Tomaszewski P, Lundberg L, Grahn H. Improving fault detection in modified code — A study from the telecommunication industry. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 22(3): 397∼409 May 2007

Improving Fault Detection in Modified Code — A Study from the Telecommunication Industry

Piotr Tomaszewski, Lars Lundberg, and Håkan Grahn

School of Engineering, Blekinge Institute of Technology, SE-372 25 Ronneby, Sweden

E-mail: piotr.tomaszewski@bth.se; lars.lundberg@bth.se; hakan.grahn@bth.se

Received March 15, 2006; revised February 15, 2007.

Abstract Many software systems are developed in a number of consecutive releases. In each release not only new code is added but also existing code is often modified. In this study we show that the modified code can be an important source of faults. Faults are widely recognized as one of the major cost drivers in software projects. Therefore, we look for methods that improve the fault detection in the modified code. We propose and evaluate a number of prediction models that increase the efficiency of fault detection. To build and evaluate our models we use data collected from two large telecommunication systems produced by Ericsson. We evaluate the performance of our models by applying them both to a different release of the system than the one they are built on and to a different system. The performance of our models is compared to the performance of the theoretical best model, a simple model based on size, as well as to analyzing the code in a random order (not using any model). We find that the use of our models provides a significant improvement over not using any model at all and over using a simple model based on the class size. The gain offered by our models corresponds to 38∼57% of the theoretical maximum gain.

Keywords fault prediction, metrics, testing and debugging

1 Introduction

Finding and fixing faults is a very expensive activity in the software development process[1]. In large telecommunication systems fault detection activities can account for a significant part of the project budget, e.g., in [2] 45% of the project resources were devoted to testing and simulation. Therefore, an increase of the fault detection efficiency can potentially bring significant savings on project cost. A well-known fact concerning faults is that a majority of the faults can be found in a minority of the code (the Pareto principle[3∼5]). Different sources report different numbers concerning the Pareto principle, ranging from 20∼60 (60% of the faults can be found in 20% of the modules) to 10∼80 (see [4] for a brief overview of the research concerning the Pareto principle). The Pareto principle shows that there is a potential for significant savings if we manage to focus our testing efforts on the most fault prone code units.

One way of helping testers to focus their efforts is to provide them with a fault prediction model. If we assume that the cost of finding faults in the class is proportional to the size of the class (like in [6, 7]) then, by selecting classes with the highest fault density, such a prediction model increases the *fault detection efficiency* (i.e., the number of faults found per the amount of code analyzed). In the long run, increasing the fault detection efficiency leads to higher quality of the products because testers focus on finding and removing faults in the classes that have the highest concentration of faults (fault density). As a result, they remove more faults within a given budget. Therefore, in this study we develop fault prediction models that predict fault density.

Fault prediction models are usually based on either different characteristics of the software that describe the structure of the code (e.g., design or code metrics[8∼10]) or historical information about the code (e.g., [11, 12]). Our models are based on design and code metrics. We perform our analysis at the class level, i.e., our predictions concern the fault-proneness of individual classes and are based on the characteristics of those classes. We predict the fault density in two ways — by predicting the fault density itself and by predicting the number of faults in a class and dividing it by the size of this class.

Our models are built and evaluated using data from two different telecommunication systems developed by Ericsson. From now on we denote them as System A and System B. In this study we have used two releases of System A (from now on called System $A1$ and System $A2$) and one release of System B. These are the most current releases of both systems (the current release of System B and the two latest releases of System A). Both systems are large telecommunication systems. Their sizes are about 800 classes (500 KLOC) and about 1000 classes (600 KLOC) for System A and System B, respectively. Both systems operate in the service layer of mobile phone network. As they are mission-critical for the customers, they undergo an extensive testing before they are released.

Both systems are mature systems that have been present in the market for several years. Over that period a number of releases of each system have been produced. Each new release usually introduces a significant amount of new functionality. Typically, new function-

Regular Paper

This paper is an extended version of paper presented at APSEC 2005 Conference.

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).

ality is introduced by modifying existing classes and/or implementing new classes. In System $A1$ the modification of classes from the previous release accounted for 65% of the code written in the current release (35% of the new code was introduced as new classes). In System $A2$ 37% of the code was introduced as a modification of previous classes, and in System B 44% of the code was introduced as a modification of the classes from the previous release. A well-known fact is that a modification of already existing code is an important source of faults[11,13]. This is supported by our data. Faults found in the modified code accounted for 86%, 62%, and 78% of all faults found in System $A1$, System $A2$, and System B, respectively. It can be noticed that in all three systems the modified code was significantly more fault-prone compared to the new code.

In this study we build and evaluate models that predict faults specifically in modified code, which is different from most studies in the area that do not distinguish between new and modified code (see Section 2). One reason for focusing on the modified code is that, as we have shown, the modified code is an important source of faults. Focusing on the modified code also gives us an opportunity to include not only usual metrics that describe the structure of the final product (e.g., size, complexity) but also metrics that describe the characteristics of the modification (e.g., the number of new and modified lines of code in the class). Also many studies in the fault prediction domain predict faults at the component or module level[5,12,14~20]. As we have shown in [21], the class level prediction, which we suggest in this paper, is of higher precision and therefore is likely to bring higher improvements.

We arbitrarily select System $A1$ as the system on which we build our models. The models are later evaluated by applying them to System $A2$ and System B. In this way we check if our models are stable across different releases of the same system as well as across different systems. We show that the models increase the efficiency of fault detection in a similar way in all three systems.

The rest of the paper is structured as follows. In Section 2 we present the work that has been done by others in the area of fault prediction. Section 3 describes the methods we have used for model building and evaluation. Section 4 presents the results we have obtained. In Section 5 we discuss our findings and different validity issues. In Section 6 we present the most important conclusions from our study.

2 Related Work

Fault prediction models that predict the number of faults or the fault density are very common in literature (e.g., [2, 10, 22~25]). The most typical methods for building prediction models are different variants of linear regression (e.g., [2, 10, 20, 22, 23, 25]). Other methods include, e.g., negative binomial regression[24]. Usually the construction of prediction model starts with selecting *independent variables* (variables that are used to predict the *dependant variable* — faults). The most common candidates are different code metrics (e.g., [10, 12, 16]) or variations of Chidamber and Kemerer (C&K)[26] object-oriented metrics (e.g., [6, 8, 10]). There are also studies that take historical information about the code fault-proneness into account (e.g., [11, 12, 24]). The initial set of independent variables is often large (e.g., over 200 metrics in [27]). A common assumption is that models based on a large number of variables are less robust and have lower practical value (more metrics have to be collected)[2,28]. Therefore, the first step of model building usually involves a reduction of the number of metrics. A commonly used method for the dataset reduction is a correlation analysis[2,8,10]. It is usually used to detect highly correlated metrics. Highly correlated metrics can, to a large extent, measure the same thing (e.g., the number of code lines and the number of statements are usually highly correlated because both measure size). Including them into the model causes a risk for multicolinearity[2]. Multicolinearity is especially risky when regression models are built. It leads to "*unstable coefficients, misleading statistical tests, and unexpected coefficient signs*"[28]. Correlation analysis is also used for selecting independent variables to predict faults (e.g., [5, 10]). Only those metrics that are correlated with faults are good fault predictors.

Below, we present studies in which fault prediction models were built. For each study we describe the set of metrics used, the metric selection criteria, and the results obtained. In the cases of prediction models built using linear regression we also quote R^2 values. R^2 is a "goodness-of-fit" measure that describes how well the model fits the data it was built on. It describes the proportion of variability of variable predicted by the model[29]. Therefore, it has values between 0 and 1[30]. The closer R^2 is to 1 the better the prediction model is. For details concerning the calculation of R^2 see Subsection 3.3.

In [10], Zhao *et al.* compare the applicability of design and code metrics to predict the number of faults. The analyzed system is one release of a large telecommunication system. The authors do not say if the code analyzed is new or modified. The design metrics collected are mostly different SDL related metrics (the number of SDL diagrams, the number of task symbols in SDL descriptions, etc.). The code metrics included the number of lines of code, the number of variables, the number of signals, and the number of if statements. The initial selection of metrics is based on the correlation analysis. To build the models the authors use the stepwise regression, which additionally eliminates the metrics that are not good as fault predictors. The authors conclude that both code and design metrics are applicable and give good results. However, in this study, the best fault prediction is obtained when both types of metrics are included in the same model. R^2 values obtained in this

study are 0.63 for the design metrics model, 0.558 for the code metrics model, and 0.68 for the model based on design and code metrics.

The applicability of object-oriented metrics for predicting the number of faults is evaluated by Yu *et al.*[25] The analyzed system consists of new classes only. The set of metrics used is largely based on C&K metrics[26]. The authors evaluate univariate and multivariate models. The best univariate model is based on the Number of Methods per Class metric ($R^2 = 0.423$). The results of univariate regression are used to select metrics for multivariate regression. For the metric to be selected, the univariate regression model based on it has to be significant (t-test) as well as it has to account for a large proportion of variability of the predicted value. However, in practice, the authors only reject the variables from the insignificant models. Finally, six different metrics are included in the proposed regression model, i.e., Number of Methods per Class, Coupling, Response for Class, Lack of Cohesion, Depth of Inheritance, and Number of Children. The R^2 statistic of this model is 0.597. The authors also show the model based on all ten metrics they collected. This model has the R^2 value equal to 0.603.

Cartwright and Shepperd[2] present a study in which they predict faults in object oriented system. The metric suite they use consists of some of the object-oriented C&K metrics (Depth of Inheritance, and Number of Children), some code metrics, and some metrics that are characteristic for the development method employed (Shlaer-Mellor). The authors obtain very high prediction accuracy. Their best univariate linear model is based on the number of events in the class and has $R^2 = 0.876$. The authors show that the accuracy of the model can be increased by adding a variable indicating if the class inherits from some other class ($R^2 = 0.897$).

Unlike our study, the studies described above do not focus specifically on modified code. However, they are very good examples of how fault prediction models are built as well as what kind of data fit can be expected from them.

There are also studies that attempt to predict faults in modified systems. Nagappan and Ball[31] 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. Their best model achieved a data fit (R^2) of 0.821. Munson and Elbaum[32] analyzed a 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[13] 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.

There are also other studies that attempt to assess the applicability of different metrics to predict faults. In most cases these are studies in which classification models were built, i.e., models that predict if there are faults in the module, not how many faults there are. From our perspective such studies are interesting, since they give an indication of metrics that are good predictors of fault-proneness. For example, El Emam *et al.*[8] observes an impact of inheritance and coupling on the fault-proneness of the class. The relation between inheritance, coupling, and probability of finding faults in the class was also identified by Briand *et al.*[6] In [15], Gunes Koru and Tian evaluate the applicability of complexity measures to predict faults. They concluded that there is a relation between complexity and faults, but it is not linear and therefore complexity measures are not likely to be good fault predictors when used in linear prediction models, like ours.

When it comes to the evaluation, most classification models are evaluated against the percentage of correctly classified classes. Briand *et al.*[6] noticed that such an evaluation may have a low practical value. Even though the model may point to a minority of classes, these classes can potentially account for a majority of the code. The prediction models used for estimating the number of faults are usually evaluated against their "goodness of fit" to the data they were built on, i.e., using R^2 statistic. Therefore, as we see, there is a lack of studies evaluating prediction models from the perspective of gain, in terms of cost reduction, that can be expected from applying them.

3 Methods

3.1 Metrics Suite

In this study we base our prediction models on the metrics that describe the structure of the system, i.e., on code and design metrics. All metrics that we collect are summarized in Table 1. All our measurements are done at the class level. The design metrics are mostly metrics that belong to the classic set of object oriented metrics suggested by Chidamber and Kemerer[26]. Lack of Cohesion (LCOM) was calculated as suggested by Graham[33,34]. The code metrics are different size metrics, metrics describing McCabe cyclomatic complexity as well as metrics describing the size of modification (Change Size — the number of new and modified lines of code in the final system as compared to the previous release of the system). For each class we collect information about the number of faults that were found in the class as well as calculate the fault density.

All product measurements mentioned in this study can be obtained automatically from the code using software tools. In this study we used the Understand C++[35] application to obtain all the design and code metrics (apart from the ChgSize quantification) from

Table 1. Metrics Collected in the Study

Name	Variable	Description
	Independent Metrics	
Coup	Coupling	Number of classes the class is coupled to
NoC	Number of Children	Number of immediate subclasses
Base	Number of Base Classes	Number of immediate base classes
WMC	Weighted Methods per Class	Number of methods defined locally in the class
RFC	Response for Class	Number of methods in the class including inherited ones
DIT	Depth of Inheritance Tree	Maximal depth of the class in the inheritance tree
LCOM	Lack of Cohesion	Metric measuring how closely the methods are related to the variables in the class
Stmt	Number of Statements	Number of statements in the code
StmtExe	Number of Executable Statements	Number of executable statements in the code
StmtDec1	Number of Declarative Statements	Number of declarative statements in the code
Comment	Number of Comments Lines	Number of lines containing comments
MaxCyc	Maximum Cyclomatic Complexity	The highest McCabe complexity of a function from the class
ChgSize	Change Size	Number of new and modified LOC (from previous release)
CtC	Ratio Comment to Code	Ratio of comment lines to code lines
	Dependent Variables	
Faults	Number of Faults	Number of faults found in the class
FaultDensity	Fault Density	Fault density of the class

the systems' code. The ChgSize was quantified using the LOCC[36] application. The information about faults was extracted from an internal Ericsson fault reporting system. We understand that it would be highly desirable[37,38] to reveal some information about the raw data we collected. However, since these data are highly confidential, due to our agreement with Ericsson we are not allowed to do that.

3.2 Model Building

We assume that the cost of performing fault detection is directly proportional to the size of the class. Therefore, our prediction models should identify the classes with the highest fault densities. Fault detection in such classes is the most efficient because it requires the least amount of code to be analysed to find a fault. Class analysis according to the model means that fault detection activities are performed on the classes in the order of their decreasing fault density predicted by the model. As we see, the fault density can be predicted in two ways:

• by predicting the fault density (Faults/Stmt) the fault density is predicted by the model;

• by predicting the number of faults (Faults) and dividing the predicted number of faults by the real class size (Stmt) — faults are predicted by the model, while size (Stmt) is measured.

In our study we evaluate both approaches. Even though they seem to predict the same thing, the prediction accuracy, given our set of metrics and our method of building models (regression), may be different for both of them. Linear regression, which we use for building models, attempts to predict the dependent variable as linear combination of independent variables. It may turn out that, e.g., linear combination of our metrics predicts fault density much more accurately than it predicts the number of faults.

We evaluate six prediction models, three predicting the fault-density and three predicting the number of faults. The models are built using:

• single metric — a model based on the single best fault (fault-density) predictor;

• selected metrics — a model based on a set of the best fault (fault-density) predictors;

• all metrics — a model based on all metrics collected.

To find the single and the selected metrics we use the simplest method, which is the correlation analysis. Since it turned out that our data were not normally distributed we use Spearman correlation co-efficient, which is not dependent on normality assumption[39]. As selected metrics, we choose those that are correlated to the independent metrics, i.e., with correlation coefficient values not close to 0. In the case of our dataset it turned out that the lowest correlation among the metrics from the selected metrics model was 0.29. Additionally, the correlations of our selected metrics with the dependent variables have to be significant at a 0.05 level (a standard significance level describing 5% risk of rejecting a correct hypothesis). In this way we eliminate the metrics that, due to a low correlation with the number of faults and the fault-density, cannot be considered useful for building prediction models.

Our univariate models are built using linear regression. The multivariate models are built using stepwise multivariate linear regression. The univariate linear regression estimates the value of the dependant variable (i.e., the number of faults or the fault-density) as a function of one of the independent variables (i.e., code and design metrics)[40]:

$$f(x) = a + b_1 x_1. \quad (1)$$

Multivariate linear regression estimates the value of the dependant variable (i.e., the number of faults or the fault-density) using linear combination of independent variables (i.e., code and design metrics)[40]:

$$f(x) = a + b_1 x_1 + b_2 x_2 + b_3 x_3 + \cdots + b_k x_k. \quad (2)$$

Stepwise regression is one of the methods that attempt to build a model on the minimal set of variables that

explain the variance of the dependant variable. Other methods of that kind are forward (backward) regression. In these methods variables are added (removed) to the model until adding (removing) the next one does not give any benefit (does not change model's ability to predict the dependent variable)[30]. We select stepwise regression because, compared to the forward regression, it additionally excludes variables that do not contribute to the model anymore[30]. Therefore, by using stepwise regression we hope to get models based on minimal sets of variables. Stepwise regression is used on the previously defined sets of metrics (All, Selected) and the final models are built on subsets of the previously defined sets of metrics, i.e., building a model on all metrics does not mean that all metrics are used in the final model but that all metrics are used as an input to the stepwise regression.

For each model we calculate a coefficient of determination, R^2, which is the standard measure of model's "goodness-of-fit". R^2 measures the strength of the correlation between the actual and the predicted number of faults. The R^2 equation is presented below

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \quad (3)$$

where y_i — the actual number of faults (or the actual fault density), $\hat{y}_i$ — the predicted number of faults (or the predicted fault density), $\bar{y}$ — the average number of faults (or the average fault density).

The practical meaning of R^2 is that it describes the proportion (percentage) of variability of the predicted variable accounted by the model[29]. The higher the R^2 value is, the better the prediction model fits the data it is built on. R^2 values range from 0 to 1[30], where 1 means the perfect model that accounts for all variability of the predicted variable (perfect prediction). R^2 equal to 0 indicates that the model is useless as a prediction model. We include R^2 for two reasons. First, it enables comparisons between our models. Second, it makes it possible to compare our results with the results obtained by other researchers, who usually quote R^2 values obtained for their models (see Section 2 for examples).

Similarly to [10], we also evaluate the significance of the entire prediction model using the F-test[30]. We select a 0.05 significance level, i.e., if the significance of the F-test has a value below 0.05 then the prediction model is significant.

All statistical operations connected with model building (i.e., correlation, stepwise regression and calculation of statistics connected with it) were performed using the statistical software package SPSS[41].

3.3 Model Evaluation

To evaluate our models we need some objective measurement of the accuracy of our models. We want to know what advantage can be expected from using our models as compared to not using any model at all. We also want to know how far our models are from the theoretical best model. A good prediction model must also give good results when it is applied to the data other than the one it was built on.

To measure the objective "goodness" of our models we introduce three reference models:

- *Random Model* — this model describes a completely random search for faults. The results obtained by this model are, on average, the results we could expect when no model is used and the order in which the classes are analyzed is random.
- *Best Model* — this is a theoretical model that makes the right choices about which classes to analyze. In this model the classes are selected according to their actual fault density. According to our criteria, it is impossible to do better than that.
- *Size Model* — a common (mis)conception[28] is that bigger classes tend to have more faults and higher fault densities. Therefore, we introduce a model in which the classes were analyzed based on their size (bigger classes are analyzed earlier).

A comparison of our models with the Random model gives us an indication if following our model is better than not following any model at all. The Best model gives us an indication of how good a model can get, and how far we are from being perfect. The Size model might often be encountered in real life situations because of its simplicity, as well as because many models suggested in literature actually tend to correlate with size[28]. By including this model we can evaluate it against our criteria of efficiency improvement as well as compare our models with it.

To check if our models are good prediction models, i.e., if they can be successfully applied to different projects, we build our models based on data from one of the projects only (System $A1$) and we apply them to:

- Project $A1$, on which the models are built;
- Project $A2$, which is a different (next) release of Project $A1$;
- Project B, which is a completely different project.

By comparing how well our models work in Project $A1$ and Project $A2$ we get an indication if they are stable across different releases of the same system. By comparing how well the models work in Project A and Project B we get an indication if they are stable across different systems. The stability is required because a prediction model is normally used to predict faults in projects/releases other than those it was built on.

In order to compare the models' performance both within and between systems we use three complementary comparison methods for assessing model "goodness". Generally, the "goodness" of the model is measured by the amount of code necessary to analyze in order to detect a certain number of faults, i.e., a model is better if by following it we are able to detect more

$$\text{Eq. 1} \quad Gain_{OurModel} = \frac{IOR_{OurModel}}{IOR_{Best}}$$

$$\text{Eq. 2} \quad IOR_{Model} = \sum_{i=1}^{n} \left(\frac{(DTR(Model, i) + DTR(Model, i-1)) \times (Code(Model, i) - \text{Code(Model}, i-1))}{2} \right)$$

$$\text{Eq. 3} \quad DTR(Model, i) = Model(i) - Random(Code(Model, i))$$

Fig.1. Calculation of the *Gain* metric. $Model(i)$ is the percentage of faults found if analyzing the i-th class according to the *Model*, $Random(Code(Model, i))$ is the expected percentage of faults detected if analyzing the same amount of code as in case of $Model(i)$ but not following any model at all, n is the number of classes. The details regarding calculation steps can be found in Subsection 3.3.

faults by analyzing the same amount of code compared to another model.

Our first comparison method is a diagram plotting the percentage of faults detected against the percentage of code that has to be analyzed to detect them. On every diagram we include our reference models (Random, Size, and Best models). By comparing how well our models do in relation to the Random model and to the Best model we are able to assess how good the models are and compare their performance in different systems.

The second method attempts to perform a quantification of the model's "goodness". Our *Gain* metric quantifies the ratio of an improvement offered by our model over the Random model to the theoretical maximum improvement possible. The calculation steps for the *Gain* metric are presented in Fig.1. Eq.1 in Fig.1 presents the way in which the Gain metric is calculated. In Eq.1 *IOR* stands for *Improvement Over Random*. The IOR_{Model} measure quantifies the overall improvement over the Random model that is offered by *Model*. On our diagrams, on which we plot the percentage of faults detected against the percentage of code that has to be analyzed to detect them, such an improvement over the Random model corresponds to the size of area between the Random model and *Model* (see Fig.2).

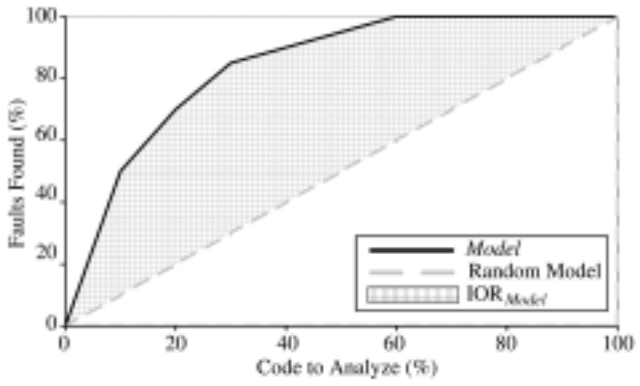

Fig.2. *Improvement Over Random* (IOR) for a *Model* is defined as the size of area between the *Model* and the Random model (checked area on the figure).

To calculate *IOR* we divide the area between the Random model and *Model* into a number of parallelograms equal to the number of classes in the system, and we sum their areas (see Eq.2 in Fig.1). In Eq.2, n is the number of classes in the system, $Code(Model, i)$ is the percentage of code that must have been covered when analyzing the i-th class according to the *Model*. It must be remembered that in order to analyze the i-th class according to *Model* we must have analyzed all the classes with predicted fault densities larger than the predicted fault density of the i-th class, which means that $Code(Model, i)$ consists of not only the size of the i-th class but also the sum of all sizes of classes with predicted fault densities larger than the predicted fault density of the i-th class. *DTR* stands for *Distance To Random*. Fig.1, Eq.3 presents the way DTR is calculated. In Eq.3, $Model(i)$ is the percentage of faults detected when analyzing the i-th class according to the *Model*. $Random(Code(Model, i))$ is the expected percentage of faults that we would detect using the Random model when analyzing the same amount of code as when analyzing the i-th class according to the *Model*.

The *Gain* metric gives a normalized value between -1 and 1, where 1 describes the Best model and -1 describes the worst possible model. It is so, because it is impossible to do better than the Best model and it is also impossible to do worse than the worst possible model, in which all the classes are selected according to their increasing actual fault density. Therefore, $\text{IOR}_{\text{Worst}} = -\text{IOR}_{\text{Best}}$, which explains the -1 value. The Random model in this scale gets value 0, which means that all models with *Gain* lower than 0 are worse than the Random model and all those with gain over 0 are an improvement over the Random model. The *Gain* metric quantifies only the average gain from using the model. As every average, it might be missing some important details. Therefore, we use it together with previously described diagrams presenting the gain from using the model for different percentages of the code. They give more insight in how the models actually perform.

Our final, third method of assessing model "goodness" is by checking the statistical significance of the difference between the performance of a model and the performance of Random model. The analysis of the graphs described before can give some conclusions regarding the model "goodness" but based on them it is hard to say to what extent the improvement over the Random model is statistically significant. Therefore, for each model, we perform the statistical analysis in which we test the following null hypothesis:

H_0: the expected mean distance between tested model and Random model equals zero

where as distance we understand Distance To Random

(DTR), defined before. Statistical tests appropriate for testing this hypothesis according to [39] are paired t-test and its non-parametric alternative Wilcoxon test (Wilcoxon Signed-Rank Test). Since our data were not normally distributed we applied the non-parametric test, i.e., Wilcoxon test.

The Wilcoxon test is performed in the following way[39]: first for every data point the distance between the Random model and the examined model is calculated. The distances are basically DTRs, as we defined them before. Absolute values of DTRs are ranked, and the sums of positive ranks (T^+) and negative ranks (T^-) are calculated. As the test statistic T of the Wilcoxon test the smaller of these two values is used, i.e., $T = \min(T^+, T^-)$. This value can be compared against tabularized values for desired significance level. For large samples it can be approximated by a normal random variable as described in [42]. SPSS, the statistical package used by us, reports the significance level for each test. Therefore, we do not need to pre-select the desired significance level for our test — we base our analysis on the highest confidence with which we can reject null hypothesis, i.e., if SPSS reports the significance of 0.05 it means that with 95% confidence we can reject the null hypothesis that our model's performance does not differ from the performance of the Random model.

4 Results

4.1 Model Building

As described in Subsection 3.2 our models are built using the data from System $A1$. We begin the model building with a correlation analysis. The results of the correlation analysis are presented in Table 2. The main purpose of the correlation analysis is to identify metrics that are the best single predictors of the number of faults and the fault-density (we look for single metrics with the highest correlations with the number of faults and the fault density). From Table 2 it can be noticed that ChgSize is the best predictor for both values (correlation values in bold in Table 2). Therefore, we select this metric to build the prediction models for the number of faults and the fault-density based on one metric.

The second reason for performing the correlation analysis is to eliminate the metrics that cannot be considered useful for building our prediction models (see Subsection 3.2 for details). The remaining metrics are used to build the model based on "selected metrics". It turned out that we removed the same metrics for the model that predicts the number of faults and the model that predicts the fault-density. We have decided not to use the following metrics in "selected metrics" models:

- Base, NOC, CtC, DIT — due to their low correlation with the faults and with the fault density and due to low significance of the correlation;
- LCOM — due to the low correlation with the faults and with the fault-density;
- Comment — due to an unsure meaning of this metric and its correlation to size.

We find the high positive correlation between Comment and Faults quite surprising. There are some possible explanations of that phenomenon, like considering the number of comments as a measure of human perceived complexity. We exclude this metric from selected metrics, because it is difficult to assure that the "commenting style" is maintained between the projects (there are no explicit guidelines concerning this in either of the analyzed projects). Therefore, it is difficult to say if prediction models based on Comments would be stable also in other products/releases of the same product.

In the study we build six different prediction models based on the data from System $A1$. Their names, independent variables and outputs are summarized in Table 3. The models are built using stepwise regression. The significance of each model's coefficient is checked using the t-test. The hypothesis tested is that the coefficient could have value 0, which would imply a lack of relationship between the independent and dependent

Table 2. Correlation Analysis (Spearman Correlation Co-Efficient) of the Metrics Collected from System $A1$

	Base	Coup	NOC	WMC	RFC	Comment	Stmt	Stmt Decl	Stmt Exe	Max Cyc	DIT	LCOM	CtC	Chg Size
Base	1													
Coup	0.35	1												
NOC	−0.13	0.06	1											
WMC	0.23	0.76	0.18	1										
RFC	0.63	0.68	0.07	0.82	1									
Comment	0.21	0.73	0.05	0.80	0.68	1								
Stmt	0.15	0.67	0.09	0.74	0.62	0.84	1							
Stmt Decl	0.01	0.57	0.08	0.61	0.44	0.73	0.86	1						
Stmt Exe	0.25	0.72	0.10	0.79	0.70	0.83	0.92	0.64	1					
Max Cyc	0.21	0.65	0.09	0.65	0.56	0.73	0.83	0.51	0.93	1				
DIT	0.97	0.35	−0.13	0.22	0.61	0.20	0.11	−0.01	0.22	0.12	1			
LCOM	−0.08	0.3	0.18	0.37	0.15	0.32	0.20	0.38	0.11	0.07	−0.08	1		
CtC	0.18	−0.01	−0.06	−0.02	0.07	0.12	−0.36	−0.34	−0.24	−0.26	0.21	0.06	1	
Chg Size	0.07	0.48	0.02	0.47	0.40	0.59	0.68	0.7	0.50	0.42	0.04	0.28	−0.25	1
Faults	0.00	0.43	0.02	0.47	0.41	0.54	0.52	0.48	0.48	0.38	−0.01	0.15	−0.1	**0.6**
Fault Density	−0.03	0.35	0.01	0.39	0.32	0.46	0.42	0.4	0.36	0.29	−0.04	0.15	−0.05	**0.53**

Note: The correlations with a grey background are NOT significant at 0.05 significance level.
The correlation of the best individual predictor of fault number and fault density is in bold.

variables (and therefore would make the original model the best, but not a meaningful mathematical relation between both variables)[40]. It turned out that all coefficients were significant at the 0.05 level. The significance of the entire model is tested using the F-test. The goodness-of-fit of each model is assessed using the R^2 statistic. The actual models are presented in Table 4.

Table 3. Summary of Our Models

Name	Based on	Predicts
AllNumber	All Metrics	Number of Faults
SelectedNumber	Selected Metrics	Number of Faults
SingleNumber	Single Metric	Number of Faults
AllDensity	All Metrics	Fault Density
SelectedDensity	Selected Metrics	Fault Density
SingleDensity	Single Metric	Fault Density

Note: Single metric: ChgSize.
Selected metrics: Coup, WMC, RFC, Stmt, StmtDecl, StmtExe, MaxCyc, ChgSize.

4.2 Model Evaluation

As it can be noticed in Table 4 all our models are significant according to the F-test. By looking at the R^2 values we can see that the goodness-of-fit is better for the models predicting the number of faults compared to those predicting fault-densities. Apparently, given our set of metrics, it is easier to predict the number of faults than the fault-density. As we expected (see Subsection 3.2) the models based on all metrics (AllNumber and AllDensity) have a better fit compared to their counterparts based on a limited number of metrics (see the R^2 values in Table 4). They may, however, suffer from the multicolinearity problem, e.g., according to the AllNumber model the number of faults increases with Comments and decreases with StmtExe, which is difficult to explain since both StmtExe and Comments are positively correlated with the number of faults (see Table 2).

Our main model evaluation is performed from the perspective of the fault detection efficiency improvement that they offer. We use each model as an indicator of the order in which the classes should be analyzed. For the models that predict the fault-density (AllDensity, SelectedDensity, SingleDensity) we order the classes according to the output of the model, so that we analyze classes with the highest predicted fault-density first. In the models that predict the number of faults (AllNumber, SelectedNumber, SingleNumber) the predicted number of faults is divided by the class size (Stmt). This partially predicted density measure is used to select the classes for analysis.

Our evaluation consists of three steps. First, for each model we plot a graph in which the percentage of faults detected is mapped to the percentage of code that has to be analyzed to detect them. The model is considered better than the other one if, by following it, we are able to detect more faults by analyzing the same amount of code. Later we calculate the *Gain* for each of our models. For details concerning the *Gain* metric see Subsection 3.3. Finally, we check if the differences between our models and the Random model are statistically significant.

To benchmark our models we include three reference models in the evaluation. The reference models are presented in Fig.3. By comparing the Best and the Random models in Fig.3 we can see that there is a large room for improvement that can be filled using a fault prediction model. For example, if we inspect 20% of code randomly, on average we would find 20% of faults. However, by inspecting the most fault prone 20% of code we can find 60%, 80%, or even almost 100% of faults for System $A1$, System $A2$, and System B, respectively (see Fig.3). That is three, four, and five times as much as by inspecting the code randomly. Therefore, a model that tells us which part of the code to analyze first can potentially result in cost savings and increased quality of software. In all future figures we include the Best, Random, and Size models (always in dashed line) in order to provide reference points for evaluating our models.

The second conclusion from analyzing Fig.3 is that the Size model does not help very much when it comes to increasing the efficiency of fault detection. In fact, it is either about as good as the Random model (System $A1$ and System B) or even worse than the Random model in the case of System $A2$. Neither of our cases supports the theory that the size affects fault density and that the Size model can be used to predict fault density.

When evaluating our models we start with evaluating the fault detection efficiency improvement gained by using the models that predict the number of faults (AllNumber, SelectedNumber, SingleNumber). The results are presented in Fig.4. As it can be noticed all three models present an improvement over both the Random model and the simple Size model. This holds true not

Table 4. Prediction Models Obtained Using Stepwise Regression Based on Data from System $A1$

Model	Equation	R^2	F	*Sig.*
AllNumber	$FaultDensity = (0.004 \times Comment + 0.003 \times ChgSize - 0.677 \times Base + 0.276 \times Ctc - 0.003 \times StmtDecl - 0.005 \times LCOM + 0.010 \times RFC - 0.001 \times StmtExe + 0.089)/Stmt$	0.752	75.944	0.0
SelectedNumber	$FaultDensity = (0.004 \times ChgSize + 0.001 \times StmtExe - 0.002 \times StmtDecl + 0.008)/Stmt$	0.585	96.412	0.0
SingleNumber	$FaultDensity = 0.005 \times ChgSize/Stmt$	0.550	252.84	0.0
AllDensity	$FaultDensity = 54.040 \times CtC + 0.115 \times ChgSize - 42.431 \times DIT - 0.096 \times Stmt + 3.036 \times Coup + 0.670 \times RFC - 19.685$	0.479	30.997	0.0
SelectedDensity	$FaultDensity = 0.184 \times ChgSize - 0.138 \times Stmt + 2.429 \times Coup + 1.057 \times WMC + 2.748$	0.280	19.815	0.0
SingleDensity	$FaultDensity = 0.081 \times ChgSize + 12.739$	0.058	12.742	0.0

Note: R^2 describes goodness-of-fit (values closer to 1 indicate better fit), *Sig.* is singificance level of F-test.
F, R^2, and *Sig.* quoted for AllNumber, SelectedNumber, and SingleNumber concern the models that predict the number of faults.
These models are divided by *Stmt* in Equation section in order to provide the fault density prediction.

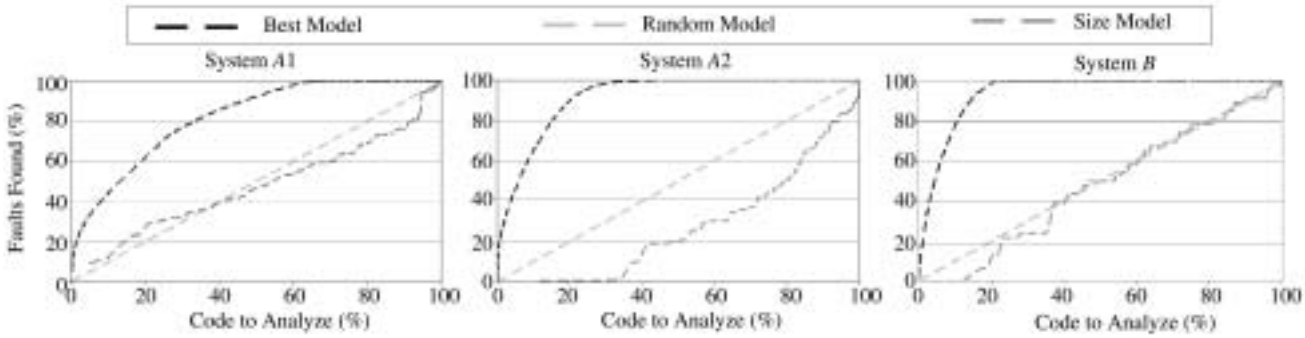

Fig.3. Reference models. The Best model describes a perfect model that makes only the right choices and selects classes in the order of their decreasing fault density. Random model represents a case in which code for inspection is picked randomly. The Size model is a model in which the biggest classes are analyzed first (see Subsection 3.3 for details concerning reference models.)

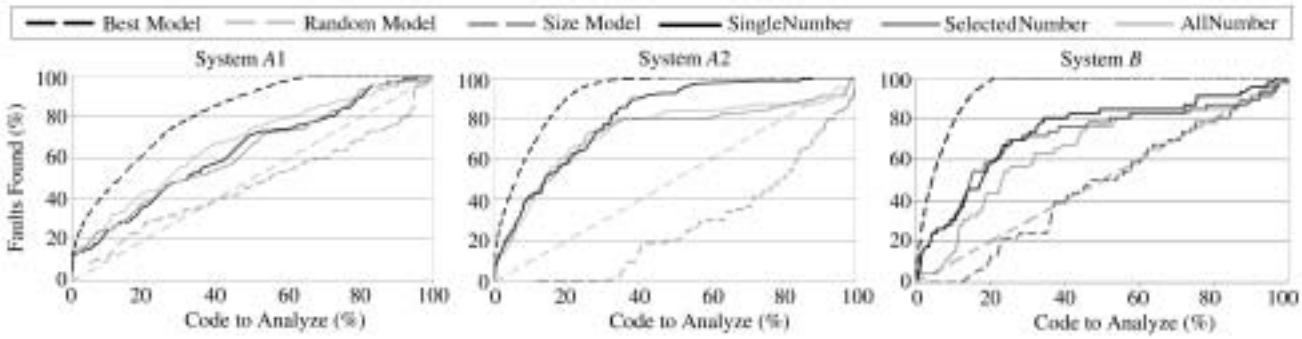

Fig.4. Efficiency comparison of the models that predict the number of faults. The models were built on the data from System $A1$. Three reference models (Best, Random, and Size) are introduced to provide a baseline for comparison.

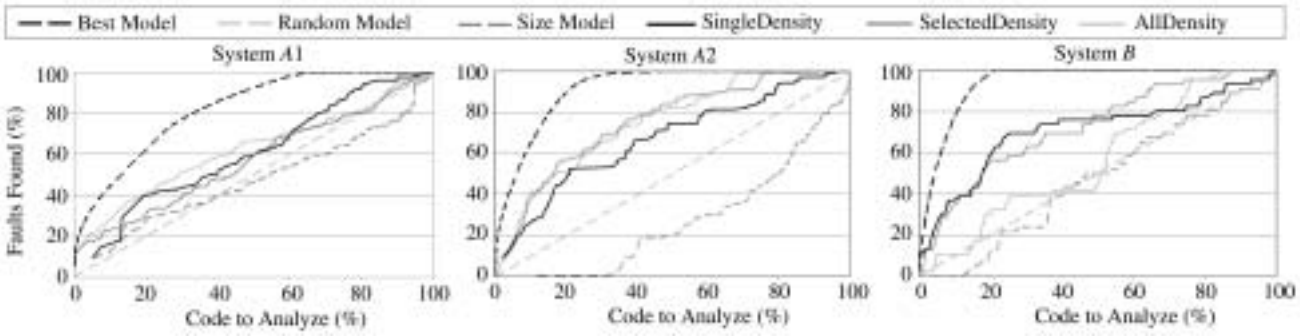

Fig.5. Efficiency comparison of the models that predict the fault density. The models were built on the data from System $A1$. Three reference models (Best, Random, and Size) are introduced to provide a baseline for comparison.

only for System $A1$, on which the models are built, but this is also very clear for System $A2$ and System B. Our models seem to be stable and to work similarly well in all systems.

The evaluation of the models' performance when applied to System $A2$ and System B indicates which models are the most promising ones as prediction models, i.e., which models are the best for predicting faults in projects other than the project they are built on. It seems that the least complex model (SingleNumber) works best. For example, in both System $A1$ and System B the model makes it possible to detect over 80% of faults by examining only 40% of the code. It is, on average, twice as many faults as we would detect when inspecting the code randomly and only about 10%~15% less than the possible maximum described by the Best model.

After evaluating the models that predict the number of faults we perform an evaluation of the models that predict fault density. The performance of the SingleDensity, SelectedDensity, and AllDensity models is presented in Fig.5. As in the case of models that predict the number of faults, the models that predict the fault density in most cases have a clear advantage over the Random model. Although the gain from using them is slightly but noticeably lower compared to the models

Table 5. Quantification of the Gain from Using the Model

	SingleNumber	SelectedNumber	AllNumber	SingleDensity	SelectedDensity	AllDensity	Size
System $A1$	0.42	0.43	0.52	0.32	0.22	0.38	−0.09
System $A2$	0.75	0.55	0.57	0.41	0.61	0.60	−0.54
System B	0.55	0.49	0.35	0.47	0.51	0.16	−0.06
Average Gain	**0.57**	**0.49**	**0.48**	**0.40**	**0.45**	**0.38**	**−0.23**

Note: Gain measures the improvement of the efficiency over the random model as a percentage of the improvement offered by the Best model. The closer the values are to 1 the better the models is. Values larger than 0 indicate that the model offers an improvement over Random model. For details concerning the Gain metric see Subsection 3.3.

that predict the number of faults (compare with Fig.4) the density prediction models are still an improvement over not using any model at all (i.e., using Random model).

When it comes to evaluating the stability, the SingleDensity and SelectedDensity models are stable in providing improvement over the Random model in all three systems. The AllDensity model works well in System $A1$ and System $A2$, but it does not in System B. It is probably an example of overfitting — the model is very much based on the unique characteristics of System A, which are present in two subsequent releases of the same system A ($A1$ and $A2$) but are not present in System B.

As a next step, we calculate the *Gain* for our models (see Subsection 3.3 for details concerning the *Gain* metric). The results are presented in Table 5. In Table 5 we quantify the gain for each model when applied to each system as well as provide an average gain for each model.

The numbers from Table 5 support our findings from the analysis of the diagrams. From the models predicting the number of faults, SingleNumber dominates over the other two models. The SelectedDensity model is the best model from the models that predict fault density. The SingleNumber model seems to be the best model of all, since it provides, on average, 57% of the maximum gain possible. SelectedNumber comes second, providing on average 49% of the maximum efficiency gain. Once again the poor performance of the Size model is confirmed — in all cases it is actually worse than the Random model.

Finally, we check if the differences between our models and the Random model are statistically significant. The hypothesis about equality of models was tested using Wilcoxon test (see Subsection 3.3 for details regarding the test and the interpretation of results). For all models (SingleNumber, SelectedNumber, AllNumber, SingleDensity, SelectedDensity, AllDensity, Size model, Best model) applied to all systems (System $A1$, System $A2$, System B) SPSS reported that the hypothesis about the equality of models can be rejected at 0.000 level, which practically means that the differences are significant for any conventional significance level. Additionally, the hypothesis for the Size model was rejected based on positive ranks, while for all other models it was rejected based on the negative ranks. It might be considered an indication, that the Size model is worse than the Random model (the sum of negative ranks was greater than the sum of positive ranks — see Subsection 3.3 for details), while all other models are better, which supports our conclusions from the analysis of Figs. 3∼5.

5 Discussion

5.1 Findings

The results obtained in our study are promising. All our models (AllNumber, SelectedNumber, SingleNumber, AllDensity, SelectedDensity, SingleDensity) represent a significant improvement compared to the Random model. It means that, when focusing fault detection efforts on a portion of the code only, more faults would be detected when using our model compared to analysing the classes in a random order. The exact value of the gain depends on the model selected, and the percentage of code analysed.

By analyzing Table 5 we can see that the best results are obtained when using the SingleNumber model. When applied to our three systems on average it produces 57% of the improvement of the Best model. The application to System $A2$ brings 75% of the maximum possible improvement. Application to System B brings 55% of the maximum possible improvement. The second best model, SelectedNumber, in the same situation brings 55% and 49% of the maximum possible improvement.

It is worth noticing that both our best models work well when relatively small percentages of code are analysed (see Fig.4 for SingleNumber and Fig.5 for SelectedDensity). For example, when we analyze about 40% of the code, then by following our two best models we should detect about 80% of faults. This is twice as many as if we were not following any model. A good performance when analyzing small percentages of code is probably of the largest practical value. This is the practical situation in which prediction models are most useful. If we decide to inspect 80% of the code even without using any model we already have a large statistical chance of finding many faults (80% on average). Therefore, using models in such a case must lead to a smaller benefit, basically because there is a much smaller room for improvement.

Another interesting finding from Table 5 is that in case of almost all models their performance is better when they are applied to System $A2$ compared to their performance when they are applied to System B. This seems to be reasonable, as System $A2$ is the next release of System $A1$ on which the models were built. It might mean that models produced within one product line have the best potential accuracy. However, this does

not need to be a rule — in our case too we can see that in some cases our models work better in System B than in System $A2$, e.g., SingleDensity. The models that do exceptionally bad when applied to System B are the models based on all metrics (AllNumber, and AllDensity). An explanation might be that such models tend to overfit the dataset they were built on and therefore lack generality. AllNumber, and AllDensity work reasonably well in the case of System $A2$ but significantly worse in System B. This is the most apparent in case of the AllDensity model, which provides 60% of the maximum improvement in System $A2$ and only 16% in the case of System B. That would suggest that the models based on large number of metrics (i.e., AllDensity, AllNumber) be tightly fit to the unique characteristics of System A, which are present in Systems $A1$ and $A2$ but are not present in System B. Therefore, it seems that models based on a smaller number of metrics have better potential for stability and transferability to systems other than the system they were built on.

One more general finding from our study is that for modified code the class size is not a good predictor of fault density. This can be observed in Table 5, where the application of the Size model brings bad results in all our systems. In all cases the Size model is, on average, even worse than the Random model. Table 5, however, only presents average values. It might be that the Size model works well when a small percentage of code is analyzed and becomes really bad afterwards. Such a situation would indicate some applicability of the Size model. However, by analyzing Fig.3 we clearly see that it is not the case. In neither of our systems the Size model is significantly better than the Random model for small percentage of the code (only in System $A1$ it is slightly better for the first 30% of the code, but the improvement is not large).

Our results also support findings of other researchers[13,31] that considered relative modification measures (i.e., the size of modification divided by the size of a code unit) as the best for predicting fault densities of modified classes. Our most successful model, SingleNumber, is based on such a relative modification measure.

Another general finding is that our dataset supports the Pareto principle (majority of faults are accumulated in a minority of code) for modified code. It is, however, difficult to pin-point *which* Pareto principle it exactly supports. It seems that System $A1$ follows the 60/20 rule stating that 60% of the faults can be found in 20% of the code. System $A2$ is closer to the classical 80/20 rule, while System B actually supports the extreme 80/10 rule.

5.2 Validity

As suggested in [39] we distinguish between four types of validity: internal, external, construct and conclusion validity.

The *internal validity* "concerns the causal effect, if the measured effect is due to changes caused by the researcher or due to some other unknown cause"[43]. Since our study is mostly based on correlations, by definition we cannot claim the causal relationship between our dependent and independent variables. However, it is also not our ambition to claim that. There can be (and probably is) an underlying third factor that demonstrates itself in both dependent and independent variables and therefore it is possible to predict one of them using another. Because of that, by finding correlations we are able to build a useful prediction model.

The *external validity* concerns the possibility of generalising the findings. The study was performed on two systems, which are representative for systems of their class (i.e., telecommunication systems). The systems are rather large (up to 600 KLOC). In order to increase the external validity we have evaluated the models using the data different from the data used to build the models. One threat to external validity can be that all systems used in this study are telecommunication systems and that they were produced in the same company, which may make them somewhat similar. In the future we plan to evaluate our models in other kinds of systems developed by other companies.

The *construct validity* "reflects our ability to measure what we are interested in measuring"[43]. One thing that may be worth discussing is the assumption that an effort connected with the fault detection activities is proportional to the size of the class. Many other studies consider the cost of detecting faults in the class to be a fixed value and therefore evaluate models only by how well they detect faults. We believe that the size of a class is a better cost indicator. At first we also considered the size of a change as a possible effort estimation metric. It is, however, not enough to analyse only the modified code, since the modification can violate some more general class assumption and result in fault in a part of the class that was not modified. Therefore, we selected size of the class for estimating analysis effort.

The *conclusion validity* concerns the correctness of conclusions we have made. When discussing conclusion validity we want to assess to what extent our conclusions are believable. The conclusion validity is mostly interested in checking if there is a correct relationship (i.e., statistically significant) between the variables. Therefore, where possible, we have presented the statistical significance of our findings.

6 Conclusions

The goal of this study was to build prediction models that would increase the efficiency of fault detection in modified code. We have built a number of models based on data collected from one release of a large telecommunication system. The objective of the model was to predict fault density in the classes. The models were evaluated using the next release of the system on which the

models were built, as well as another large telecommunication system. The evaluation was performed against three reference models: a model based on random selection of the classes for analysis, the theoretical best model, and a simple model based on the size of the class.

We have found that our models provide a stable improvement compared to both the random-and the size-based models. Our models are able to provide, on average, 38% to 57% of the maximal theoretical improvement in fault detection efficiency. The difference in performance of our models as compared to the random model was shown to be statistically significant.

As the most promising, we have found a model that predicts the number of faults based on the number of new and modified lines of code. The output of this model is divided by the class size to obtain the fault density. This model made it possible to achieve 75% of the maximum possible improvement when applied to the next release of the system on which it was built. When applied to a completely different system it achieved 55% of the maximum improvement. In both cases, these were the best results obtained for respective systems by any of our models.

The second most promising model was the one that predicted the number of faults based on the number of new and modified lines of code, the number of declarative and the number of executable statements in the class. This model made it possible to achieve 55% of the maximum possible improvement when applied to the next release of the system on which it was built, and 49% when applied to a different system.

We have also found yet another indication that models consisting of a small number of metrics are highly correlated to faults tend to behave better when applied to a new dataset, as compared to models which use a large number of metrics. Models that use many metrics tend to overfit the dataset on which they were built, which makes them less stable when applied to other datasets.

In this study we have also managed to find empirical evidence for a number of popular hypotheses concerning faults. Our findings support the findings of those researchers that consider the relative size of modification as the best fault density predictor in modified code. Our datasets also comply with the Pareto principle. We have found an evidence of the 60/20 rule (60% of the faults can be found in 20% of the code), but also the 80/20 and even the 80/10 rule. Another finding concerns the applicability of the size metric to predict the fault density. We have shown that for modified classes the class size is a poor predictor of class fault-density.

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.

References

[1] Sommerville I. Software Engineering. Boston: Addison-Wesley, 2004.

[2] Cartwright M, Shepperd M. An empirical investigation of an object-oriented software system. *IEEE Trans. Software Engineering*, 2000, 26(8): 786∼796.

[3] Boehm B, Basili V R. Software defect reduction top 10 list. *Computer*, 2001, 34: 135∼137.

[4] Fenton N, Ohlsson N. Quantitative analysis of faults and failures in a complex software system. *IEEE Trans. Software Engineering*, 2000, 26(8): 797∼814.

[5] Ohlsson N, Eriksson A C, Helander M. Early risk-management by identification of fault-prone modules. *Empirical Software Engineering*, 1997, 2(2): 166∼173.

[6] Briand L C, Wust J, Daly J W, Porter D W. Exploring the relationship between design measures and software quality in object-oriented systems. *The Journal of Systems and Software*, 2000, 51(3): 245∼273.

[7] Briand L C, Wust J, Ikonomovski S V *et al.* Investigating quality factors in object-oriented designs: An industrial case study. In *Proc. the 1999 Int. Conf. Software Eng.*, Los Angeles, USA, 1999, pp.345∼354.

[8] K El Emam, W L Melo, J C Machado. The prediction of faulty classes using object-oriented design metrics. *The Journal of Systems and Software*, 2001, 56(1): 63∼75.

[9] Tomaszewski P, Lundberg L, Grahn H. Increasing the efficiency of fault detection in modified code. In *Proc. Asian Pacific Software Engineering Conference*, Taipei, 2005, pp.421∼430.

[10] Zhao M, Wohlin C, Ohlsson N, Xie M. A comparison between software design and code metrics for the prediction of software fault content. *Information and Software Technology*, 1998, 40(14): 801∼809.

[11] Pighin M, Marzona A. An empirical analysis of fault persistence through software releases. In *Proc. the Int. Symp. Empirical Software Engineering*, Rome, Italy, 2003, pp.206∼212.

[12] Pighin M, Marzona A. Reducing corrective maintenance effort considering module's history. In *Proc. Ninth European Conference on Software Maintenance and Reengineering*, Manchester, UK, 2005, pp.232∼235.

[13] Selby R W. Empirically based analysis of failures in software systems. *IEEE Trans. Reliability*, 1990, 39(4): 444∼454.

[14] Gascoyne S. Productivity improvements in software testing with test automation. *Electronic Engineering*, 2000, 72(885): 65∼67.

[15] A Gunes Koru, J Tian. An empirical comparison and characterization of high defect and high complexity modules. *Journal of Systems and Software*, 2003, 67(3): 153∼163.

[16] Khoshgoftaar T M, Allen E B, Deng J. Controlling overfitting in software quality models: Experiments with regression trees and classification. In *Proc. The 7th Int. Software Metrics Symposium*, London, UK, 2001, pp.190∼198.

[17] Khoshgoftaar T M, Allen E B, Jianyu D. Using regression trees to classify fault-prone software modules. *IEEE Trans. Reliability*, 2002, 51(4): 455∼462.

[18] Khoshgoftaar T M, Allen E B, Jones W D, Hudepohl J P. Accuracy of software quality models over multiple releases. *Annals of Software Engineering*, 2000, 9(1-4): 103∼116.

[19] Khoshgoftaar T M, Seliya N. Fault prediction modeling for software quality estimation: Comparing commonly used techniques. *Empirical Software Engineering*, 2003, 8(3): 255∼283.

[20] Ohlsson N, Zhao M, Helander M. Application of multivariate analysis for software fault prediction. *Software Quality Journal*, 1998, 7(1): 51∼66.

[21] Tomaszewski P, Håkansson J, Lundberg L, Grahn H. The accuracy of fault prediction in modified code — Statistical model vs. expert estimation. In *Proc. 13th Annual IEEE Int. Conf. and Workshop on the Engineering of Computer Based Systems*, Potsdam, Germany, 2006, pp.334∼343.

[22] Chidamber S R, Darcy D P, Kemerer C F. Managerial use of metrics for object-oriented software: An exploratory analysis.

IEEE Trans. Software Engineering, 1998, 24(8): 629∼639.
[23] Nikora A P, Munson J C. Developing fault predictors for evolving software systems. In *Proc. The Ninth Int. Software Metrics Symposium*, Sydney, Australia, 2003, pp.338∼349.
[24] Ostrand T J, Weyuker E J, Bell R M. Predicting the location and number of faults in large software systems. *IEEE Trans. Software Engineering*, 2005, 31(4): 340∼355.
[25] Yu P, Systa T, Muller H. Predicting fault-proneness using OO metrics. An industrial case study. In *Proc. The Sixth European Conference on Software Maintenance and Reengineering*, Budapest, Hungary, 2002, pp.99∼107.
[26] Chidamber S R, Kemerer C F. A metrics suite for object oriented design. *IEEE Trans. Software Engineering*, 1994, 20(6): 476∼494.
[27] Fioravanti F, Nesi P. A study on fault-proneness detection of object-oriented systems. In *Proc. Fifth European Conf. Software Maintenance and Reengineering*, Lisbon, Portugal, 2001, pp.121∼130.
[28] Fenton N, Neil M. A critique of software defect prediction models. *IEEE Trans. Software Engineering*, 1999, 25(5): 675∼689.
[29] Keppel G. Design and Analysis: A Researcher's Handbook. Upper Saddle River: Prentice Hall, N.J., 2004.
[30] Milton J S, Arnold J C. Introduction to Probability and Statistics: Principles and Applications for Engineering and the Computing Sciences. Boston: McGraw-Hill, 2003.
[31] Nagappan N, Ball T. Use of relative code churn measures to predict system defect density. In *Proc. the 27th Int. Conf. Software Engineering*, St. Louis, USA, 2005, pp.284∼292.
[32] Munson J C, Elbaum S G. Code churn: A measure for estimating the impact of code change. In *Proc. the Int. Conf. Software Maintenance*, Bethesda, USA, 1998, pp.24∼31.
[33] Graham I. Migrating to Object Technology. Wokingham, England; Reading, Mass.: Addison-Wesley Pub. Co., 1995.
[34] Henderson-Sellers B, Constantine L L, Graham I M. Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design). *Object Oriented Systems*, 1996, 3(3): 143∼158.
[35] Scientific Toolworks Inc. Understand C++ Homepage. Last accessed on 2006-03-13.
[36] LOCC Project Homepage. The Collaborative Software Development Laboratory. University of Hawaii, USA, last accessed on 2005-11-10, http://csdl.ics.hawaii.edu/Tools/LOCC/.
[37] Kitchenham B, Pickard L, Pfleeger S L. Case studies for method and tool evaluation. *IEEE Software*, 1995, 12(4): 52∼62.
[38] Kitchenham B A, Pfleeger S L, Pickard L M *et al.* Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Software Engineering*, 2002, 28(8): 721.
[39] Wohlin C, Runeson P, Höst M *et al.* Experimentation in Software Engineering: An Introduction. Boston: Kluwer, 2000.
[40] Rees D G. Essential Statistics, London; New York: Chapman & Hall, 1995.
[41] SPSS Inc., SPSS Package Homepage, http://www.spss.com/, last accessed on 2006-03-13.
[42] Aczel A D, Sounderpandian J. Complete Business Statistics. Boston: McGraw-Hill, 2006.
[43] Host M, Regnell B, Wohlin C. Using students as subjects—A comparative study of students and professionals in lead-time impact assessment. *Empirical Software Engineering*, 2000, 5(3): 201∼214.

Piotr Tomaszewski received an M.Sc. degree in software engineering from Wroclaw University of Technology, Poland in 2002, and Ph.D. degree in systems engineering from Blekinge Institute of Technology, Sweden in 2006. His research interests include: software development productivity evaluation and improvement as well as software engineering decision support. He received the Best Paper Award at ECBS 2006 (IEEE International Conference on the Engineering of Computer Based Systems). Dr. Tomaszewski is a member of the IEEE.

Lars Lundberg received an M.Sc. degree in computer engineering from Linköping University, Sweden in 1986, and a Ph.D. degree in computer systems engineering from Lund University, Sweden in 1993. He has previously worked as a consultant for embedded systems, and is now a full professor in computer systems engineering at Blekinge Institute of Technology. From 1999 to 2004 he was the Dean of the Technical Faculty at Blekinge Institute of Technology. Professor Lundberg has published close to 100 articles in scientific journals and conferences. His research interests include: real-time systems, software engineering, high performance computing, scheduling and combinatorics.

Håkan Grahn is an associate professor of computer engineering at Blekinge Institute of Technology. He received an M.Sc. degree in computer science and engineering in 1990 and a Ph.D. degree in computer engineering in 1995, both from Lund University. His main interests are computer architecture, shared-memory multiprocessors, software architecture, and performance evaluation. He has authored and co-authored more than forty papers on these subjects and supervised one Ph.D. student. He received the Best Paper Award at the PARLE'94 (Parallel Architectures and Languages, Europe) Conference. From January 1999 to June 2002 he was head of the Department of Software Engineering and Computer Science. Dr. Grahn is a member of the ACM, the IEEE. For more information, please refer to URL http://www.ipd.bth.se/~hgr/.