Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction

Tibor Gyimóthy, Rudolf Ferenc, and István Siket

Abstract—Open source software systems are becoming increasingly important these days. Many companies are investing in open source projects and lots of them are also using such software in their own work. But, because open source software is often developed with a different management style than the industrial ones, the quality and reliability of the code needs to be studied. Hence, the characteristics of the source code of these projects need to be measured to obtain more information about it. This paper describes how we calculated the object-oriented metrics given by Chidamber and Kemerer to illustrate how fault-proneness detection of the source code of the open source Web and e-mail suite called Mozilla can be carried out. We checked the values obtained against the number of bugs found in its bug database—called Bugzilla—using regression and machine learning methods to validate the usefulness of these metrics for fault-proneness prediction. We also compared the metrics of several versions of Mozilla to see how the predicted fault-proneness of the software system changed during its development cycle.

Index Terms—Fact extraction, metrics validation, reverse engineering, open source software, fault-proneness detection, Mozilla, Bugzilla, C++, compiler wrapping, Columbus.

1 Introduction

Open source software systems are becoming evermore important these days. Many large companies are investing in open source projects and many of them are also using this kind of software in their own work. As a consequence, many of these projects are being developed rapidly and are quickly becoming very large. But, because open source software is usually developed outside companies—mostly by volunteers—and the development methodology employed is quite different from the usual methods applied in commercial software development, the quality and reliability of the code needs to be investigated. Various kinds of code measurements can be quite helpful in obtaining information about the quality and fault-proneness of the code.

In this paper, we describe how we calculated and validated the object-oriented metrics suite given by Chidamber and Kemerer [8] for fault-proneness detection from the source code of the well-known open source Web and e-mail suite called *Mozilla* [17], [20]. This metrics suite has already been validated by Basili et al. [1] and Briand et al. [3] [4], [5] for this purpose and similar results were also presented by Fioravanti and Nesi [13] and Yu et al. [22], but our aim was to supplement their work with measurements obtained from a real-size open source software system (one with over a million lines of code). Besides using well-known statistical methods for analysis (logistic and linear regression), we also employed machine learning techniques (decision trees and neural networks) to predict the fault-proneness of the code.

In order to perform our analyses we collected the number of bugs found and corrected in each class of the system from the *Bugzilla* [6] database, which contains all the bugs that arose during the development of Mozilla.

We analyzed the source code of Mozilla with the help of our reverse engineering framework called *Columbus* [9], [10], which we also used to calculate the required metrics. The Columbus framework has been further improved recently with a so-called *compiler wrapping* method [11] which allows us to automatically analyze and extract information from practically any software system that compiles with GCC on the GNU/Linux platform (the idea is also applicable to other compilers and operating systems). Moreover, we can do this without modifying any of the source code or makefiles. The details of this method will be given later on. It should be mentioned here that we performed a full analysis of the source code of Mozilla and the results obtained can be used for any re and reverse engineering task like architecture recovery and visualization. Here, we used them only for calculating metrics, but their use is by no means limited to this task.

The main contributions of this paper are summarized in the following: First, we performed the analysis of, and calculated metrics from, an open source real-word software system. Previous studies mostly analyzed smaller software packages and, when the size was similar, it was proprietary software and no details were published. Analyzing open source software is interesting because it is not being developed with the usual standard methodologies. Second, besides the common statistical methods, we applied machine learning techniques to predict the fault-proneness of the code. Third, we analyzed the changes in the fault-proneness of Mozilla through seven versions using our results.

• *The authors are with the Department of Software Engineering at the University of Szeged, H-6720 Szeged, Árpád tér 2, Hungary. E-mail: {gyimi, ferenc, siket}@inf.u-szeged.hu.*

Manuscript received 25 Mar. 2005; revised 9 Sept. 2005; accepted 13 Sept. 2005; published online 3 Nov. 2005.
Recommended for acceptance by Harman, Korel, and Linos.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0080-0305.

We will proceed as follows: In the next section, we will describe the metrics suite presented by Chidamber and Kemerer [8] and validated by Basili et al. [1], and also state our hypotheses about their expected connection with fault-proneness. In Section 3, we will present our methods of fact extraction from the C++ source code of Mozilla and its bug database called Bugzilla. Next, in Section 4, we will compare the results of some basic statistical calculations with those published by Basili et al. [1]. Then, in Section 5, we will present the results of further statistical analyses and compare them with those found in other articles. We will also describe the results of our experiments in applying machine learning methods to fault-proneness prediction. In Section 6, we will study the changes of the metrics over different releases of the Mozilla internet suite. Afterward, in Section 7, we will discuss some works similar to ours. Finally, in Section 8, we will present some conclusions and then outline directions for future work.

2 Studied Metrics

In this section, we define the eight metrics that we investigated.[1] Six of these metrics were first presented by Chidamber and Kemerer [8], but they were modified slightly by Basili et al. [1] to reflect the special features of the C++ language. We also used these modified metrics and added one more object-oriented metric (LCOMN) and the well-known lines of code metric (LOC) because we were also interested in comparing object-oriented metrics with the traditional code-size metric. The metrics we investigated were the following:

- **WMC (Weighted Methods per Class)**. The WMC is the number of methods defined in each class. More precisely, WMC is defined as being the number of all member functions and operators defined in each class. However, "friend" operators (C++ specific constructs) are not counted. Member functions and operators inherited from the ancestors of a class are also not counted.
- **DIT (Depth of Inheritance Tree)**. The DIT measures the number of ancestors of a class.
- **RFC (Response For a Class)**. This is the number of methods that can potentially be executed in response to a message being received by an object of that class. The RFC is the number of C++ functions directly invoked by member functions or operators of a C++ class.
- **NOC (Number Of Children)**. The NOC is the number of direct descendants for each class.
- **CBO (Coupling Between Object classes)**. A class is coupled to another one if it uses its member functions and/or instance variables. The CBO gives the number of classes to which a given class is coupled.
- **LCOM (Lack of Cohesion on Methods)**. 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 this subtraction is negative.
- **LCOMN (Lack of Cohesion on Methods allowing Negative value)**. The LCOMN is calculated in the same way as LCOM except that its value is not set to zero when the subtraction is negative.
- **LOC (Lines Of Code)**. The LOC of a class is the number of all nonempty, noncomment lines of the body of the class and all of its methods.

Basili et al. [1] drew up six hypotheses (one for each metric) that represented the expected connection between the metrics and the fault-proneness of the code. They tested these hypotheses and found that some of the metrics were very good predictors, while others were not. We adopted five of their six hypotheses but we modified their NOC hypothesis because, in the case of NOC, their observed trend was contrary to that stated by their NOC hypothesis. We set up our own hypotheses for the two additional metrics (LCOMN and LOC). Our hypotheses[2] were the following (the corresponding null hypotheses are given in parentheses):

- **WMC hypothesis**. A class with more member functions than its peers is more fault-prone than they are. (Null hypothesis: A class with more member functions than its peers is no more fault-prone than they are.)
- **DIT hypothesis**. A class located lower in a class inheritance lattice than its peers is more fault-prone than they are. (Null hypothesis: A class located lower in a class inheritance lattice than its peers is no more fault-prone than they are.)
- **RFC hypothesis**. A class with larger response sets than its peers is more fault-prone than they are. (Null hypothesis: A class with larger response sets than its peers is no more fault-prone than they are.)
- **NOC hypothesis**. A class with a larger number of children than its peers is less fault-prone than they are. (Null hypothesis: A class with a larger number of children than its peers is no less fault-prone than they are.)
- **CBO hypothesis**. A class which is more coupled than its peers is more fault-prone than they are. (Null hypothesis: A class which is more coupled than its peers is no more fault-prone than they are.)
- **LCOM and LCOMN hypotheses**. A class with lower cohesion than its peers is more fault-prone than they are. (Null hypothesis: A class with lower cohesion than its peers is no more fault-prone than they are.)
- **LOC hypothesis**. A class with a larger number of code lines than its peers is more fault-prone than they are. (Null hypothesis: A class with a larger number of code lines than its peers is no more fault-prone than they are.)

In the following, we will test the null hypotheses on Mozilla and either accept the null hypotheses or reject them

1. We should mention here that our *Columbus* reverse engineering framework is able to compute over 80 additional metrics.

2. These are considered as the alternative hypotheses of the null hypotheses.

(in this case, we will accept the corresponding alternative hypotheses), but first we will describe how we calculated the metrics from the source code of Mozilla and how we extracted the bugs that arose during its development from its bug database Bugzilla.

3 Fact Extraction from Mozilla and Bugzilla

Fact extraction from a real-world system's source code is a very difficult task. Such software products usually contain several configurations to be able to compile themselves on different platforms (hardware and/or operating systems). Moreover, properly analyzing the source code written in such a complex programming language as C++ introduces additional difficulties.

In previous work [11], we showed that there are several important steps that have to be performed to obtain the information needed about the source code. These steps include the acquisition of configuration information, the analysis of the source files with analyzer tools, the creation of some kind of representation for the extracted information, the merging of these representations, and various further calculations performed on this merged representation to put the collected data into a useable form. In our case, this last step means the calculation of the necessary metrics for each class in the software system being analyzed. In this section, we will briefly describe how we managed to perform these tasks.

Bugzilla stores the bugs in an SQL database, so extracting the bugs from it was a straightforward task. The real challenge, however, was to associate these bugs with the classes found in the source code. We will describe how we achieved this later in this section. But first, we will briefly describe the toolset we developed to support the fact extraction process.

3.1 The Columbus Framework

Columbus [9], [10] is a reverse engineering framework that has been developed in cooperation between the University of Szeged, the Nokia Research Center, and FrontEndART [14]. The main motivation behind developing the Columbus framework was to create a toolset which supports fact extraction in general and provides a common interface for other reverse engineering tasks as well.

The framework contains all the necessary components to be able to perform the analysis of arbitrary C/C++ source code and to present the extracted information in any desired form. In this study, we used the compiler wrapper module of Columbus to perform the extraction of facts from Mozilla's source code. This will be described in the following.

3.2 Compiler Wrapping

The source code of a software system is usually divided into several files and these files are arranged into folders and subfolders. Furthermore, different preprocessing configurations may apply to them. In the case of Mozilla, the information on how these files are related to each other and what settings apply to them are stored in *makefile*s (used by the *make* tool). An important issue that we addressed was to *not change* anything in the subject system (not even the makefiles). The technique described below successfully deals with this issue. It was tested with the GCC compiler in the GNU/Linux environment, but the idea is applicable as well to other compilers and operating systems.

The make tool and the makefiles represent a powerful pair for configuring and building software systems. Makefiles may contain not just the references to files to be compiled and their settings, but also various commands like those invoking external tools. A typical example is when the source file to be compiled is generated on-the-fly from IDL descriptions by another tool. These powerful possibilities are problematic for reverse engineers because every action in the makefile must somehow be simulated in the reverse engineering tool. This may be extremely hard or even impossible to do in some cases.

We solved this problem by *wrapping* the compiler. This means that we temporarily hide the original compiler by a wrapper toolset. This toolset consists basically of several scripts. Among the scripts, there is a key one that is responsible for hiding the original compiler by changing the PATH environment variable. Actually, all the user has to do is to run this script. The script inserts the path of the folder in which the other scripts can be found at the beginning of the PATH environment variable. The names of the other scripts correspond to the executable files of the compiler (for instance *g++*, *gcc*, *ar*, and *ld*). Afterwards, if the original compiler is invoked, one of our wrapper scripts will start instead.

The scripts first execute the original compiler tool (e.g., *g++*) with the same parameters and in the same environment so the output remains the same; hence, the user will not notice any difference. After calling the original compiler, the scripts also call our corresponding analyzer tool which creates a file containing the extracted information. These files are merged together by our linker tool invoked by the linker wrapper script in parallel with the original linker program. At the end of the analysis, the results consist of several linked files which contain the extracted information for each component/subsystem (e.g., shared libraries and executables) of the analyzed system.

As a last step, these linked files are merged together by executing only one script to produce the full representation of the system. This so-called superlinked file was used to calculate the needed metrics. The result of this calculation is a table containing the classes found in the source code along with their position/interval in the code (path and line information) and the eight calculated metrics. We repeated this whole process for all the seven examined Mozilla versions (1.0-1.6) and created a table for each version.

In the following, we will describe how we extended these tables by associating the bugs extracted from the Bugzilla database with the classes found in the source code.

3.3 Mining Bugs from Bugzilla

To be able to perform our analyses, we also needed to collect the number of bugs found and corrected in each class of the system in all of the analyzed versions. At the time of writing, the actual version of Mozilla was 1.7 and we collected all the bugs beginning from version 1.0 (released in June, 2002). We gathered the bugs in the following way.

TABLE 1
Filtering Steps of the Bug Reports

Filtering aspect	No. of bugs
All bugs	256,613
Product	231,021
Fixed bugs	57,151
Patch	22,553
Date (after June, 2002)	9,539
Date (before May, 2004)	8,936

The developer community of Mozilla kindly provided us with the full Bugzilla [6] database, which contained all the bugs reported since the beginning of the development of the software.

The Bugzilla database contained 256,613 different bug entries, but not all of these were of interest to us (see Table 1). First, we removed the bugs reported for accessory software like Bonsai and Tinderbox. After this step, 231,021 bugs remained. Next, we considered only those bugs that were fixed (57,151 remained) and the fix of the bug contained a patch file that corrected the bug (22,553 remained). Version 1.0 of Mozilla was released in June 2002, so we filtered out those bugs which were fixed before this date (9,539 remained). Similarly, we filtered out those bugs which were reported after Mozilla 1.7 was released in June 2004. We found 8,936 different bug entries which fulfilled all these requirements.

By analyzing the patch files, we assigned the bugs to an interval in the code. A patch file contained the name of the changed file and it described how many lines were deleted starting from a given line number and how many lines were inserted at a given line number. From these four numbers, we determined an interval of changes in the file for localizing the bug.

We wanted to associate the bugs to classes in concrete release versions, but the database did not explicitly state which version the patch file was applied to. Fortunately, it contained the date when the bug was reported and also the date when it was fixed. So, we considered a bug to be present in the actual release at the time when it was reported and in all subsequent releases up to the date of the fix (see Fig. 1). Regarding the bugs reported before version 1.0, we took into account only those which were fixed after version 1.0.

Next, we had to associate the bugs to the classes found in the source code (see Section 3.2). We did this one at a time for each bug in each affected version. Concretely, we examined the bugs one by one and, in each version concerned, we searched for a class whose interval in the source code overlapped the interval of the bug. If we found such a class, then we increased the number of bugs

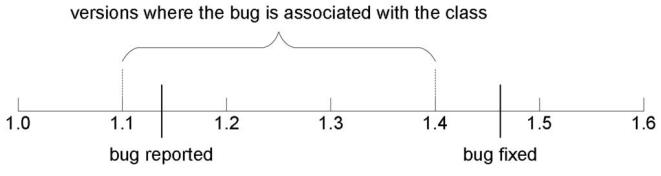

Fig. 1. Mozilla versions where a bug is associated with a class.

TABLE 2
Distribution of the Bugs in Version 1.6

No. of classes	%	No. of associated bugs
1,850	57.96	0
666	20.86	1
236	7.39	2
136	4.26	3
78	2.44	4
49	1.54	5
44	1.38	6
26	0.81	7
28	0.88	8
18	0.56	9
14	0.44	10
4	0.13	11
7	0.22	12
2	0.06	13
5	0.16	14
9	0.28	15-19
8	0.25	20-24
8	0.25	25-29
4	0.13	30-34
3,192	100.00	3,961

in that class. If the bugfix changed more than one class, then the bug was associated with all these classes. With this method, we extended the tables obtained in the previous section with a new column containing the number of bugs for each class.

This way, we could associate 4,429 different bugs to classes, which is about half of the number of bugs extracted from Bugzilla (8,936—see Table 1). If we take into account that we associated bugs only with classes (which are object-oriented constructs, written in C++) and that, in Mozilla, there are many C source files as well (more than 1,500), then this result is acceptable.

As a last step, we filtered out those classes which were generated on-the-fly during compilation because there could be no bugs associated with them. We also filtered out all bug-free classes which existed in all seven analyzed versions of Mozilla and where none of the metrics changed. This way, we arrived at 3,192 remaining classes (out of 3,677 extracted classes) for Mozilla version 1.6. We used these classes in our further analyses described in the next sections. We summarized our findings in Table 2. More than half of the classes (1,850 of them) contained no bugs, and about one fifth (666 of them) contained only one bug.

Table 3 shows the Pearson's linear correlations (R^2: Coefficient of determination) between the numbers of

TABLE 3
Correlations between the Number of Bugs Associated to the Different Analyzed Versions

ver.	1.0	1.1	1.2	1.3	1.4	1.5	1.6
1.0	1.00	0.89	0.84	0.76	0.73	0.69	0.70
1.1		1.00	0.90	0.81	0.77	0.74	0.73
1.2			1.00	0.85	0.81	0.77	0.73
1.3				1.00	0.83	0.75	0.74
1.4					1.00	0.84	0.76
1.5						1.00	0.80
1.6							1.00

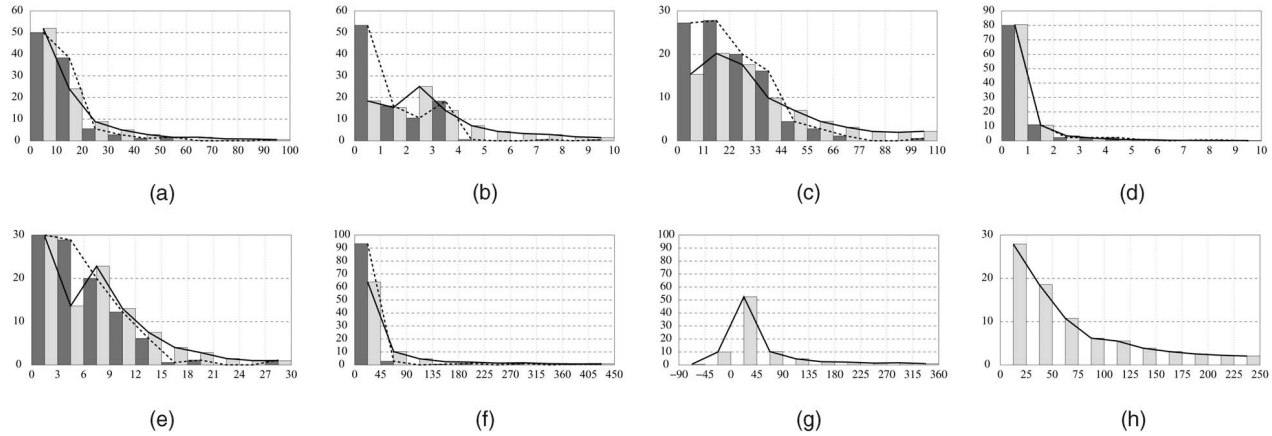

Fig. 2. Distribution of the metrics. The X axes represent the values of the metrics. The Y axes represent the percentage of the number of classes having the corresponding metric value. The darker columns (and dashed lines) represent the original values of Basili et al. [1], while the brighter ones (and solid lines) represent the values calculated form Mozilla 1.6.

bugs associated with the classes in the versions we analyzed. (For calculating correlation between two versions, we took into account classes which exist in both versions and calculated the correlation between the numbers of bugs associated with these classes.) All correlations are significant (p-value < 0.001) and they are high, which means there is strong linear association between the bugs in the different versions. This is why we chose to use only the results for version 1.6 in our further analyses. (We should mention here that, at the end of our analyses, we tested these results on version 1.0 as well. See Section 5 for details.)

Versions In the following section, we will compare some basic statistical data and the distribution of the metrics collected from Mozilla 1.6 with the results published by Basili et al. [1].

4 Comparison of the Metrics

Basili et al. [1] studied object-oriented systems written by students in C/C++. They carried out an experiment in which they set up eight project groups each consisting of three students. Each group had the same task—to develop a small/medium-sized software system. Since all the necessary documentation (for instance, reports about faults and their fixes) was available, they could search for relationships between the fault density and metrics. They used the same Chidamber and Kemerer metrics suite [8] as we did and analyzed the distribution of the metrics and also the correlations between them. Afterwards, they employed logistic regression—a standard technique based on maximum likelihood estimation—to analyze the relationship between metrics and the fault-proneness of classes.

Now, we will compare the metrics obtained from Mozilla 1.6 with those presented by Basili et al. [1]. Fig. 2 shows a comparison of the histograms of the metrics. As the reader will notice, the distributions of WMC, NOC, and LCOM are fairly similar in both cases. On the other hand, the distributions of DIT, RFC, and CBO are quite different. The only difference between LCOM and LCOMN is that more than 10 percent of the classes have negative LCOMN values, while the LCOM values of these classes are 0. In the rest of the cases—when LCOM is positive—the two metrics are the same. The distribution of LOC shows that there are many small classes (about two third of all classes are shorter than 100 lines of code).

Table 4 shows the basic statistical information about the two systems.[3] The *Minimum* values are almost the same but the *Maximum* values increased dramatically. This is not surprising because, in Mozilla, we analyzed about 18 times more classes than Basili et al. [1] did (they examined 180 classes, while, in Mozilla 1.6, there were 3,192 of them). Since LCOM and LCOMN are proportional to the square of the size (number of methods) of a class, their very large values are to be expected. (This is also the reason for the very low Min. value of LCOMN.) In Mozilla, we examined about three thousand classes, so the extremely high Max. value of NOC may seem surprising at first. But, the second biggest value of NOC is just 115 and the next is only 37; hence, we deduced that the class with the largest value is probably a common base class from which almost all other classes are inherited. (The source code was checked afterwards, which confirmed this assumption.) *Median* and *Mean* are more or less similar to the values of Basili et al. [1], except for the LCOM value (whose case is similar to the Max. value case). Since, in Mozilla, there are many more classes and these are more variegated, the metrics change over a wider range. The *Standard Deviation* values support this view.

The reader may notice that the Max. LOC value is 9,371—which is quite high—and that the average class size is only 183.27. Unfortunately, Basili et al. [1] did not study this metric so we could not compare our results with theirs.

Basili et al. [1] also calculated the *correlations* of the metrics (see Table 5), which is also an important statistical quantity. They found that Pearson's linear correlations (R^2: Coefficient of determination) between the object-oriented metrics studied were, in general, very weak. Three coefficients of determination appeared somewhat larger than the others. But, they concluded that these metrics are mostly statistically independent. We also calculated the same correlations for Mozilla 1.6 and found that all correlations

3. The values were different from our previous work [11] because, here, we filtered out some classes (see the previous section).

TABLE 4
Descriptive Statistics of the Classes

Basili et al. [1]	WMC	DIT	RFC	NOC	CBO	LCOM
Max.	99.00	9.00	105.00	13.00	30.00	426.00
Min.	1.00	0.00	0.00	0.00	0.00	0.00
Median	9.50	0.00	19.50	0.00	5.00	0.00
Mean	13.40	1.32	33.91	0.23	6.80	9.70
Standard deviation	14.90	1.99	33.37	1.54	7.56	63.77

Mozilla	WMC	DIT	RFC	NOC	CBO	LCOM	LCOMN	LOC
Max.	337.00	33.00	1,103.00	1,214.00	70.00	55,198.00	55,198.00	9,371.00
Min.	0.00	0.00	0.00	0.00	0.00	0.00	-54,614.00	0.00
Median	9.00	2.00	30.00	0.00	6.00	21.00	21.00	57.00
Mean	17.36	3.13	66.66	0.92	7.80	364.66	344.19	183.27
Standard deviation	25.32	3.42	97.01	21.65	8.35	1,875.40	2,113.59	425.31

TABLE 5
Correlations between the Metrics

Basili [1]	DIT	RFC	NOC	CBO	LCOM
WMC	0.02	**0.24**	0.00	0.13	**0.38**
DIT	1.00	0.00	0.00	0.00	0.01
RFC		1.00	0.00	**0.31**	0.09
NOC			1.00	0.00	0.00
CBO				1.00	0.01
LCOM					1.00

Mozilla	DIT	RFC	NOC	CBO	LCOM	LCOMN	LOC
WMC	0.16	**0.54**	0.00	**0.43**	**0.64**	**0.37**	**0.56**
DIT	1.00	**0.52**	0.00	0.17	0.09	0.07	0.08
RFC		1.00	0.00	**0.48**	**0.33**	0.21	**0.40**
NOC			1.00	0.00	0.00	0.00	0.00
CBO				1.00	0.19	0.15	**0.58**
LCOM					1.00	**0.79**	**0.46**
LCOMN						1.00	**0.36**
LOC							1.00

The bold numbers denote large correlations.

were significant (p-value < 0.001) except for the correlations of NOC which were not significant. This means that there is no linear association between NOC and the other metrics. DIT correlates only with RFC (its other correlation values are very small). The correlation between LCOM and LCOMN is high—as was expected—but the LCOM correlation values compared with the other metrics are much larger than the LCOMN ones. The other four metrics (WMC, RFC, CBO, and LOC) have more or less notable correlations with each other and with LCOM. What is more, there are some large values (for instance, between WMC and LCOM and between WMC and LOC), so it follows that these metrics are not totally independent and represent redundant information.

5 Analyses

In this section, we will describe the analyses we performed to discover the relationships between the values of the metrics and the numbers of bugs found in the classes. We first employed *regression analysis* methods, which are widely used to predict an unknown variable based on one or more known variables. We chose *logistic regression* [16] to study the relationships between the metrics and fault-proneness of classes. Basili et al. [1] performed the same analyses, so we could compare our results with their conclusions and examine whether their results are generally valid for large open source software.

The logistic regression method only predicts if a class is faulty or not, but does not say anything about the possible number of faults in that class. In the case of Mozilla, there are several classes which contain a lot of bugs, thus we applied *linear regression* [18] as well where the number of bugs could also be predicted. Linear analysis was applied by Yu et al. [22] too, but, since we calculated only four of the eight metrics in the same way as they did, we could compare only these.

Naturally, other statistical methods can be applied as well. For instance, Subramanyan and Krishnan [21] validated four of the eight metrics we analyzed (WMC, DIT, CBO, and LOC[4]) with a slightly modified linear regression analysis where the reciprocal of the number of the bugs is estimated. Hence, we will compare only their conclusions with ours.

In logistic and linear analyses, we applied both *univariate* and *multivariate* regressions. Univariate regression analysis is used to examine the effect of each metric separately, while multivariate regression analysis examines the common effectiveness of the metrics.

Besides the two statistical approaches, we also employed two *machine learning* methods to predict the fault-proneness of the classes. These methods are rarely applied in this area. We chose the *decision tree* method [19], which is more or less based on statistics, and the *neural networks* method [2]. Both methods are able to predict the faults using just one

4. In their study, the LOC metric is called SIZE.

TABLE 6
Result of Univariate Logistic Regression

	WMC	DIT	RFC	NOC	CBO	LCOM	LCOMN	LOC
Coefficient	1.069	0.598	0.868	-0.041	1.082	2.120	1.400	1.642
Constant	-0.252	-0.316	-0.278	-0.321	-0.283	-0.135	-0.248	-0.162
p-value	0.000	0.000	0.000	0.551	0.000	0.000	0.000	0.000
R^2	0.114	0.067	0.108	0.000	0.152	0.076	0.049	0.128

metric—which is just like univariate regression—and it is also possible to consider several metrics together for prediction—which is similar to multivariate regression.

We compared the results of the four methods and obtained very similar results in all cases. We also compared our observations with those presented by Basili et al. [1], Yu et al. [22], and Subramanyan and Krishnan [21]. But first, we will present the results of the methods one at a time.

5.1 Logistic Regression Analyses

In logistic regression, the unknown variable, called the *dependent variable*, can take only two different values. Therefore, we divided the classes into two groups according to whether a class contained at least one bug or not. The known variables, called *explanatory variables*, are the metrics. Since these metrics change over different ranges, they were standardized, which means that each metric has zero mean and unit variance.

The multivariate logistic regression model is based on the

$$\pi(X_1, X_2, \ldots, X_n) = \frac{e^{C_0+C_1\cdot X_{i_1}+\ldots+C_n\cdot X_{in}}}{1+e^{C_0+C_1\cdot X_{i_1}+\ldots+C_n\cdot X_{in}}}$$

relationship equation where the X_is are the explanatory variables and π is the probability that a fault was found in a class during validation. Logistic regression is a widely used statistical method so we will not describe it here in detail (a detailed description is given by Basili et al. [1] and Hosmer and Lemeshow [16]). Univariate logistic regression is a special case of multivariate regression in the case when there is only one explanatory variable in the model.

First, we performed univariate logistic regression (see Table 6). The *Coefficient* is the estimated regression coefficient. The larger the absolute value of the coefficient, the stronger the impact (positive or negative, according to the sign of the coefficient) of the explanatory variable on the probability of a fault being detected in a class. The *p-value* is related to the statistical hypothesis and tells us whether the corresponding coefficient is significant or not. We used the $\alpha = 0.05$ significance level to assess the p-values we obtained. The R^2 coefficient is defined as the proportion of the total variation in the dependent variable y that is explained by the regression model. The bigger the value of R^2, the larger the portion of the total variance in y that is explained by the regression model and the better the dependent variable y is explained by the explanatory variables.

We can see that seven of the eight metrics are very significant (p-value < 0.001) and NOC is the only metric which is not significant (p-value = 0.551). CBO has the largest R^2 value—larger than the value of LOC—which suggests that CBO is the best predictor. The R^2 values of DIT, LCOM, and LCOMN are significantly smaller than the others, hence, they seem less useful. We should mention here that, in logistic regression, high R^2 values are rare opposed to the R^2 values of least-square regression because they are built on very different formulae. For this reason, these values should not be compared with the R^2 values of other regression analyses like the R^2 values of our linear regression.

We know that the examined metrics are not totally independent (see Table 5) and, hence, must capture redundant information. Thus, not all of them are required in multivariate analyses. A standard procedure called *stepwise selection* was used to select the necessary variables for multivariate analysis (see Table 7). The metrics were selected in the CBO, DIT, WMC, LOC order. As can be seen, all four metrics are significant at $\alpha = 0.05$. The R^2 value of the multivariate logistic regression is 0.175, which is only slightly better than the R^2 value of CBO in univariate analysis.

Logistic regression provides models for classifications; one for each metric in univariate logistic regression and one for the multivariate logistic regression. These models were applied with a threshold value of 0.5, which means that, if $0.5 < \pi$, the class is classified as faulty, otherwise, as not faulty. Table 8 shows the results of the multivariate logistic regression model, where the numbers in parentheses are the sum of faults that were found in that group of classes. As can be seen, the model classified 2,222 (1,624 + 598) of the 3,192 classes correctly, that is, with a precision of 69.61 percent.

From a testing point of view, there are two more important quantities which say more about the quality of the model. The first one is the *correctness* [3] value, which describes what percentage of the faulty predicted classes is really faulty—more precisely, it is the number of classes observed and predicted faulty divided by the number of all

TABLE 7
Result of Multivariate Logistic Regression

	const.	CBO	DIT	WMC	LOC
Coefficient	-0.229	0.631	0.310	0.246	0.461
p-value	0.003	0.000	0.000	0.000	0.002

TABLE 8
Classification Result

	Predicted	
Observed	Not faulty	Faulty
Not faulty	1,624	226
Faulty	744 (1,377)	598 (2,584)

TABLE 9
Precision, Correctness, and Completeness on Mozilla 1.6

Metric	Precision	Correctness	Completeness
WMC	65.38%	68.84%	55.24%
DIT	64.04%	65.06%	45.17%
RFC	66.01%	71.89%	53.60%
NOC	57.96%	–	–
CBO	69.77%	70.38%	69.12%
LCOM	64.69%	81.34%	43.68%
LCOMN	63.82%	85.02%	39.01%
LOC	66.85%	72.98%	54.58%
Multiv. reg.	69.61%	72.57%	65.24%

faulty predicted classes. The larger the correctness, the fewer faultless classes have to be tested, which, in turn, improves the efficiency of the testing. In this case, 824 (226 + 598) classes were predicted as faulty and 598 of them were really faulty, meaning that the correctness was 72.57 percent.

The other important question is what percentage of the total number of faults can be captured in this way. This is expressed by the *completeness* [3] value, which is defined as the number of faults in faulty predicted classes divided by the number of faults in all classes. The larger the completeness values, the bigger the rate of the faults that are predicted. Although the multivariate model identified less than half of the faulty classes correctly—598 out of 1,342 (744 + 598)—this smaller group contains many more faults—2,584 out of 3,961 (1,377 + 2,584)—so the completeness is 65.24 percent.

Table 9 shows the values for the precision, correctness, and completeness of the univariate models and the multivariate logistic regression model in the last row as well. Since, in the case of NOC, the model classified all classes as faultless, there is no point in calculating the correctness and the completeness for NOC. LCOM and LCOMN have very high correctness values (81.34 percent and 85.02 percent), meaning that only a small percentage (18.66 percent and 14.98 percent) of the faultless classes were predicted as faulty. This is very good, but the low completeness values (43.68 percent and 39.01 percent) mean that only a small fraction of the faults could be discovered by these metrics. DIT's completeness value was also low (45.17 percent) and its precision and correctness value were also poorer. These values correspond the low R^2 values of LCOM, LCOMN, and DIT. On the other hand, the values of LOC, WMC, and RFC are more or less equal and these values are acceptable. The precision and completeness values of CBO are the largest—larger than the values of the multivariate model—and its correctness value is also good, once again confirming that CBO is the best predictor.

TABLE 10
Precision, Correctness, and Completeness on Mozilla 1.0

Mozilla 1.0	Precision	Correctness	Completeness
Multiv. reg.	57.96%	83.76%	55.94%

Out of interest, we decided to test the model built on Mozilla 1.6 on version 1.0. The result of the precision, correctness, and completeness can be seen in Table 10. We chose version 1.0 because it is the most distant analyzed version in time. We can say that, even though the precision and completeness values worsened, the model is still useful for predicting the fault-proneness of classes.

5.2 Linear Regression Analyses

In Mozilla, there are many classes which contain a lot of bugs and the number of bugs varies over a wide range (see Table 2). This situation cannot be modelled by logistic regression, so we also performed linear regression analyses. In linear regression, the explanatory variables are the standardized metrics as well, but the dependent variable is the number of bugs in a class.

First, we applied the univariate linear regression analysis (see Table 11). Similar to the results of the logistic regression, seven of the eight metrics are very significant and only NOC is not significant as before. CBO has the largest R^2 value, but it is only slightly better than the value of LOC. On the other hand, the value of DIT has the smallest one, which is also similar to the results of the univariate logistic regression analysis.

We also performed a multivariate linear regression analysis (see Table 12). Four of the five selected metrics are the same as in the case of the logistic regression and CBO was selected first as well but the selection order of the other metrics changed.

The R^2 value is 0.43, which means that 57 percent of the total variation was not captured. There are several possible reasons of this. One might be that we used heuristics to assign classes to bugs and bugs to versions. Classes which were faultless and did not change over the seven versions were filtered out, but the remaining ones may not be homogeneous. This means that the number of bugs in the new classes was counted precisely, but, for classes which existed before version 1.0, we only counted the bugs which were corrected after version 1.0. Thus, the number of bugs counted by us might depend on the class creation time.

5.3 Machine Learning Models

In this section, we will present our results of employing machine learning tools (C4.5 [19] for building decision trees

TABLE 11
Result of Univariate Linear Regression

	WMC	DIT	RFC	NOC	CBO	LCOM	LCOMN	LOC
Coefficient	1.641	1.082	1.533	-0.018	1.712	1.328	1.147	1.694
Constant	1.241	1.241	1.241	1.241	1.241	1.241	1.241	1.241
p-value	0.000	0.000	0.000	0.728	0.000	0.000	0.000	0.000
R^2	0.321	0.139	0.280	0.000	0.349	0.210	0.157	0.342

TABLE 12
Result of Multivariate Linear Regression

	const.	CBO	WMC	LOC	DIT	RFC
Coefficient	1.241	0.744	0.597	0.678	0.505	-0.214
p-value	0.000	0.000	0.000	0.000	0.000	0.012

TABLE 13
Overall Learning Precision

Classification	Decision tree	Neural network
<0,1-34>	69.58%(4.92%)	68.77%(4.93%)
<0,1,2-12,13-34>	62.91%(4.43%)	63.75%(4.25%)

TABLE 14
Learning Precision for Individual Metrics ($< 0, 1\text{-}34 >$)

Metrics	Decision tree	Neural network
WMC	66.51%(4.32%)	66.20%(4.33%)
DIT	63.66%(3.77%)	63.47%(3.40%)
RFC	66.45%(5.06%)	66.76%(4.60%)
NOC	57.95%(6.00%)	57.96%(6.00%)
CBO	69.77%(4.63%)	69.46%(4.60%)
LCOM	66.67%(4.10%)	66.17%(3.71%)
LCOMN	66.67%(4.10%)	67.17%(3.90%)
LOC	67.98%(4.93%)	67.58%(4.58%)

and a neural network system developed at our university) to predict the probable number of bugs in a class with the help of metrics as predictors. We trained the systems in two different ways. First, we set up a training database in which we treated a class as faulty if it contained at least one bug. Second, we considered four levels of faultiness. This way, we had two classifications. These were:

1. We considered only two categories: one category which contained classes without bugs and the other which contained classes with bugs (at least one). We will use the $< 0, 1\text{-}34 >$ notation for this case.
2. We considered four categories: one with classes without bugs, another with only one bug, one which contained 2 to 12 bugs, and a category with 13 or more bugs. We will use the $< 0, 1, 2\text{-}12, 13\text{-}34 >$ notation for this case.

In our experiments, we employed the method of 10-fold cross-validation for learning and testing. This means that we divided the training data into 10 equal parts and then performed the learning process ten times. Each time, we chose another part for testing and used the remaining nine parts for learning. After, we calculated the average values and the deviation values from the ten different testing results. We applied this procedure to both learning systems.

Table 13 shows the overall precision of the learning with all metrics used as predictors (the numbers in parentheses show the deviation). We analyzed the models and found that the two learning approaches make mistakes mostly in classifying classes where the number of bugs is one or two, but are more reliable with those classes which are bug-free or contain three or more bugs.

We also tested the efficiency of our learning approaches using only one metric at a time to find out which metrics are good predictors of fault-proneness. Table 14 gives the results for the $< 0, 1\text{-}34 >$ classification, while Table 15 provides the results for the $< 0, 1, 2\text{-}12, 13\text{-}34 >$ case.

TABLE 15
Learning Precision for Individual Metrics ($< 0, 1, 2\text{-}12, 13\text{-}34 >$)

Metrics	Decision tree	Neural network
WMC	62.91%(4.50%)	62.22%(4.59%)
DIT	58.45%(6.13%)	60.15%(4.39%)
RFC	61.94%(4.24%)	62.16%(4.25%)
NOC	57.95%(6.00%)	57.96%(6.00%)
CBO	63.79%(4.68%)	63.25%(5.08%)
LCOM	62.75%(4.22%)	62.16%(4.80%)
LCOMN	62.75%(4.22%)	62.16%(4.58%)
LOC	62.78%(5.94%)	62.44%(5.58%)

TABLE 16
Correctness and Completeness for Individual Metrics ($< 0, 1\text{-}34 >$)

Correctness	Decision Tree	Neural network
WMC	62.34%(10.21%)	65.75%(12.51%)
DIT	63.13%(12.70%)	61.36%(17.35%)
RFC	65.15%(13.46%)	63.99%(10.07%)
NOC	-	-
CBO	69.13%(11.88%)	70.63%(10.84%)
LCOM	63.96%(12.07%)	63.92%(12.50%)
LCOMN	63.96%(12.07%)	66.70%(11.09%)
LOC	66.81%(08.37%)	65.29%(08.05%)
Multi	68.38%(11.47%)	68.94%(13.46%)
Completeness	Decision Tree	Neural network
WMC	65.33%(10.80%)	60.19%(08.59%)
DIT	41.09%(18.24%)	40.52%(18.08%)
RFC	56.91%(15.64%)	61.66%(16.13%)
NOC	-	-
CBO	67.02%(11.88%)	65.13%(11.81%)
LCOM	60.59%(12.66%)	60.36%(13.34%)
LCOMN	60.59%(12.66%)	60.62%(09.24%)
LOC	64.41%(10.95%)	65.85%(12.28%)
Multi	67.84%(11.96%)	64.76%(11.43%)

As can be seen, the most precise metric in both models is CBO. Its average learning precision is 69.77 percent in the first classification ($< 0, 1\text{-}34 >$) for the decision tree model and 69.46 percent for the neural network, which are both slightly better than the overall learning precisions (69.58 percent for the decision tree and 68.77 percent for the neural network).

We also calculated the same *correctness* and *completeness* values as those described in Section 5.1 for our learning results. These are presented in Table 16 (we calculated these values only for the $< 0, 1\text{-}34 >$ classification because the definitions of correctness and completeness deal only with this case). This table lists the average values and deviation values for the correctness and completeness of the 10 different cases. It shows the same values for each metric separately as well.

Upon analyzing Table 16, we notice that CBO is once again the best metric in our study. In the decision tree model, the learning correctness value of CBO (69.13 percent) is surprisingly larger than the overall correctness value (68.38 percent) and its learning completeness value (67.02 percent) is very close to the overall completeness value (67.84 percent). In the neural network model, the learning correctness value of CBO is also the best and its completeness value is larger than the

TABLE 17
Overall Learning Precision for Version 1.0

Classification	Decision tree	Neural network
<0,1-34>	58.01%(0.41%)	57.81%(0.64%)
<0,1,2-12,13-34>	50.88%(0.44%)	51.28%(0.27%)

TABLE 18
Correctness and Completeness for Version 1.0 ($< 0, 1\text{-}34 >$)

Model Accuracy	Decision tree	Neural network
Correctness	80.95%(0.64%)	81.03%(2.28%)
Completeness	58.15%(1.10%)	57.05%(2.06%)

TABLE 19
Results of the Different Validations

Metric	Our results	[1]	[22]	[21]
WMC	++	+	++	++
DIT	+	++	0	−
RFC	++	++	+	
NOC	0	++	−−	
CBO	++	+	+	+
LCOM	+	0		
LCOMN	+			
LOC	++			+

+ denotes that this metric is significant (++ means that it is more useful). − denotes that this metric is significant, but in an inverse way (−− means that it is more useful). 0 denotes that this metric is not significant. A blank entry means that our hypothesis was not examined or the metric was calculated in a different way.

TABLE 20
Summary of Precision, Correctness, and Completeness

Metric	Model	Precision	Correctness	Completeness
WMC	Log. reg.	65.38%	68.84%	55.24%
	Dec. tree	66.51%	62.34%	65.33%
	Neural n.	66.20%	65.75%	60.19%
DIT	Log. reg.	64.04%	65.06%	45.17%
	Dec. tree	63.66%	63.13%	41.09%
	Neural n.	63.47%	61.36%	40.52%
RFC	Log. reg.	66.01%	71.89%	53.60%
	Dec. tree	66.45%	65.15%	56.91%
	Neural n.	66.76%	63.99%	61.66%
NOC	Log. reg.	57.96%	–	–
	Dec. tree	57.95%	–	–
	Neural n.	57.96%	–	–
CBO	Log. reg.	69.77%	68.84%	55.24%
	Dec. tree	69.77%	69.13%	67.02%
	Neural n.	69.46%	70.63%	65.13%
LCOM	Log. reg.	64.69%	81.34%	43.68%
	Dec. tree	66.67%	63.96%	60.59%
	Neural n.	66.17%	63.92%	60.36%
LCOMN	Log. reg.	64.69%	85.02%	39.01%
	Dec. tree	66.67%	63.96%	60.59%
	Neural n.	67.17%	66.70%	60.62%
LOC	Log. reg.	66.85%	72.98%	54.58%
	Dec. tree	67.98%	66.81%	64.41%
	Neural n.	67.58%	65.29%	65.85%
Multi	Log. reg.	69.61%	72.57%	65.24%
	Dec. tree	69.58%	68.38%	67.84%
	Neural n.	68.77%	68.94%	64.76%

overall value, but LOC has the largest completeness value here. The correctness and completeness of NOC cannot be interpreted because, when using NOC for learning, both models always predict that every class will be nonfaulty. The correctness values of DIT (63.13 percent and 61.36 percent) are only slightly worse than other correctness values, but its completeness values are very low (41.09 percent and 40.52 percent). Overall, we may conclude that CBO is the best metric here, NOC seems quite unimportant in predicting the number of bugs in a class, and the usability of DIT is somewhat limited.

Afterward, we performed the same interesting experiment as in Section 5.1. That is, we tested the precision, the correctness, and the completeness of the models trained on version 1.6 on Mozilla 1.0. Tables 17 and 18 list the results.

Similar to our findings in Section 5.1, the values worsened a little (except for the correctness value), but the model is still useful for predicting the fault-proneness of classes.

5.4 Discussion and the Validation of the Hypotheses

In this section, we will exploit the results of the previous three sections to validate our hypotheses stated in Section 2. At the same time, we will also compare our conclusions with those of Basili et al. [1], Yu et al. [22], and Subramanyan and Krishnan [21] (see Table 19).

Apart from the results of the regression analyses, the precision, correctness, and completeness values can also help us in the validation process. These values were calculated via the logistic regression, decision tree, and neural network approaches (see Sections 5.1 and 5.3). We summarized the descriptive values of these three models in Table 20. (Note that the linear regression analyses did not produce these values.)

- **WMC hypothesis**. WMC (Weighted Methods per Class) was found to be very significant in our regression analyses. On the other hand, Basili et al. [1] found it less significant, but, for extensively modified classes and for UI classes, it was more significant. In the study by Yu et al. [22], WMC was significant and it was found to be the best predictor—which is similar to our linear regression results where it is one of the best predictors. Subramanyan and Krishnan [21] found WMC to be significant for C++ (but not significant for Java).

  The results of machine learning are also in line with our statistical results where the precision values—which are around 66 percent—are almost the same as those in the statistical models. The correctness and the completeness values changed slightly in the three models but there is no extremely high or low value (see Table 20).

  All of our four analyses of WMC yielded the same result; hence, we rejected the null hypothesis of WMC and accepted the alternative hypothesis.
- **DIT hypothesis**. DIT (Depth of Inheritance Tree) was found to be very significant in logistic regression. This finding is similar to those given by Basili et al. [1]. Subramanyan and Krishnan [21] found that DIT was significant as well but in an inverse way, which contradicts our result. Furthermore, DIT was significant in linear regression as well, contradicting the result of Yu et al. [22], where DIT was

found to be insignificant. In spite of the good p-value, the R^2 values were smaller than those of the other metrics in the two regression analyses, which follows the smaller precision value of 64 percent. The precision of the machine learning models is around 63 percent, which is significantly smaller than the others (except NOC). The correctness values are more or less similar to the values of the other metrics, but the completeness values are worse in both the logistic regression and machine learning models (see Table 20).

We can only say that we rejected the null hypothesis of DIT and accepted the alternative hypothesis, but DIT is not such a good predictor as the others and further investigation is needed.

- **RFC hypothesis**. We found RFC (Response For a Class) to be very significant—which is the same as the result of Basili et al. [1]. Yu et al. [22] found RFC to be significant as well but they calculated the RFC in a different way.[5] We examined the difference between the two RFCs in the case of Mozilla and we found a high ($R^2 = 0.66$) and significant (p-value < 0.001) correlation between them, so we decided to ignore the differences between the two definitions of RFC. The precision values of the machine learning models—which are about 66 percent—are the same as the precision value of logistic regression. The completeness values of the three models are fairly similar and the difference in the correctness values is not too large (see Table 20).

  Taking into account these results, we rejected the null hypothesis of RFC and accepted the alternative hypothesis.
- **NOC hypothesis**.[6] We found that NOC (Number Of Children) was not significant in the linear and logistic regression analyses. The machine learning models predicted that all classes would be faultless, which agrees with the inference about the insignificance of the statistical models. These extreme classifications say that the precision is the same as the percentage of faultless classes (see Table 2) and that the correctness and completeness values mean nothing. In accordance with our findings, we saw that NOC could not be used for fault-proneness prediction and we could not validate our NOC hypothesis.

  Basili et al. [1] found NOC to be very significant and they noticed that, the larger the value of NOC, the lower the probability of fault detection. According to Yu et al. [22], NOC was significant as well but they found that, the more children a class has, the more fault prone it is—which contradicts the result of Basili et al. [1].

  We accepted the null hypothesis of NOC and our conclusion is that not only is NOC a bad predictor for fault-proneness detection, but it is also unreliable because the three different studies evaluated this metric in three different ways.
- **CBO hypothesis**. CBO (Coupling Between Object classes) was found to be very significant and it is the best predictor in both linear and logistic regression analyses. Just like WMC, this metric was found significant in all three studies [1], [21], [22][7]—which also confirms that the CBO is the best among these metrics.

  The machine learning models confirmed the findings of the regression analyses because the precision, correctness, and completeness values of CBO are the best among the metrics (except for completeness in the case of the neural network where LOC is slightly better—see Table 20). Moreover, the precision values of CBO on its own are better than those of the multivariate models, and the correctness and completeness values are also better in some cases.

  Thus, we can say that not only we did reject the null hypothesis of CBO (and accepted the alternative hypothesis), but that CBO is the best predictor out of the eight metrics in every aspect.
- **LCOM hypothesis**. LCOM (Lack of Cohesion on Methods) was found to be very significant during the regression analyses, contradicting the result of Basili et al. [1], where LCOM was shown to be insignificant. Yu et al. [22] calculated LCOM in a totally different way, hence, we could not compare our results with theirs.

  The precision values of our models are almost the same, but the correctness value of the logistic regression model is significantly larger than those of the machine learning models, while the completeness is much smaller (see Table 20). We do not know the reason for these big differences, so it needs further investigation.

  Overall, though, we rejected the null hypothesis of LCOM and accepted the alternative hypothesis.
- **LCOMN hypothesis**. LCOMN (Lack of Cohesion on Methods allowing Negative value) was found to be very significant (like LCOM) but, in both regression analyses, its R^2 coefficient values were worse than LCOM's values. The correctness of the regression model was larger and the completeness was smaller than those of LCOM. On the other hand, the values from the machine learning models are the same in almost every case. Our conclusion is that LCOM and LCOMN are similar and we cannot say which one of them is better.

  Similar to LCOM, we rejected the null hypothesis of LCOMN and accepted the alternative hypothesis.
- **LOC hypothesis**. LOC (Lines Of Code) was found to be very significant in both regression analyses and only CBO was better than LOC. Subramanyan and Krishnan [21] also found that LOC was significant, but neither Basili et al. [1] nor Yu et al. [22] examined this metric.

5. The methods of the class in question were excluded from RFC by Yu et al. [22].

6. In our NOC hypothesis, the relationship between the NOC metric and the number of bugs was the inverse of that of Basili et al. [1] and Yu et al. [22].

7. In the study of Yu et al. [22], $\mathrm{CBO}_{\mathrm{out}}$ represents the same metric as CBO here.

TABLE 21
Changes in the Mean Value of the Metrics over Seven Versions of Mozilla

ver.	No. of classes	No. of bugs	Bugs per class	WMC	
				Mean	Dev.
1.0	3585	6612	1.84	**15.88**	23.68
1.1	3624	5720	1.58	**15.89**	23.77
1.2	3451	5549	1.61	**16.76**	24.31
1.3	3491	4960	1.42	**16.58**	24.28
1.4	3666	4243	1.16	**16.03**	23.79
1.5	3689	3300	0.89	**15.98**	23.96
1.6	3677	3961	1.08	**15.99**	24.06

ver.	DIT		RFC		LCOM	
	Mean	Dev.	Mean	Dev.	Mean	Dev.
1.0	**2.89**	2.97	**59.12**	86.01	**322.47**	1754.78
1.1	**2.88**	2.98	**59.23**	86.19	**324.93**	1776.92
1.2	**2.76**	3.03	**61.45**	88.53	**332.63**	1603.85
1.3	**2.75**	3.06	**61.39**	89.04	**332.00**	1626.51
1.4	**2.75**	3.16	**60.63**	90.59	**314.54**	1609.34
1.5	**2.77**	3.25	**61.10**	93.67	**319.15**	1726.65
1.6	**2.76**	3.25	**60.69**	92.44	**321.70**	1752.48

ver.	LCOMN		CBO		LOC	
	Mean	Dev.	Mean	Dev.	Mean	Dev.
1.0	**319.09**	1755.62	**6.52**	7.95	**163.93**	394.42
1.1	**321.64**	1767.72	**6.55**	7.99	**164.63**	398.54
1.2	**313.34**	1857.45	**6.92**	8.05	**174.52**	411.29
1.3	**312.92**	1879.91	**7.13**	8.14	**171.88**	409.50
1.4	**296.21**	1853.55	**6.98**	7.98	**166.32**	406.01
1.5	**301.05**	1949.89	**6.97**	7.99	**165.32**	402.89
1.6	**303.51**	1973.51	**6.99**	8.10	**164.44**	400.29

The precision values of LOC were good and only the values of CBO were better. The correctness and completeness values indicate that LOC is one of the best metrics for fault prediction.

All analyses yielded the same result, therefore, we rejected the null hypothesis of LOC and accepted the alternative hypothesis.

We used the results of the univariate models to validate the hypotheses. Now, we will also compare the results of the multivariate analyses to each other. The multivariate logistic analysis chose four metrics (CBO, DIT, WMC, and LOC) while the multivariate linear analysis chose five metrics (CBO, WMC, LOC, DIT, and RFC). Although the same four metrics were chosen, the order of the selection was different.

The precision and correctness values of the logistic regression model are slightly better than those obtained from the machine learning models, but the completeness value of the decision tree model is better than that from logistic regression (see the last row in Table 20). The neural network approach produced the weakest model, although its correctness value was slightly better than that from the decision tree model.

6 Studying Mozilla's Evolution

In this section, we will analyze how Mozilla evolved from version 1.0 released in June of 2002 to version 1.6 released in January of 2004 using the results of our analyses discussed in the previous section.

In Table 3, we listed the correlations between the number of bugs associated with the classes in the different versions. The table shows that the correlations are very large even in the case of the most distant versions in time (1.0-1.6). We also tested the effectiveness of our models (built on version 1.6) on version 1.0 (see Sections 5.1 and 5.3) and concluded that, even when the precision and completeness values decreased, the correctness values of the models were still very good.

We analyzed the distribution of the metrics in the different versions (see Fig. 2 for version 1.6), but, because of the large amount of data we had to deal with, it was hard to draw any sound conclusions from it. Hence, we decided to examine aggregated data. We know that we cannot draw completely trustworthy conclusions from the averages of the metrics, but we tried to identify some trends in these values. Table 21 lists the mean and standard deviation values of the Mozilla versions (Fig. 3 shows the same information using graphical charts).[8] The deviations are high, but this is normal for such a large software system as Mozilla (there are more than 3,000 classes and they are highly variegated).

Looking at Fig. 3, we noticed an interesting phenomenon. The values of WMC, RFC, LCOM, CBO, and LOC increased significantly with version 1.2. (The values of DIT and LCOMN changed in the opposite direction, but, as previously stated, these metrics are not really useful for fault prediction.) With that version, the number of bugs also increased, which is interesting too because, at the same time, the number of classes decreased (see Table 21).

We presumed from these observations that there must have been a bigger reorganization of the Mozilla source code with version 1.2, causing a significant increase in the metric values and the number of the bugs. Of course, many other factors might also influence the number of submitted bugs and fixed bugs, so further investigation is needed to obtain more precise results.

7 Related Work

Many researchers have sought to analyze the connection between object-oriented metrics and code quality. A summary of the empirical literature was given by Subramanyan and Krishnan [21]. Here, we considered only those which are closely connected to our work. The work that is most similar to ours is Basili et al.'s paper [1], which we discussed in Section 4.

Yu et al. [22] chose eight metrics (actually, 10, because CBO and RFC were divided into two different types) and they examined the relationship between these metrics and the fault-proneness. The subject system was the client side of a large network service management system developed by three professional software engineers. It was written in Java and consisted of 123 classes and around 34,000 lines of code. First, they examined the correlation among the metrics and found four highly correlated subsets. Then, they used univariate analysis to find out which metrics could detect faults and which could not. They found that

8. We did not include the values of NOC because we rejected the NOC hypothesis in the previous section.

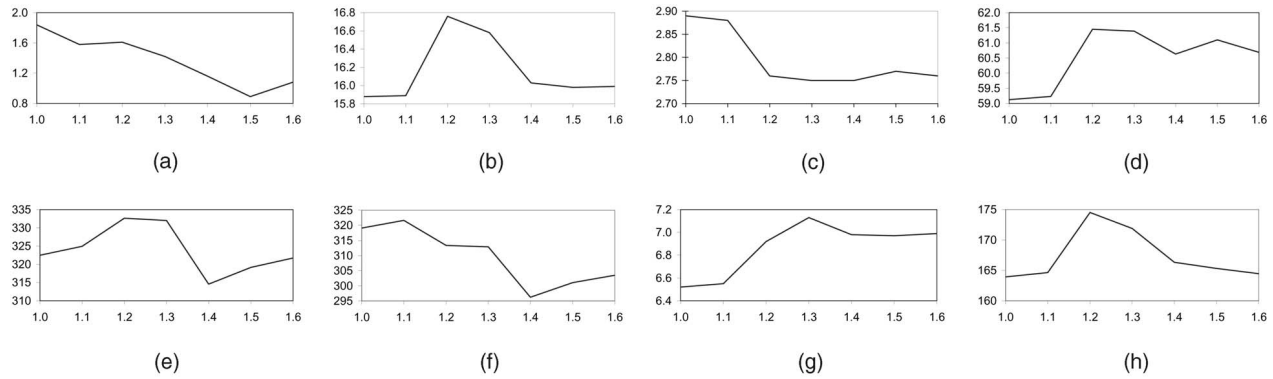

Fig. 3. Changes in the mean value of the metrics over seven versions of Mozilla.

three of the metrics (CBO_{in}, RFC_{in}, and DIT) were unimportant, while the others were significant but to different extents (WMC, LOC, CBO_{out}, RFC_{out}, LCOM, NOC, and Fan-in).

Subramanyam and Krishnan [21] chose a relatively large e-commerce application developed in C++ and Java and collected metrics from 405 C++ and 301 Java classes. They examined the effect of the size along with the WMC, CBO, and DIT values on the faults by using multivariate regression analysis. Besides validating the usefulness of metrics, they compared the applicability of the metrics in different languages; thus, they validated their hypotheses for C++ and Java classes separately. They concluded that the size was a good predictor in both languages, but WMC and CBO could be validated only for C++.

Fioravanti and Nesi [13] used the results of the same projects as Basili et al. [1] to examine how metrics could be used for fault-proneness detection. They calculated 226 metrics and their aim was to find a minimum number of metrics for obtaining a good identification of faulty classes in medium-sized projects. First, they reduced the number of metrics to 42 and attained a very high accuracy score (over 97 percent). But, this model was still too large to be useful in practice. Applying statistical techniques based on logistic regression, they created a hybrid model which consisted of only 12 metrics with an accuracy that was still good enough to be useful (close to 85 percent). The metrics suite they obtained was not the same as the one used by Basili et al. [1] but there are many similarities.

This article is the continuation of previous work [11], where we described our fact extraction process and wrapper tools in detail. We analyzed seven different versions of Mozilla and calculated metrics from them. At that time, we had no information about the faults, so we used the results of Basili et al. [1] to analyze Mozilla. We compared some basic statistics of the metrics with those of Basili et al. [1] and examined how Mozilla changed over its seven version evolution.

Mozilla was investigated earlier by Godfrey and Lee [15]. They examined the software architecture model of Mozilla Milestone-9. The authors used the *PBS* [12] and *Acacia* [7] reverse engineering systems for fact extraction and visualization. They created the subsystem hierarchy of Mozilla and looked at the relationships among them. Their model consisted of 11 top-level systems which could be divided into smaller subsystems. They created the subsystem hierarchy by taking into consideration things like source directory structure and software documentation. It turned out that the dependency graph was nearly complete, which means that almost all the top-level systems used each other.

8 Conclusion and Future Work

The main contributions of this paper are the following:

1. We presented a method and toolset with which metrics (and also other data) can be automatically calculated from the C++ source code of real-size software (the toolset is freely available for academic purposes and can be downloaded from the home page of FrontEndART [14]).
2. By processing the Bugzilla database, we associated the bugs with classes found in the source code.
3. We employed statistical (logical and linear regression) and machine learning (decision tree and neural network) methods to assess the applicability of the well-known object-oriented metrics to predict the number of bugs in classes.
4. Using the calculated metrics, we studied how Mozilla's predicted fault-proneness changed over seven versions covering one and a half years of development.

Our main observations are the following:

1. All four assessment methods employed yielded very similar results.
2. The CBO metric seems to be the best in predicting the fault-proneness of classes.
3. The LOC metric performed fairly well and, because it can be easily calculated, it seems to be suitable for quick fault prediction. However, for fine-grained analyses, the multivariate models perform much better (e.g., in the case of linear regression, the R^2 value of LOC was 0.34, while the R^2 value of the multivariate model was 0.43).
4. The correctness of the LCOM metric is good, but its completeness value is low.
5. The DIT metric is untrustworthy, and NOC cannot be used at all for fault-proneness prediction.
6. In Mozilla version 1.2, we noticed significant changes in the metrics—which we believe reflects a

fall in quality—but it slowly restored in the later versions.

The precision of our models is not yet satisfactory, so we have to analyze what the reasons are for the most common errors in the models and examine whether other metrics can improve them. We will also check whether multiple models perform better when combined in some way (e.g., using voting majority).

We are currently performing the same kind of investigation on other large software systems (OpenOffice.org and two industrial systems). In the future, we plan to scan Mozilla (and other open source systems) regularly for fault-proneness and make these results publicly available for the software developer community.

ACKNOWLEDGMENTS

The authors would like to thank the developer community of Mozilla who kindly provided us the Bugzilla [6] database. They would also like to thank Lajos Fülöp for his assistance in performing various machine learning tasks, László Viharos for his statistical advice, Ernö Jójárt for his help in processing the Bugzilla database, László Tóth for providing us with his neural network system, and David Curley for correcting this paper from a linguistic point of view. This work was supported by grants No. GVOP-3.3.1.-2004-04-0024/3.0 and No. GVOP-3.1.1.-2004-05-0345/3.0.

REFERENCES

[1] V.R. Basili, L.C. Briand, and W.L. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators," *IEEE Trans. Software Eng.,* vol. 22, no. 10, pp. 751-761, Oct. 1996.
[2] C.M. Bishop, *Neural Networks for Pattern Recognition.* Oxford, U.K.: Clarendon Press, 1995.
[3] L.C. Briand, W.L. Melo, and J. Wüst, "Assessing the Applicability of Fault-Proneness Models Across Object-Oriented Software Projects," *IEEE Trans. Software Eng.,* vol. 28, no. 7, pp. 706-720, July 2002.
[4] L.C. Briand and J. Wüst, "Empirical Studies of Quality Models in Object-Oriented Systems," *Advances in Computers,* vol. 56, Sept. 2002.
[5] L.C. Briand, J. Wüst, J.W. Daly, and D.V. Porter, "Exploring the Relationships between Design Measures and Software Quality in Object-Oriented Systems," *The J. Systems and Software,* vol. 51, pp. 245-273, 2000.
[6] Bugzilla for Mozilla, http://bugzilla.mozilla.org, 2005.
[7] Y.-F. Chen, E.R. Gansner, and E. Koutsofios, "A C++ Data Model Supporting Reachability Analysis and Dead Code Detection," *IEEE Trans. Software Eng.,* vol. 24, no. 9, pp. 682-693, Sept. 1998.
[8] 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, 1994.
[9] R. Ferenc and Á. Beszédes, "Data Exchange with the Columbus Schema for C++," *Proc. Sixth European Conf. Software Maintenance and Reeng. (CSMR 2002),* pp. 59-66, Mar. 2002.
[10] R. Ferenc, Á. Beszédes, M. Tarkiainen, and T. Gyimóthy, "Columbus—Reverse Engineering Tool and Schema for C++," *Proc. 18th Int'l Conf. Software Maintenance (ICSM 2002),* pp. 172-181, Oct. 2002.
[11] R. Ferenc, I. Siket, and T. Gyimóthy, "Extracting Facts from Open Source Software," *Proc. 20th Int'l Conf. Software Maintenance (ICSM 2004),* pp. 60-69, Sept. 2004.
[12] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Mueller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong, "The Software Bookshelf," *IBM Systems J.,* vol. 36, pp. 564-593, Nov. 1997.
[13] F. Fioravanti and P. Nesi, "A Study on Fault-Proneness Detection of Object-Oriented Systems," *Proc. Fifth European Conf. Software Maintenance and Reeng. (CSMR 2001),* pp. 121-130, Mar. 2001.
[14] FrontEndART Software Ltd., http://www.frontendart.com, 2005.
[15] M.W. Godfrey and E.H.S. Lee, "Secrets from the Monster: Extracting Mozilla's Software Architecture," *Proc. Second Int'l Symp. Constructing Software Eng. Tools (CoSET 2000),* pp. 15-23, June 2000.
[16] D. Hosmer and S. Lemeshow, *Applied Logistic Regression.* Wiley-Interscience, 1989.
[17] The Mozilla Homepage, http://www.mozilla.org, 2005.
[18] J. Neter, W. Wasserman, and M.H. Kutner, *Applied Linear Statistical Models,* third ed. Richard D. Irwin, 1990.
[19] J.R. Quinlan, *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1993.
[20] C.R. Reis and R. Pontin de Mattos Fortes, "An Overview of the Software Engineering Process and Tools in the Mozilla Project," *Proc. Workshop Open Source Software Development,* pp. 155-175, Feb. 2002.
[21] R. Subramanyan and M.S. Krishnan, "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects," *IEEE Trans. Software Eng.,* vol. 29, pp. 297-310, Apr. 2003.
[22] P. Yu, T. Systä, and H. Müller, "Predicting Fault-Proneness Using OO Metrics: An Industrial Case Study," *Proc. Sixth European Conf. Software Maintenance and Reeng. (CSMR 2002),* pp. 99-107, Mar. 2002.

Tibor Gyimóthy is the head of the Software Engineering Department at the University of Szeged in Hungary. His research interests include program comprehension, slicing, reverse engineering, and compiler optimization. He has published more then 60 papers in these areas and was the leader of several software engineering R&D projects. He is the program cochair of the 21th International Conference on Software Maintenance (ICSM 2005).

Rudolf Ferenc obtained the PhD degree in computer science from the University of Szeged in 2005. His chosen field of research is source code analysis, modeling, measurement, and design pattern recognition. He is also interested in software quality assurance and open source software development. He is member of the program committee and tools cochair of the 21th International Conference on Software Maintenance (ICSM 2005).

István Siket obtained the MSc degree in mathematics and in computer science from the University of Szeged in 2003 and is currently a PhD student. His main research interests are source code measurement and software quality assurance.