

An Empirical Evaluation of Object Oriented Metrics in Industrial Setting[†]

Giovanni Denaro
Università di Milano Bicocca
Dip. di Informatica,
Sistemistica e Comunicazione
I-20126, Milano, Italy
denaro@disco.unimib.it

Luigi Lavazza
CEFRIEL and Politecnico di
Milano
Dip. di Elettronica e
Informazione
I-20133, Milano, Italy
lavazza@elet.polimi.it

Mauro Pezzè
Università di Milano Bicocca
Dip. di Informatica,
Sistemistica e Comunicazione
I-20126, Milano, Italy
pezze@disco.unimib.it

1. INTRODUCTION

Advances in distributed object technologies (e.g., the Common Object Request Broker Architecture [15] and the Enterprise Java Bean Specification [19]) dramatically impact the development process of distributed software applications. In particular, time for providing new distributed services is decreasing because applications are not built from scratch any longer. Rather, they are developed based on pre-existing middle tier software (middleware) and integrate components and services provided off-the-shelf by third parties [9]. The increasing demand for rapid provision of new products entails rigid constraints on the activities of quality assurance for this class of applications. It becomes crucial to optimize the allocation of resources for testing and analysis to meet the required quality goals, while reducing time-to-market.

Measuring the fault-proneness of the software may facilitate the allocation of resources for testing and analysis. If the distribution of faults in the software can be accurately estimated in advance, resources can be allocated accordingly, i.e., more resources to the more fault-prone parts of the software. For example, in the case of code inspection, more thorough inspection sessions could be scheduled for the more fault-prone modules. Although fault-proneness cannot be directly measured, it can be estimated based on other measurable attributes of the software, based on expected correlations between such attributes and fault-proneness. Many software metrics have been proposed for this purpose (e.g., [14, 10, 20]). However, the best predictors of fault-proneness may vary according to the class of applications and the target application domain, as demonstrated by many empirical studies [13, 18, 16, 17, 7, 6].

In the nineties, researchers started investigating software metrics to capture the specific complexity of object-oriented systems [5, 11, 4, 2]. Object-Oriented (OO) metrics capture characteristics of class hierarchies, of the internal cohesion

of classes and of the degree of coupling between different classes. An preliminary set of empirical studies support the hypothesis that OO metrics are better related to external attributes of object-oriented systems, such as fault-proneness and maintainability, than traditional metrics [1, 8, 3]. However, these studies have been conducted on small applications, containing less than 100 classes, which may not adequately represent large-scale industrial systems.

This paper reports an empirical study on a large object-oriented industrial system for telecommunications. The target software consists of more than 2,000,000 lines of code organized in 3,344 modules, containing one or more C++ classes each. We analyzed the relationships between the OO metrics defined by Chidamber and Kemerer in [5] and fault-proneness across three different versions of this target application. The result of the experiment suggests that OO metrics do not outperform traditional metrics as fault-proneness predictors for large industrial systems: in our experiments, none of the experimented OO metrics appears to be a better predictor of fault-proneness than lines of code (LOC).

2. EMPIRICAL STUDY

The empirical study was conducted on an industrial telecommunication application. For confidentiality reasons, we cannot disclose neither functional details nor the producer of this application. In what follows, we refer to this application as RC.

2.1 Target Data

We collected data from three different versions of RC, referred to as RCx.01, RCx.02 and RCx+1.0, respectively. Such versions represent the evolution of the same software over time. At the time of this writing, all three versions operate in the field and all of them are maintained by the company as branching versions. All three versions are significantly different from each other in terms of offered functionality.

Each version of RC consists of more than 2 million lines of C++ code. The set of source files common to the three versions (common core files) accounts for 93.3% of the files in RCx.01, 92.9% in RCx.02 and 86.5% in RCx+1.0. This suggests that the evolution of RC largely favored modification and adaptation of available code over implementation of new functionality in separate modules. To the end of analyzing

[†] This work has been partially founded by the Italian Government in the context of the QUACK project. (QUACK: A Platform for the Quality of New Generation Integrated Embedded Systems.)

the trends of faults and metrics across the three version of RC, only the common core files have been considered in our analysis.

We considered header and implementation source files with the same base name (for example, the file `x.hpp` and `x.cpp`) as a single software module. This is consistent with the common C++ programming practice of separating interface definitions and method implementations corresponding to the same class in header and implementation file. The common core of RC consists of 3,344 modules. Among these 2,937 contain exactly one class and less than 100 contain more than 2 classes, confirming the general validity of the above assumption.

A database stores faultiness data for the three version of RC. These data have been collected during system and integration testing. Whenever a fault was revealed, references to the version under test and the corresponding faulty code were traced in the database. Thus, we have been able to count the number of faults discovered for versions and modules of RC: For version `x.01`, we have 2,087 faults distributed among 600 faulty modules; For version `x.02`, we have 1,962 faults distributed among 590 faulty modules; For version `x+1.0`, we have 12,600 faults distributed among 1,800 faulty modules.

2.2 OO Metrics and Hypotheses

Our analysis focus on the set of OO metrics defined by Chidamber and Kemerer in [5] (C&K OO suite). Here, we briefly recall the semantics of these metrics:

WMC, weighted methods per class. This metric measures the complexity of a class. Its value is computed as the sum of the complexity of each individual method of the class. For the sake of simplicity, we consider all methods of a class to be equally complex. Thus the value of WMC represents the number of methods of the class.

DIT, depth of inheritance tree. In our study, this metric measures the number of the ancestors of a class.

NOC, number of children. This metric measures the number of direct descendants of a class.

RFC, response for a class. This metric measures the number of methods that can be potentially executed in response to a message received by an object of a class.

LCOM, lack of cohesion in methods. This metric measures the number of pairs of methods of a class that do not share any instance variables, minus the pairs of methods that do. The value is zero when the subtraction yields a negative result.

CBO, coupling between objects. This metric measures the number of other classes used by a class. A class uses another class if one of its member uses a member of the other class.

For each module in the dataset we measured both the metrics of the C&K OO metrics and the size in terms of LOC. We have been not able to measure CBO and LCOM, because the available tools failed in computing these two metrics due to the very large dimension of the application.

2.3 Data Analysis Methodology

We evaluated the impact of each single metric on the fault-proneness of the software modules, and we used multivariate statistical models for understanding whether the conjunct use of set of metrics can explain fault-proneness better than a single metric alone.

Studying the Impact of Single Metrics

One important application of fault-proneness models is to help focus analysis and testing activities. For example, we would like to be able to analyze or test the most faulty parts of the software under verification earlier and more deeply than the rest. The better a predictor helps recognize the most faulty part of a software system, the more it helps focus analysis and testing.

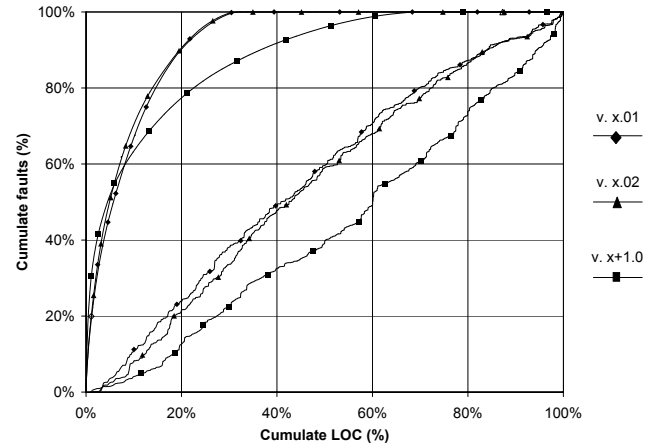


Figure 1: LOC as fault-proneness predictor

The graph in Figure 1 plots the cumulate percentage of faults versus the cumulate percentage of lines of code, for the three versions of RC¹, when modules are sorted according to different criteria. The three upper lines correspond to the modules sorted by decreasing density of known faults for the three versions. Allocating resources according in this order would optimize testing and analysis activities. Unfortunately, this order is known too late for planning analysis and testing activities. The three lower lines correspond to the modules sorted by decreasing size (LOC). The figure confirms that the intuitive idea *the bigger, the more faulty*, is generally valid for RC modules, even though the fault density increases only linearly. It also confirms that to use LOC as fault predictor allows to stay on the *safe-side*, avoiding the risk of focusing first on big amounts of non-faulty modules. In fact, planning testing and analysis according to the size of software modules is a common practice in industry.

We used LOC for benchmarking OO metrics as fault-proneness predictors, i.e., we consider a metric as a good fault-proneness predictor when it outperforms LOC in anticipating the most fault dense modules of the system.

¹The version corresponding to each line is identified by the particular symbol drawn in random points of the line itself, according to the association defined by the legend on the right side of the figure.

Multivariate Regression Analysis

For multivariate analysis we use logistic regression. A logistic regression model estimates the probability that an object belongs to a specific class (dependent variable), based on the values of some quantified attributes of the object (explanatory variables). We classified as faulty the modules with at least one fault in the dataset and used this class as dependent variable. We used the software metrics as explanatory variables. Thus, our models estimate fault-proneness as the probability of finding a fault in a software module, based on given sets of metrics. For example, let M_1, \dots, M_n be a set of n metrics measured on the module M , then according to the logistic regression equation,

$$p(M \text{ is faulty}) = \frac{e^{C_0 + C_1 M_1 + \dots + C_n M_n}}{1 + e^{C_0 + C_1 M_1 + \dots + C_n M_n}}.$$

Where the coefficients C_0, C_1, \dots, C_n are computed by maximizing the likelihood of the model on the set of available observations. Full details on logistic regression can be found in [12].

We used logistic regression to investigate the combined impact of pairs of metrics on fault proneness. We then compared the fault-proneness indicators provided by the models and LOC, with respect to the ability of anticipating the most fault dense modules of the system.

2.4 Results

Figure 2 shows the impact of single OO metrics on fault-proneness for RCx.01. The two thick lines represent the impact of ordering the modules by decreasing fault density (the optimal order) and by decreasing LOC (the baseline order). The four thin lines represent the modules sorted according to the decreasing order provided by the four investigate OO metrics. The diagram indicates that none of the OO metrics performs better than LOC in anticipating fault-proneness. Thus, in this case there is no advantage in using any OO metric instead of LOC as fault-proneness indicator. Results for RCx.02 and RCx+1.0 show analogous trends.

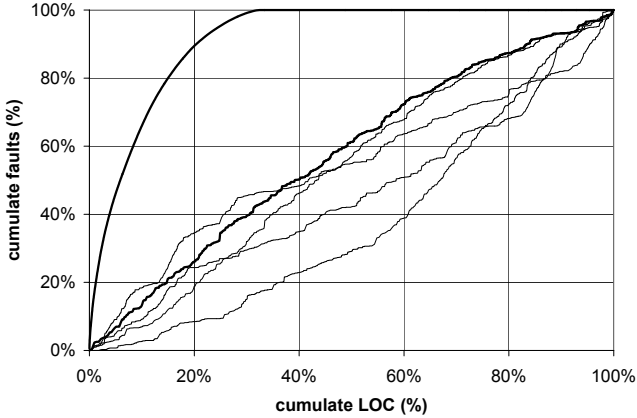


Figure 2: Impact of OO metrics for RCx.01

We then performed multivariate analysis using all the possible pairs of metrics in the dataset as independent variables. We produced 10 logistic regression models. We excluded

four models, because they were not statistically significant, according to the goodness indicators provided by the logistic regression procedure. Figure 3 shows the cumulative fault density according to the fault-proneness indicators provided by the logistic regression models (thin lines), in comparison with LOC (thick line) for RCx.02. None of the models provides a better fault-proneness indicator than LOC. Moreover, models that do not include LOC among the explanatory variables perform sensibly worse than LOC as fault-proneness indicators. Thus in this case, multivariate models do not represent any advantage over LOC, as fault-proneness indicator. Results for RCx.01 and RCx+1.0 show analogous trends.

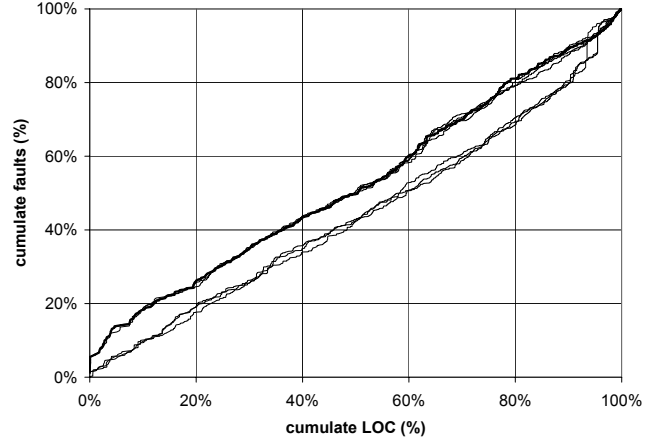


Figure 3: Quality of LR models for RC6.02

3. THREATS TO VALIDITY

We identify some threats that may limit the generalization of the results of this study.

RC was born as evolution of a legacy C application previously maintained by the producer company. This raises the question whether we analyzed an application that adequately represents object-oriented software. If the internal structure of RC was inherited from the procedural program, this could explain the low significance of OO metrics in representing the complexity of the software. Although the organization of the code (mainly articulated in single class modules with separated interface and implementation) supports the assumptions of object-orientation, we cannot definitely exclude this threat.

We also notice that in the analyzed data the amount of non-faulty modules largely outstands faulty modules. For example, faulty modules represent less than 20% of the dataset in RCx.01 and RCx.02. This could induce a bias in the logistic regression models: being non-faulty the most likely category in the dataset, models tend to predict low probabilities of fault-proneness such to stay on the safe-side. We plan to execute new experiments excluding the non-faulty modules from the building procedure, but we are now performing a preliminary study to understand more in detail the type of conclusions that we can draw in this latter case.

Finally, we cannot ignore the absence of LCOM and CBO in the set of the analyzed metrics. These metrics measure the degree of coupling between classes that is expected to

be an important dimension of complexity. Thus, our results cannot be generalized to all the metrics of the C&K OO suite.

4. CONCLUSIONS

This paper reported an empirical study of the relationships between OO metrics and fault-proneness for an industrial application for telecommunications. The results of this study challenges the hypothesis that OO metrics are better predictors of fault-proneness than traditional software metrics. To the best of our knowledge, our study represents the first empirical evaluation of OO metrics on an software system of industrial size.

5. ACKNOWLEDGMENTS

The authors would like to thank Simone Bordin for his contribution to the collection of data for the empirical study presented in this paper.

6. REFERENCES

- [1] V. Basili, L. Briand, and W. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, Oct. 1996.
- [2] L. Briand, J. Daly, and J. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, Jan./Feb. 1999.
- [3] L. Briand, J. Wüst, S. Ikononovski, and H. Lounis. Investigating quality factors in object-oriented designs: an industrial case study. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE'99)*, pages 345–355, New York, May 1999. Association for Computing Machinery.
- [4] S. Chidamber, D. Darcy, and C. Kemerer. Managerial use of metrics for object-oriented software: An exploratory analysis. *IEEE Transactions on Software Engineering*, 24(8):629–639, Aug. 1998.
- [5] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.
- [6] G. Denaro, S. Morasca, and M. Pezzè. Towards industrially relevant fault-proneness models. *International Journal of Software Engineering and Knowledge Engineering*, 13(4):395–418, Aug. 2003.
- [7] G. Denaro and M. Pezzè. An empirical evaluation of fault-proneness models. In *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*, pages 241–254, New York, 2002. ACM Press.
- [8] K. E. Emam, W. Melo, and J. Machado. The prediction of faulty classes using object-oriented design metrics. *The Journal of Systems and Software*, 56(1):63–75, 2001.
- [9] W. Emmerich. Distributed component technologies and their software engineering implications. In *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*, pages 537–546. ACM Press, 2002.
- [10] M. Halstead. *Elements of Software Science*. Elsevier North-Holland, New York, 1 edition, 1977.
- [11] M. Hitz and B. Montazeri. Chidamber and Kemerer's metrics suite: A measurement theory perspective. *IEEE Transactions on Software Engineering*, 22(4):267–271, Apr. 1996.
- [12] D. Hosmer and S. Lemeshow. *Applied Logistic Regression*. Wiley-Interscience, 1989.
- [13] T. Khoshgoftaar, E. Allen, K. Kalaichelvan, and N. Goel. Early quality prediction: a case study in telecommunications. *IEEE Software*, 13(1):65–71, Jan. 1996.
- [14] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, Dec. 1976.
- [15] P. Merle. CORBA 3.0 new components chapters. Technical report, TC Document ptc/2001-11-03, Object Management Group, 2001.
- [16] S. Morasca and G. Ruhe. A hybrid approach to analyze empirical software engineering data and its application to predict module fault-proneness in maintenance. *The Journal of Systems and Software*, 53(3):225–237, Sept. 2000.
- [17] N. Ohlsson and H. Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Transactions on Software Engineering*, 22(12):886–894, Dec. 1996.
- [18] R. Selby and A. Porter. Learning from examples: Generation and evaluation of decision trees for software resource analysis. *IEEE Transactions on Software Engineering*, 14(12):1743–1757, Dec. 1988.
- [19] B. Shannon. Java 2 platform enterprise edition specification, 1.4 - proposed final draft 2. Technical report, Sun Microsystems, Inc., 2002.
- [20] M. Woodward, M. Hennell, and D. Hedley. A measure of control flow complexity in program text. *IEEE Transactions on Software Engineering*, 5(1):45–50, Jan. 1979.