A Method for an Accurate Early Prediction of Faults in Modified Classes

Piotr Tomaszewski, Håkan Grahn, Lars Lundberg
School of Engineering
Blekinge Institute of Technology
SE-372 25 Ronneby, Sweden
{piotr.tomaszewski, hakan.grahn, lars.lundberg}@bth.se

Abstract

In this paper we suggest and evaluate a method for predicting fault densities in modified classes early in the development process, i.e., before the modifications are implemented. We start by establishing methods that according to literature are considered the best for predicting fault densities of modified classes. We find that these methods can not be used until the system is implemented. We suggest our own methods, which are based on the same concept as the methods suggested in the literature, with the difference that our methods are applicable before the coding has started. We evaluate our methods using three large telecommunication systems produced by Ericsson. We find that our methods provide predictions that are of similar quality to the predictions based on metrics available after the code is implemented. Our predictions are, however, available much earlier in the development process. Therefore, they enable better planning of efficient fault prevention and fault detection activities.

1. Introduction

A majority of software systems evolve during their lifetime. This system evolution causes many changes to be introduced in the original source code. Such code modifications are an important source of faults [9, 13, 20, 21]. It is widely known that faults are one of the major cost drivers in software development projects. Activities connected with fault handling account for a significant part of the project budget, e.g., in the study reported in [4] 45% of the project resources were devoted to testing and simulation. Therefore, any method that reduces the cost associated with faults handling is likely to bring significant project cost savings.

The fact that about 60%-80% of the faults can be found in about 20% of the code modules [1, 11] and that about half of the code modules are usually defect free [1] shows that there is a potential for savings if we manage to focus our fault handling efforts on the portion of the code that actually contains faults. A popular method for identifying fault-prone code is using a fault prediction model (e.g., [6, 11, 13-15, 22]). If we assume that the cost of finding faults in a class is proportional to the size of the class (like in [2, 3]) then, by selecting classes with the highest fault densities, such a prediction model increases the *fault detection efficiency* (i.e., the number of faults found per amount of code analyzed). As a result, more faults are removed within a given budget. Therefore, in this study we build models that predict fault density.

Fault prediction models are usually based on different characteristics of the software, e.g., design or code metrics (e.g., [6, 22]). Some of those metrics are available only after the system is implemented, e.g., the number of lines of code or McCabe complexity [7]. There are also metrics that are available before the coding has started. For example, many design metrics, like the number of methods or coupling [5], can be calculated from the design documentation. Prediction models based on such design metrics are able to identify fault-prone classes even before these classes are actually modified. Being able to identify the most fault-prone classes so early in the development process makes it possible to apply preventive measures to such classes. For example, they can be assigned to more experienced developers or an increased number of code reviews/inspections can be planned for such classes.

There are a lot of studies that attempt to predict faults in the modified code units [8, 10, 17, 19-21]. One general conclusion from these studies is that the most promising indicator of fault density of a modified code unit is the relative size of the modification of this code unit, i.e., the size of the modification divided by the size of the whole code unit (see Section 2 for details concerning these studies).

In this paper we apply the idea of a relative modification size to the metrics that are available before the system is implemented. We define a number of metrics, available at design time, that approximate the relative size of the modification. We evaluate their ability to predict fault densities of classes before these classes are implemented. We show that our metrics are able to predict fault densities of classes with accuracy similar to the accuracy of a prediction based on metrics that are available after the code is implemented.

Our evaluation is based on data describing three releases of two telecommunication systems developed by Ericsson. These are large systems (about 1000 classes, 500 KLOC each) that are mission-critical for mobile network operators. Because of that, they undergo extensive and therefore expensive quality assurance before they are released to the market. The systems are mature and have been available on the market for over six years.

The rest of the paper is structured as follows: in Section 2 we present work that has been done by others in the area of fault prediction in modified code. Section 3 describes the metrics we have defined to predict fault densities in modified classes. In Section 4 we present our evaluation method. Section 5 presents the results of the evaluation. In Section 6 we discuss our findings. In the last section (Section 7) we present the most important conclusions from our study.

2. Related work

As we indicated in the introduction there is a lot of research that aims at predicting faults in evolving systems. Nagappan and Ball [10] evaluated the applicability of relative code churn measures to predict the fault densities of software units. As relative code churn measures they understand the amount of code change normalized by the size of the code unit the change was introduced to. Their study was based on the code churn between Windows Server 2003 and Windows Server 2003 Service Pack 1. The authors concluded that the relative code churn measure could be used as predictor of a system's fault density. The measures described in [10] are typical code metrics. To calculate them the system must be implemented, which limits the usage of the prediction models to after the system is implemented.

Munson and Elbaum [9], analyzed large software system and they also noticed that relative measures are very good predictors of the fault-proneness of modified code. The metric they evaluated was the relative complexity of modified modules. They showed that this metric was highly correlated with the fault density.

Selby [17] reached a similar conclusion. He observed that the number of faults in a modified class tends to increase with the size of the modification of the class. The information about the modification of a file was also considered very useful by Ostrand at al. [12]. They noticed that modified files are very fault-prone – more fault prone than new files.

We also performed studies [19, 21], in which we built models that predict fault densities in modified classes. We found that the most promising metric for estimating the number of faults in the modified code was the size of the modification, which we calculated as a number of new and modified lines of code in the class. As a consequence, the best fault density prediction metric was the relative modification size, obtained by dividing the size of the modification by the size of the class.

In all studies described above the faults are predicted in modified code, but only after the system is implemented. There are also studies that report promising results when it comes to predicting faults before the implementation has started. For example, Zhao at al. [22] compared the accuracy of fault prediction using design metrics with the accuracy of fault prediction using code metrics. The authors concluded that the results obtained from models based on design metrics are even more accurate than the results obtained using code metrics only. The authors, however, did not say if the modules analyzed were new or modified. Also the design metrics collected are mostly different SDL related metrics (the number of SDL diagrams, the number of task symbols in SDL descriptions, etc.), which limits their usage to systems designed using SDL.

There are studies that evaluate the applicability of other metric suits to predict faults. For example, Yu *et al.* [15] evaluated the applicability of the most common object-oriented metrics for predicting the number of faults. The authors obtained rather promising results but their study was based on new classes only.

To check if object-oriented metrics are also applicable for predicting faults in modified code we performed a study [20], in which we compared the accuracy of fault predictions using object oriented metrics with the accuracy of predictions using code metrics. It turned out that our results were similar when we used design or code metrics that described the characteristics of a final system. However, when we introduced the code metric describing the size of modification, it largely increased the quality of prediction using code metrics. This metric, alone, achieved higher prediction accuracy than all metrics describing the characteristics of a final system combined into one multivariate prediction model. Therefore, we concluded that to improve the quality of early (i.e., available before implementation) prediction of faults we must look for metrics that:

- describe the characteristics of the modification

- are available before the implementation is done

In this paper we suggest such metrics and we evaluate their ability to predict fault proneness of modified classes.

3. Predictor metrics

As we indicated in previous sections, our goal is to find metrics that are available at the time when the new release of the system is already designed, but not yet implemented. The metrics should describe the relative size of modification (*RelMod*), i.e., the size of the modification divided by the size of the class:

$$\mathrm{RelMod} = \frac{\mathrm{Size(Modification)}}{\mathrm{Size(Class)}} \quad (1)$$

In studies where the prediction is performed after the code is implemented, such a metric was shown to be very successful for predicting fault densities of modified files (see Section 2 for details). However, the task of obtaining such a metric is significantly simpler when the code is implemented. At that time we can simply measure Size(Modification), i.e., the number of added and changed lines of code in the class, and Size(Class), i.e., the number of code lines in the class. Both values are easily available from version control systems. However, at the design time none of these metrics are available. For that reason, they must be approximated by some other metrics.

Typically size metrics measure the length of code and therefore they are based on counting the number of some language constructs, e.g., the number of statements, the number of code lines, or the number of operands. Even though all these metrics do not measure exactly the same thing, they usually tend to be highly correlated, which makes it possible to predict one of them using another. One size metric of that kind that is available from the design documentation is the number of methods (*NoM*). This metric was shown to be a very good predictor of the final size of the system measured in the number of code lines [16].

In our study two metrics are based on the concept of counting methods:

- *NoM*– the Number of Methods in the Class, which we use as a Size(Class) metric
- *NoACM* – the number of Added or Changed Methods in the Class, which we use as a Size(Modification) metric

One can argue that one problem with using NoM as a size metric is that the average size of a method (in lines of code) may be different in different classes. Studies like [16] show, that these differences tend to average out at the project level. However, since for modified classes we actually have information about the average size of the method, we decided to check if using this information improves the accuracy of a prediction. The average size of a method can be calculated from the previous release of the system. Therefore, we introduced a new metric *ApproxSize* (approximated size of the class) which we define in the following way:

$$\mathrm{ApproxSize} = \mathrm{NoM}_{\mathrm{CurRel}} \bullet \frac{\mathrm{Size}_{\mathrm{PrevRel}}}{\mathrm{NoM}_{\mathrm{PrevRel}}} \quad (2)$$

where *CurRel* indicates that the metric concerns the release for which we perform predictions, while *PrevRel* indicates that a certain metric concerns the previous release of the system. Obviously, we use ApproxSize as Size(Class) metric.

Based on the metrics introduced above (NoM, NoACM, and ApproxSize) we defined two metrics describing the relative size of the modification.

The first one, *RelMod*$_{NoM}$, measures the modification as the number of new or modified methods in the class in relation to the number of all methods in the class:

$$\mathrm{RelMod}_{\mathrm{NoM}} = \frac{\mathrm{NoACM}}{\mathrm{NoM}} \quad (3)$$

The second one, *RelMod*$_{ApproxSize}$, uses the ApproxSize metric to approximate the size of the class. Therefore, RelMod$_{\mathrm{ApproxSize}}$ is defined in the following way:

$$\mathrm{RelMod}_{\mathrm{ApproxSize}} = \frac{\mathrm{NoACM}}{\mathrm{ApproxSize}} \quad (4)$$

4. Evaluation method

The evaluation of our metrics is performed using the data collected from three releases of two large telecommunication systems developed by Ericsson. From now on, we call these systems System A1, System A2, and System B, where System A1 and System A2 are two consecutive releases of one system. As we indicated in Section 1, these systems are large, they comprise of about 1000 classes and about half a million code lines each. In the releases under study a significant amount of code was introduced as a modification of already existing classes. In System A1 44% of the code was introduced as the modifications of existing classes, in System A2 43% of the code introduced in this release was introduced in existing classes. In System B 37% of the code written in this release was written in existing classes. An interesting thing is that 78%, 60% and 62% of faults that were found in System A1, System A2, and System B, respectively, were located in modified classes. This clearly suggests that modified classes are an important source of faults.

We evaluate our metrics (*$RelMod_{NoM}$*,, and $\mathrm{RelMod}_{\mathrm{ApproxSize}}$) from the perspective of their applicability to predict the fault proneness of modified classes. We order classes in the order of their decreasing fault density. We evaluate the different metrics by plotting the percentage of faults that would be detected if analyzing a system according to its suggestion against the accumulated percentage of the code that would have to be analyzed. Since our prediction method is meant for modified classes in our evaluations we use only modified classes from the respective systems.

To obtain a point of reference for our evaluations, we introduce two theoretical reference models:

- *Random model* – the model describing a completely random search for faults
- *Best model* – the model that makes only the right choices about which classes to analyze first

The Random model provides a baseline for evaluating our predictions, as it describes what results, on average, we could expect if we analyzed the code not following any model at all. On average, by analyzing *n*% of code we find *n*% of faults. Therefore, the Random model looks the same for all systems. By comparing the performance of our prediction with the Random model we can see if our prediction method provides an improvement over not using any prediction method at all.

The Best model provides a boundary of how good the prediction can be. In this theoretical model the code units are selected according to their actual fault density. The Best model looks differently for different systems, because it depends on the actual distribution of faults in the system. By comparing the performance of our prediction with the Best model we can see how far our prediction is from the best possible prediction.

The models described above are theoretical models. Other studies (see Section 2 for details) indicate that the best prediction practically available can be obtained by using the actual relative size of code modification. Therefore, we additionally include this metric as a point of reference. The relative size of code modification (*$RelMod_{Code}$*) is defined as:

$$\mathrm{RelMod}_{\mathrm{Code}} = \frac{\mathrm{NoACLOC}}{\mathrm{NoLOC}} \qquad (5)$$

where NoACLOC is the number of added and changed lines of code in the class, while NoLOC is the total number of lines of code in the class. The reader must bear in mind that $\mathrm{RelMod}_{\mathrm{Code}}$ is available only after the code is implemented. It can be seen as the current "state-of-the-art" in prediction of fault densities in the modified classes. Therefore, it is not evaluated in our study but it is included in our evaluations as a point of reference.

5. Results

The results of the evaluation using System A1 are presented in Figure 1. As can be noticed, there is no visible difference in the prediction quality between our metrics ($\mathrm{RelMod}_{\mathrm{NoM}}$ and $\mathrm{RelMod}_{\mathrm{ApproxSize}}$) and the relative modification metric measured after the code is implemented ($\mathrm{RelMod}_{\mathrm{Code}}$). This indicates that the fault densities of the classes in System A1 could be predicted equally accurately before the system was implemented and after the system was implemented. There is no obvious difference between the performance of $\mathrm{RelMod}_{\mathrm{NoM}}$ and $\mathrm{RelMod}_{\mathrm{ApproxSize.}}$

On average, our prediction models provide about half of the maximum possible improvement over the Random model. This is not any formal quantification, but an observation based on the fact that in Figure 1 our predictions are placed more or less half way between the Random model and the Best model.

The results of evaluation using System A2 are presented in Figure 2. By analyzing Figure 2 we can see that $\mathrm{RelMod}_{\mathrm{Code}}$ and $\mathrm{RelMod}_{\mathrm{ApproxSize}}$ predict fault densities with a similar accuracy. Therefore, the best prediction available before the code is implemented gives similar results as the best prediction available after the code is implemented. The accuracy of $\mathrm{RelMod}_{\mathrm{NoM}}$ is actually similar to the accuracy of the two remaining prediction models, apart from between 30% and 40% of code where it is clearly worse.

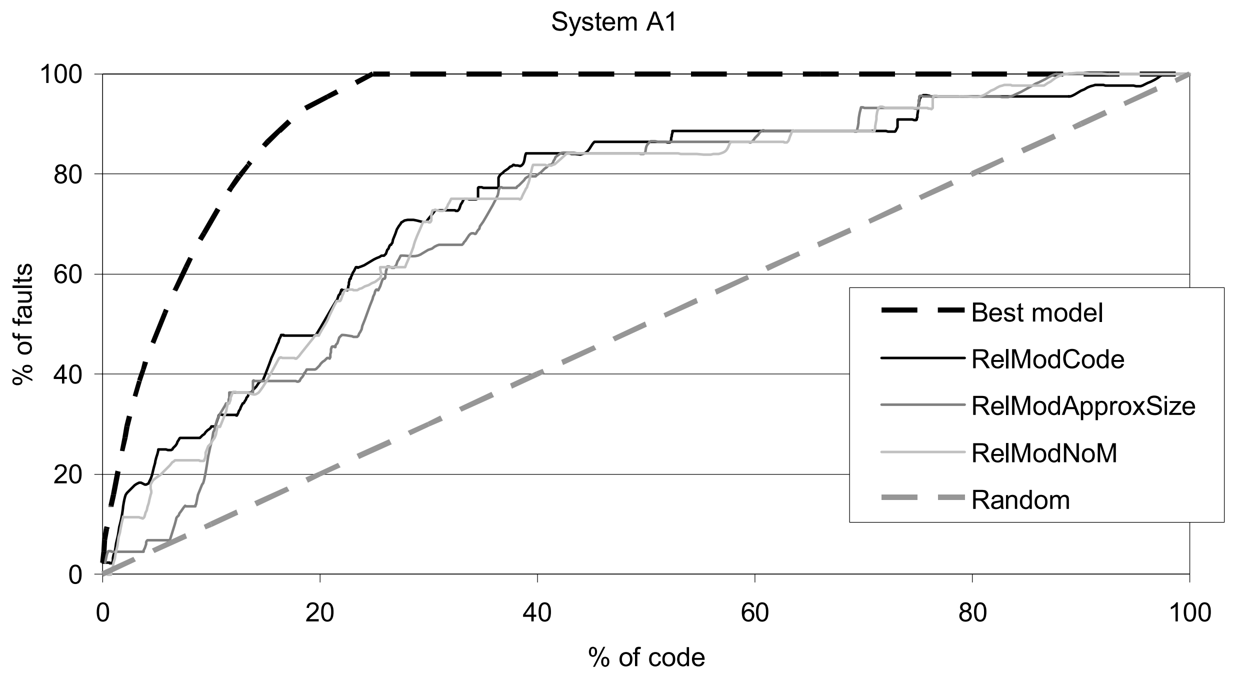

Figure 1. Evaluation of the applicability of metrics to predict the fault-densities of modified classes in System A1.

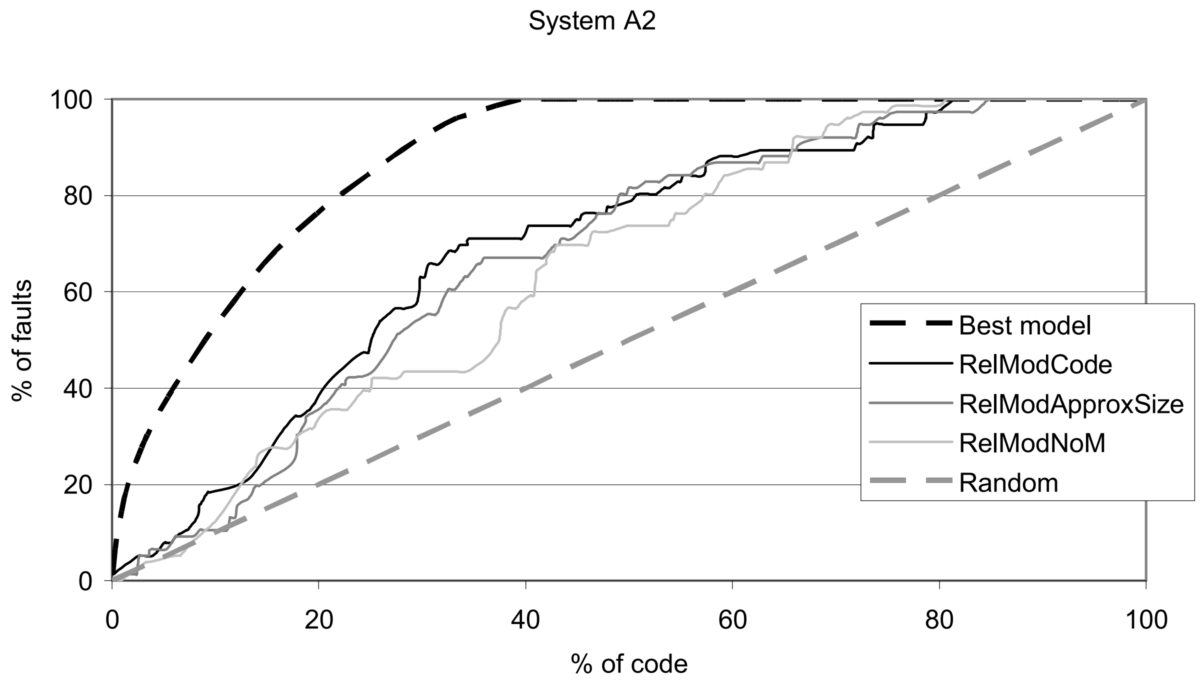

Figure 2. Evaluation of the applicability of metrics to predict the fault-densities of modified classes in System A2.

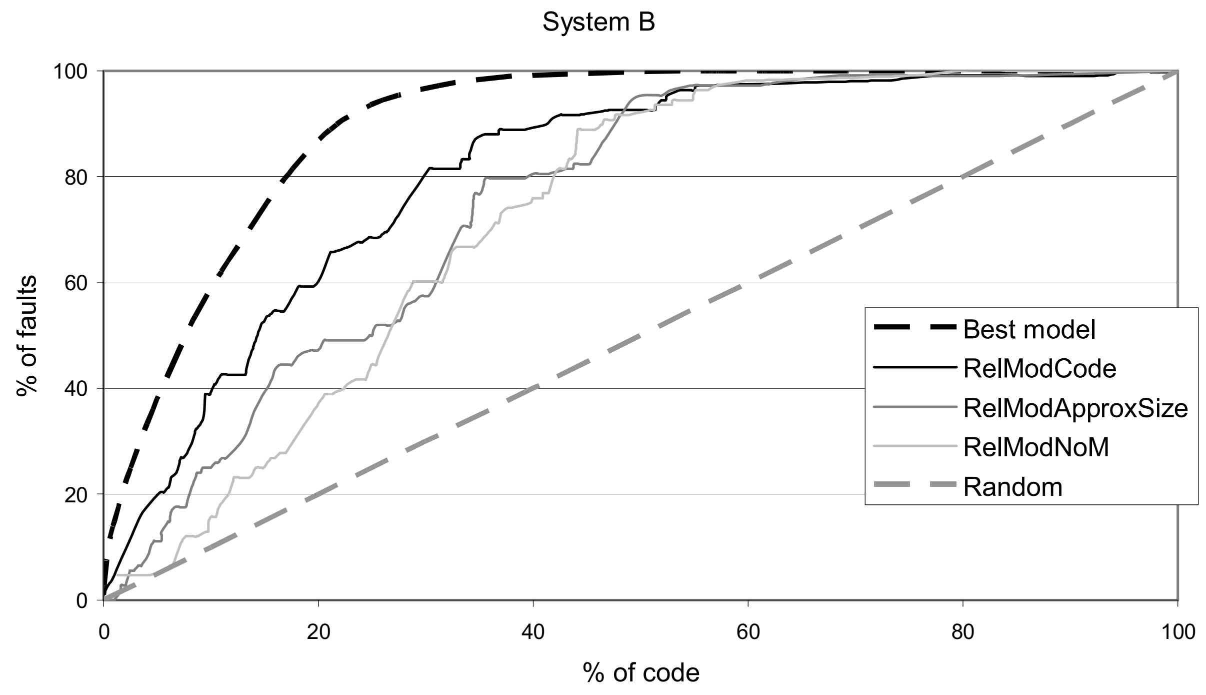

Figure 3. Evaluation of the applicability of metrics to predict the fault-densities of modified classes in System B.

Similarly to the results obtained when evaluating our prediction method using data from System A1, in System A2 our predictions offer about half of the maximal possible improvement.

The results of the evaluation of our prediction methods using data collected from System B are presented in Figure 3. In Figure 3 we can see that the prediction using $\text{RelMod}_{\text{Code}}$ is more accurate than any of the two prediction methods available at the design time. In practice, however, it is visible only when between 20% and 40% of the code is considered.

In System B the prediction using $\text{RelMod}_{\text{ApproxSize}}$ seems to be more accurate compared to the prediction using $\text{RelMod}_{\text{NoM}}$, especially when low percentages of the code are considered (up to 30%). However, as in case of Systems A1 and A2, in System B the overall difference in performance between $\text{RelMod}_{\text{ApproxSize}}$ and $\text{RelMod}_{\text{NoM}}$ is not large.

Also similarly to the previous cases (i.e., System A1 and System A2) in System B the early prediction methods are stable in providing about a half of the maximum possible improvement over the Random model. The prediction using $\text{RelMod}_{\text{Code}}$ seems to be more accurate here than in the previous cases – in Figure 3 the $\text{RelMod}_{\text{Code}}$ for all percentages of the code is closer to the Best model than to the Random model.

6. Discussion

Our findings clearly show that it is possible to perform accurate predictions concerning the fault densities of modified classes at the design stage, i.e., before these classes are actually implemented. Our evaluation, in which we used three releases of large telecommunication systems, showed that in all three cases the quality of the prediction based on the data available before the implementation was comparable with the quality of the best prediction available after the code was implemented. These findings are promising, as they indicate that it is possible to obtain the information that can be used for planning fault detection and fault prevention activities at the time when this information is most needed, i.e., early in the development process.

The results indicate that our method of approximating the size of code modification by using the information about the number of new and modified methods in the class works well and is accurate enough for making predictions. Also both our methods for approximating the final size of the class are accurate enough. It seems, however, that the method, in which we use the information about the size of the class from before the modification is slightly more accurate compared to the method that takes only the number of methods in the class into account. It can be observed because the predictions obtained using this approximation (i.e., $\mathrm{RelMod}_{\mathrm{ApproxSize}}$) are very similar to the predictions using the actual size of the class after modification (i.e., $\mathrm{RelMod}_{\mathrm{Code}}$).

One reason for the higher accuracy of predictions based on the number of methods and the size of the class from previous release of the system as compared to only using the number of methods might be that the spread of sizes of methods seems to be smaller within the classes than between classes. This can be explained by the fact that there is usually one person responsible for implementing a class and, therefore, this person's "programming style" may make the methods similar in size. This is, however, only a hypothesis, which we have not evaluated in this study.

On the other hand, by looking at figures 1-3 we see that the actual difference between $\mathrm{RelMod}_{\mathrm{ApproxSize}}$ and $\mathrm{RelMod}_{\mathrm{NoM}}$ is, in practice, very small. It would indicate that the sizes of the methods are not very different even between classes. It can mean that there are some common design practices that are followed by different designers within the company, which make their methods somewhat similar in size.

Even though, based on our evaluations, we would rather suggest using $\mathrm{RelMod}_{\mathrm{ApproxSize}}$, we must clearly state that using $\mathrm{RelMod}_{\mathrm{NoM}}$ also provides an improvement over not using any prediction method at all (i.e., following the Random model). The improvement is not much smaller compared to using $\mathrm{RelMod}_{\mathrm{ApproxSize}}$. The main difference, as we see it, is that $\mathrm{RelMod}_{\mathrm{ApproxSize}}$ seems to be more stable (see Figure 2 and Figure 3). This is, however, only our subjective judgment based on the observation of figures 1-3, not supported by any formal statistical analysis.

One can argue that one of the greatest advantages of fault prediction models based on code metrics, as well as those based on some design metrics, is that the measurements necessary for predictions can be obtained automatically. For example, for our $\mathrm{RelMod}_{\mathrm{Code}}$ it is possible to write an application that will get as an input the code from current and previous releases of the system, and as output will produce the prediction. The information about class sizes and modification sizes can be measured by a software tool, e.g., LOCC [18], or can be obtained from a version control system.

Such a full automation in case of our prediction method will be hard to achieve. Some things, like class size in the previous release of a system or the number of functions in the planned release are relatively easy to obtain automatically. Class size in the previous release of the system can be measured using some code measuring tool. If the design of the system is done using, e.g., UML modeling language, it is also relatively easy to extract the information about the number of methods in the class in the designed system. We are, however, not aware of any method for automatically obtaining the information regarding the number of new and modified methods in the class at design time. Therefore, if such prediction method is to be implemented, the company must introduce a process, in which each designer manually quantifies the number of methods to be modified and added to a class when planning the modification of this class. This should be a neither difficult nor expensive process. It must, however, be used rigorously for our prediction method to work.

One validity threat to our study is that the systems on which we evaluate our models come from the same company (i.e., Ericsson) and the same application domain (i.e., telecommunications). As we indicated before, it is possible that within this particular company there is some kind of "style guide" that e.g., makes the differences between the method sizes small and therefore makes the number of methods an accurate predictor of the size measured in code lines. We investigated this factor and, to our knowledge, there is no such guide stated explicitly. It is however, still possible that there is some implicit "programming style" within the company that is followed by the designers. This could potentially limit the applicability of our findings to this company only. Therefore, to further evaluate the models, an evaluation using data describing systems developed in some other companies and for different application domains would be recommended.

7. Conclusions

The goal with this paper is to suggest and evaluate a method for predicting fault densities in modified classes early in the development process. In this study we focus on predicting fault densities of classes before they are actually modified. Access to information about the fault-proneness of the classes before they are modified enables more efficient planning of different fault prevention and fault detection activities. For example, in order to assign more experienced developers to especially fault-prone classes, the information about fault-proneness of the classes in the system must be available before the coding actually begins.

In our study we establish the current "state-of-the-art" when it comes to predicting the fault densities of modified classes. We find that the relative size of code modification is considered as the best fault density predictor, i.e., the size of the code modification divided by the size of the class. This metric is available only after the system is implemented, so it is not applicable for the early prediction of fault-proneness.

Since the relative size of the modification is considered as the best fault density predictor for modified classes, we want metrics that approximate this measure but that are available before the coding starts. We suggest two such measures. Both of them approximate the size of modification by counting the number of added and modified methods in the modified classes. As class size metric one of them uses the number of methods in the class, while the other one also incorporates the information about the average size of the method in the previous release of a certain class.

We evaluate both our prediction methods and obtain promising results. Both our methods provide a prediction of quality similar to the quality of the prediction using the "state-of-the-art" solution that is only available after the code is implemented. It means that, by using our method, it is possible to obtain the information of similar quality much earlier in the development process.

Since the measurements necessary for our prediction can not be obtained automatically we also discuss the changes that need to be introduced to the development process in order to collect all the data we need for making our predictions. We conclude that, even though the data must be collected manually, the process of obtaining it is very simple and inexpensive. It must, however, be followed rigorously for our method to work.

Acknowledgments

The authors would like to thank Ericsson for providing us with the data for the study and The Collaborative Software Development Laboratory, University of Hawaii, USA (http://csdl.ics.hawaii.edu/) for the LOCC application.

This work was partly funded by The Knowledge Foundation in Sweden under a research grant for the project "Blekinge - Engineering Software Qualities (BESQ)" (http://www.bth.se/besq).

References

[1] B. Boehm and V. R. Basili, "Software Defect Reduction Top 10 List", *Computer*, vol. 34, 2001, pp. 135-137.

[2] L. C. Briand, J. Wust, J. W. Daly, and D. V. Porter, "Exploring the relationship between design measures and software quality in object-oriented systems", *The Journal of Systems and Software*, vol. 51, 2000, pp. 245-273.

[3] L. C. Briand, J. Wust, S. V. Ikonomovski, and L. H., "Investigating quality factors in object-oriented designs: an industrial case study", *Proc. of the 1999 Int'l Conf. on Software Eng.*, 1999, pp. 345-354.

[4] M. Cartwright and M. Shepperd, "An empirical investigation of an object-oriented software system", *IEEE Transactions on Software Engineering*, vol. 26, 2000, pp. 786-796.

[5] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design", *IEEE Transactions on Software Engineering*, vol. 20, 1994, pp. 476-494.

[6] K. El Emam, W. L. Melo, and J. C. Machado, "The prediction of faulty classes using object-oriented design metrics", *The Journal of Systems and Software*, vol. 56, 2001, pp. 63-75.

[7] N. Fenton and S. L. Pfleeger, *Software metrics: a rigorous and practical approach*, 2. ed. London; Boston: PWS, 1997.

[8] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history", *IEEE Transactions on Software Engineering*, vol. 26, 2000, pp. 653-661.

[9] J. C. Munson and S. G. Elbaum, "Code churn: a measure for estimating the impact of code change", *Proceedings of the International Conference on Software Maintenance*, 1998, pp. 24-31.

[10] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density", *Proceedings of the 27th International Conference on Software Engineering ICSE 2005.*, 2005, pp. 284-292.

[11] N. Ohlsson, A. C. Eriksson, and M. Helander, "Early Risk-Management by Identification of Fault-prone Modules", *Empirical Software Engineering*, vol. 2, 1997, pp. 166-173.

[12] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the Location and Number of Faults in Large Software Systems", *IEEE Transactions on Software Engineering*, vol. 31, 2005, pp. 340-355.

[13] M. Pighin and A. Marzona, "An empirical analysis of fault persistence through software releases", *Proceedings of the International Symposium on Empirical Software Engineering*, 2003, pp. 206-212.

[14] M. Pighin and A. Marzona, "Reducing Corrective Maintenance Effort Considering Module's History", *Proc. of Ninth European Conference on Software Maintenance and Reengineering*, 2005, pp. 232-235.

[15] Y. Ping, T. Systa, and H. Muller, "Predicting fault-proneness using OO metrics. An industrial case study", *Proc. of The Sixth European Conference on Software Maintenance and Reengineering*, 2002, pp. 99-107.

[16] M. Ronchetti, G. Succi, W. Pedrycz, and B. Russo, "Early estimation of software size in object-oriented environments a case study in a CMM level 3 software firm", *Information Sciences*, vol. 176, 2006, pp. 475-489.

[17] R. W. Selby, "Empirically based analysis of failures in software systems", *IEEE Transactions on Reliability*, vol. 39, 1990, pp. 444-454.

[18] The Collaborative Software Development Laboratory, University of Hawaii, USA;LOCC Project Homepage, http://csdl.ics.hawaii.edu/Tools/LOCC/;2005

[19] P. Tomaszewski, J. Håkansson, L. Lundberg, and H. Grahn, "The Accuracy of Fault Prediction in Modified Code – Statistical Model vs. Expert Estimation", *Proceedings of the 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, 2006, pp.

[20] P. Tomaszewski, L. Lundberg, and H. Grahn, "The Accuracy of Early Fault Prediction in Modified Code", *Fifth Conference on Software Engineering Research and Practice in Sweden*, 2005, pp. 57-63.

[21] P. Tomaszewski, L. Lundberg, and H. Grahn, "Increasing the Efficiency of Fault Detection in Modified Code" presented at Asian Pacific Software Engineering Conference, APSEC, Taipei, Taiwan, 2005.

[22] M. Zhao, C. Wohlin, N. Ohlsson, and M. Xie, "A comparison between software design and code metrics for the prediction of software fault content", *Information and Software Technology*, vol. 40, 1998, pp. 801-809.