber and Kemerer's OO metrics appear to be useful to predict class fault-proneness during the high- and low-level design phases of the life-cycle. In addition, Chidamber and Kemerer's OO metrics show to be better predictors than the best set of "traditional" code metrics, which can only be collected during later phases of the software development processes.

This empirical validation provides the practitioner with some empirical evidence demonstrating that most of Chidamber and Kemerer's OO metrics can be useful quality

indicators. Furthermore, most of these metrics appear to be complementary indicators which are relatively independent from each other. The results we obtained provide motivation for further investigation and refinement of Chidamber and Kemerer's OO metrics.

Finally, results seem to show that one would likely be able to make inspections of design or code artifacts more efficient if they were driven by models such as the one we built in Section 3.2.3, based on Chidamber and Kemerer's OO metrics. However, how to help focus inspections on error-prone parts in large programs is still an important issue to be further investigated. Our results should be interpreted as maximum possible gains and not as expected gains.

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. Replication should help us achieve the following objectives:
 - 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., single versus multiple inheritance) on different types of defects and rework. In this study, because the data collection process was not fully adequate, we were unable to analyze the relationships of OO design metrics with rework and different defect categories. With regard to rework, we believe that this drawback could be overcome by refining our data collection process to capture the amount of effort spent debugging each class individually. With regard to defect categories, we would need to collect additional information about defect origin (e.g., specification, design, implementation, previous change), defect type (e.g., omission/commission), defect class (e.g., external interface, internal interface, etc.), etc.
 - Investigating the prediction usefulness of Chidamber and Kemerer's OO design metrics with regard to different types of faults, e.g., fault severity. The fault-proneness prediction capabilities of any suite of OO may be different depending on the type of fault used.
- 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 programming language used. Work must be undertaken to validate this suite of OO design metrics across different OO languages, e.g., Ada95, Smalltalk, C++, etc.
- Extending the empirical investigation to other OO metrics proposed in the literature and develop improved metrics, e.g., more language specific, based on more sophisticated hypotheses.

ACKNOWLEDGMENTS

We want to thank (1) Dr. Gianluigi Caldiera for helping us teach the OMT method; (2) A. Belkhelladi, K. El Emam, C. Kemerer, J. Kontio, C. Seaman, F. Shull, and R. Tesoriero for their suggestions that helped improve both the content and the form of this paper; and (3) the students of University of Maryland for their participation on this study. This work will not be possible without the help of Prem Devanbu (AT&T Labs). Finally, we wish to thank the anonymous reviewers and Hausi Muller, the editor of this paper, for their valuable comments. This work was supported, in part, by the National Aeronautics and Space Administration under Grant No. NSG-5123, the National Science Foundation under Grant No. 01-5-24845, Fraunhofer Gesellschaft, UMIACS, and Westinghouse Corporation.

REFERENCES

- [1] F.B. Abreu and R. Carapuça, "Candidate Metrics for Object-Oriented Software within a Taxonomy Framework," *J. System and Software*, vol. 26, no. 1, pp. 87-96, Jan. 1994.
- [2] Amadeus Software Research, *Getting Started With Amadeus*, Amadeus Measurement System, 1994.
- [3] V. Basili and D. Hutchens, "Analyzing a Syntactic Family of Complexity Metrics," *IEEE Trans. Software Eng.*, vol. 9, no. 6, pp. 664-673, June 1982.
- [4] V. Basili, R. Selby, and T.-Y. Philips, "Metric Analysis and Data Validation Across Fortran Projects," *IEEE Trans. Software Eng.*, vol. 9, no. 6, pp. 652-663, June 1983.
- [5] V. Basili, L. Briand, and W. Melo, "Measuring the Impact of Reuse on Software Quality and Productivity," *Comm. ACM*, vol. 39, no. 10, pp. 104-116, Oct. 1996.
- [6] J.M. Bieman and B.-K. Kang, "Cohesion and Reuse in an Object-Oriented System," *Proc. ACM SIGSOFT Symp. Software Reusability*, Seattle, Wash., pp. 259-262, 1995.
- [7] B. Boehm, *Software Eng. Economics*, Prentice-Hall, 1981.
- [8] L. Briand, V. Basili, and C. Hetmanski, "Developing Interpretable Models with Optimized Set Reduction for Identifying High Risk Software Components," *IEEE Trans. Software Eng.*, vol. 19, no. 11, pp. 1,028-1,044, Nov. 1993.
- [9] L. Briand, K. El Emam, and S. Morasca, *Theoretical and Empirical Validation of Software Product Measures*, ISERN Technical Report 95-03, 1995.
- [10] L. Briand, S. Morasca, and V. Basili, *Defining and Validating High-Level Design Metrics*, Technical Report CS-TR-3301, Univ. of Maryland, Dept. of Computer Science, College Park, Md., 1994.
- [11] L. Briand, S. Morasca, and V. Basili, "Property Based Software Engineering Measurement," *IEEE Trans. Software Eng.*, vol. 22, no. 1, pp. 68-86, Jan. 1996.
- [12] I. Brooks, "Object-Oriented Metrics Collection and Evaluation with a Software Process," *Proc. OOPSLA '93 Workshop Processes and Metrics for Object-Oriented Software Development*, Washington, D.C., 1993.
- [13] S.R. Chidamber and C.F. Kemerer, "A Metrics Suite for Object-Oriented Design," *IEEE Trans. Software Eng.*, vol. 20, no. 6, pp. 476-493, June 1994.
- [14] S.R. Chidamber and C.F. Kemerer, "Authors Reply," *IEEE Trans. Software Eng.*, vol. 21, no. 3, p. 265, Mar. 1995.
- [15] N.I. Churcher and M.J. Shepperd, "Comments on 'A Metrics Suite for Object-Oriented Design,'" *IEEE Trans. Software Eng.*, vol. 21, no. 3, pp. 263-265, Mar. 1995.
- [16] S.D. Conte, H.E. Dunsmore, and V.Y. Shen, *Software Eng. Metrics and Models*, Benjamin/Cummings, 1989.
- [17] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood, "The Effect of Inheritance Depth on the Maintainability of Object-Oriented Software," *Empirical Software Eng.: An Int'l J.*, vol. 1, no. 2, Feb. 1996.
- [18] P. Devanbu, "GENOA/GENII—A Customizable Language and Front-End Independent Code Analyzer," *Proc. 14th Int'l Conf. Software Eng.*, Melbourne, Australia, 1992.

- [19] P. Devanbu, S. Karstu, W. Melo, and W. Thomas, "Analytical and Empirical Evaluation of Software Reuse Metrics," *Proc. 18th Int'l Conf. Software Eng.*, pp. 189-199, Berlin, Germany, 1996.
- [20] R.B. Grady, *Practical Software Metrics for Project Management and Process Improvement*, Prentice Hall, 1992.
- [21] W. Harrison, "Using Software Metrics to Allocate Testing Resources," *J. Management Information Systems*, vol. 4, no. 4, pp. 93-105, Apr. 1988.
- [22] W. Harrison, "Software Measurement: A Decision-Process Approach," *Advances in Computers*, vol. 39, pp. 51-105, 1994.
- [23] G. Heller, J. Valett, and M. Wild, *Data Collection Procedure for Software Eng. Laboratory (SEL) Database*, SEL Series, SEL-92-002, 1992.
- [24] M. Hitz and B. Montazeri, "Chidamber and Kemerer's Metrics Suite: A Measurement Theory Perspective," *IEEE Trans. Software Eng.*, vol. 22, no. 4, pp. 267-271, Apr. 1996.
- [25] D. Hosmer and S. Lemeshow, *Applied Logistic Regression*, Wiley-Interscience, 1989.
- [26] N.E. Fenton, *Software Metrics: A Rigorous Approach*, Chapman & Hall, 1991.
- [27] C.M. Judd, E.R. Smith, and L.H. Kidder, *Research Methods in Social Relations*, Harcourt Brace Jovanovich College Publishers, 1991.
- [28] T.M. Khohgoftaar, A.S. Panday, and H.B. More, "A Neural Network Approach for Predicting Software Development Faults," *Proc. Third Int'l IEEE Symp. Software Reliability Eng.*, North Carolina, 1992.
- [29] F. Lanubile and G. Visaggio, "Evaluating Predictive Quality Models Derived from Software Measures: Lessons Learned," to appear in the *J. Software and Systems*; also available as Technical Report CS-TR-3606, Univ. of Maryland, Computer Science Dept., College Park, Md., 1996.
- [30] W. Li and S. Henry, "Object-Oriented Metrics that Predict Maintainability," *J. Systems and Software*, vol. 23, no. 2, pp. 111-122, Feb. 1993.
- [31] F. McGarry, R. Pajersk, G. Page, S. Waligora, V. Basili, and M. Zelkowitz, *Software Process Improvement in the NASA Software Eng. Laboratory*. Carnegie Mellon Univ., Software Eng. Inst., Technical Report CMU/SEL-95-TR-22, Dec. 1994.
- [32] J. Munson and T. Khoshgoftaar, "The Detection of Fault-Prone Programs," *IEEE Trans. Software Eng.*, vol. 18, no. 5, May 1992.
- [33] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.
- [34] R. Selby and A. Porter, "Learning from Examples: Generation and Evaluation of Decision Trees for Software Resource Analysis," *IEEE Trans. Software Eng.*, vol. 14, no. 2, pp. 1,743-1,747, Feb. 1988.
- [35] N. Schneidewind, "Methodology for Validating Software Metrics," *IEEE Trans. Software Eng.*, vol. 18, no. 5, pp. 410-422, May 1992.
- [36] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley Series in Computer Science, second edition, 1991.
- [37] D.A. Young, *Object-Oriented Programming with C++ and OSF/MOTIF*, Prentice Hall, 1992.



Victor R. Basili is a professor in the Institute for Advanced Computer Studies and the Computer Science Department at the University of Maryland, College Park, Maryland, where he has served as chairman for six years.

He is one of the founders and principals in the Software Engineering Laboratory, a joint venture between NASA Goddard Space Flight Center, the University of Maryland, and Computer Sciences Corporation, established in 1976. He is a recipient of the first Process Improvement

Achievement Award by the IEEE and the SEL, and the NASA Group Achievement Award.

He has served as editor-in-chief for the *IEEE Transactions on Software Engineering*; general chair of the 15th International Conference on Software Engineering in 1993, in Baltimore, Maryland; program chair for the Sixth International Conference on Software Engineering in 1982, in Japan; and general/program chair for several other conferences. He is an IEEE fellow and member of the IEEE Computer Society and Golden Core.



Lionel C. Briand received the BS degree in geophysics and the MS degree in computer science from the University of Paris VI, France, in 1985 and 1987, respectively. He received the PhD degree, with high honors, in computer science from the University of Paris XI, France, in 1994. Dr. Briand started his career as a software engineer at CISI Ingénierie, France. Between 1989 and 1994, he was a research scientist at the NASA Software Engineering Laboratory, a research consortium: NASA Goddard Space Flight Center, University of Maryland, and Computer Science Corporation. He then held the position of lead researcher of the software engineering group at CRIM, the Computer Research Institute of Montreal, Canada. He is currently the head of the Quality and Process Engineering Department at the Fraunhofer Institute for Experimental Software Engineering, an industry-oriented research center located in Rheinland-Pfalz, Germany. His current research interests and industrial activities include measurement and modeling of software development products and processes, software quality assurance, domain-specific architectures, reuse, and reengineering.



Walcélio L. Melo received the following degrees in computer science: BSc in 1983 from the University of Brasilia, Brazil; MSc in 1988 from the Federal University of Rio Grande do Sul, Brazil; and a PhD in 1993 from Joseph Fourier University, Grenoble, France. He is now the lead researcher in software engineering at the Centre de Recherche Informatique de Montréal (CRIM), Canada. At CRIM, he develops research and technology transfer projects related to software measurement, software reuse, object-oriented technology, software architecture, and software process modeling. He is also an adjunct professor at the McGill University School of Computer Science and a former member of the NASA Software Engineering Laboratory. From 1983 to 1989, he worked as a system analyst for the Brazilian Federal Bureau of Data Processing, where he developed several business-oriented software applications. From 1989 to 1993, he developed research related to software process programming languages at the Software Engineering Laboratory of Grenoble, France. From 1994 to 1996, he worked as a research associate at the University of Maryland Institute for Advanced Computer Studies, where he developed research projects related to software maintenance, software metrics, and object-oriented technology. He is a member of the IEEE Computer Society.