periodica polytechnica

Electrical Engineering
52/3-4 (2008) 177–186
doi: 10.3311/pp.ee.2008-3-4.06
web: http://www.pp.bme.hu/ee

RESEARCH ARTICLE

Evaluating the effectiveness of object-oriented metrics for bug prediction

István Siket

Received 2008-07-22

Abstract

In our experiments we examined the general relationship between object-oriented metrics and the fault-proneness of classes. We analyzed a large open-source program called Mozilla, calculated 58 object-oriented metrics for Mozilla at the class level [9]*, collected the reported and corrected bugs from the bug tracking system of Mozilla and associated them with the classes. We applied logistic regression to examine which metrics could be used to predict the fault proneness of the classes. We found that 17 of the 58 object-oriented metrics were useful predictors, but to a different extent. The CBO (Coupling Between Object classes) metric was the best, but it was only slightly better than NOI (Number of Outgoing Invocations) and RFC (Response Set for a Class), which proved useful as well.*

We also examined the metrics in terms of their categories and we found that coupling metrics were the best predictors for finding bugs, but the complexity and size metrics also gave good results. On the other hand, in tests all the inheritance-related metrics were statistically insignificant.

Keywords

Fact extraction · object-oriented metrics · fault-proneness detection · Mozilla · Bugzilla · Columbus

Acknowledgement

This study was supported in part by the Hungarian national grants RET-07/2005, OTKA K-73688, and TECH_08-A2/2-2008-0089-SZOMIN08.

István Siket

Department of Software Engineering, University of Szeged, H-6720 Szeged, Árpád tér 2., Hungary
e-mail: siket@inf.u-szeged.hu

1 Introduction

A fair number of object-oriented metrics have been defined and published in the past few decades. The essence of these metrics is that they characterize certain properties of the programs in numeric form, hence the values of different programs or the parts of programs (e.g. classes) can be compared. From these values we can infer things about them. However, before doing this it is necessary to find the relationship between the metrics and the software quality (e.g. fault-proneness or reusability) via concrete empirical results, and it is not enough to try to discover these relations just by using the definition of each metric. Unfortunately, there are very few publications to date in which the usefulness of metrics have been studied and evaluated. The main reason for this is that determining the quality of large program is difficult, and so is calculating the metrics for it.

A sizable part of the cost of software development arises from testing, and within it, finding software faults. Therefore, anything which increases the efficiency of testing is helpful as it can decrease the overall development cost. Consequently, in this paper we shall focus on the issue of the fault-proneness of the classes of a program. The fault-proneness of the classes can be characterized by the number of bugs found and corrected in the past during development. Hence, the fault-proneness quality of the classes can to some extent be expressed in numerical or statistical terms.

There are various methods available for examining the relationship between object-oriented metrics and software quality (fault-proneness) represented by a number. Usually, regression analysis and machine learning are applied to investigate this problem [3, 6, 10, 12, 15, 18, 19]. In a previous paper [12], we applied regression analysis together with neural networks and decision trees (both are machine learning methods), but we did not find any significant difference in the results. For this reason we shall use only one of them, namely regression analysis.

We employed our own reverse engineering framework called Columbus [8] to analyze the source code of Mozilla [14]. We developed a compiler wrapping method [9] to be able to automatically analyze the software system, which means that we can extract the necessary information without modifying anything in

the source code. This way, we analyzed nine versions of Mozilla and calculated 58 object-oriented metrics.

The bugs reported during the development of Mozilla are stored in the Bugzilla database [4]. We collected the corrected bugs from Bugzilla and associated them with the classes in order to see how many bugs the classes contained. We used regression analysis to examine the relationship between the object-oriented metrics and the number of bugs found in the classes. We also investigated what kinds of metrics were good at predicting the fault-proneness of the classes.

We will show that 17 of the 58 examined object-oriented metrics were found to be statistically significant. The coupling metrics (e.g. CBO and RFC) seemed the best predictors for indicating the fault-proneness of classes, but the complexity and the size-related metrics also yielded significant results.

We will proceed as follows. In the next section, we will describe how we extracted the data from the source code of Mozilla and how we collected bugs from Bugzilla. In Section 3, we will provide the necessary technical background about metrics and logistic regression. In the next section, we will present the results of our statistical analysis, and list our conclusions about the metrics and compare the actual results with the results of an earlier paper [12]. In Section 5, we will discuss several other articles which tackled the same problems. In Section 6, we will present out main conclusions and then outline our plans for future study.

2 Fact Extraction from Mozilla and Bugzilla

It is a rather difficult and complex task to extract facts from a real-world system's source code and for the present, we shall outline just the main difficulties we had to overcome during the analysis of Mozilla source code. The first step of the build process is the configuration where all the necessary information about the environment (e.g. operating system, existence and version of the tools required to build the system) is verified, collected and stored into files. Later all this information is taken into account when building the system. After the configuration, the system can be built automatically because the information on how to build it is also part of the source code and it is stored in separate files (makefiles or project files). Unfortunately, these files can be very different and can store almost any kind of data, hence it would be a great challenge to "understand" them. We will describe the main idea behind our solution to these problems and also how we calculated the necessary metrics. For a more detailed description of all this, see Ferenc et al. [9].

Besides analyzing the source code of Mozilla, we collected the bugs reported and corrected during its development. These bugs were reported into Bugzilla [4], which stores detailed information about the bugs. Bugzilla consists of a Web interface used to notify users of a new bug or to manage the existing ones and an SQL database which stores the data. Extracting data from an SQL database is a straightforward task, and first we chose this approach of bug collection [12]. We were able to do it because the developer community of Mozilla provided us with this database, but the great drawback of this approach is that if we need more up-to-date information about the bugs we have to repeatedly ask for the database. To solve this problem we improved our bug collection toolset so that we could collect bugs using the Web interface of the official Bugzilla of Mozilla. In the second part of this section, a detailed description is given about the bug extraction method and we will list the number of bugs for the classes of Mozilla 1.6.

2.1 Source Code Analysis

Columbus [7, 8] is a reverse engineering framework that was developed in cooperation between the University of Szeged, the Nokia Research Center and FrontEndART Software Ltd. [11]. The main motivation behind developing the Columbus framework was to create a general toolset which supports fact extraction 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.

The main idea of *wrapping* is that we temporarily hide the original compiler by inserting the directory of the wrapper toolset at the beginning of the PATH environment variable. In this directory, the script files have the same names as the executables of the compiler, hence if the original compiler is invoked, one of its wrapper scripts will start instead. The scripts first execute the original compiler tool (e.g. *g++* or *ld*) 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. Based on this technique, an arbitrary system can be analyzed without modifying it. In fact, we successfully applied it with other industrial and open-source C/C++, Java and C# projects and on both Linux and Windows platforms. A more detailed description about wrapping was presented earlier by Ferenc et al. [7].

After analyzing Mozilla's source code we calculated the metrics needed. 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 58 calculated metrics. We repeated this whole process for all the nine Mozilla versions (1.0 – 1.8) investigated by us and created a table for each version.

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

2.2 Mining Bugs from Bugzilla

In order to carry out our analysis we also had to collect the number of bugs found and corrected in each class of the system in all of the analyzed versions from 1.0 (released in June 2002) to 1.8[1] (released on February 26, 2005). We will describe the heuristic used in our approach.

Since the bug database contains all the known bugs about every Mozilla product from the beginning of its development, we first had to filter them. We analyzed just those bugs which were reported for Mozilla[2], which were `FIXED` and had at least one non-obsolete[3] patch file associated with them. They were reported before July 2005 (four months – which is a typical release period – later than the release date of Mozilla 1.8, February 26, 2005) and they were corrected (more precisely, the last modification date) only after the release date of Mozilla 1.0, meaning after June 5, 2002 (see Fig. 1). In addition, we examined `DUPLICATED` bugs as well to improve the accuracy of our heuristic. A bug is labeled as a `DUPLICATED` bug of another bug if both of them describe the same bug. In this case we refined the information about the bug (e.g. reporting date, correction date) with the data of `DUPLICATED` bugs, and if the `DUPLICATED` bug had any patch file, this patch file was also assigned to the bug. This way, we collected bugs more precisely (and refined the method we used earlier [12]). The result was 10,503 bugs in all.

Our earlier solution [12] for bug extraction, where the bugs were extracted from the SQL database of Bugzilla, had certain limitations because if we wanted to repeat the experiment several months later we had to ask for the database again, which was very circuitous and sometimes required a lot of effort from the owner of the database (in this case, from the community of Mozilla). We wanted to overcome this drawback of the process, so we developed a toolset which collected bugs over the Internet directly from Bugzilla. We were able to do it by creating a URL which contained the address of Bugzilla and all the filter conditions (e.g. product, `FIXED`, date) in a special form. We inserted an additional parameter into the URL to get the search result in an XML form. When we got the XML file, it was processed, and all the bug *id*s (identifiers) were collected. Then all the bugs and their details were downloaded in XML form as well. By processing XMLs of bugs we got all the data we needed about them including the *id*s of the patch files (similar to bugs, every patch file has a unique identifier), so the necessary patch files could be downloaded as well.

The drawback of this method of bug collection is that it is very slow and it overburdens Bugzilla and its infrastructure (e.g. the SQL database, the Bugzilla server or SQL server it is running on) a lot on the first run. For example, it took days to collect all the information about the bugs required for Mozilla analysis. On the other hand, if we have already collected the bugs this way, we can archive them and if we want to repeat or extend this bug collection later, we do not have to repeat the whole process again. It is enough to collect the "difference" (the new or changed bugs or patch files), which is much faster and does not adversely affect Bugzilla.

After collecting information about bugs and getting all the patch files, we had to find out which classes were affected by the bugs and in which version. By analyzing the patch files we located the bugs within a part of the source code. A patch file contained the name of the fixed 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. With these four numbers we defined an interval of changes in the file for localizing the bug.

We wanted to associate the bugs with the classes in concrete release versions, but the bug report in Bugzilla did not explicitly state which version the patch file had been 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. 2). As regards 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 with the classes found in the source code. We did this one at a time for each bug in each affected version. More precisely, we examined the bugs one by one and in each given version we looked for a class whose interval in the source code fell within the interval of the bug. If we found such a class, we increased the number of bugs in that class. If the bugfix changed more than one class, the bug was associated with all these classes. With this method we extended each table mentioned in the previous section with a new column containing the number of bugs for each class.

To carry out a precise bug analysis, first we had to analyze each patch file to find out the Mozilla revision the patch files had been applied to. (We could do it because this information was also stored in the patch file.) After getting the revision number we had to analyze the source code of this revision and associate the patch file with this revision to get the precise result. Taking into account the fact that we had over 10,000 bugs and a bug may have more than one patch file, running the algorithm would be very time-consuming and would require a lot of resources (e.g. cpu time and free space on hard disk). Therefore we choose a faster and easier way, and associated bugs using just the main release versions of Mozilla. We knew that in this case the result might contain several imprecise values, but this did cause a problem here.

Although we had every metric value and bug numbers for each nine version, using just one of them was sufficient to carry out our experiments. We chose version 1.6 because it had a lot

[1] We analyzed Mozilla 1.8 Beta 1 because there was no non-beta release of version 1.8.

[2] Bugzilla contains the bugs of all the products developed by the Mozilla community (e.g. Sunbird, Rhino and Bugzilla itself), but as we only needed the bugs of Mozilla, we filtered it as well.

[3] If a patch file is created for a given bug but it is not used later (e.g. it does not properly correct the bug or a better one is created), it is marked as obsolete.

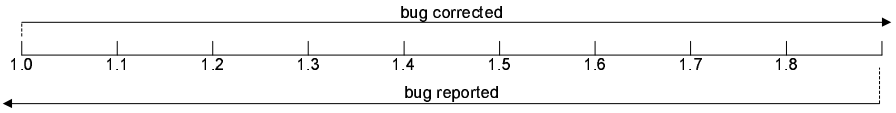

Fig. 1. Interval restrictions of bug extraction

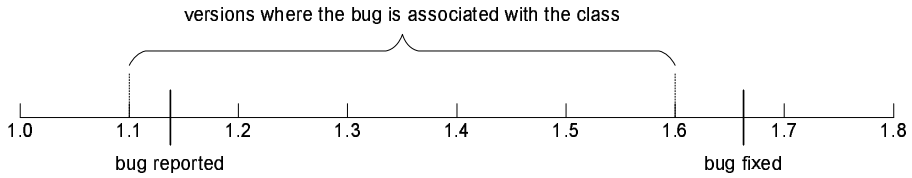

Fig. 2. Mozilla versions where a bug is associated with a class

of bugs and we had already examined it.

Although the sum of the number of bugs in Mozilla 1.6 classes was 7,662, this does not mean that there were this many different bugs because a bug can be associated with one or more classes. If we take into account the fact that we associate bugs just with classes (which are object-oriented constructs, written in C++) and that in Mozilla there are many C source files as well (about 1,550 out of the approximately 5,700 C/C++ source files), then this result is acceptable.

Tab. 1. Distribution of the bugs in version 1.6

No. of classes	%	No. of associated bugs
1,284	40.01	0
760	23.68	1
415	12.93	2
197	6.14	3
115	3.58	4
92	2.87	5
71	2.21	6
35	1.09	7
35	1.09	8
28	0.87	9
25	0.78	10
23	0.72	11
16	0.50	12
12	0.37	13
14	0.44	14
11	0.34	15
31	0.97	16-20
12	0.37	21-25
10	0.31	25-30
10	0.31	31-40
5	0.16	41-50
6	0.19	51-60
2	0.06	60-72
3,209	100.00	7,662

As a last step, we filtered out the classes which were generated on-the-fly during compilation because there were no bugs associated with them. We also filtered out all the bug-free classes which existed in each of the nine analyzed versions of Mozilla and where none of the metrics had changed. This way we arrived at 3,209 remaining classes (out of 3,624 extracted classes[4]) for Mozilla version 1.6.

We used these classes in our other analysis described in the following sections. Also we summarized our findings in Table 1 above. About 40% of the classes (1,284 of them) contained no bugs at all, and about one fifth (760 of them) contained just one bug.

3 Background

In this section, we will describe the types of metrics used in our experiments, some basic theory of logistic regression and state some important definitions. We should add that in an earlier paper [12] we wanted to compare our results with those of Basili et al. [3]. This was why we used their terminology to represent the different qualities of the models. Here we will use the well-known information retrieval terminology [16]. For each definition, we will give the corresponding pair we used earlier [12] to aid the comparison.

3.1 Metrics

We calculated 58 object-oriented metrics for the classes, but due to lack of space Appendix A just contains the definitions of 18 of them (which proved useful in our analysis). The 58 metrics were classified into five groups based on their detailed properties.

We have 30 *size*-related metrics which usually tell us elementary things about the system by counting the number of given items (e.g. lines, classes, methods, attributes). The main advantage of these metrics is that they are easy to calculate (or at least easier to calculate than the others) and it is not difficult to understand them. A good example of this is the traditional and well-known lines of code metric (LOC).

Complexity metrics measure some kind of complexity of the program or a given part of the program. Although many different complexities of a class can be defined and measured (e.g. data complexity), here we shall calculate just one complexity metric

[4]The C/C++ analyzer has been improved since our last experiment [12], hence the number of classes is now slightly different.

(WMC), which is the sum of the McCabe cyclomatic complexity (McCC) of the methods of the class.

Inheritance metrics (we calculated 8 of them) provide information about the inheritance tree of the system. Though an inheritance tree cannot be reconstructed based on these values, they describe its main characteristics, which is sufficient to express the inheritance complexity of the system.

One of the aims of object-oriented techniques is encapsulation, which suggests that data belonging together and the functions that operate on them should be incorporated into one unit called a class. *Cohesion* metrics (we examined 11 of them) measure whether a class implements just one functionality. Otherwise, there is weak cohesion among its members (methods and attributes). Weak cohesion occurs when the class stores data or implements a functionality which does not belong to it. In extreme cases, the members can be classified into two or more groups without any connection (e.g. call or attribute access) between them, which suggests that the class should be redesigned and refactored.

Yet another indication of a bad design is when the classes use each other too much. They can do this in many different ways e.g. when a class calls the methods of other classes or uses the attributes of other classes. The 8 *coupling* metrics we calculated here measure these kinds of relations.

3.2 Logistic Regression Analysis

In logistic regression, the unknown variable called the *dependent variable* can take just two distinct 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.

The multivariate logistic regression model is based on the relationship equation

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

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 elaborate on it here (a detailed description is given by Basili et al. [3], and Hosmer and Lemeshow [13]). Univariate logistic regression is a special case of multivariate regression for situations when there is just one explanatory variable in the model.

First, we performed univariate logistic regression (the results are given in Table 3). The *coefficient* (not shown in Table 3, but discussed later on) is the estimated regression coefficient. The larger the absolute value of this 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* tells us whether the result is statistically significant or not (we chose a significance level of $\alpha = 0.05$). For these 17 metrics, the *p-value* was always less than 0.001. 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 that is explained by the explanatory variables.

3.3 Definitions

Apart from the statistical figures, logistic regression gives us a model for each classification. Thus for each classification we know the values of the coefficients (C_1, ..., C_n) and the constant (C_0) of formula (1). We applied these models with a 0.5 threshold, which means that if $0.5 < \pi$ then the class is classified as faulty, otherwise it is not considered faulty. Table 2 shows the classification results for the CBO metric where the figures in brackets are the sums of the faults that were found in that particular group of classes.

Tab. 2. Classification results obtained by using CBO

	Predicted	
Observed	Not faulty	Faulty
Not faulty	757	527
Faulty	478 (695)	1,447 (6,967)

We can see from this table that 2,204 (757+1,447) of the 3,209 classes were classified correctly, which means that the *correctness* of the model is 68.68% (2,204/3,209). (In an earlier paper [12], we used the term *precision* instead.)

From a testing point of view there are three more important quantities which say more about the quality of the models. The first one is the *precision* score, which describes how many of the faulty predicted classes are really faulty in percentage terms. That is, it is the number of classes observed and predicted faulty divided by the number of all faulty predicted classes times 100. The larger the precision score, the fewer error-free classes have to be tested, which in turn improves the efficiency of the testing phase. In this case, 1,974 (527+1,447) classes were predicted as faulty and 1,447 of them were really faulty, meaning that the precision score was 73.03%. (In an earlier paper [12], we used the term *correctness* instead.)

The second one is *recall*, which tells us how many of the faulty classes have been found in percentage terms. That is, it is the number of classes observed and predicted faulty divided by the number of all faulty classes times 100. The larger the recall score, the more faulty classes have been found by the model. In the case of CBO, 1,447 classes were predicted as faulty out of the 1,925 (478+1,447) really faulty classes, which means that the recall score was 75.17%. (Previously in [12], we did not calculate this.)

In spite of the fact that the models classify the classes by just taking into account the fact whether a class contains a bug or not, we know precisely how many bugs they found in percentage

terms. This is also helpful for us. This quantity is characterized by a *completeness* parameter, which is defined as the number of faults in faulty predicted classes divided by the number of faults in all classes times 100. The larger the completeness score, the bigger the ratio of the faults captured by the model. In the case of CBO, 6,967 out of 7,662 bugs were found, which means that 90.93% of the bugs were captured. (In an earlier paper [12], we used the same terminology here.)

The *classified faulty* figure (column *C.f.* in Table 3) tells us how many classes out of the 3,209 the model classified as faulty (although Coh could be calculated just for 2,400 classes because it is defined for classes that have at least one attribute and at least one method (see Appendix A)). The *true positive* value (column *T.p.*) tells us the number of faulty classes which we predicted as faulty and really are faulty. These two figures are important from a testing point of view. By using the figures of Table 3, tables like Table 2 for the remaining 16 metrics can be created.

4 Results

In this section, we will examine the relationship between the metrical values and the number of bugs found in the classes. In spite of the fact that we knew exactly how many bugs were in a class, we just looked at whether a class was faulty (contained at least one bug) or not. Here, we employed logistic regression, which is commonly used for predicting an unknown variable based on one or more known variables.

Table 3 lists just the best 17 out of the 58 metrics, given in terms of a decreasing R^2 score. (Later we will explain why only these 17 metrics are listed here.) As all the p-values are less than 0.001, these metrics are significant. The coefficient of each metric except the Coh metric is positive, which means that the larger the metric value, the more fault-prone the class is. The negative coefficient of Coh displays an inverse relation between the metrical value and the fault-prone property of the class. More precisely, the larger the Coh value is, the less fault-prone the class is. According to the R^2 values, CBO is the best predictor but NOI is only slightly worse. On the other hand, the R^2 values do not tell us much about the bugs or the classes themselves, nor can we compare the usefulness of the metrics using the R^2 values alone. To do this, we examined other aspects in our regression analysis.

In the following we shall analyze the results of univariate logistic regression and then draw some conclusions.

4.1 Evaluation

In Table 3, we listed just 17 out of the 58 metrics because the models of the other 41 metrics classified fewer than one hundred classes (less than 3.2% of the classes) as not faulty. Hence there is little use analyzing their scores, and we can say that in the case of Mozilla they are not really suitable for predicting faults.

Although the R^2 scores dramatically decrease from the CBO to the Coh metrics, the correctness scores decrease by less than 5%, and the worst score of 63.79%, which is associated with LCOM4, is still better than the value of trivial classification (59.99%) where all classes would have been predicted as faulty. Interestingly, the correctness score of CBO (68.68%) is almost 9% better than that for a trivial classification.

LOC has the largest precision score (74.64%), meaning that most of the faulty predicted classes really are faulty. On the other hand, it has the smallest recall score – only 69.87% – so some 30% of the faulty classes are not captured. The score for lLOC is very similar, which is not surprising as both of them measure lines of the code but in a slightly different way. The precision score of CBO is just 1.5% worse than the score for LOC, but its recall score is over 5% better. Coh has the largest recall score – over 90% – but seeing that only 295 classes are predicted error-free its large value is not so helpful here. It would be good to somehow improve the precision and recall scores at the same time. This does not really contradict the definition of precision and of recall, but experience tells us that if precision is increased, recall will decrease, and vice versa. This general relationship can be seen in Table 3 as well (although the scores are in terms of descending R^2-values and not in terms of precision or recall).

Completeness can provide a better idea about how many bugs have been captured. The CBO model classified 1,974 (527+1,447, see Table 2) classes as faulty which is only 61.51% of all the classes, but these 1,974 classes contain 75.17% of the faulty classes (recall) and, what is more, 90.93% of the bugs (completeness). The correctness score of Coh is the biggest here (97.07%), but it classified 87.71% (2,105/2,400) of the classes as faulty. LCOM4 got the second biggest completeness score (94.39%), which is slightly worse, but it predicted 79.93% of the classes as faulty, which is almost 8% less than the value of Coh, hence it is better overall. The completeness scores of the other metrics vary from 88.79% to 91.80%, which is a small difference, so their strength depends on how many of the classes predicted faulty are really faulty in percentage terms. The fewer classes that are predicted to be faulty, the more valuable a big completeness score is.

Though we cannot choose the best metric in general here, we can still analyze their categories, which will tell us what kinds of metrics are useful. As we saw previously, coupling metrics have the largest R^2 scores, the largest correctness and a "balanced" precision and recall pair. And while the coupling metrics measure more or less different kinds of couplings between classes, it seems that any kind of dependency between classes increases the probability of faults. WMC, which is the only examined complexity metric, also yields good scores. This is natural because it is generally more difficult to create, understand and maintain a complex class than a less complicated one. The size metrics whose scores we found significant can be divided into two groups. The first group (LOC, lLOC and AvgLOC) counts the lines of the code of a class and its methods in different ways, and we think it is obvious that larger classes contain more faults. The second group of size metrics contains

Tab. 3. Results obtained using Univariate Logistic Regression

Metrics	Cat.	R^2	Corr.	Prec.	Recall	Compl.	C.f.	T.p.
CBO	coup.	0.186	68.68%	73.30%	75.17%	90.93%	1,974	1,447
NOI	coup.	0.184	67.90%	72.82%	74.18%	89.62%	1,961	1,428
RFC	coup.	0.176	67.68%	72.09%	75.27%	90.92%	2,010	1,449
$NFMA_{ni}$	coup.	0.174	66.87%	72.26%	72.68%	88.80%	1,936	1,399
WMC	compl.	0.161	67.28%	72.63%	72.94%	89.39%	1,933	1,404
NFMA	coup.	0.158	67.22%	72.19%	73.77%	89.27%	1,967	1,420
LOC	size	0.154	67.68%	74.64%	69.87%	89.05%	1,802	1,345
lLOC	size	0.147	67.19%	73.12%	71.64%	89.51%	1,886	1,379
NML	size	0.128	66.38%	69.93%	77.09%	91.16%	2,122	1,484
RFC3	coup.	0.125	67.93%	71.35%	77.77%	91.80%	2,098	1,497
NMLD	size	0.122	65.78%	68.94%	78.18%	91.48%	2,183	1,505
NAML	size	0.114	66.19%	69.37%	78.13%	91.56%	2,168	1,504
NML_{pub}	size	0.110	66.19%	69.23%	78.55%	91.36%	2,184	1,512
$NMLD_{pub}$	size	0.104	65.60%	68.52%	78.91%	91.24%	2,217	1,519
AvgLOC	size	0.090	64.07%	66.92%	79.32%	88.79%	2,228	1,527
LCOM4	coh.	0.085	63.79%	64.87%	86.44%	94.39%	2,565	1,664
Coh[5]	coh.	0.050	65.96%	67.36%	91.60%	97.07%	2,105	1,418

metrics which measure the number of methods (NML, NMLD, NAML, NML_{pub}, $NMLD_{pub}$). It was also shown earlier by Basili [3] that the more methods a class has, the more faults the class contains. However, we see that the number of public methods used is a more useful predictor than the number of protected or private methods (among the 58 metrics there were metrics which measured this as well). Only two of the 11 cohesion metrics (LCOM4 and Coh) gave acceptable, but not such good results. The negative coefficient of the Coh metric is in accordance with the positive correlation of LCOM4 because LCOM4 measures the lack of cohesion. Thus we think that the less coherent classes are more fault-prone. And finally, we did not find any inheritance-based metric which could help to predict the faults of the classes, though we examined metrics which took this into account, such as the number of parents (NOP), ancestors (NOA), children (NOC), descendants (NOD) and the depth of the inheritance tree (DIT).

We shall compare the main results of the above experiments with those of an earlier paper [12]. Previously, we found that CBO was the best predictor and the other coupling metric (RFC) was good as well. The results agree with our earlier findings about coupling metrics, and what is more, CBO is the best in both cases. In our earlier paper, we examined two size metrics, LOC and WMC[6], and found that LOC performed only slightly worse than CBO, and WMC also gave good results. This is similar to our new findings because many of the size metrics proved to be good predictors and some of them are only a little worse than the coupling metrics. So far we have found that only 2 of the 11 cohesion metrics are acceptable, which does not contradict the results of our earlier analysis where the two cohesion metrics also gave poor results. The fault-proneness capability of DIT in our earlier analysis produced the worst scores and the performance of other inheritance metric was not significant. Similarly, we found that all the inheritance metrics are not of much worth here either. Summarizing our comparison, we can say overall that the new experiment confirms our earlier conclusions about software metrics.

[5]Coh was calculated just for 2,400 of the 3,209 classes because it is defined for classes which have at least one attribute and have at least one method (see Appendix A).

[6]The weight was one so the WMC counted the number of methods instead of their complexity. Hence it is a size metric.

4.2 Multivariate Logistic Regression

In a multivariate logistic regression analysis we utilized all 58 metrics but we knew beforehand that these metrics were not totally independent and captured similar information about the classes. Thus, not all of them were required in the multivariate regression, so a stepwise selection was applied to select the necessary variables for the multivariate analysis. The chosen metrics were CBO, AvgLOC, RFC, CLD (see Appendix A for the standard definition of each metric). The R^2 value is 0.208, which is a little better than that for CBO (which is the best among the results of our univariate analysis). The correctness and precision scores of CBO are slightly better than those for the multivariate model, but the recall and completeness scores of CBO are slightly worse. The classified faulty (c.f.) and true positive (t.p.) scores are practically the same.

4.3 Metrics in Practice

Our main conclusion here is that with the help of metrics we can choose a set of classes of the system which contains the major portion of the bugs. This means that if we focus on testing these classes instead of uniformly testing the entire system, we can find more bugs with the same effort. This way we can improve the efficiency of the testing phase, which should then improve the quality of the software and reduce the cost of the testing phase.

Tab. 4. Results obtained using multivariate logistic regression

	R^2	Corr.	Prec.	Recall	Compl.	C.f.	T.p.
Multi	0.208	68.37%	72.89%	75.27%	91.14%	1,988	1,449

If we regularly calculate metrics for a given system, we can see how the metrics change over time, from which we can estimate the quality of the system without testing. Moreover, this way we can discover problematic parts of the source code as soon as they appear and, what is more important, before the testing phase. This would be of great help to the practicing software engineer.

5 Related Work

A number of studies have been carried out in the past fifteen years to discover the precise relationship between object-oriented metrics and the fault-proneness of classes. The first article about object-oriented metrics was published by Chidamber and Kemerer (CK) [5].

Basili et al. [3] examined the relationship between the six CK metrics and the fault-proneness of classes. They utilized a medium-size management information system which had been written in C++ by university students and consisted of 180 classes. They extracted CK metrics by GEN++ and applied logistic regression to examine the correlation between the identified bugs and their classes. They found that DIT, NOC[7] and RFC were very significant, CBO was significant, WMC was to some extent significant and LCOM was insignificant. Their conclusions and our conclusions about inheritance metrics seem to contradict each other, but their conclusions about the other four metrics are quite similar to ours.

Fioravanti and Nesi [10] analyzed the same project, but they looked at 226 metrics. One goal of theirs was to create a model whose precision score was at least 90% (meaning that it classifies over 90% of the classes correctly) and another was to reduce the number of metrics used to an acceptable amount while keeping the precision score for the model above 80%. They employed principal component analysis (PCA) and multivariate logistic regression, and first they got a model consisting of 42 metrics with the precision score of 97.35%. After reducing the number of metrics, their model contained only 12 metrics and its precision score was 84.96%. We examined their metrics and discovered that 5 of their 12 metrics are similar to ours. We found that 3 of them (RFC3, NAML, NML) were good predictors, but the other 2 metrics (LCOM1, LCOM2) were not significant. However, we did not calculate the other 7 metrics of theirs.

Olague et al. [15] studied the Rhino project [17], which is also an open-source product of Mozilla. It is written is Java and was developed using an agile software development process. They examined its six different versions, where the number of classes rose from one hundred to two hundred. They also used the Bugzilla of Rhino to collect the bugs and to associate them with the classes. They studied three groups of software metrics, namely the CK metrics [5], Brito e Abreus MOOD metrics [1], and the Bansiya and Davis's quality model for object-oriented design (QMOOD) metrics [2]. In their univariate logistic regression they found that the coefficient of CBO[8] was significant for five of the six cases, the LCOM98[9] coefficient was significant in four of the six cases, the RFC coefficient was significant in every case and the WMC[10] coefficient was significant in four of the six cases, but the other two CK metrics – DIT and NOC – were of little value. MOOD metrics coefficients were significant in only two of the six versions. CIS[11] was significant in five of the six versions. Summarizing these results, we can say that their general conclusion about univariate logistic regression is similar to ours here. They studied several different models built using multivariate logistic regression and they found that the CK metrics-based models were the best for predicting the quality of classes, but the QMOOD metrics-based models were just marginally worse. They evaluated the usefulness of the metrics and the models for the different versions as well and found that the CK and QMOOD metrics were better in the initial phase. They divided the classes into two groups based on their size, and they looked at how the metrics performed on small and large classes.

6 Conclusions and Future Work

The main contributions of this paper are the following. Firstly, 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. Secondly, we improved our earlier bug collection method [12] and now we are able to collect bugs directly from Bugzilla. Thirdly, we associated these bugs with classes found in the source code. Fourthly, we employed logistic regression to assess the suitability of 58 object-oriented metrics for predicting the number of bugs in the classes (in an earlier paper we evaluated just 8 metrics).

Our main observations are the following. First of all, the CBO metric performed best when predicting the fault-proneness of classes, and we can say in general that the coupling metrics are the best predictors. Secondly, WMC, which is the only examined complexity metric, is also a good predictor. Thirdly, the observation that the classical size metrics (LOC and number of

[7] In the case of NOC, there was an inverse correlation.

[8] Their definition for CBO differs slightly from ours.

[9] Their LCOM98 metric is equivalent to our LCOM4 metric.

[10] They used one for a weight instead of real complexity, so WMC there counted the number of methods.

[11] CIS is quite similar to our NML and NML_{pub} metrics.

methods) were useful predictors is not surprising and it is not a new result. Furthermore, we found that the number of public methods used also displayed some correlation with the fault-proneness of classes. Fourthly, only two of the 11 cohesion metrics were found to be slightly significant. Fifthly, we did not find any inheritance metric which was significant in our new study.

In the future we plan to do experiments with function and method-level metrics in order to learn about their bug prediction capabilities. In addition, we would like to examine open-source projects written in Java (e.g. Eclipse and Derby) to find out which metrics are better at predicting faults.

References

1 **Brito e Abreu F, Melo W L**, *Evaluating the Impact of Object-Oriented Design on Software Quality*, Proceedings of the third international software metrics symposium, IEEE Computer Society, March 1996, 90–99, DOI 10.1109/METRIC.1996.492446, (to appear in print).

2 **Bansiya J, Davis C**, *A Hierarchical Model for Object-Oriented Design Quality Assessment*, Ieee transactions on software engineering, IEEE Computer Society, January 2002, 4–17, DOI 10.1109/32.979986.

3 **Basili V R, Briand L C, Melo W L**, *A Validation of Object-Oriented Design Metrics as Quality Indicators*, Ieee transactions on software engineering, October 1996, 751–761, DOI 10.1109/32.544352.

4 *Bugzilla for Mozilla*, available at `http://bugzilla.mozilla.org`.

5 **Chidamber S R, Kemerer C F**, *A Metrics Suite for Object-Oriented Design*, Ieee transactions on software engineering 20,6(1994), 1994, 476–493, DOI 10.1109/32.295895, (to appear in print).

6 **Denaro G, Pezzè M**, *An empirical evaluation of fault-proneness models*, Icse 2002: Proceedings of the 24th international conference on software engineering, March 2002, 241–251, DOI 10.1109/ICSE.2002.1007972, (to appear in print).

7 **Ferenc R, Beszédes Á, Tarkiainen M, Gyimóthy T**, *Columbus – Reverse Engineering Tool and Schema for C++*, Proceedings of the 18th international conference on software maintenance (icsm 2002), IEEE Computer Society, October 2002, 172–181.

8 **Ferenc R, Beszédes Á**, *Data Exchange with the Columbus Schema for C++*, Proceedings of the 6th european conference on software maintenance and reengineering (csmr 2002), IEEE Computer Society, March 2002, 59–66, DOI 10.1109/CSMR.2002.995790, (to appear in print).

9 **Ferenc R, Siket I, Gyimóthy T**, *Extracting Facts from Open Source SoftwareExtracting Facts from Open Source Software*, Proceedings of the 20th international conference on software maintenance (icsm 2004), IEEE Computer Society, September 2004, 60–69, DOI 10.1109/ICSM.2004.1357790, (to appear in print).

10 **Fioravanti F, Nesi P**, *A Study on Fault-Proneness Detection of Object-Oriented Systems*, Fifth european conference on software maintenance and reengineering (csmr 2001), March 2001, 121–130, DOI 10.1109/CSMR.2001.914976, (to appear in print).

11 *FrontEndART Software Ltd.*, available at `http://www.frontendart.com`.

12 **Gyimóthy T, Ferenc R, Siket I**, *Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction*, Ieee transactions on software engineering, IEEE Computer Society, October 2005, 897–910, DOI 10.1109/TSE.2005.112.

13 **Hosmer D, Lemeshow S**, *Applied Logistic Regression*, Wiley-Interscience, 1989.

14 *The Mozilla Homepage*, available at `http://www.mozilla.org`.

15 **Olague H M, Etzkorn L H, Gholston S, Quattlebaum S**, *Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes*, Ieee transactions on software engineering, June 2007, 402–419, DOI 10.1109/TSE.2007.1015.

16 *Precision and Recall Wiki*, available at `http://en.wikipedia.org/wiki/Precision_and_recall`.

17 *Rhino Home Page*, available at `http://www.mozilla.org/rhino`.

18 **Subramanyan R, Krishnan M S**, *Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects*, Ieee transactions on software engineering, April 2003, 297–310, DOI 10.1109/TSE.2003.1191795.

19 **Yu P, Systä T, Müller H**, *Predicting Fault-Proneness using OO Metrics: An Industrial Case Study*, Sixth european conference on software maintenance and reengineering (csmr 2002), March 2002, 99–107.

A Definition of the metrics mentioned above

- *Coupling between object classes (CBO)*: A class is coupled to another if the class uses any method or attribute of the other class or it directly inherits one of these from it. CBO is the number of coupled classes.

- The *number of outgoing invocations (NOI)*: NOI for a class is the cardinality of the set of all function and method invocations in the class of its methods. This means that if there is more than one invocation to the same function or method, they are counted just once. Invocations in a nested or local class are not counted.

- The *response set for a class (RFC)*: RFC is the cardinality of the set M of the methods of the class (inherited ones are not taken into account) and the set of the methods directly invoked by the methods in M.

- The *number of foreign methods accessed (without inheritance) ($NFMA_{ni}$)*: $NFMA_{ni}$ for a class is the cardinality of the set of method invocations of any of the class' methods, where the invoked methods belong to classes other than the method itself, or they are inherited methods (thus only the locally defined methods are not counted). This means that if there is more than one invocation to the same method, they are counted only once.

- *Weighted methods for a class (WMC)*: The weight is the McCabe cyclomatic complexity. Consider a class C with methods $M_1, \ldots, M_n$ that are defined in the class. Let $c_1, \ldots, c_n$ be the McCabe cyclomatic complexity of the methods. Then WMC = $c_1 + c_2 + \cdots + c_n$.

- The *number of foreign methods accessed (NFMA)*: NFMA for a class is the cardinality of the set of method invocations of any of the class' methods, where the invoked methods belong to classes other than the method itself (the inherited methods are not counted either). This means that if there are more than one invocation to the same method, they are counted only once.

- *Lines of code (LOC)*: This metric is calculated using the original source files. Here the lines between the beginning and the end of a class or a method are counted regardless of the number of lines in the file after preprocessing. LOC for a

defined class counts the end-line of the class definition minus the begin-line of the definition plus 1. In addition to this, the LOC metric of a method of the class is also added if the method is implemented outside the class definition. If there is a nested class (struct or union) in the class or a local class in one of its methods, the LOC of this class is decreased by the length of the nested/local class (struct or union), which is the end line of the nested class (struct or union) minus the begin line of the class (struct or union) plus 1.

- *Logical lines of code (lLOC)*: This metric is calculated after preprocessing the file (after files have been included, macros have been substituted, etc.). lLOC for a defined class counts all nonempty, non-comment lines of the class and all its methods implemented outside the class definition (but the lines of the local classes are not counted).
- The *number of local methods (NML)*: NML is the number of local methods. Both declarations and definitions are counted, but if there is an implementation for a declaration, the declaration is not counted.
- The *response set for a class (without inheritance) (RFC3)*: RFC3 is the cardinality of the set M of the methods of the class (the locally defined and the inherited ones are taken into account), and the set of the methods directly or indirectly invoked by the methods in M.
- The *number of locally defined methods (NMLD)*: NMLD is the number of methods locally defined (the inherited methods are not counted).
- The *number of locally defined attributes and methods (NAML)*: NAML is the number of the locally defined attributes and methods of a class.
- The *number of public local methods (NML_{pub})*: NML_{pub} is the number of the public local methods of a class. Both declarations and definitions are counted, but if there is an implementation for a declaration, the declaration is not counted.
- The *number of public locally defined methods ($NMLD_{pub}$)*: $NMLD_{pub}$ is the number of the public methods locally defined.
- *Average of lines of code (AvgLOC)*: the AvgLOC of a class is the average of the lines of the method definitions in the class.
- *Lack of cohesion in methods (LCOM4)*: Consider an undirected graph G, where the vertices are the methods of a class, and there is an edge between two vertices if the corresponding methods use at least one attribute in common. LCOM4 is defined as the number of connected components of G.
- *Cohesion (Coh)*: Consider a set of methods $\{M_i, i = 1, \ldots, m\}$ accessing a set of attributes $\{A_j, j = 1, \ldots, a\}$. Let $v(A_j)$ be the number of methods which reference attribute A_j. Then Coh = $\frac{\sum_{j=1}^{a} v(A_j)}{m*a}$.
- *Class-to-leaf depth (CLD)*: the CLD of a class is the maximum number of levels in the hierarchy that are below that class.