University of Szeged
Department of Software Engineering

Applying Software Product Metrics in Software Maintenance

Summary of the Ph.D. Thesis

István Siket

Supervisor:

Dr. Tibor Gyimóthy

Szeged
2010

Introduction

One of the biggest challenges of software engineering is the difficulty of fulfilling the customers' precise demands. Large and complex systems have to be developed in a short time, and after delivering them to the customer, they have to be maintained and improved based on the new requirements. Due to the strict deadlines and the complexity of problems that have to be solved, developers cannot always choose the best solution for each problem, so the quality of the software system may deteriorate over time. Since quality, which is important from the customer point of view (e.g. that of usability or reliability), must be maintained to a high level, developers may sacrifice the "internal" quality (e.g. maintainability) as it is less important for a customer. But the "internal" qualities are as important as the others, and none of them should be neglected during the lifecycle of the software system. For example, the cost of finding and correcting a bug, or the cost of implementing a new feature is heavily dependent on the maintainability of the software system. If we do not look after our software system over time, we cannot maintain or develop it efficiently. Short-term goals like quality should not be strived for at the expense of long-term goals like maintainability.

In the thesis we are principally concerned with software maintenance, including testing and discovering a software system's bugs. Since a significant part of software development costs arises from this, anything that increases the efficiency of testing is helpful, so it can decrease the overall development cost. Previously many studies confirmed [1, 3, 16] that there is correlation between the object-oriented metrics and the faults found in the classes. This means that with the regular measurement and analysis of object-oriented metrics, the quality and the maintainability of a software system can be improved and its testing can be made more efficient. They can have a positive impact on the development of a software system and best practice of the software development community.

The first problem is that if experiments were carried out on small- or medium-sized systems, relationships cannot be validated in an industrial environment. The second problem is that if large and complex systems cannot be analyzed, we will not be able to calculate their metrics values, which is indispensable for the practical application of the results. Without first solving these difficulties, the set of metrics cannot be usefully applied in the software engineering process.

In this thesis we present a solution for both these problems. First, the Columbus technology is introduced which allows us to analyze a large software system and to calculate the object-oriented metrics even for the largest systems encountered by us. For the validation of the metrics, seven different versions of Mozilla [9] were analyzed and the metric values were calculated. All reported and corrected bugs were collected from the bug tracking system of Mozilla called Bugzilla [4] and by applying our heuristic, they were associated with classes. This way, we demonstrated that our technique was of good practical use, but what was more important was that we had all the necessary data to carry out our own experiments. First, we examined the object-oriented metrics, defined by Chidamber and Kemerer [5], and the standard lines of code metric. Later we extended these in our experiments, where the aim was to examine metric categories instead of the metrics themselves. Using these results, we were able to construct metric-based quality models to aid software development.

Afterwards, the practical aspect of these metrics was examined by asking the developers for their views about the relationship between four metrics and software comprehension & testing. The findings raised several issues connected with the practical use of the metrics, and led us to conclude that we cannot just give the metrics to the developers without some sort of proper training. The other beneficial outcome of the survey was that its results can be used to improve our quality models.

The four main results of the thesis are the following:

1. **The analysis of large C++ software systems, developing a language-independent model for a high-level representation of object-oriented languages, and calculating object-oriented metrics using this model. We developed a heuristic that associates corrected bugs with classes.**
2. **We developed fault-predictor models based on object-oriented design metrics.**
3. **We analyzed the fault-prone properties of the object-oriented metric categories.**
4. **We examined the views of software developers and applied the results to improve our quality models.**

1 The Columbus technology

To be able to examine the relationship between the object-oriented metrics and the number of bugs, we have to calculate these values. First, we will present a solution for the analysis of any large and complex software system and we will describe a language-independent object-oriented model that allows us to readily calculate object-oriented metrics even for large systems. Nine Mozilla versions were analyzed and the metric values were calculated for each one, and with the help of our heuristic, the reported and corrected bugs were associated with the classes of Mozilla.

The Columbus framework

It is a rather difficult and complex task to analyze the source code of a real-world system. The source code is usually logically split into several files and these files are then arranged into folders and subfolders. All the information about how these files are related to each other and what settings are applied to them is usually stored either in makefiles or in different project files. All this information is taken into account when building the system. 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. This is only one example of the many problems that have to be addressed if we would like to analyze a real-world system in detail.

After examining several software systems, we developed the *Columbus framework* [6] which allows us to automatically analyze and extract information from an arbitrary software system without modifying anything in its source code. Although the first version could only analyze C++ systems compiled with GCC on the Linux platform, now it supports the Windows platform and Visual Studio compiler as well; furthermore, Java and C# languages can also be analyzed. Many large open-source (e.g. Mozilla [9] and OpenOffice.org [11]) and industrial systems, among which the largest one contained 30 million lines of code, were analyzed successfully (and the metric values were calculated), which indeed proves the feasibility and the power of the Columbus technology.

Information obtained from an analysis can be used for a variety of re- and reverse engineering purposes [13]. Now we will describe how the object-oriented metrics were calculated.

Language Independent Model

We worked out a *Language Independent Model* (LIM) that is a high-level representation of an object-oriented system. The model consists of four main packages. The *base package* contains general classes like the common base class of all other classes in the schema. Another example may be the class that represents comments of the source code. The *physical package* contains classes that can be used to represent the physical structure of the system; more precisely its files, folders and their relations (for example, a file that includes another one). The main object-oriented elements (e.g. packages, classes, and methods) of a system and their properties and connections can be represented by the classes of the *logical package*. Furthermore, some low-level information, which is necessary for metric calculations (e.g. the number of branches of a method), but cannot be calculated using LIM, is also stored in the appropriate class as an attribute. The *type package* combines the type representation of the C++, Java and C# languages in a slightly simpler form, so all but "the most rarely used" types can be built from these classes. Such a precise type representation is not necessary for metrics, but, for example, source documentation is generated from LIM as well, and it should represent the types correctly.

One of the main advantages of LIM (in contrast to the language representation model) is that it uses much less memory so huge systems, which cannot be handled in one C++ schema, might be represented in LIM. The other benefit is that LIM stores only high-level information, therefore the tools can calculate object-oriented results faster and easier. If we implement a new exporter on LIM, it works for all languages that can be represented in LIM; so by doing this we only have to maintain one program.

We implemented the conversion from C++, Java and C# languages to LIM, so it can be used, among other things, for calculating object-oriented metrics for these languages. Currently, 16 system level, 65 class level, and 17 method level metrics can be calculated using our program.

Analysis of Mozilla

Mozilla, which is an open-source Web and e-mail suite, was the first industrial system we analyzed successfully. During our first experiments seven different Mozilla versions (from 1.0 to 1.6) were chosen and all the metric values were calculated. All Mozilla versions contained over 1 million non-empty and non-comment lines of code, over 4,500 classes, and almost 70,000 methods.

The next step was to determine the number of bugs found and corrected in the classes of Mozilla. We devised a heuristic to collect the necessary bugs from the bug tracking system of Mozilla (called Bugzilla [4]) and to associate them with the classes. The developer community of Mozilla provided us with the full Bugzilla database, which contained all the reported bugs since the beginning of its development. The Bugzilla database contained 256,613 different bug entries, but not all of these were of interest to us. Hence we filtered out those bugs which were reported for other Mozilla products (231,021 bugs remained) but were not fixed (57,151 remained) and those bugs that did not contain a patch file that may have corrected the bug (22,553 remained). Those bugs that were fixed before version 1.0 of Mozilla were also filtered out (9,539 remained) and similarly, those bugs that were reported after Mozilla 1.7 were filtered out as well. We found 8,936 different bug entries, which fulfilled all of these requirements.

We associated the bugs with classes in concrete release versions by using the reporting and the

correction date. We considered a bug to be present in the actual release at the time when it was reported and in all subsequent releases up till the date of the fix. The patch files were used to find out which classes were affected by the bug. This way we could associate 3,961 bugs with classes of Mozilla 1.6, which was used in our first experiment (Section 2). The results were validated by hand on random samples and we found that the heuristic could assign the bugs to the correct classes.

One big drawback of this method was that it needed the Bugzilla database. Therefore we improved our method to make it able to collect the same information over the Internet. Later we repeated our analysis procedure with nine Mozilla versions, but this time we collected the bugs with the new method. In our second investigation (Section 3) we utilized the results of the extended analysis.

Own contributions

We developed the Columbus framework, which can analyze a real-sized software system. We introduced the Language Independent Model, which is suitable for representing object-oriented languages, and we used it for metric calculations. We created a heuristic to associate the reported and corrected bugs with the classes of Mozilla.

The Columbus framework and the analysis of Mozilla are not the author's work. But the idea and the development of the Language Independent Model, the conversion from C++ language to this model, and the metric calculation using the Language Independent Model are the work of the author. The bug collection from the bug tracking system of Mozilla and the heuristic used for associating the bugs with classes are also the author's work.

2 Fault-predictor models based on object-oriented metrics

Previous studies had already validated the fault-prone capabilities of metrics, but we wanted to test their results for a large industrial system. For this experiment eight metrics were selected. Six of these metrics were first presented by Chidamber and Kemerer [5], and this metric suite is one of the most frequently examined and applied software tools. The seventh metric was LCOMN, which was a modified version of one of the six metrics. We examined the non-empty and non-comment lines of code metric as well to learn the difference between the traditional size metric and the object-oriented metrics. Each of these metrics may be defined in the following way:

- *Number of Methods Local (NML)* is the number of methods defined in a class.
- *Number Of Ancestors (NOA)* measures the number of ancestors of a class.
- *Number Of Children (NOC)* is the number of direct descendants for each class.
- *Coupling Between Object classes (CBO)* is the number of coupled classes. 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.
- *Response set For a Class (RFC)* is the cardinality of the set, which contains the methods of the class and methods directly invoked by them.
- Lack of COhesion in Methods (LCOM) is the number of member function pairs without shared instance variables, minus the number of member function pairs with shared instance variables. The metric is set to zero whenever this subtraction result is negative.

- *Lack of COhesion in Methods allowing Negative value (LCOMN)* is calculated in the same way as LCOM, but its value is not set to zero when the subtraction result is negative.
- *Logical Lines Of Code (LLOC)* of a class is the number of all non-empty, non-comment lines of the body of the class and all of its methods.

Although we calculated the metrics values and determined the number of bugs for each given Mozilla version, only version 1.6 was used for the analysis. Since the source code of Mozilla contained classes which were generated on-the-fly during compilation, these classes could not contain any bug. Therefore these classes were filtered out. Similarly, 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 for Mozilla version 1.6, which contained 3,961 bugs.

Four different methods were applied to investigate the metrics and all the results were used to judge the metrics as well. In the case of Mozilla there were several classes that contained a lot of bugs, thus we applied *linear regression* [10] in situations where the number of bugs could be predicted. First, the metrics were analyzed one by one, with univariate linear regression, to see which metrics could be used for bug prediction. Then we tried to improve the models by using more metrics together, so multivariate analysis was applied as well. Table 1 shows the results of applying linear regression. We can see that for the NOC metric, there is no apparent correlation between the metric values and the number of bugs. On the other hand, all the other metrics were able to predict bugs (p-values are less than 0.01) but to a different degree. The best predictor was CBO (it has the highest R^2 value), but LLOC was only slightly worse. As we expected, applying more metrics together in one bug predictor model, we can achieve a much better result (the R^2 value of multivariate regression is much better than any other value).

	NML	NOA	NOC	CBO	RFC	LCOM	LCOMN	LLOC	Multiv.
p-value	0.000	0.000	0.728	0.000	0.000	0.000	0.000	0.000	0.000
R^2	0.321	0.139	0.000	0.349	0.280	0.210	0.157	0.342	0.430

Table 1: Results of linear regression

With the other three methods (*logistic regression* [8], *decision trees* [12], and *neural networks* [2]) the fault-prone property of the classes was examined, therefore the classes were divided into two groups, where the first group contained the faulty classes and the other one represented the faultless classes. In all three methods the metrics were first examined individually and then the extended model with more metrics was also tested. Though these methods proved incapable of predicting the number of bugs, if we count the faults in the classes classified as faulty by the model, we can still determine how many bugs the given model found.

To be able to compare the results of the different methods, we need to calculate three scores. The *correctness* score tells us how many classes were classified correctly by the model. The *precision* score describes what percentage of the faulty predicted classes is really faulty. The *completeness* score tells us what percentage of the faults was found by the model in practice.

Table 2 summarizes the results of the models. The *CBO (Coupling Between Object classes)* metric was the best predictor in linear regression and the correctness, precision, and completeness scores of CBO are almost always the best among the metrics. Moreover, the correctness scores of CBO on its own are better than those of the multivariate models, and the precision and completeness scores are

Metric	Model	Correctness	Precision	Completeness
NML	Log. reg.	65.38%	68.84%	55.24%
	Dec. tree	66.51%	62.34%	65.33%
	Neural n.	66.20%	65.75%	60.19%
NOA	Log. reg.	64.04%	65.06%	45.17%
	Dec. tree	63.66%	63.13%	41.09%
	Neural n.	63.47%	61.36%	40.52%
NOC	Log. reg.	57.96%	–	–
	Dec. tree	57.95%	–	–
	Neural n.	57.96%	–	–
CBO	Log. reg.	69.77%	70.38%	69.12%
	Dec. tree	69.77%	69.13%	67.02%
	Neural n.	69.46%	70.63%	65.13%
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%
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.	63.82%	85.02%	39.01%
	Dec. tree	66.67%	63.96%	60.59%
	Neural n.	67.17%	66.70%	60.62%
LLOC	Log. reg.	66.85%	72.98%	54.58%
	Dec. tree	67.98%	66.81%	64.41%
	Neural n.	67.58%	65.29%	65.85%
Multiv.	Log. reg.	69.61%	72.57%	65.24%
	Dec. tree	69.58%	68.38%	67.84%
	Neural n.	68.77%	68.94%	64.76%

Table 2: Summary of the correctness, precision and completeness scores

also better in some cases. Hence, it seems CBO is the best predictor. *LLOC (Logical Lines Of Code)* got excellent scores in each method applied and only the CBO metric was better. *NML (Number of Methods Local)* and *RFC (Response set For a Class)* gave worse but still usable results. The scores for the *LCOM (Lack of COhesion in Methods)* and *LCOMN (Lack of COhesion in Methods allowing Negative value)* metrics differed by 3-4% at most, but both of them were poor. This means that they are not really good for predicting bugs. We can just say that some statistical connection was found between NOA (Number Of Ancestors) and the number of bugs, but it is not such a good predictor as the others and further investigation is required. In the case of NOC (Number Of Children) all three method classified all the classes as faultless, therefore NOC cannot be seriously used for bug prediction.

Summarizing our results, we can state that 7 out of the 8 metrics can be usefully applied for fault-prediction, and the best one seems to be the CBO metric, which is better than LLOC. As we mentioned before, several previous studies corroborate our general findings. The main value of our results is that we are one of the first to demonstrate a concrete statistical connection between metrics

and the bugs contained in a large system.

Studying the evolution of Mozilla

We examined the metric statistics for Mozilla to see how it changed over the seven versions we studied. The values of NML, CBO, RFC, LCOM, and LLOC increased significantly with version 1.2. With this version the number of bugs also increased, which is also curious because at the same time the number of classes decreased.

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 bugs. Of course, other factors might have also influenced the number of submitted and fixed bugs, so further investigation is needed to identify the precise reason for this.

Own contributions

We examined the relationship between Chidamber and Kemerer metric suite and the traditional lines of code metric values and the number of bugs found in the classes. The regression analysis and various machine learning methods applied in this study gave very similar results. We found that 7 out of the 8 metrics in question can be used for fault-prediction, but to a different degree.

The analysis of the fault-prone property of the metrics, with the help of regression analysis and machine learning methods, and the construction of fault-predictor models are the author's own work. The analysis of Mozilla's software evolution is not the author's result.

3 Examining the fault-prediction capabilities of metric-categories

Previously we examined only eight metrics but we could not form any definite general conclusions. Therefore we decided to extend our experiments so as to examine metric categories as well. First, we analyzed two more versions of Mozilla, so we had all the versions from 1.0 to 1.8. Since lots of bugs had been corrected since the previous experiment, the bug reports were collected again and they were associated with classes. Thanks to our new way of bug collection, we did not need the Bugzilla database any longer because the necessary information could be extracted directly from Bugzilla. We chose version 1.6 again, and the generated and unchanged classes were filtered out in the same way as before. So 3,209 classes and 7,662 bugs remained in version 1.6.

We examined five metric categories (*size*, *complexity*, *coupling*, *inheritance* and *cohesion*) and the fault-proneness capability of a category was determined via its metric capability. 58 metrics were chosen and they were classified according to the five metric categories.

As the four previously applied methods gave very similar results, we decided just to employ logistic regression. We found statistical connections between the metrics and the bugs in 17 cases. Based on these results, we reasonably concluded that the coupling metrics are the best bugs predictors, which supports our previous results, where we found that CBO was the best. The complexity category achieved good results as well. The metrics from size category, which measured the lines of code of a class in a different way or counted the methods and public library routines, were useful too.

Unfortunately, the other size metrics we tested were of little practical worth. Cohesion metrics gave very poor results so we do not suggest using any of them for fault-prediction. Lastly, none of the inheritance metrics was suitable for bug prediction. These results are broad agreement with our previous findings.

Own contributions

We extended our previous analysis to metric categories and found that coupling metrics are the best predictors, but complexity metrics and some of the size metrics may also be beneficial for fault-prediction. The predictive capabilities of cohesion metrics are negligible and the inheritance metrics proved unsuitable for bug prediction.

The entire analysis of the fault-prediction capabilities on the metric categories is the author's own work.

4 Using software developers' view to improve quality models

We demonstrated that the metric values for arbitrarily large software systems can be easily calculated with the help of the Columbus technology. For an industrial system we found that there was a correlation between the metric values and the quality of the software, hence we might think that all the difficulties would be eliminated if we applied the metrics in practice. But before introducing metrics in our software process, we should investigate how the developers would use them. If they cannot use the metrics properly, we just slow down their work and the software quality will not be improved. What is worse, the use of metrics might have unexpected consequences.

Therefore we created a *Survey* to examine the developers' knowledge and opinions about the use of three object-oriented (LLOC, CBO, and WMC) metrics and one code duplication (CI) metric. The Survey contained over 50 questions. It examined the participants' views in order to discover what kinds of relations exist between the metrics and program comprehension & testing. The Survey can be divided into three parts. The first part contains several general questions about the participants' experience and skills in order to get an overall picture about them. In the second part, the questions examined the participants' views about each metric separately; more precisely, we asked exactly the same questions about all four metrics to learn their opinion of each. In the third part the metrics were examined together and the participants had to rank the them in order of their importance.

The 50 participants who filled out the questionnaire at our Software Engineering Department all work on industrial and R&D projects. They ranged from beginner students to experienced programmers so the participants' experience and skills differed greatly. Therefore besides the analysis of the answers and their distribution, statistical methods were applied to see how the different levels of experience influenced their assessment on the practical worth of each metric.

First the general questions and their relationships were analyzed and we found that the replies broadly reflected the structure of our department. Based on these replies we divided the participants into two groups (junior and senior), according to their knowledge and experience in a particular area. For each question we examined whether there was any significant difference between the replies of the two groups.

In the second part, the questions asked how they would assess the problem of the understanding

& testing of an unknown piece of software by just using metrics. We were interested in what kind of reasons (for example, generated code or a well-known design pattern) would be accepted for "wrong" metric values. The third set of questions asked how the participants would share their testing resources based on metrics. All these questions were asked for each metric separately. In many cases we found that the junior and senior participants' answers differed significantly, which probably means that the experience gained in different areas has a direct influence on the practical use of metrics.

In the third part, among other questions we investigated the importance of the four metrics from a testing aspect. According to the participants, the WMC metric was the most important, while the scores for the other three metrics (LLOC, CBO, CI) were almost the same, but they were significantly worse than the WMC score. This was surprising because it seems to contradict the results of our earlier studies.

The Survey has many interesting findings which should be taken into account before applying metrics in practice. One conclusion drawn from the answers and the fact that the metrics were frequently applied differently by the senior developers is that we should first explain to the developers how the metrics should be used in their everyday work.

In addition, it often turned out that the metrics should be applied differently. In the replies over half of the participants said they would accept wrong metric values for generated source code. When investigating the developers' views about these cases, our quality models ought to be improved to handle these parts of the software differently.

Own contributions

We were curious about how developers could benefit from applying software metrics, so a Survey was devised to learn how the developers would use the metrics in different software maintenance areas, and to see which of the metrics were important for them. We showed that the experience gained in different areas often influences the judgement of the practical use of metrics. The results also made clear that developers should be taught about their proper use. For us, these results can be used to improve the efficiency of our quality models.

Although the idea and topic of the Survey is not the author's, the detailed elaboration of the Survey, its implementation and the evaluation of its results are the author's own work.

Summary

The thesis examined object-oriented metrics from several aspects. First, we presented our Columbus technology which is suitable for analyzing large C++, Java and C# systems and we showed how object-oriented metrics can be calculated. We developed a heuristic to automatically collect the reported and corrected bugs from a bug tracking system and to associate them with the source code elements.

We proved that there is correlation between the metrics and the number of bugs, which means that the quality models based on metrics can be used to improve the quality of the software. Later we extended our experiments to metric-categories as well. Using these results, we created the kind of tools that are able to measure the software quality of the package in question, and at the same time it can be used to follow its evolution.

In addition, our Survey about the practical use of metrics demonstrated that it is not enough to give the developers the metrics; we also have to show them how to use them. The results of the survey can also be used to improve our quality model by taking into account the views of developers.

Lastly, Table 3 below summarizes which publications cover which results of the thesis. We should mention here that there is a great international interest in the topic of the thesis, which is supported by the fact Ferenc et al.'s article [6] has been independently cited 34 times, while Gyimóthy et al.'s article [7] has been cited over 100 times to date.

	[6]	[7]	[14]	[15]
1.	•	•		
2.		•		
3.			•	
4.				•

Table 3: The relation between the thesis topics and the corresponding publications

Acknowledgements

First, I would like to thank for my supervisor, Tibor Gyimóthy, who introduced me to this fascinating topic and who steered the direction of my research work. I also wish to thank Rudolf Ferenc, Lajos Fülöp , Judit Jász, Péter Siket, Zoltán Sógor, and Fedor Szokody for their kind help and support that I received during my Ph.D. studies. I will always treasure their friendship and the good advice offered over the years.

István Siket, March 19, 2010.

References

[1] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. In *IEEE Transactions on Software Engineering*, volume 22, pages 751–761, October 1996.

[2] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Clarendon Press, Oxford., 1995.

[3] Lionel C. Briand and Jürgen Wüst. Empirical Studies of Quality Models in Object-Oriented Systems. In *Advances in Computers*, volume 56, September 2002.

[4] Bugzilla for Mozilla.
`http://bugzilla.mozilla.org`.

[5] S.R. Chidamber and C.F. Kemerer. A Metrics Suite for Object-Oriented Design. In *IEEE Transactions on Software Engineering 20,6(1994)*, pages 476–493, 1994.

[6] Rudolf Ferenc, István Siket, and Tibor Gyimóthy. Extracting Facts from Open Source Software. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004)*, pages 60–69. IEEE Computer Society, September 2004.

[7] Tibor Gyimóthy, Rudolf Ferenc, and István Siket. Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. In *IEEE Transactions on Software Engineering*, volume 31, pages 897–910. IEEE Computer Society, October 2005.

[8] D. Hosmer and S. Lemeshow. *Applied Logistic Regression*. Wiley-Interscience, 1989.

[9] The Mozilla Homepage.
`http://www.mozilla.org`.

[10] J. Neter, W. Wasserman, and M. H. Kutner. *Applied Linear Statistical Models, 3rd Ed.* Richard D. Irwin, 1990.

[11] OpenOffice.org Home Page.
`http://www.openoffice.org/`.

[12] J.R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[13] Ferenc Rudolf. *Modelling and Reverse Engineering C++ Source Code*. PhD thesis, University of Szeged, 2004.

[14] István Siket. Evaluating the Effectiveness of Object-Oriented Metrics for Bug Prediction. *Periodica Polytechnica*, Budapest, 2009 Accepted for publication.

[15] István Siket and Tibor Gyimóthy. The Software Developers' View on Product Metrics; *A Survey-based Experiment*. *Annales Mathematicae et Informaticae*, Eger, 2010 Accepted for publication.

[16] Ping Yu, Tarja Systä, and Hausi Müller. Predicting Fault-Proneness using OO Metrics: An Industrial Case Study. In *Sixth European Conference on Software Maintenance and Reengineering (CSMR 2002)*, pages 99–107, March 2002.