Predicting Fault-prone Components in a Java Legacy System

Erik Arisholm[§] and Lionel C. Briand[§‡]

[§] Simula Research Laboratory
Department of Software Engineering
Martin Linges v 17, Fornebu
P.O.Box 134,
1325 Lysaker, Norway
erika@simula.no

[‡] Carleton University
Department of Systems and Computer Engineering
Software Quality Engineering Laboratory
1125 Colonel By Drive
Ottawa, ON K1S5B6, Canada
briand@sce.carleton.ca

ABSTRACT

This paper reports on the construction and validation of fault-proneness prediction models in the context of an object-oriented, evolving, legacy system. The goal is to help QA engineers focus their limited verification resources on parts of the system likely to contain faults. A number of measures including code quality, class structure, changes in class structure, and the history of class-level changes and faults are included as candidate predictors of class fault-proneness. A cross-validated classification analysis shows that the obtained model has less than 20% of false positives and false negatives, respectively. However, as shown in this paper, statistics regarding the classification accuracy tend to inflate the potential usefulness of the fault-proneness prediction models. We thus propose a simple and pragmatic methodology for assessing the cost-effectiveness of the predictions to focus verification effort. On the basis of the cost-effectiveness analysis we show that change and fault data from previous releases is paramount to developing a practically useful prediction model. When our model is applied to predict faults in a new release, the estimated potential savings in verification effort is about 29%. In contrast, the estimated savings in verification effort drops to 0% when history data is not included.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics – *process metrics, product metrics.* D.2.9 [**Software Engineering**]: Management – *software quality assurance (SQA).*

General Terms

Measurement, Design, Verification.

1. INTRODUCTION

The study presented in this paper was motivated by a practical problem encountered in a large telecom company. A large Java legacy system was being maintained and there was a constant shortage of resources and time for verification (testing, inspection). The quality assurance engineers wanted to investigate means to focus verification on parts of the system where faults were more likely to be detected. Though many studies on predicting fault-prone classes on the basis of the structural properties of object-oriented systems have been reported (Section 5), one specificity of the study presented here is the fact that we need to predict fault-proneness of a changing legacy system. We therefore not only need to account for the structural properties of classes across the system, but also for changes and fault corrections on specific releases and their impact on the code, among a number of factors potentially impacting fault-proneness. Another interesting issue to be investigated is related to the fact that past change and fault data are typically available in legacy systems and such data could be useful to help predicting fault-proneness, e.g., by identifying what subset of classes have shown to be inherently fault prone in the past.

The legacy system studied is a middleware system serving the mobile division in a large telecom company. It provides more than 40 client systems with a consistent view across multiple back-end systems, and has evolved through 17 major releases during the past seven years. At any time, somewhere between 30 and 60 software engineers have been involved in the project. The core system consists of about 1700 Java classes in about 110K SLOC[1]. The project had used the XRadar system [16] for some time to monitor the quality of the software as it evolved [11], but as the system expanded in size and complexity, QA engineers felt they needed more sophisticated techniques to focus verification activities on fault-prone parts of the system.

This paper will attempt to build a fault-proneness prediction model for this system and assess not only its accuracy but also its potential cost-effectiveness to focus verification on future releases. We will do so using measures of the history of changes and faults, code quality, class structure, and changes in class structure. We will discuss the factors that appear to be important indicators of fault-proneness and explain why that is the case. A simple and pragmatic methodology for assessing the cost-effectiveness of such fault-proneness models will also be described and used on our case study.

Note that this paper is the first one that reports on the construction and validation of fault proneness prediction models in the context of an object-oriented, evolving, legacy system and that explores in

[1] In addition, the system consists of 1000K SLOC of generated code, but this code is not considered in our study.

this context the issue of cost-effectiveness of using such models. We believe this is a crucial issue that is far too rarely addressed. As shown in this paper, results regarding the fault-proneness classification of components are often misleading and tend to inflate the potential usefulness of such predictions. The relationship between our study and related works will be further discussed in Section 5.

The remainder of this paper is organized as follows. Section 2 explains how we developed the fault-prediction model for the abovementioned system. Section 3 evaluates the resulting prediction model. Section 4 describes the main threats to validity. Section 5 relates our approach and results to existing research. Section 6 concludes and outlines directions for future research.

2. METHODOLOGY

2.1 Goal

Our goal is to build a prediction model for the quality assurance staff to determine where to focus verification effort in one important legacy, Java system. We therefore want to identify fault prone classes which can then be targeted by specific verification activities. Though many studies have been reported on detecting fault-prone classes in object-oriented systems, the specificity of this work is the legacy nature of the system under study. This has a number of implications that will be further described below.

2.2 Fault-proneness factors

The fundamental hypothesis underlying our work is that the fault-proneness of classes in a legacy, object-oriented system can be affected by the following factors:

- the structural characteristics of classes (i.e., their coupling)
- the amount of change (requirements or fault corrections) undertaken by the class to obtain the current release
- the coding quality of classes: coding style and practices, presence of redundant code
- other, unknown factors that are captured by the fault history of classes in previous releases
- the skills and experience of the individual performing the changes

Furthermore, it is also likely that these factors interact in the way they affect fault-proneness. For example, changes may be more fault-prone on larger, more complex classes. However, there is little theory we can rely on to help us uncover such interactions and we will see below how this is addressed in our analysis. We have no data on the skills and experience of developers in this study and therefore the last factor listed above will not be considered at this point in our analysis.

2.3 Dependent and independent variables

The dependent variable in our analysis is the occurrences of fault corrections in classes of a release. Typically a fault correction involves several classes and in our case, since our level of analysis is at the class level, we count the number of times a distinct fault correction was required in that class for developing a given release. This aims at capturing the fault-proneness of a class.

The independent variables are summarized in Table 1 and attempt to measure the factors in Section 2.2. They include various measures of class size, inheritance, coupling, and cohesion. These were captured on each release using two code analyzers: XRadar [16] and JHawk [10]. In addition, measures capturing the level of redundancy in the code, conformance to desirable coding practices and coding style have been captured using the XRadar system. Note that some of the variable names in Table 1 are not following standard terminology (e.g., *LOC*, which refers to *number of local methods*). However, this is how the variables are defined in the tools, and we decided to keep the names to facilitate replications in future studies that might use the same tools.

In addition, for each class, we capture the number of requirement changes and fault corrections performed on release n-1 that were required to obtain the current release n. The amount of change across classes in release n-1 is expected to affect the likelihood of fault corrections in release n.

Ideally, we would also want to measure the *size* of changes and fault corrections, for example in terms of lines of code changed, deleted, and added. However, because this data could not easily be obtained, we use surrogate measures that compute the variation in coupling, cohesion, and size measures between release n and n-1. These measures aim at capturing the size of change undergone by each class to build release n.

We also thought that history data, telling us about requirement changes and fault corrections in previous releases might be useful as they would tell us about the inherent fault-proneness and change-proneness of classes in past releases. This type of data is usually available for legacy systems as there is usually a long history of changes and fault corrections for most classes in a release. In this paper, due to the current limitations of our data, we cannot look further back than release n-2.

Last, we also collect information on the number of releases in which the class had been present, assuming that older classes were more stable and less likely to contain faults than newer classes.

2.4 Assumptions and caveats

Our analysis strategies and the independent variables we have defined above assume that most of the faults corrected in a release n are related to changes and code characteristics of release n-1 and to a lesser extent n-2. This is, however, not true in general and it would have been desirable to obtain precise information regarding the cause-effect relationships between changes and fault corrections. However, this assumption was deemed reasonable based on our discussion with developers and more precise information was impossible to obtain at the time of writing. This assumption may, however, affect the ability of our model to accurately predict fault-prone classes.

2.5 Design of the study

Recall that we model the probability of a fault correction in a class as a function of the independent variables mentioned above. Class fault-proneness in release n is therefore modeled as the probability that a given class will undergo a fault correction in release n+1.

Table 1. Summary of the independent variables in the study

Variable	Description	Source
Violations	Number of violations in the code, e.g., "X defined but not used"	XRadar
Duplications	Number of “copy+paste” duplication segments in the code	XRadar
StyleErrors	Number of coding style errors, e.g., "Line is longer than 80 characters"	XRadar
Functions	Number of implemented methods in a class	XRadar
Ncss	Number of non-commentary source statements (SLOC)	XRadar
Javadocs	Number of formal Javadoc comments	XRadar
Ccn	Cyclomatic complexity	XRadar
LCOM	Lack of Cohesion	JHawk
EXT	Number of external methods called	JHawk
HIER	Number of methods called in class hierarchy	JHawk
LOC	Number of local methods called	JHawk
INST	Number of instance variables	JHawk
MOD	Number of modifiers, i.e., the number of methods that can change the objects’ state	JHawk
INTR	Number of interfaces implemented	JHawk
PACK	Number of packages imported	JHawk
RFC	Response for a class	JHawk
MPC	Message passing coupling	JHawk
FIN	The sum of the number of unique methods that call the methods in the class	JHawk
FOUT	Number of distinct non-inheritance related classes on which a class depends (CBO)	JHawk
NSUP	Number of superclasses	JHawk
NSUB	Number of subclasses	JHawk
dLCOM	Abs(LCOM(n)-LCOM(n-1))	JHawk
dLOC	Abs(LOC(n)-LOC(n-1))	JHawk
dMOD	Abs(MOD(n)-MOD(n-1))	JHawk
dFOUT	Abs(FOUT(n)-FOUT(n-1))	JHawk
dNSUP	Abs(NSUP(n)-NSUP(n-1))	JHawk
dNSUB	Abs(NSUB(n)-NSUB(n-1))	JHawk
FaultCorrections	Number of faults corrected in release n-1 to build release n	Release Data
ChangeCount	Number of requirement changes in release n-1 to build release n	Release Data
PrevVersionCount	Number of releases in which the class has been present before release n	Release Data
n1FaultCorrections	Number of faults corrected in release n-2 to build release n-1	Release Data
n1ChangeCount	Number of requirement changes in release n-2 to build release n-1	Release Data

We use logistic regression [6] to derive an optimal prediction model from the available data. More precisely, we have four releases (denoted R1 to R4) on which we collected fault and change data. We build a prediction model with R3 fault corrections as the dependent variable and R2 measurements (including number of requirement changes and fault corrections, code measures and structural change measures since R1) plus the change and fault history of R1 as the independent variables. Then the model is *applied* to predict R4 fault corrections, using R3 measurements (including number of requirement changes and fault corrections, code measures and structural change measures since R2) plus the change and fault history of R2 as the predictor variables. The rationale is that we want not only to see how well such a model can fit our data (on R3) but we also want to determine how well it can predict the future (R4) and help focus verification on future releases.

2.6 Data analysis

Many of our independent variables have distributions skewed to the right, with a number of extreme values. Because regression is in general sensitive to outliers (e.g., very large values), we perform a logarithm transformation of independent variables[2] in order to obtain less skewed distributions [6]. This has also the advantage to account for interactions between variables without having to explicitly specify them[3], an important issue in our context where we know interactions to be plausible but difficult to predict beforehand.

[2] $x' = \ln(x+1)$. We refer to these variables as log-transformed independent variables.

[3] $a \ln x1 + b \ln x2 + c \ln x1x2 = (a+c) \ln x1 + (b+c) \ln x2$

Following a common analysis procedure [2], we perform a Principal Component Analysis (PCA) to identify the dimensions actually present in the data (for release R2) and to help us interpret subsequent results. We then use logistic regression to perform a univariate analysis of each independent variable to identify which ones are significant predictors of fault-proneness (fault corrections to build release R3) and whether their relationship is in the expected direction. This allows us to check whether our initial hypotheses are supported. The next step is then to build a multivariate prediction model, using stepwise logistic regression, in order to predict class fault-proneness using all available measures[4]. Note that in order to obtain a balanced model for both faulty and non-faulty classes, we randomly extract a subset of non-faulty classes of identical size to the number of faulty classes. This is necessary in order to avoid biasing the fitted model towards non-faulty classes as those typically represent the vast majority of classes in a release[5]. We thus obtain a total of 82 observations to build the model. The goodness of fit of the model is then assessed by computing the percentage of false positives and false negatives when using it to classify classes in the modeling dataset as fault-prone or not. To get a more realistic result of what classification accuracy to expect on other datasets, we perform the same assessment again but using a Leave-one-out cross validation procedure [6].

We then apply the multivariate model to determine how well it can predict the fault corrections (to build release R4), now including all 1758 observations. This step of our analysis is related to assessing the cost-effectiveness of using the prediction model that was built. Typically, fault-proneness models are found useful to focus verification activities, such as testing and inspections, on parts of a large system. The acceptance criteria for testing (e.g., control flow coverage) may, for example, be more demanding for parts predicted as fault-prone. Regardless of the details of its application, the cost-benefit of using such a prediction model to focus verification decreases as fault-prone parts represent an increasingly larger part of the system and contain a lower percentage of the faults. Therefore, we will investigate the percentage of classes, functions, and lines of code classified as fault-prone and the relationship of such percentages to the percentage of faults contained in these artifacts. These faults represent the fault subset that can potentially be detected by additional verification activities targeting fault-prone components.

3. RESULTS

3.1 Principal Component Analysis

A Principal Component Analysis (PCA) shows that the data captures a number of distinct dimensions (principal components or PCs) which is far lower than the number of independent variables considered, given the criterion of an eigenvalue above 1.0 to determine the number of components. That type of redundancy in software engineering data is very common [2] and needs to be identified to better interpret the results of our study. PCA is performed here on the log-transformed variables which are used as independent variables, and then rotated using the VariMax rotation to facilitate the interpretation. The raw PCA results are provided in Appendix A. The PCs can be described as follows:

- PC1: The first principal component mostly captures class size (e.g., in terms of methods, line of code, control flow complexity) and import coupling from other classes (e.g. PACK, RFC, MPC, EXT, FOUT), which is in most studies associated with size [2]. Violations, Duplications, and StyleErrors are also part of this PC. So is dLOC, indicating that the amount of change in local methods called is related to class size. ChangeCount and n1ChangeCount are also part of this PC, due probably to the strong impact of class size on the likelihood of a class to undergo change. We note that FaultCorrections and n1FaultCorrections do not load above 0.5 on any of the components, but seem to be partly related to PC1, indicating that fault proneness is related to class size but also other dimensions (PC6).
- PC2: The number of releases in which the class has been present (PrevVersionCount), plus a number of structural change measures (dLCOM, dFOUT, dMOD) we collect to assess the impact of change from one release to the next on class structure. This is to be expected as, in our data, older classes tend to be much more stable and show less structural change.
- PC3: Cohesion (LCOM) and number of instance variables (INST), as classes with large numbers of instance variables tend to have lower LCOM values.
- PC4: The sum of the number of unique methods that call the methods in the class (FIN)
- PC5: The number of ancestors classes (NSUP) and its change (dNSUP). This indicates a shallow inheritance hierarchy in the system, and that the classes are seldom moved within an inheritance hierarchy (both are close to zero).
- PC6: The number of modifier methods (MOD) (and to some extent, FaultCorrections)
- PC7: The number of descendent classes (NSUB) and its change (dNSUB). As for PC5, this indicates a shallow inheritance hierarchy, and that the classes are seldom moved within an inheritance hierarchy (both are close to zero).
- PC8: Number of methods called in class hierarchy (HIER)

3.2 Univariate Analysis

From Table 2 we can see that code quality measures (violations, duplication, and style errors) are all significantly related to the probability of fault correction (p-value < 0.05) and the relationship is in the expected direction: poor code quality leads to increased probability of correction (odds-ratio[6] > 1).

[4] We do not make use of PCA to select a subset of independent variables since, as discussed in [2], experience has shown this usually leads to suboptimal prediction models, though regression coefficients are easier to interpret.

[5] In the univariate case, we also tried an alternative (and computationally expensive) approach known as *exact* logistic regression [9], which computes unbiased coefficients even with sparse or skewed data sets. With our data, the differences between the two approaches are negligible.

[6] It is the number by which we would multiply the odds for a class to contain a fault for each one-unit increase in the independent

Table 2. Summary of univariate results

Variable	Odds-ratio	p-value
Violations	1.8	0.0037
Duplications	2.8	0.0101
StyleErrors	1.4	0.0387
Functions	3.2	0.0001
Ncss	3.1	<.0001
Javadocs	4.4	<.0001
Ccn	2.9	<.0001
EXT	2.3	<.0001
LOC	2.6	0.0003
INST	1.7	0.0409
PACK	2.2	0.0004
RFC	3.1	<.0001
MPC	2.1	<.0001
FOUT	2.5	<.0001
dLOC	2.2	0.0116
dFOUT	1.9	0.0048
FaultCorrections	5.3	0.0055
ChangeCount	2.3	0.0072

Size measures (Functions, Ncss, Javadocs, Ccn) also show a very strong impact on the probability of fault correction, as this has been the case in many previous studies [2].

The number of external methods (EXT) and local methods (LOC) called also show a significant relationship where the higher the number of calls, the higher the probability of fault correction. This is also the case of PACK, the number of packages imported, and other types of import coupling measures: RFC, MPC, FOUT. Note, however, that all these measures belong to the same PC as size measures, which is to be expected as larger classes tend to perform more calls, import more packages, and so forth.

No inheritance or cohesion measure appears to be significantly related to fault-proneness. Recall that there is limited use of inheritance in this system, as also reported by other studies [4, 5, 8]. As for cohesion, as discussed in [2], our result is consistent with previous studies that have shown that measures of cohesion were rarely selected as significant predictors of fault-proneness.

Some of the delta measures, aiming at measuring the change incurred by classes between two releases, are significantly related to fault-proneness: dLOC, dFOUT. However, the fact that dLOC is related to size (PC1) might explain part of this result.

Change counts and fault corrections show that the higher the number of change and error corrections on a class in the previous release (R2), the higher the likelihood of fault correction in the current release (R3). This is intuitive as we expect that the amount of change performed to build R2, whether requirements changes or corrections, would impact fault-proneness. However, historic data on change and fault counts to build release R1 (n1ChangeCount and n1FaultCorrections) do not seem to have a significant relationship with fault-proneness.

variable. For example, an odds-ratio of 1.8 means an increase of 80% in the odds of a class to contain a fault.

Table 3. Variables selected in the multivariate model

Variable	p-value
Ncss	0.0069
Javadocs	0.0088
Ccn	0.0137
LCOM	0.0817
HIER	0.0297
INST	0.0221
NSUP	0.0654
NSUB	0.0425
FaultCorrections	0.0098
ChangeCount	0.0225
n1FaultCorrections	0.0099
n1ChangeCount	0.0297

3.3 Multivariate Analysis

As expected, the stepwise logistic regression procedure selected a number of size measures (PC1) in the multivariate prediction model (Ncss, Javadocs, Ccn). The number of instance variables (INST) is also selected. Also, not surprisingly, variables capturing the amount of change (ChangeCount) and fault correction (FaultCorrections) are selected. A number of variables which appeared significant in the univariate analysis are not selected and this is not surprising as many of them belong to the same principal components. However, more surprisingly, a number of variables who did not appear significant in the univariate analysis are also included. Recall that our model automatically accounts for interactions due to the log-transformations of independent

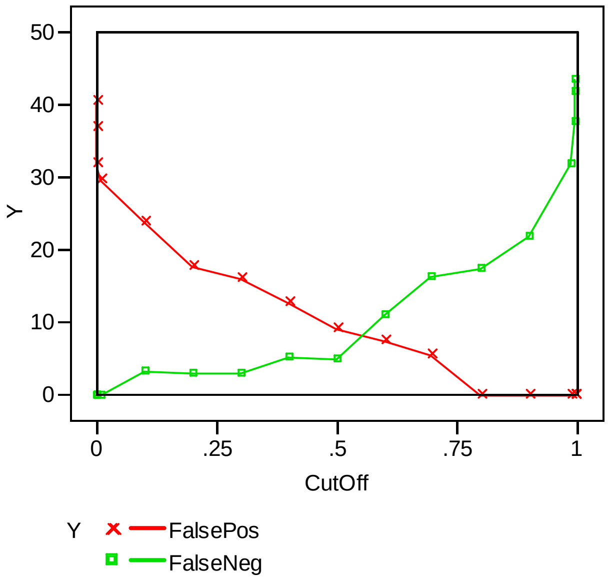

Figure 1. Goodness of fit in terms of false positives and false negatives on release R3

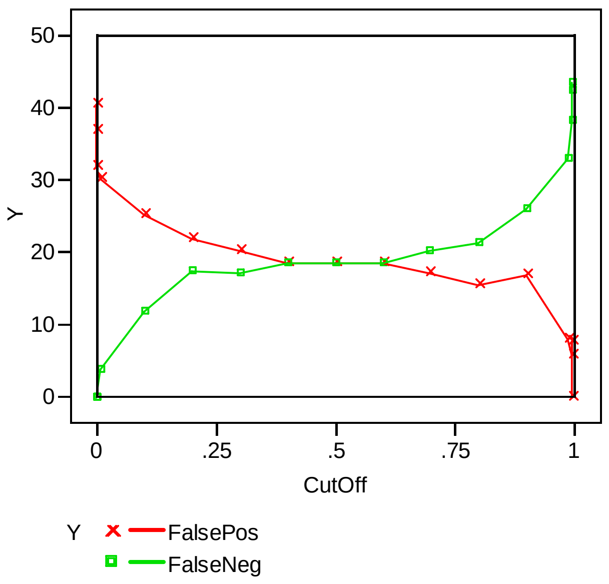

Figure 2. Leave-one-out cross validation false positives and false negatives on release R3

variables and this probably explains such a result. A number of code measures are thus selected regarding inheritance (NSUP, NSUB, HIER) and cohesion (LCOM). Furthermore, variables capturing change and error history before the development of the analyzed release (n1ChangeCount, n1FaultCorrections) are also selected.

There are many ways to look at the goodness of fit of a logistic regression model [6]. One practical way is to use the probability predictions of the model to classify classes as faulty or not and check the false positives and false negatives. This is illustrated in Figure 1 for different probability thresholds. We can see that when using a balanced probability threshold value for classification (around 0.5), we have less than 9% and 5% of false positives and false negatives, respectively. In other words, when predicting classes as fault-prone there is a 9% chance of being wrong and waste verification resources. Similarly, there is a 5% chance of missing a faulty class and not detect the faults it contains.

In order to obtain a more realistic assessment of the fault-proneness prediction accuracy, we use a Leave-one-out cross validation procedure, leaving one observation out and building a model on the remaining observations, doing so iteratively for all observations. From Figure 2, we can see that when using a balanced threshold value (around 0.5), we have less than 20% of false positives and false negatives, respectively. This is a sharp increase compared to Figure 1 but the results are still reasonably accurate.

3.4 Cost-Benefit Analysis

The cross validation results presented above provide insights on the accuracy of our fault-proneness models if all releases were alike in terms of change process, changes, personnel, and so on. However, we know this is never the case. In this case study, we were informed that process changes were taking place across releases. So, from a practical perspective we wanted to answer the following questions:

- how useful is such a prediction model when predicting future releases?
- what is the cost-benefit of using such a model to focus verification?

We built a prediction model using R3 fault corrections as dependent variable and R2 data for the independent variables. Then we applied the model to predict R4, the latest release for which we have fault correction data. Since we are looking at ways to get insights into the cost-effectiveness of using such model we have to define surrogates measures for verification cost. Depending on the specific verification undertaken on classes predicted as fault prone, one may want to use a different size measure that would be proportional to the cost of verification. In Figure 3, we plot the number of classes predicted as fault prone as well as the cumulative number of lines of code (Ncss), methods (Functions), and control flow complexity (Ccn) they contain. We compute this for different threshold probability cutoff values, as for the computation of false positives and false negatives. From Figure 3 we can see that at a cutoff value of 0.5, more than 70% of the faults are found in less than 30% of the classes. Those classes, however, represent roughly 50% of the lines of code, control flow complexity, and methods. So depending on whether you intend to apply, for example, test criteria such as Block coverage, Edge coverage, or simply inspect class interfaces [15], one may want to use a different size measure that is a proper surrogate of the effort of a specific verification activity. In our case the measurement happens to be very consistent for all our size measures, except the number of classes which shows lower percentages. This is not surprising as larger classes tend to be more fault-prone than smaller classes.

Going back to our example, assuming the number of lines of code is an appropriate size measure that is proportional to the verification effort of the predicted fault-prone classes, a random selection of classes on which to apply a specific verification strategy (e.g., test or inspection) would require the verification of 70% of the code to detect a maximum of 70% of the faults. Using our model we bring this percentage down to 50% of the code, thus potentially reducing the verification effort by 29%. An alternative to using a model would be to ask the maintainers of the system to subjectively select classes to verify. However, in the context where this study took place, no maintainer seems to have the adequate overview of the system and release changes to order or classify parts of the system according to their fault-proneness. We believe that this is a common problem when the system being maintained is large and involve a large number of developers.

In order to assess the results presented above, we should compare them against a baseline capturing the best we could achieve. In release R4, fault-prone classes represent 13% of the lines of code. We therefore see that, despite the estimated potential savings in verification effort, there exists substantial room for improvement in our prediction model.

As discussed above, a specificity of legacy systems is that we have access to change and fault history data from previous releases. When predicting the fault-proneness of a given release we can attempt to make use of change and fault data from past releases to improve our models. As discussed in 3.3, recall that

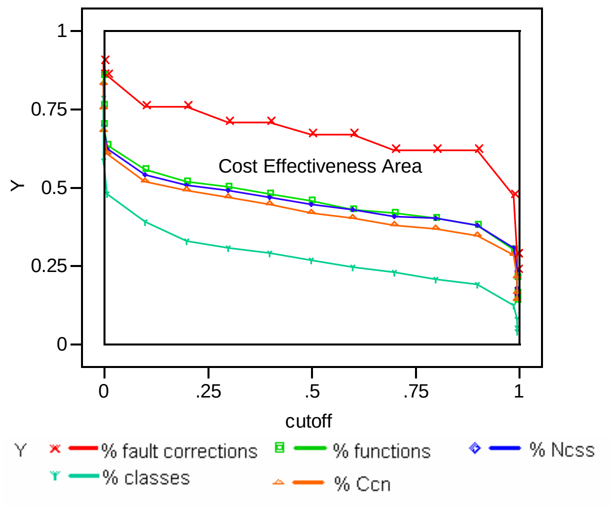

Figure 3. Cost-effectiveness of fault-proneness prediction model

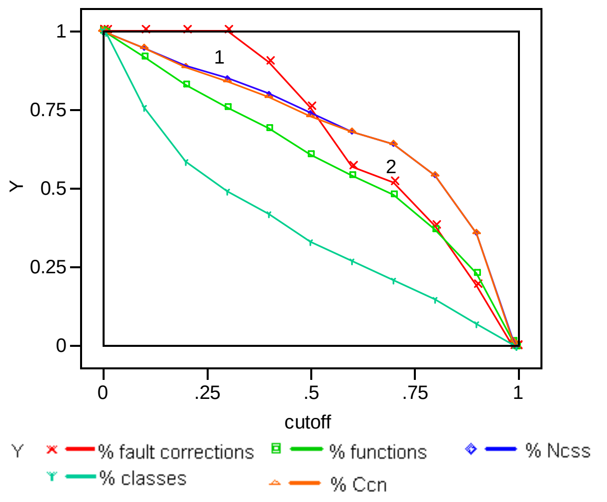

Figure 4. Cost-effectiveness of fault-proneness prediction model, without historical data

two covariates capturing fault corrections and changes in release R1 were selected in the prediction model along with other independent variables based on R2 data. Now an interesting, practical question is to determine how important such history data is for building prediction models for legacy systems in terms of cost-effectiveness. Though our history data here is limited to one past release (R1), what is the consequence of not using such information? Answering such a question could help us provide practical advice to practitioners regarding whether to collect and use such history data to build similar prediction models. Figure 4 shows the same cost-effectiveness curves as in Figure 3 but on a basis of building a model without history data. From this figure, the results clearly show that the size curves (except for the class percentage curve, for reasons already discussed above) across different thresholds are not very much above (and often below) the model fault percentage curve. Based on the assumptions listed earlier, this implies that such a model is not likely to be of practical usefulness as its cost-effectiveness to drive verification is questionable. Though additional studies with longer fault and change history are needed, it seems of practical importance to exploit such history data to build high cost-effectiveness prediction models.

For the sake of comparing the cost-effectiveness of models, it would be convenient to have an assessment that is independent from a specific probability threshold. This can be achieved by computing the surface area between the Fault correction curve and any appropriate size curve (e.g., Ncss) across probability thresholds. For example, in Figure 3, the area labeled "Cost Effectiveness Area" would quantify the cost effectiveness of our model in a way that can be easily compared to other models. The larger this area, the more cost-effective the fault-proneness model across thresholds. From Figure 4, we can see that the fault correction and Ccn curves cross each other at threshold 0.5. In general, when a size curve lies above the fault correction curve, the surface area should count negatively towards cost-effectiveness (area "2" in Figure 4) whereas when it lies below the fault correction curves (area "1" in Figure 4), then there is a positive contribution.

4. THREATS TO VALIDITY

The design of case studies is complex and many issues may affect the validity of results [17]. Construct validity concerns establishing correct operational measures for the concepts being studied. Many of our independent variables are structural measures collected using the XRadar and JHawk tools. The descriptions of the measures as provided in the tools are very informal and not sufficiently precise to determine exactly *how* the tools collect the measures, which would be important to provide an accurate assessment of their construct validity, as advocated in [3], for example. In any case, the repeatability of the measurements, also referred to as reliability in [17], is probably more important than construct validity in our case, as the goal is to build a prediction model rather than discovering cause-effect relationships.

The change and fault data in this study are related to the number of changes and number of faults corrected at the class-level in a given release. The reliability of the measures depends on how consistently the developers reported changes and fault corrections in their configuration management system. Due to the extensive use of the XRadar system in the releases being studied and due to the efforts of a dedicated release manager [11], we have good reasons to believe that the reported data are fairly accurate. However, some classes changed names or package location from release to release, and in such cases history data was lost. We expect to fix this problem in the future and thus improve our predictions.

There is no concept of fault *severity* in our models, and such a distinction would be important to improve the value of the prediction model [1]. It also concerns both internal validity and external validity, as the distribution of fault severity may change over time in the given project, and may be entirely different in other projects, respectively. Future prediction models should thus account for the severity of faults.

5. RELATED WORK

There is a lot of research on fault-proneness models in OO systems. A survey is provided in [2]. In summary, structural properties of software, such as cyclomatic complexity or coupling, do seem to have an impact on fault proneness. Due to space constraints, the remainder of this section focuses on providing an in-depth discussion of research that has attempted to build fault-proneness prediction models in the context of evolving software systems for which change and/or fault history data were available.

Instead of using product measures such as lines of code or cyclomatic complexity, one study took a slightly different approach, and attempted to build change risk prediction models on the basis of simple change data [12]. The results suggest that the *risk* of performing a given change, in the sense that the change will cause a future software failure, can be successfully modeled on the basis of simple change measures such as the size, duration, diffusion and type of change. The final risk prediction model also included developer experience (measured as the number of deltas previously performed by a given developer on the system before the given change) as a covariate.

A study on fault proneness prediction in a large, evolving system was reported in [13]. The dependent variable was the module-level cumulative number of faults (from the initial build to the last build). The independent variables were based on 12 different size and control-flow measures, but the measures were not used directly. Instead, principal component regression [6] was performed. More specifically, the 12 initial measures were transformed into three new variables on the basis of three identified principal components. The advantage of such an approach is that the derived measures are uncorrelated, which in turn makes it simpler to interpret the individual contribution of each measure. But as previously discussed, the drawback is that it often leads to suboptimal prediction models [2].

Finally, for each of the three variables, the sum (across all builds) of the absolute values of the differences between all pairs of successive builds was computed. These sums of differences, denoted as *code churn* in [13], formed the independent variables. The first independent variable (mainly representing size changes in the modules across the builds) was found to be a significant predictor of the cumulative number of faults in a module. The resulting multiple principal regression model had an adjusted R-Sq [6] equal to 0.61, suggesting that the model potentially could have practical value as a management tool to help identify fault-prone modules. There are several similarities and differences between the study reported in [13] and our paper. The *code churn* measures in [13] are similar to the structural change measures used as candidate independent variables in this paper (e.g, dLCOM), although we used the changes in the individual measures since the previous release instead of the cumulative changes in the principal component-based measures as independent variables. Thus the results are difficult to compare directly. However, two of the structural change measures (dLOC and dFOUT) were significant in the univariate case (Table 2), suggesting that changes in structural properties do affect fault proneness, thus confirming results in [13]. However, they were not significant in the multiple regression model (Table 3), that is, when also accounting for other measures such as the structural properties (in release n) and change and fault history, in which case the measures of structural change no longer seem to be significant predictors of fault-proneness in our case. No attempts was made to perform a cross-validation or evaluate the cost-benefits of the model on new releases in [13], so no further direct comparisons of the two studies are possible.

A study of fault-proneness in a very large and long-lived software system was reported in [7]. The dependent variable was the number of fault incidences within a two-year period. The independent variables consisted of various product and process measures collected from repositories just before that two-year period. The product measures included module-level size and complexity measures, e.g., lines of code and cyclomatic complexity. The process measures included the number of *past* faults in a module, the number of changes or deltas to a module over its entire history, and the *age* of the code. Thus, the variables used in [7] are similar in type to the ones used in our study. However, the goals were different as the main focus in [7] was to identify the *reasons* for faults whereas the main goal in this paper is to build an optimal prediction model. This has an impact on the type of statistical analyses performed. For example, [7] did not attempt to evaluate the accuracy or cost-effectiveness of the obtained models by applying them to predict *future* faults, though this was vital in our case. Despite the differences, a very important result in [7] was that process measures based on the change and fault history were much better explanatory variables of the faults than were the product measures. This is supported by a study reported in [18], which among others concludes that modules with large numbers of faults in the past are likely to also contain faults in the future. As can be seen when comparing Figure 3 and Figure 4, our results support the conclusions in [7, 18] in the context of predicting future faults. The inclusion of change and fault history data is essential in order to build practically useful fault-proneness prediction models for evolving legacy-systems. In [7], the best model was a so-called *weighted time damp model*, which allowed different weight to the change history data, depending on how *old* the changes were. The impact of changes on fault-proneness was downweighed by a factor of about 50 percent per year. In our case, the history data available to build our prediction model was only based on the two prior releases, so it was feasible to simply assign time-related weights by having different coefficients for n1ChangeCount, n1FaultCorrections, ChangeCount and FaultCorrections, respectively.

In [14], a case study on modeling fault-proneness over a sequence of four releases was presented. The system was composed of 800 KLOC of C code. The data was based on change reports as no design or code was available to the researchers. The independent variables thus only included measures such as the number of times a component was changed together with other components, number of files fixed in a given component and the number of lines of code added and deleted in a component, for a given release. A component was defined as a collection of files in the same directory, and it was assumed that the directory structure reflected the functional architecture. The relative size of the directories was not known. The components in the system were classified as fault-prone if the number of faults exceeded a given threshold value. One objective of the paper was to compare different statistical techniques (classification trees and discriminant analysis with or without the use of PCA to determine the set of candidate variables) for building fault-proneness models over a sequence of releases. Amongst others, they built fault-proneness classification trees (for each release) and evaluated the stability of the classification trees over successive releases. The results suggest that PCA combined with classification trees is a viable approach for modeling fault proneness as a software system evolves. However, due to the limitations in the data, it would not be possible to assess the cost-effectiveness of the obtained models on future releases.

On the basis of results summarized in [2], it appears that structural properties of software are useful predictors of fault proneness. However, even simple measures of the size, diffusion, and type of changes may be useful predictors of fault-proneness, as illustrated in [12]. This study also suggests that the experience of the developer performing changes should ideally be considered. Measures of the cumulated difference in structural properties over time can also be significant predictors of fault proneness [13]. However, on the basis of results in [7] and in this paper, it appears that whenever historic data of changes and faults are available, they should also be included as candidate predictors of fault proneness, because they will probably result in practically useful improvements in prediction accuracy. If history data from many releases are available, a weighted time damp model [7] seems to be a viable approach for prediction purposes.

To summarize, our study differs from existing work in several ways. It takes place in the context of an *object-oriented* legacy system and goes beyond predicting the fault-proneness of classes to look into the cost-effectiveness of predictions when using them to drive verification effort.

6. CONCLUSIONS

The main goal of this paper is to report on a study performed in a large telecom company and which focuses on predicting fault-prone parts of an object-oriented, legacy system after a new release is completed. The goal is to help QA engineers focus their limited verification resources on parts of the system likely to contain faults. Though many studies exist on the topic of predicting fault proneness, this is the first study doing so in the context of an *object-oriented*, evolving legacy software system. Such systems are bound to become increasingly more important in the future.

We make use of a variety of measures as independent variables ranging from structural measures, structural impact measures, code quality measures, change and fault measures, and history data from previous releases. Using logistic regression on log-transformed variables we build a multivariate prediction model to predict the probability of fault correction across classes, assess its goodness of fit, and its prediction capability on the release subsequent to the release on which the prediction model was built. Using a novel, simple, and pragmatic approach, we then assess the potential cost-effectiveness of using such a model to focus verification effort. This is rarely done in studies predicting fault-proneness but we believe this is essential to get a realistic estimate of how useful such models can be in practice.

A Leave-one-out cross validation yields less than 20% of false negatives and false positives when selecting a balanced fault correction probability threshold. The cost-effectiveness analysis suggests that, if such a prediction model would be used to focus verification, given a certain number of assumptions, it could result into a cost reduction of about 29%.

However, these results show there is substantial room for improvement in terms of false negatives and false positives. Given the limitations of our data this is not surprising. Nevertheless, our study shows that building such fault-proneness models is promising as it could potentially save verification effort in the context of a constantly changing legacy system. It also suggests that using history change and fault data about previous releases is paramount to developing a useful prediction model on a given release.

Future work will include collecting additional data on other releases and collect more precise change and fault data regarding the size and the cause of changes. Thus, we hope to develop more accurate prediction models leading to increased cost-effectiveness.

7. ACKNOWLEDGMENTS

Thanks to Kristoffer Kvam and Paul Skrede who made the data available to us. Thanks also to Valery Buzungu who assisted with the data extraction and to Hans Christian Benestad who helped us with the JHawk tool. The research was partly financed by the Research Council of Norway through the research project EVISOFT.

8. REFERENCES

[1] B. Boehm, "Value-Based Software Engineering," *ACM Software Engineering Notes*, 28(2), 2003, 1–12.

[2] L. C. Briand and J. Wuest, "Empirical Studies of Quality Models in Object-Oriented Systems," *Advances in Computers*, 59, 2002, 97–166.

[3] L. C. Briand, S. Morasca, and V. R. Basili, "Property-based Software Engineering Measurement," *IEEE Trans. on Software Eng.*, 22(1), 1996, 68–85.

[4] M. Cartwright and M. Shepperd, "An Empirical Investigation of an Object-Oriented Software System," *IEEE Trans. on Software Eng.*, 26(8), 2000, 786–796.

[5] S. R. Chidamber, D. P. Darcy, and C. F. Kemerer, "Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis," *IEEE Trans. on Software Eng.*, 24(8), 1998, 629–637.

[6] R. J. Freund and W. J. Wilson, *Regression Analysis: statistical modeling of a response variable*: Academic Press, 1998.

[7] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting Fault Incidence Using Software Change History," *IEEE Trans. on Software Eng.*, 26(7), 2000, 653–661.

[8] R. Harrison, S. Counsell, and R. Nithi, "Experimental Assessment of the Effect of Inheritance on the Maintainability of Object-Oriented Systems," *Journal of Systems and Software*, 52(2-3), 2000, 173–179.

[9] K. F. Hirji, C. R. Mehta, and N. R. Patel, "Computing Distributions for Exact Logistic Regression," *JASA*, 82, 1987, 1110–1117.

[10] JHawk, http://www.virtualmachinery.com/jhawkprod.htm.

[11] K. Kvam, R. Lie, and D. Bakkelund, "Legacy system exorcism by Pareto's principle," *proc. OOPSLA'05*, pp. 250–256.

[12] A. Mockus and D. M. Weiss, "Predicting Risk of Software Changes," *Bell Labs Technical Journal*, April-June 2000, 169–180.

[13] A. P. Nikora and J. C. Munson, "Developing fault predictors for evolving software systems," *proc. METRICS'03*, 2003, pp. 338–350.

[14] M. C. Ohlson, A. Amschler Andrews, and C. Wohlin, "Modelling fault-proneness statistically over a sequence of releases: a case study," *Journal of Software Maintenance and Evolution: Research and Practice*, 13, 2001, 167–199.
[15] J. Tian, *Software Quality Engineering.* Wiley, 2005.
[16] XRadar, http://xradar.sourceforge.net/.
[17] R. K. Yin, *Case Study Research, Design and Methods, 3rd edition*. Thousand Oaks, CA.: Sage Publications, 2003.
[18] T. J. Yu, V. Y. Shen, and H. E. Dunsmore, "An analysis of several software defect models," *IEEE Trans. on Software Eng.*, 14(9), 1988, 1261–1270.

Appendix A – Principal Component Analysis of the independent variables on R2

Table 4. Principal Component Analysis, using the VariMax rotation, eigenvalues >= 1.0

Rotated Factor Pattern								
Variable	**PC1**	**PC2**	**PC3**	**PC4**	**PC5**	**PC6**	**PC7**	**PC8**
LogViolations	**0.79**	-0.16	0.22	0.10	0.12	0.01	-0.02	-0.11
LogDuplications	**0.68**	-0.25	0.19	0.03	0.25	-0.07	0.12	0.08
LogStyleErrors	**0.67**	-0.30	0.15	-0.02	-0.08	-0.04	-0.30	-0.02
LogFunctions	**0.85**	-0.05	0.20	0.00	-0.07	-0.23	0.19	-0.14
LogNcss	**0.93**	-0.01	0.25	-0.04	-0.04	0.06	0.06	0.03
LogJavadocs	**0.84**	-0.01	0.14	-0.07	-0.07	-0.18	0.18	-0.09
LogCcn	**0.94**	0.01	0.14	-0.04	-0.06	-0.06	0.09	-0.04
LogLCOM	0.09	0.03	**0.89**	0.03	-0.06	0.13	-0.07	0.04
LogEXT	**0.93**	0.12	-0.04	-0.09	-0.06	0.22	-0.04	0.04
LogHIER	-0.10	0.06	0.01	-0.10	0.11	-0.03	0.06	**0.80**
LogLOC	**0.79**	-0.01	0.26	0.32	0.16	0.02	0.10	-0.01
LogINST	0.45	-0.02	**0.80**	0.12	0.03	0.03	0.12	-0.05
LogMOD	0.19	0.04	0.24	-0.10	0.12	**0.79**	0.00	0.06
LogINTR	0.23	0.06	0.35	-0.37	-0.08	-0.14	0.34	-0.40
LogPACK	**0.83**	0.04	-0.14	-0.35	-0.01	-0.20	-0.11	0.03
LogRFC	**0.97**	0.07	0.03	-0.04	-0.08	0.02	0.02	-0.01
LogMPC	**0.94**	0.11	-0.04	-0.10	-0.06	0.20	-0.05	0.03
LogFIN	-0.08	-0.02	0.15	**0.76**	-0.25	-0.13	-0.10	-0.16
LogFOUT	**0.88**	0.12	-0.05	-0.20	-0.01	0.22	-0.14	0.09
LogNSUP	-0.12	-0.06	-0.10	0.01	**0.83**	0.14	0.07	0.18
LogNSUB	-0.10	-0.18	0.01	-0.17	-0.05	0.17	**0.66**	0.40
LogdFOUT	0.50	**0.64**	-0.03	-0.27	-0.01	0.08	-0.01	0.03
LogdLCOM	0.04	**0.86**	0.15	0.02	0.01	0.06	0.05	0.09
LogdNSUP	0.11	0.03	0.06	-0.43	**0.72**	-0.05	-0.06	-0.04
LogdLOC	**0.70**	0.30	0.13	0.15	0.27	-0.09	0.17	-0.11
LogdMOD	-0.04	**0.94**	-0.03	0.00	-0.01	0.04	0.05	-0.04
LogdNSUB	0.15	0.25	0.01	0.02	0.07	-0.02	**0.77**	-0.13
LogFaultCorrections	0.45	-0.11	0.15	-0.15	0.04	**-0.50**	-0.21	0.30
LogChangeCount	**0.71**	0.08	0.09	-0.40	0.07	-0.19	-0.04	-0.15
LogPrevVersionCount	0.08	**-0.94**	0.06	-0.02	0.02	0.05	-0.04	0.05
Logn1FaultCorrections	0.45	-0.15	0.03	0.05	0.03	-0.40	-0.10	0.44
Logn1ChangeCount	**0.61**	**-0.51**	0.03	0.03	0.08	-0.24	0.05	0.17
Variance Explained	11.88	3.63	2.06	1.55	1.55	1.53	1.50	1.46
Cumulative	0.39	0.50	0.57	0.63	0.68	0.72	0.75	0.79