UNIVERSITY OF OSLO
Department of informatics

Implementing and Evaluating a Fault-proneness Prediction Model to Focus Testing in a Telecom Java Legacy System

Master thesis
60 credits

Magnus J. Fuglerud

1. May 2007

Table of Contents

Abstract

Assuring high quality software is perceived as a key factor to succeed in the software industry. However, deliveries of products of poor quality are still common due to time constraints and limited resources. One way to address this problem is to try to make the software testing process more efficient, and hence find more faults in less time or with fewer resources. For example, fault proneness prediction models can be built on the basis of historic data about changes and faults combined with measures of the structural properties of the software. Assuming that a sufficiently accurate model can be built on the basis of the available data, the model can be applied on a forthcoming release of the software system – giving predictions that identify those software modules that are likely to contain faults. Having identified the fault-prone modules, the testing activities can focus on those modules to improve testing efficiency.

This thesis investigates how one can build, evaluate and use class level fault-proneness prediction models for the purpose of focusing testing in the context of a large, evolving Java legacy system. First, different data mining techniques for building the prediction model were evaluated both in terms of *classification accuracy* and *cost-effectiveness*. The results of this evaluation suggested that the best model was very accurate and could, for example, identify as many as 60 percent of the faults in only 10 percent of the system. Second, the *practical* usefulness of a prediction model based unit testing activity was evaluated: The model was used to rank classes in an upcoming release according to their fault proneness. The most fault prone classes then received additional unit testing, by augmenting the test suite to achieve higher branch and loop coverage on the fault-prone classes, assuming this would reveal additional faults. A relatively large number of faults were identified and corrected within a small subset of the most fault-prone classes, confirming the more theoretical cost-effectiveness estimates: in practice, the model had selected classes with at least ten times higher fault density than what would be expected based on fault data collected from previous releases. The time spent on the focused unit testing activity was recorded. Then interviews were conducted with the developers to elicit expert estimates on the potential cost *savings* of having performed the focused testing activity and thus preventing some faults from slipping through to later phases. The evaluation indicated that cost saved was 91.7 hours and time consumed during unit testing was 49.5 hours, indicating more than 50 percent cost reduction due to the focused testing activity.

Acknowledgements

This thesis was conducted as part of the EviSoft research project financed by the research council of Norway.
I would like to thank to my advisors Erik Arisholm and Lionel C. Briand for their helpfulness and constant encouragements during my work. I also want to thank everybody at Simula Research Laboratory and Telenor Mobil who contributed to this work by sharing their experience and ideas with me.
Finally, I would like to thank my girlfriend, Marianne Bråten, for her great support and bearing out with me during the work on this thesis.

Oslo, May 1st, 2007

Magnus Fuglerud

1 Introduction

Assuring high quality software at a realistic cost is a crucial issue to all organizations delivering software. Harder competition among software developing organizations in delivering the right product, at the right time, at the cheapest price is currently experienced by the software-industry. To meet these demands organizations have to improve their processes and products continually. The effectiveness of the software testing process is a key issue in meeting the increasing demand of quality without augmenting the overall costs of software development. Research has shown that at least 50 percent of the total software cost is comprised of testing activities. Planning and allocating resources for testing and analysis is difficult and it is usually done in an unsystematic manner, often leading to unsatisfactory results. Delay in testing and delivery of products of poor quality are common experiences in software production.

1.1 Study design

The motivation for our study was a practical problem occurring in a large Telecom company who is maintaining a Java legacy middleware system used for their mobile division. This system has evolved over several years and has been subject to frequent and substantial changes. The maintenance of old legacy systems like the one studied here is challenging. New requirements needs to be implemented due to business environment changes, and at the same time faults that are introduced as the software evolves has to be repaired. A constant shortage of resources for verification and validation yielded the need for a better way to focus the resources in a more cost-effective manner.

Wanting to reduce costs when developing object-oriented systems, fault-proneness prediction models could be beneficial to adapt to allocate resources to risky parts of software where e.g. faults are more likely to occur, thus making e.g. verification and validation more cost-effective.

In this study, fault-proneness of a software module is defined as the probability that the module will result in a fault. Fault-proneness cannot be directly measured in the software. However, fault-proneness can be estimated based on directly measurable attributes. More specifically, the process of building fault-proneness prediction models involves investigating whether a relationship (e.g., correlation) exists between the future event that a module (or in our case, a Java class) will result in a fault, and existing product- and process metrics in that software. Examples of such metrics include the structural characteristics of classes, the amount of changes a class may have been through (both requirements and fault corrections), the coding quality of classes, as in coding style and practices, and presence of redundant code and more, the fault history of classes in previous releases, and the skills and experience of the individual performing modifications to the product.

The statistical techniques used to build and evaluate fault-proneness prediction models are many. Several studies have conducted research on fault-proneness models [Ostrand et al., #1, 2004], [Gill & Kemerer, 1991], but to this author's knowledge, no existing studies have been performed to evaluate the practical usefulness of actually *applying* such models to focus verification and validation activities such as testing. Furthermore, the use of extensive historic change and fault data to build such models is uncommon. In legacy systems past change and fault data are typically available and such data could be useful to help predict fault-proneness.

The study reported in this thesis consists of two parts. First, we present results from evaluating different data mining techniques for building fault-proneness prediction models in Java telecom software. Second, the results from a practical application of using such a technique to focus unit testing are presented.

1.1.1 Model building and assessment

This thesis documents the process of trying to build a fault-proneness prediction model in the context of an evolving legacy telecom system. The most common way to evaluate the prediction models is by assessing their classification accuracy by means of the so-called confusion matrix criteria (e.g., precision, recall). However, as the results presented in this thesis will show, such criteria do not clearly and directly relate to the cost effectiveness of using class fault-proneness prediction models when applied to focus verification and validation activities. To compare the potential cost-effectiveness of alternative prediction models, we need to consider (surrogate) measures of verification cost for the subset of classes selected for verification. For many verification activities, e.g., structural coverage testing or even simple code inspections, the cost of verification is likely to be roughly proportional to the size of the class. What we want are models that capture other fault factors in addition to size, so that the model would select a subset of classes where we are likely to find faults but not just because they are large classes.

To build such models there are a large number of modelling techniques to choose from, including standard statistical techniques such as logistic regression and data mining techniques such as decision trees [Witten & Frank, 2005]. The data mining techniques are especially useful since we have little theory to work with and we want to explore many potential factors (and their interactions) and compare many alternative models so as to optimize cost-effectiveness.

In this thesis different data mining techniques and machine learning techniques is evaluated both in terms of classification accuracy and cost-effectiveness. Having good classification accuracy and cost effectiveness over different percentages of the code that will be subject to additional, focused verification is important in the context of practical adaptability. For example, if the model built is very accurate on just one specific percentage of the code, it would only be useful if having resources to achieve this exact percentage, but not if resources available upfront is unknown.

The C4.5 decision tree algorithm was the best overall technique for different percentages of code (and across different test sets), suggesting that more complex modelling techniques may not be required. On the other hand, in our study, the Neural Network model building technique proved to be very useful if only very small percentages (say, 1 percent) of the code were subject to additional verification, but it performed much poorer than the C4.5 decision tree for larger percentages.

Note that no “general” models have been developed yet that can be applied to a broader set of application domains. The research conducted so far (including what is presented in this thesis) only applies to specific application domains and is tailored to fit in just that context.

1.1.2 Practical evaluation

This thesis also reports the practical evaluation of using fault-proneness prediction models in industry to focus verification and validation. At present, there exists much research on the theory behind building fault-proneness models, but no empirical evaluations of *applying* such models in industry to focus testing on the fault-prone classes. If fault-proneness prediction models are to be adapted in industry in general, evaluations of practical use are important.

We adopted a strategy of *selective unit retest* of the most fault-prone classes. Selective retest techniques differ from a *retest-all* approach, which runs e.g. all unit tests in a software system. This strategy supports, to some extent, the cost-effective perspective we wanted to illustrate. When release 22 was finished (*code-freeze*), we collected the data required as input variables in the model and applied the model. Our initial strategy was to use the fault-proneness predictions so as to ensure that the unit test of fault-prone classes was reasonably complete, to minimize the number of faults that slipped through to system test. A sorted set of classes with the highest fault probability were thus provided to the participants of the additional unit testing to focus on the fault-prone classes. These classes were published on an editable webpage easing the unit testing. The participants could then update the webpage and keep track on which class was next and justifications made when skipping classes. Initially they were asked to perform a test coverage analysis to identify parts of these classes not fully covered in terms of loop and branch coverage. A coverage analysis tool was used to measure this.

Further they were told to augment the initial test suites to increase test coverage. This typically involved modifying test cases within the test suite to exercise more of the code with tests and validating this by running a test coverage analysis. This process was repeated until the highest practically possible loop and branch coverage was achieved. We recorded effort used per class and how many faults found. We assumed that by exercising fault-prone classes with full loop and branch coverage, faults would be revealed. More specific we suspect faults to be associated with loops in practice, and by exercising loops zero times, one time, a representative number of times and a maximum number of times we assumed that faults could be detected.

Seven faults were identified and corrected within a small subset consisting of the 26 most fault-prone classes, confirming the previous evaluation based on historic data: in practice, the model had selected classes with much higher fault density than average: In a typical release, there would be somewhere between 7 – 83 faulty classes among the 2600 core Java classes in the studied system, thus the historic data indicated a fault density of between 7/2600 – 83/2600, or between 0.0027 – 0.032 faults per class. Among the 26 most fault-prone classes, as selected by the prediction model, the additional unit-testing activity revealed 7 faults, implying a fault density of 7/26, or 0.27 faults per class. Thus, the practical evaluation clearly confirmed that the prediction model was sufficiently accurate to be a practically useful way to identifying fault-prone classes.

However, we also attempted to estimate potential *cost-savings* of the additional unit testing activity, which had incurred a certain cost that we had recorded. The question then was whether this cost would be less than the potential cost savings of detecting and correcting the faults early rather than in later testing phases or even after system delivery. When evaluating the costs of the additional unit testing relative to the potential cost of detecting and correcting the faults in other testing phases, potential cost savings could be estimated. To obtain estimates of the cost savings, we conducted interviews to collect expert estimates on probabilities and costs of detecting and correcting faults in the subsequent testing phases and in the delivered system. These data were parameters in a cost-effectiveness model developed to estimate potential cost savings by performing the unit testing. Having three participants involved, we performed a pilot interview with one of them to ensure we asked the right questions in getting the data we needed for our cost-effective model. The three interviews were compared to validate consistency among participant answers (also known as interrater reliability).

The results from the practical evaluation showed that the potential cost-effectiveness of using such a technique appears to be beneficial. The additional unit testing cost was 49.5 hours. The evaluation indicated that cost saved from finding the faults at that point in time rather than later was 91.7 hours, indicating more than 50 percent cost reduction due to the focused testing activity.

1.2 Contribution

This thesis is part of a longitudinal research project that aims to build and evaluate fault-proneness prediction models and evaluate the costs and benefits of using such models to improve the quality of object-oriented software, e.g., by focusing testing on fault-prone classes. By means of an empirical study conducted in the Telecom application domain, this thesis demonstrates that the models can be built and cost-effectively applied to focus testing on fault-prone classes, resulting in a more efficient testing process.

More specifically, my contribution to the project has been two-fold: First, I collected the data necessary to build the fault-proneness prediction models for the system under study and assisted in building and evaluating the accuracy and cost-effectiveness of the resulting models. The results of this work have been submitted to the International Symposium on Software Reliability Engineering (ISSRE 2007) [Arisholm et al., 2007]. Second, I helped defining the testing process for the practical evaluation, ensured that the developers followed it, and collected the data needed to evaluate costs and benefits. These data consists of both process data and expert estimates to better be able to conclude about practical usage.

1.3 Thesis structure

The reminder of this thesis is organized as follows. Section 2 considers quality models and more specific fault-proneness prediction models and its applications in terms of related work. Section 3 describes the empirical study in terms of the model building and assessment. Section 4 describes the practical evaluation. Section 5 concludes the research and presents further work.

2 Fault-proneness Prediction Models and Applications

Improvement of software quality is important to all organizations that rely on good software products to succeed – quality products depend on quality software. Although organizations agree upon the importance of software quality, they do not deterministically agree upon how to define and how to measure it. Consequently, different quality models have been developed using different approaches and different measurements. In addition differences in application domains yields for the necessity of these differences.

A Software Quality Model was defined as a *statistical relationship between a quality-factor – a dependent variable – and product-and process metrics – independent variables* [Khoshgoftaar et al., 1996].

A software quality factor is to be understood according to the definition of software quality. This includes i.e. *maintainability*, as the ability to facilitate updating to satisfy new requirements, *reliability,* as the ability to perform its intended functionality in a satisfactorily way, *portability,* as the ability to operate easily on computer configurations other than its current one, *testability, as* facilitating the establishment of acceptance criteria and supports evaluation of its performance and *usability*, as being convenient and practicable to use [Arisholm et al., 2005].

Software metrics are often divided into two categories: *software product metrics* and *software process metrics* [B.Henderson-Sellers, 1995]. Software product metrics are used to measure aspects of software products, e.g. source code metrics and design documents metrics. Software process metrics are used to measure e.g. software development processes, including development effort, staffing levels and developer experience. Software product metrics can be further divided into two categories: those measuring dynamic attributes and those measuring static attributes. Dynamic metrics can only be evaluated at run-time and are thus difficult to measure. These metrics are not widely used, except those measuring code coverage in software testing. Static metrics, in turn, are used to measure static attributes of software and they are widely used in software engineering.

Another categorization of software metrics involves *internal* and *external metrics*. An internal metric only applies to the software product or process itself. These metrics are often easy to define and objectively measurable, i.e. software size and elapsed time are internal metrics of a product and process, respectively. An external metric of software can involve factors such as people or environments and describes how a software product or process relates to its surroundings. This metric is dependent on human and environmental factors making it more difficult to define than internal attributes. I.e. maintainability and costs are external metrics of product and process, respectively. Even tough external metrics are difficult to measure, decision-making people is very interested in these measures and the relationship between them. Hence, identifying connections between internal and external measurements is one of the important research areas of software metrics [B.Henderson-Sellers, 1995].

Using structural design properties, such as coupling, cohesion or complexity is considered to be a promising approach towards early quality assessment. To be able to use such measurements effectively, quality models are needed to quantitatively describe how these structural properties relate to external system qualities, such as reliability or maintainability [Briand & Wüst, 2002]. The rationale is that a quality-factor, as in external system qualities of software, is related to certain characteristics, as in metrics of that software, being e.g. product- and process metrics, making it possible to objectively predict various external quality aspects. Such prediction models can be used to help decisions-making during development and to reduce costs.

In general and as outlined in [Slaugther et al., 1998], the costs of software quality is divided into *conformance* and *non-conformance* cost, where the first represents the costs associated with the amount spent to achieve quality products i.e. testing and inspection, and the second is associated with the costs incurring when things go wrong i.e. errors occurring after the system has been put into use. To illustrate, in an informal way, the twofold ness in cost of software quality, let the conformance costs be costs that are intended and non-conformance costs be costs that is not intended. In general, conformance costs are budgeted, while non-conformance costs are not. Looking at it this way, one may observe the fact that by eliminating non-conformance costs, one could also eliminate exceeding the budget. Wanting to improve e.g. reliability of software to eliminate non-conformance costs, quality assurance teams can use different techniques such as additional testing, inspection and reviews of code and design documents, as well as strategically assigning personnel to different programming tasks (conformance cost). Due to time constraints it is not always practically possible to assure such activities to all the parts of the software. In a large-scale system, e.g. inspecting all the code is not feasible.

The *Pareto* principle suggests that 80 percent of all faults in a system stems from 20 percent of the code. Fault-proneness prediction models is one type of quality model that is motivated by the Pareto principle, and the assumption that certain characteristics of the software product and process used to develop it, is correlated to faultiness in the software. Predicting where e.g. faults *are* makes it possible to tune resource allocation e.g. testing to focus on those parts and detecting and fixing more faults with less time consumed, leaving additional time to test the reminder of the system. Potential consequences would be more faults found with less effort, making more reliable software at a decreased cost. Fault-proneness, or the number of defects detected in a software module (e.g., class), is the most frequently investigated dependent variable in research on quality models [Briand & Wüst, 2002].

In [Mockus & Weiss, 2000] fault-proneness for IMR's (Initial Maintenance Requests) was predicted using Logistic Regression. Only metrics regarding the changes themselves was used as indicators of risk and the presence or absence of faults in the past, as in historical data was used as the dependent variable. The most significant predictors of risky faultiness were *size*, as in deltas or how much code was added due to the change, *diffusion*, as in the number of distinct parts of the software, such as files, that need to be touched, or altered, to make the change, the *type* of change, as in large or small change or new functionality, and *developer experience*.
The applications of use included a web-based tool providing the probability of fault for an IMR before the software update is set to production, supplied by a risk flag explaining what might be the cause of the potential failure, suggesting actions to be taken trying to eliminate the fault. Such actions could be i.e. delaying the IMR for a later release or performing an additional code inspection. Fault-proneness was indicated as a probability on a continuous scale.

In [Khoshgoftaar et al., 1998] the configuration management system was analyzed and revealed that 99 percent of the unchanged modules had no faults, suggesting modules not changed in release n would not result in faults in release n+1. Knowing this only metrics from changed or new modules where used in the quality model as was in [Mockus & Weiss, 2000], but the modules focused on software modules not software changes. Classification trees where used to model quality software in terms of reliability. The dependent variable was based on historical data in the sense that a module classified to be fault-prone if faults were discovered by costumers, and not fault-prone if faults were not discovered by costumers. Different product-and process metrics were used as classifiers of the dependent variable, and classification miss-accuracy was measured using Type I and Type II errors. To build the model CART (Classification and Regression Trees) algorithm was used. When evaluating the classification accuracy it appeared that a model with only 2 product metrics had similar data-splitting accuracy, than did a model with 40 candidate product- and process metrics. It was pointed out that this was due to confounding, as in correlation among variables. The number of distinct include files was the most significant individual explanatory variable. Distinct include files is a file-level statement metric and describes what is called coupling in object-oriented software. If this fault-proneness model was to be used to guide extra reviews and testing, it was suggested that modules classified as Type I errors and those modules actually predicted as fault-prone would receive extra treatment. Further, in stating potential cost-effectiveness, it was pointed out that 30 percent of the total number of modules would be given extra treatment to discover faults early, as in pre-release, and effective reviews potentially would reveal 73 percent of the total number of faults in a release as opposed to randomly selecting 30 percent of the modules and potentially revealing 30 percent of the faults. Other models were not built to support usage with fewer percents of the code treated more thoroughly. Given the assumption of limited time available to perform such additional testing, this model has some limitations. Being able to build different models, also using different statistical techniques, to support cost-effectiveness on smaller parts of the total number of modules is considered more adaptive to practical usage. Figure 4 in this thesis illustrates the usefulness of building different models and assessing their potential cost-effectiveness on 1,5,10 and so forth up to 100 percent of the code. Knowing this you could actually chose the model best suited for the percentage of the code you can afford to deploy more thoroughly testing on.

In [Ostrand et al., 2005] fault-proneness was indicated in terms of number of faults a file probably would contain, as on a discrete scale. To calculate this, a negative binomial regression model using information from previous releases was developed. Different measurements were used as predictors of fault-proneness on file level, such as the file’s size, the files age, whether or not the file is new to the current release, and if it is not new, whether it was changed during the prior releases. Other measures considered were the number and magnitude of changes made to the file, the number of detected faults during early releases and the number of faults detected during early development stages.

Instead of focusing on fault-proneness predictions in terms of a probability on module level, the number of faults per file was presented in a tool based on fault-proneness modelling in [Ostrand & Weyuker #1, 2002]. The scalable aspect of this approach was described in terms of the tester being able to ask for the top 20 percent of the files with the *most* faults in the next release, thus being able to focus on these for testing purposes. This is an important disruption among studies on fault-proneness prediction models. Some models predict fault-proneness as a probability and others predicts number of faults per module. In other words, some use a discrete indicator while others use continuous indicator of fault-proneness. In [Denaro et al., 1994] it was argued that discrete classifications often were too coarse to be flexible enough to be applied in industrial environments. On the contrary, continuous indicators would allow for a much finer allocation of testing time and resources thus being preferable, or even required for many industrial applications. The continuous presentation of the fault-proneness can be more useful than discrete presentation as it allows us to plan testing activities since they can be adapted according to the available resources. For example, if we are supposed to increase testing effort only for a given percentage X of modules, continuous presentation allows us to directly consider X% of most fault-prone modules, while the discrete presentation cannot do that.

An empirical comparable study on quality models in object-oriented systems [Briand & Wüst, 2002] found several measures being consistent among studies. Explicitly, when predicting fault-proneness quality models based on structural measures, coupling proved to be a useful predictor. More specifically suggested, great emphasis should be put on method invocation import coupling since it has proved to be a strong, stable predictor of fault-proneness. In addition a separation of the coupling aspect into more specific measures could prove to be useful because of their ability to capture distinct dimensions in the data. A separation of import versus export coupling, coupling to library classes versus application classes and a separation of method invocation versus aggregation coupling is recommended due to validity among studies. Cohesion did not seem to be a useful predictor of fault-proneness. Two facts was pointed out to reflect this: (1) today the understanding of what this attribute is supposed to capture is weak, (2) when measuring the cohesion attribute, difficulties occur due to the fact that static analysis do not capture such an attribute e.g. the code must be analyzed at runtime to reveal this attribute. Inheritance measures appear not to be consistent indicators of fault-proneness between studies. In specific, the significance level of this attribute seems strongly to be related to developer experience and the use of inheritance as a strategy when developing, e.g. one could chose not to keep inheritance at a minimal level for project strategy purposes. Finally, measures of size proved consistently being a good predictor of fault-proneness. But the combination of the abovementioned coupling and inheritance with size outperformed a model with size only as predictor of fault-proneness. In the same work it was discussed the concept of confounding. It was stated that the number of measures safely could be reduced without corrupting the potential overall quality-related effect.

Considering the research briefly presented above and others [Gill & Kemerer, 1991], [Basili & Hutchens, 1983], [Frankl & Lakounenko, 1998], the process of building and using fault-proneness models consists of some general aspects being consistent over a lot of research conducted in the area [Denaro et al., 1994]. Generally, a *construction* and a *usage* phase are comprised in the use of fault-proneness models. Because no generally valid models exist a construction phase is required to account for different application domains.

Some key steps in the construction phase are:

- Identifying the target domain: models are valid only within specific classes of applications. Using fault-proneness models for a set of programs that do not meet these requirements requires special care. This is especially important for the model validation and model tuning phase.
- Analysis of historical data: models are built based upon fault-data from past applications. The completeness of the available data must be checked, and obviously the data must exist. This suggests that e.g. a configuration management system containing these data must be present to actually being able to build fault-proneness models.
- Construction of fault-proneness models: this step involves using statistical techniques and tools to construct models based upon the data collected.
- Selection of a significant model: the best model is selected by using methods for validating the quality of the constructed models.

In the usage phase these steps normally applies:

- The model selected above is used to predict fault-proneness of new releases or new products within the same application domain, before testing of these.
- Validate the results during the testing phase or based upon previous the fault-data: Possible disputes can be revealed at this stage.
- Tune the model at the end of the testing phase: the introduction of new techniques may change the validity of the model. Fault-proneness models must be modified periodically to increase their precision by adding new data.

In a cost-effective perspective, the applications of fault-proneness prediction quality models have proved to be useful in both theory and to some extent in practice [Mockus & Weiss, 2000]. There exists more research on the theory behind building fault-proneness models than do empirical studies on the usage of these models in industry. The majority of these studies conclude by suggesting the usage of such techniques in testing, but few or none have evaluated such an initiative, at least not in the context of an evolving telecom legacy system under maintenance.
In [Denaro et al., 1994] it was argued that the reason fault-proneness models did not get adapted more in industry was due to bias occurring due to lack of empirical evaluations and tuning of models in industry. This yields the need for more empirical evaluation being conducted.

2.1 Using Fault-proneness Prediction Models to Focus Verification and Validation

Software development activities like verification and validation plays a critical role in the production of high quality dependable systems, and account for a significant amount of resources including time, money, and personnel. Research has shown that at least 50 percent of the total software cost is comprised of testing activities [Tahat et al., 2001]. Additionally, the fact that testing is the last phase in the development life cycle often results in that whenever schedule slips throughout the project and the release date is fixed, the testing is squeezed down to the bare minimum.

To monitor and control the quality of software and the effectiveness and costs of analysis and testing, the ability to both measure the amount of faults found with testing, as in *software faultiness* and predicting the distribution of faults before testing, as in *software fault-proneness*, is a key factor.

It is relatively easy to measure software faultiness and this could help tune the software development process. If software faultiness data is recorded accurately and kept up to date, the software team is always aware of what problems are currently outstanding, and decisions about whether the product is ready for release can be made. However software faultiness is of little help in resource allocation and in anticipating costs and problems of analysis and testing, as in early project phases.
Software fault-proneness cannot be directly assessed before testing and cannot be easily estimated, but it is extremely helpful in anticipating analysis and testing problems, and in planning effort [Denaro & Pezzê, 2002].

Knowing in which parts of the system faults are most likely to occur one could test those parts earlier and more intensively than other parts, possibly revealing more faults with less effort. Since testing always is limited, more of the precious testing time could be allocated to parts of the system otherwise not covered as thoroughly, possibly leading to software with higher reliability, still using the same amount of resources. Potentially the time set aside to testing can be reduced, as the tester might uncover the same faults more quickly, leading to the same degree of reliability sooner and more cheaply than would otherwise be possible.

In [Slaugther et al., 1998] the question is asked; whether and how much is to be invested in specific quality improvement initiatives. This question is approached by looking at the financial return on investments (ROI) and is called the return on software quality (ROSQ). The rationale behind ROSQ is to provide a justification of software quality expenses. Loosely speaking, investments should be made if they are smaller than expenses occurring if the investment was not made. A ROSQ being lower than 1 the investment is not financially justified. Defining fault-proneness prediction models as a model predicting modules with a high degree of faultiness, and focusing verification and validation on these parts as a quality improvement initiative, a justification can be made that the investment, i.e. hours used to perform such an activity is lower than hours used to correct defects occurring because this activity was not accomplished. More specific the conformance costs should be smaller than the non-conformance costs.

3 Data Mining Techniques for Building Fault-proneness Models in Telecom Java Software

A case study was performed to build and evaluate alternative fault-proneness prediction models. These models are described in Section 3.1. In addition this section describes the development project, study variables, data collection, and model building and evaluation procedures. A related study is compared to ours.

3.1 Fault-proneness modelling

There exists a large number of modelling techniques to build a fault-proneness model, such as a classification model determining whether classes or files are faulty. A classical statistical technique used in many existing papers is Logistic Regression [Freund & Wilson, 1998]. But many techniques are also available from the fields of data mining, machine learning, and neural networks [Witten & Frank, 2005]. One important category of machine learning techniques focuses on building decision trees, which recursively partition a data set, and the most well-known algorithm is probably C4.5 [Quinlan, 1993]. In our context, each leaf of a decision tree would then correspond to a subset of the data set available (characterized by class source code characteristics and their fault/change history, as described in Section 3.4) and its probability distribution can be used for prediction when all the conditions leading to that leaf are met. Another similar category involves coverage algorithms that generate independent rules where a number of conditions are associated with a probability for a class to contain a fault based on the instances each rule covers. As opposed to the divide-and conquer strategy of decision trees, these algorithms iteratively identify attribute-value pairs that maximize the probably of the desired classification and, after each rule is generated, remove the instances that it covers before identifying the next optimal rule.

Both decision tree or coverage rule algorithms generate models that are easy to interpret (logical rules associated with probabilities) and that therefore tend to be easier to adopt in practice as practitioners can then understand why they get a specific prediction. Furthermore they are easy to build (many freely available tools exist) and apply as they only involve checking the truth of certain conditions. Another advantage is that, instead of providing model-level accuracy (e.g., like for Logistic Regression), each rule or leaf has a specific expected accuracy. The level of expected accuracy associated with a prediction therefore varies across predictions depending on which rule or leaf is applied.

Other common techniques include Neural networks, for example the classical back-propagation algorithm [Werbos, 1994], which can also be used for classification purposes. A more recent technique that has received increased attention in recent years across various scientific fields [Vapnik, 1995], [Joachims, 2002], [Shipp et al., 2002] is the Support Vector Machine classifier (SVM), which attempts to identify optimal hyperplanes with nonlinear boundaries in the variable space in order to minimize misclassification.

Based on the above discussion, we will compare here one classification tree algorithm, namely C4.5 as it is the most studied in its category, the most recent coverage rule algorithm (PART) which has shown to outperform older algorithms such as Ripper [Witten & Frank, 2005], Logistic Regression as a standard statistical technique for classification, Back-propagation neural networks as it is a widely used technique in many fields, and SVM. Furthermore, as the outputs of leaves and rules are directly comparable, we will combine C4.5 and PART predictions by selecting, for each class instance to predict, the rule or leaf that yields a fault probability distribution with the lowest entropy (i.e., the fault probability the furthest from 0.5, in either direction). This allows us to use whatever technique works best for each prediction instance.

Machine learning techniques, such as classification trees, can be improved in terms of accuracy by using metalearners. For example, decision trees are inherently unstable due to the way their learning algorithms work: a few instances can dramatically change variable selection and the structure of the tree. The Boosting [Witten & Frank, 2005] method combines multiple trees implicitly seeking trees that complement one another in terms of the data domain where they work best. Then it uses voting based on the classifications yielded by all trees to decide about the final classification of an instance. How the trees are generated differ depending on the algorithm and one of the well-know algorithm we use here is AdaBoost [Freund & Schapire, 1995] that is designed specifically for classification algorithms. It iteratively builds models by encouraging successive models to handle instances that were incorrectly handled in previous models. It does so by re-weighting instances after building each new model and builds the next model on the new set of weighted instances.

Another metalearner worth mentioning is named Decorate. This recent technique is claimed [Melville & Mooney, 2005] to consistently improve not only the base model but also outperform other techniques such as Bagging and Random forest [Witten & Frank, 2005], which we will not include here for that reason. Since it is also supposed to outperform boosting on small training sets and rivals it on larger ones, it is also considered in our study.

Another way to improve classifier models is to use techniques to pre-select variables or features, to eliminate most of the irrelevant variables before the learning process starts. When building models to predict fault components or files, we often do not have a strong theory to rely on and the process is rather exploratory. As a result, we often consider a large number of possible predictors, which often turn out not to be useful or are strongly correlated. Though in theory the more information one uses to build a model, the better the chances to build an accurate model, studies have shown that adding random information tends to deteriorate the performance of C4.5 classifiers [Witten & Frank, 2005]. This happens because as the tree gets built, the algorithm works with a decreasing amount of data, which may lead to chance selection of irrelevant variables. The number of training instances needed for instance-based learning increases exponentially with the number of irrelevant variables present in the data set. Strong inter-correlations among variables also affect variable selection heuristics in regression analysis [Freund & Wilson, 1998]. A recent paper [Hall & Holmes, 2003] has compared various variable selection schemes. The authors concluded by recommending a number of techniques which vary in terms of their computational complexity. Among them, two efficient techniques were reported to do well: CFS (Correlation-based Feature Selection [Hall, 2000], ReliefF [Kononenko, 1995]. In our case these two techniques yielded the same selection of variables. Though we used CFS for all modelling techniques, we will only report it for C4.5 as the results were not significantly different for others (i.e., the impact of CFS was at 3 best small) and as, a discussed below, C4.5 will be the technique we ultimately retain to focus testing.

3.2 Telenor COS development Environment

A large Java legacy system (COS) is being maintained at Telenor, Oslo, Norway, and there is a constant shortage of resources and time for testing and inspections. The quality assurance engineers wanted to investigate means to focus verification on parts of the system where faults were more likely to be detected. As a first step, the focus was on unit testing in order to eliminate as many faults as possible early on in the verification process by applying more stringent test strategies to code predicted as fault-prone. Though many studies on predicting fault-prone classes on the basis of the structural properties of object-oriented systems have been reported [Briand & Wüst, 2002], one specificity of the study presented here is the fact that we need to predict fault-proneness for 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 22 major releases during the past eight years. We used 12 recent releases of this system for model building and evaluation. At any time, somewhere between 30 to 60 software engineers have been involved in the project. The core system currently consists of more than 2600 Java classes amounting to about 148K SLOC2. As the system expanded in size and complexity, the developers felt they needed more sophisticated techniques to focus verification activities on fault-prone parts of the system.

3.3 Dependent Variable

The dependent variable in our analysis is the occurrences of corrections in classes of a specific release which are due to field error reports. Since our main current objective is to facilitate unit testing and inspections, the class was a logical unit of analysis. However, it is typical for a fault correction to involve several classes and we therefore count the number of distinct fault corrections that was required in that class for developing the next release n+1. This aims at capturing the fault-proneness of a class in the current release n. Furthermore, in this project, only a very small portion of classes contained more than one fault for a given release, so class fault-proneness in release n is therefore treated as a classification problem and is estimated as the probability that a given class will undergo one or more fault corrections in release n+1.

3.4 Explanatory Variables

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 (e.g., their coupling)
- the amount of change (requirements or fault corrections) undertaken by the class to obtain the current release
- the experience of the individual performing the changes
- other, unknown factors that are captured by the change history (requirements or fault corrections) of classes in previous releases

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. The data mining techniques used to build the models will account for such interactions. Explanatory variables are defined in Table 1.

3.5 Collection Procedures

Perl scripts were developed to collect file-level change data for the studied COS releases through the configuration management system (MKS [MKS]). In our context, files correspond to Java public classes. The data model is shown in Figure 1. Each change is represented as a change request (CR). The CR is related to a given releaseId and has a given changeType, defining whether the change is a critical or non-crititical fault correction, small, intermediate or large requirement change, or a refactoring change. An individual developer can work on a given CR through a logical work unit called a change package (CP), for which the developer can check in and out files in relation to the CR. For a CP, we record the number of CRs that the responsible developer has worked on prior to opening the given CP, and use this information as a surrogate measure of that person’s coding experience on the COS system. For each Class modified in a CP, we record the number of lines added and deleted, as modelled by the association class CP_Class. Data about each file in the COS system is collected for each release, and is identified using a unique MKSId, which ensures that the change history of a class can be traced even in cases where it changes location (package) from one release to the next. Finally, for each release, a code parser (JHawk [JHawk]) is executed to collect structural measures for the class, which are combined with the MKS change information. Independent and dependent variables (Faults in release n+1) were computed on the basis of the data model presented in Figure 1.

Table 1 Summary of the explanatory variables in the study

Variable	Description	Source
No_Methods | NOQ | NOC	Number of [implemented | query | command] methods in the class	JHawk
LCOM	Lack of cohesion of methods	JHawk
TCC | MAXCC | AVCC	[Total|Max|Avg] cyclomatic complexity in the class	JHawk
NOS | UWCS	Class size in [number of Java statements | number of attributes + number of methods]	JHawk
HEFF	Halstead effort for this class	JHawk
EXT/LOC	Number of [external | local] methods called by this class	JHawk
HIER	Number of methods called that are in the class hierarchy for this class	JHawk
INST	Number of instance variables	JHawk
MOD	Number of modifiers for this class declaration	JHawk
INTR	Number of interfaces implemented	JHawk
PACK	Number of packages imported	JHawk
RFC	Total response for the 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 the class depends	JHawk
R-R | S-R	[Reuse | Specialization] Ratio for this class	JHawk
NSUP | NSUB	Number of [super | sub] classes	JHawk
MI | MINC	Maintainability Index for this class [including | not including] comments	JHawk
[nm1|nm2|nm3]_CLL_CR	The number of large requirement changes for this class in release [n-1 | n-2 | n-3]	MKS
[nm1|nm2|nm3]_CFL_CR	The number of medium requirement changes for this class in release [n-1 | n-2 | n-3]	MKS
[nm1|nm2|nm3]_CKL_CR	The number of small requirement changes for this class in release [n-1 | n-2 | n-3]	MKS
[nm1|nm2|nm3]_M_CR	The number of refactoring changes for this class in release [n-1 | n-2 | n-3]	MKS
[nm1|nm2|nm3]_CE_CR	The number of critical fault corrections for this class in release [n-1 | n-2 | n-3]	MKS
[nm1|nm2|nm3]_E_CR	The number of noncritical fault corrections for this class in release [n-1 | n-2 | n-3]	MKS
numberCRs	Number of CRs in which this class was changed	MKS
numberCps	Total number of CPs in all CRs in which this class was changed	MKS
numberCpsForClass	Number of CPs that changed the class	MKS
numberFilesChanged	Number of classes changed across all CRs in which this class was changed	MKS
numberDevInvolved	Number of developers involved across all CRs in which this class was changed	MKS
numberTestFailed	Total number of system test failures across all CRs in which this class was changed	MKS
numberPastCr	Total developer experience given by the accumulated number of prior changes	MKS
nLinesIn	Lines of code added to this class (across all CPs that changed the class)	MKS
nLinesOut	Lines of code deleted from this class (across all CPs that changed the class)	MKS
	FOR CRs of type X={CLL, CFL, CKL, M, CE, E}:	
<X>_CR	Same def as *numberCRs* but only including the subset of CR's of type X.	MKS
<X>_CPs	Same def as *numberCpsForClass* but only including the subset of CR's of type X	MKS
<X>numberCps	Same def as *numberCps* but only including the subset of CR's of type X	MKS
<X>numberFilesChanged	Same def as *numberFilesChanged* but only including the subset of CR's of type X	MKS
<X>numberDevInvolved	Same def as *numberDevInvolved* but only including the subset of CR's of type X	MKS
<X>numberTestFailed	Same def as *numberTestFailed* but only including the subset of CR's of type X	MKS
<X>numberPastCr	Same def as *numberPastCr* but only including the subset of CR's of type X	MKS
<X>nLinesIn	Same def as *nLinesIn* but only including the subset of CR's of type X	MKS
<X>nLinesOut	Same def as *nLinesOut* but only including the subset of CR's of type X	MKS

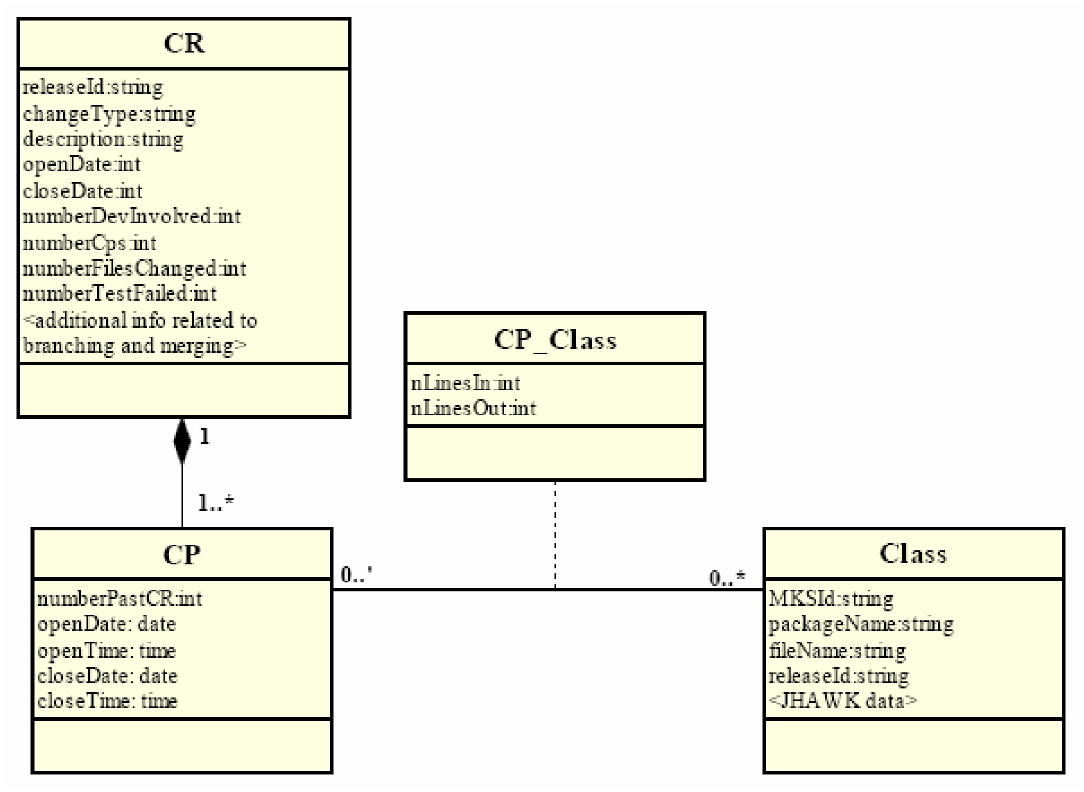


Figure 1 COS Data Model

3.6Building the Prediction Models

To build and evaluate the prediction models, class-level structural and change data from 12 recent releases of COS were used. The data was divided into three separate subsets, as follows. The data from the 11 first releases was used two form two datasets; a training set to build the model and a test set to evaluate the predictions. More specifically, 66.7 percent of the data (16311 instances) were randomly selected as the Training dataset, whereas the remaining 33.3 percent (8143 instances) formed the Excluded test dataset. Our data set was large enough to follow this procedure to build and evaluate the model without resorting to cross-validation, which is much more computationally intensive. Also, the random selection of the training set across 11 releases reduced the chances for the prediction model to be overly influenced by peculiarities of any given release. Note that in the training set, there were only 307 instances representing faulty classes (that is, the class had at least one fault correction in release n+1). This is due to the fact that, in a typical release, a small percentage of classes turn out to be faulty. For reasons further discussed in Section 5.1, to facilitate the construction of unbiased models, we created a balanced subset (614 rows) from the complete training set, consisting of the 307 faulty classes and a random selection of 307 rows representing non-faulty classes. Finally, the most recent of the 12 selected releases formed the third distinct dataset, hereafter referred to as the COS 20 test dataset, which we will also used as a test set. The Excluded test set allows us to estimate the accuracy of the model on the current (release 11) and past releases whereas the COS 20 test set indicates accuracy on a future release. This will give us insight on the level of decrease in accuracy to be expected, if any, when predicting the future.

Having described our model evaluation procedure, we now need to explain what model accuracy criteria we use. First, we consider the standard confusion matrix criteria [Witten & Frank, 2005]: precision and recall. In our context, precision is the percentage of classes classified as faulty that are actually faulty and is a measure of how effective we are at identifying where faults are located. Recall is the percentage of faulty classes that are predicted as faulty and is a measure of how many faulty classes we are likely to miss if we use the prediction model.

Another common measure is the ROC[1] area [Witten & Frank, 2005]. A ROC curve is built by plotting on the vertical axis the number of faults contained in a percentage of classes on the horizontal axis. Classes are ordered by decreasing order of fault probability as estimated by a given prediction model. The larger the area under the ROC curve (the ROC area), the better the model. A perfect ROC curve would have a ROC area of 100%.

Though relevant, the problem with the general confusion matrix criteria is that they are designed to apply to all classification problems and they do not clearly and directly relate to the cost effectiveness of using class fault-proneness prediction models in our context. Assuming a class is predicted as very likely to be faulty, one would take corrective action by investing additional effort to inspect and test the class. Such activities are likely to be roughly proportional to the size of the class. For example, that would be the case for structural coverage testing or even simple code inspections. So, if we are in a situation where the only thing a prediction model does is to model the fact that the number of faults is proportional to the size of the class, we are not likely to gain much from such a model. What we want are models that capture other fault factors in addition to size. Therefore, to assess the cost effectiveness, we compare two curves as exemplified in Figure 2.
Classes are first ordered from high to low fault probability. When a model predicts the same probability for two classes, we order them further according to size so that larger classes are selected last. The solid curve represents the actual percentage of faults given a percentage of lines of code of the classes selected to focus verification according to the abovementioned ranking procedure (referred to as the model cost-effectiveness (CE) curve). The dotted line represents a line of slope 1 where the percentage of faults would be identical to the percentage of lines of code (% NOS) included in classes selected to focus verification. This line is what one would obtain, on average, if randomly ranking classes and is therefore a baseline of comparison (referred to as the baseline). Based on these definitions, our working assumption is that the overall cost-effectiveness of fault predictive models would be proportional to the surface area between the CE curve and the baseline. This is practical as such a surface area is a unique score according to which we can compare models in terms of cost-effectiveness regardless of a specific, possibly unknown, NOS percentage to be verified. If the model yields a percentage of faults roughly identical to the percentage of lines of code, then no gain is to be expected from using such a fault-proneness model when compared to chance alone. The exact surface area to

1 Receiver Operating Characteristic

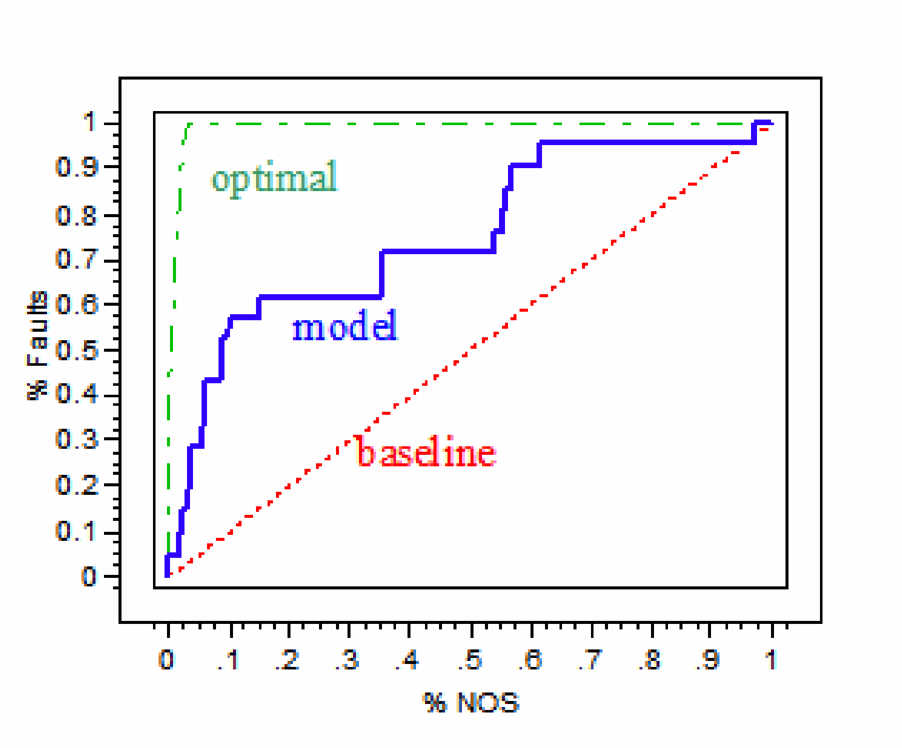


Figure 2 Computing a Surrogate Measure of Cost Effectiveness

consider may depend on a realistic, maximum percentage of lines of code that is expected to be covered by the extra verification activities. For example, if only 5% of the source code is the maximum target considered feasible for extra testing, only the surface area below the 5% threshold should be considered.

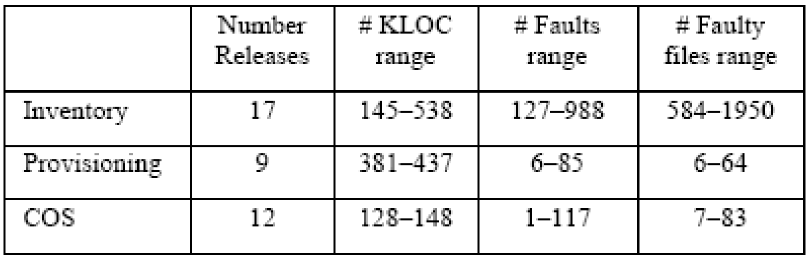
For a given release, it is impossible to determine beforehand what would be the surface area of an optimal model. For each release, we compute it by ordering classes as follows: (1) we place all faulty classes first and then order them so that larger classes are tested last, (2) we place fault-free classes afterwards also in increasing order of size. This procedure is a way to maximize the surface area for a given release and set of faulty classes, assuming the future can be perfectly predicted. Once computed, we can compare, for a specific NOS percentage, the maximum percentage of faults that could be obtained with an optimal model and use this as an upper bound to further assess a model, as shown by the dashed line in Figure 2.

3.7 Related and comparable work

There exists a lot of research on fault-proneness models in Object-Oriented-systems. As well as the related work mentioned in the introduction of this thesis, a survey is provided in [Briand & Wüst, 2002]. Due to the already mentioned differences in application domains, not all related work on fault-proneness prediction models is comparable.

One of the main pieces of related work, that is comparable, has been reported in [Ostrand et al., 2005] and we therefore perform a detailed comparison of their study with ours. Their goal was also to predict fault-proneness in evolving systems mostly composed of Java code. One difference is that they predict fault proneness in files

Table 2 System information for [Ostrand et al., 2005] and this study



	Number Releases	# KLOC range	# Faults range	# Faulty files range
Inventory	17	145–538	127–988	584–1950
Provisioning	9	381–437	6–85	6–64
COS	12	128–148	1–117	7–83

instead of classes, as their systems were not only coded in Java. This should, however, be similar for most files as Java files normally contain one public class and possibly their inner classes. Another important difference was that their main focus was to support system testing whereas in our case the main goal was to support the focus of extra unit testing to prevent as many faults as possible to reach subsequent testing phases and deployment.

They looked at two systems. For the first one (“Inventory”), all life cycle faults were considered whereas for the second one (“Provisioning”) only post unit testing faults were accounted for. They studied 17 Inventory releases and 9 Provisioning releases, but due to the small number of faults reported in the latter, they merged releases into three “pseudo” releases. The number of releases we considered in COS (12) is similar but we only accounted for post-release faults.

Descriptive statistics of the systems reported in [Ostrand et al., 2005] and in this thesis (COS) are provided in Table 2. Inventory peaks at 1950 files and 538 KLOC in its last release whereas Provisioning is slightly above 2000 files and 437 KLOCS. Because it only accounts for post unit testing faults, the latter only has between 6 to 85 faults per release whereas the former has between 127 and 988 faults per release. The COS system is smaller with 148 KLOC of Java code. The number of faults across COS releases is expectedly more comparable to the provisioning system: 1 to 117.

A fault was defined as a change made to a file because of a Modification Request (MR), which seems identical to our definition. (If n MRs changes a file, this is counted as n faults.) One difference with our COS data collection though was that Ostrand et al. had no reliable data regarding whether a change was due to a fault. As a result, they used and validated a heuristic where changes involving less than three files were considered faults.

Ostrand et al. used negative binomial regression [Ostrand et al., 2005], which is a natural technique to use when predicting small counts. Since on average, a faulty file contained 2-3 faults in their systems, their modelling approach is perfectly justified. In our case, most faulty classes contained one fault and we therefore resorted to Logistic Regression to classify a class as faulty or not. In addition to this statistical approach, because this is one important focus of this study, we tried out and compared many of the data mining techniques available to us for the sake of comparing prediction results. Some of those techniques have practical advantages discussed in Section 3.6, the main one being that they produce interpretable models, something we noticed was important for the practitioners using these models. There are, however, very few studies performing comprehensive comparisons of modelling approaches.

Based on their negative binomial regression model, for both systems, Ostrand et al. reported that the 20% most fault prone files contain an average of 83% of the faults across releases. These files represent an average of 59% of the source code statements. They used 20% as this was the “knee of the fault curve” where the number of faults contained in faulty files started to plateau. Comparisons with our results are provided in Section 3.8.4.

The analysis in [Ostrand et al., 2005] used much fewer explanatory variables: number of LOCs per file, whether the file was changed from the previous release, the age of file in terms of number of releases, the number of faults in previous release, and the programming language. It was also reported that other variables were used but turned out not to be significant additional predictors: number of changes to file in previous release, whether a file was changed prior to the previous release, and cyclomatic complexity.

3.8Prediction models

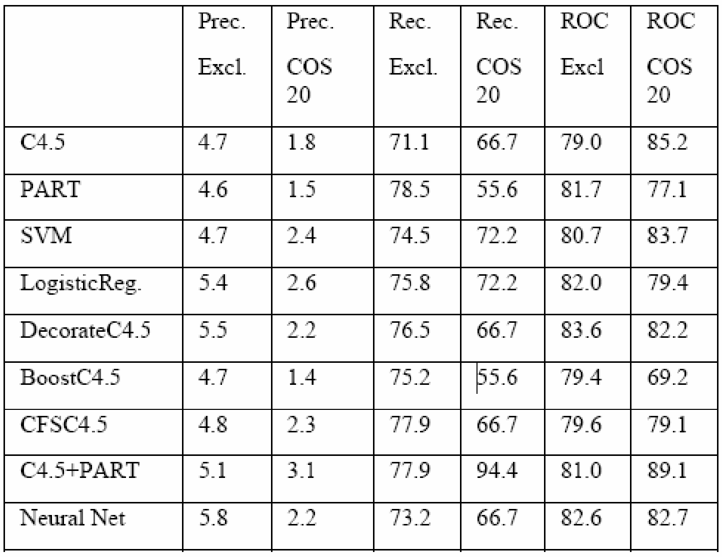
In this section, we compare the predictions of the various modelling techniques selected in Section 3.1. We first compare them in terms of the usual confusion matrix criteria (Recall, Precision, ROC area) and then in terms of cost-effectiveness as defined in Section 3.6. We then compare our results to the ones published in [Ostrand et al., 2005] in order to determine commonalities and differences. Other differences in terms of objectives and methodology were already discussed in Section 3.7.

3.8.1 Precision, Recall and ROC Area

First it is important to note that all results presented in this section are based on a balanced training set. As discussed in Section 3.7, there are a small percentage of faulty classes in COS, as it is usually the case in most systems. Nearly all the techniques we used performed better (sometimes very significantly) when run on a balanced dataset formed of all faulty classes plus a random sample of the same size of correct classes. The proportions of faulty and correct classes were therefore exactly 50% in the training set and the probability decision threshold for classification into faulty and correct classes for the test sets can therefore be 0.5 to achieve balanced precision and recall.[2]

[2] As you change the threshold, recall increases and precision decreases, or vice-versa

Table 3 Confusion Matrix for Precision, Recall and ROC Area for all Techniques



	Prec. Excl.	Prec. COS 20	Rec. Excl.	Rec. COS 20	ROC Excl	ROC COS 20
C4.5	4.7	1.8	71.1	66.7	79.0	85.2
PART	4.6	1.5	78.5	55.6	81.7	77.1
SVM	4.7	2.4	74.5	72.2	80.7	83.7
LogisticReg.	5.4	2.6	75.8	72.2	82.0	79.4
DecorateC4.5	5.5	2.2	76.5	66.7	83.6	82.2
BoostC4.5	4.7	1.4	75.2	55.6	79.4	69.2
CFSC4.5	4.8	2.3	77.9	66.7	79.6	79.1
C4.5+PART	5.1	3.1	77.9	94.4	81.0	89.1
Neural Net	5.8	2.2	73.2	66.7	82.6	82.7

Table 3 provides confusion matrix statistics for faulty class predictions and we can see that differences among techniques in terms of precision, recall, and ROC area are in most cases very small, or at least too small to be of practical significance. The results are less consistent across techniques on the COS 20 test set but this is to be expected as this release had a small number of faults and it is therefore subject to more random variation. Therefore neither the metalearners (Boosting, Decorate) nor the variable/feature selection techniques (CFS, RELIEF) seem to make a clear, practically significant difference. One exception is that the combination of C4.5 and PART seems to bring notable improvement in terms of recall for COS 20. ROC areas are overall rather high and mostly above 80%, but the question remains about how to interpret such a result to assess the applicability of models. Despite small differences in classification accuracy, as discussed in Section 3.1, it is important to recall that certain techniques are easier to use and more intuitive than others. This is the case of classification trees such as the ones produced by C4.5 or coverage rule algorithms such as PART. Furthermore, the feature selection techniques tend to simplify the models. For example, the decision tree generated by C4.5 after using CFS went from 24 leaves to 17 leaves and was built based on 29 variables instead of the original 112 variables considered. This may be of practical importance when applying the models.

The very small precision numbers are worth an explanation. This is due to the very imbalanced test sets, which are realistic, but which result nonetheless in low precision values. Even if there is a small misclassification probability of correct classes into faulty classes, when the proportion of correct classes is very large, this results into low precision. For example, based on a balanced test set with randomly selected correct classes and all faulty classes, we obtained precision numbers of 0.857 and 0.726, for the COS 20 and Extended test sets, respectively. So we can see that for imbalanced test sets, it is not easy to interpret precision values.

Table 4 Cost-Effectiveness for all Techniques

	1% Excl.	1%. COS20	5% Excl.	5%. COS20	20% Excl	20% COS20	100% Excl	100% COS20
C4.5	0.026	0.041	0.397	0.695	3.56	6.86	18.03	24.96
PART	0.010	-	0.014	-	1.55	5.27	18.32	33.61
SVM	0.005	0.023	0.117	0.670	2.43	4.68	17.78	21.14
Logistic Reg.	0.028	0.054	0.222	0.71	2.72	4.88	18.22	17.49
Decorate C4.5	0.003	0.016	0.210	0.726	4.18	6.16	19.06	15.16
Boost C4.5	0.02	-	0.406	-	2.73	-	16.04	0.34
CFS C4.5	0.026	0.041	0.32	0.635	3.19	6.62	17.46	20.36
Neural Net	0.031	-	0.193	0.069	1.50	0.70	12.79	12.55
C4.5 + PART	0.026	0.041	0.21	0.172	2.47	6.16	19.12	31.71

3.8.2 Cost-effectiveness

We now turn our attention to Table 4 where cost-effectiveness values are reported for all techniques and for selected percentages of classes. Though we also report the results for 100% of the classes (entire CE area), one would in practice focus on small percentages as such prediction models would typically be used to select a small part of the system. But recall that computing a CE area is just a way to compare models without any specific percentage of classes in mind and based on a unique score. Note that in Table 4, the symbol "-" stands for negative CE values, which we do not need to report. The reason why CE values may look small, though they have been multiplied by 100, is that the vertical and horizontal axes are percentages, and therefore values below 1. Admittedly such CE values are not easy to interpret but their purpose is only to facilitate the comparison among models based on a measure that should be directly proportional to cost effectiveness in our context.

We can see that, as opposed to confusion matrix criteria, there are wide variations in CE across techniques and class percentages. For example, for 5% NOS in the Excluded test set, CE ranges from 0.014 (PART) to 0.406 (Boosting+C4.5). What we can also observed is that C4.5, though never the best, is never far from it for all class percentages and for both the COS 20 and Excluded test sets. On the other hand we see that some techniques provide unstable results which vary a great deal depending on the test set and the selected NOS percentage. For example, PART varies from reasonably good CE values to negative CE values. Based on these results, we selected C4.5 to apply within the COS project as it is both simple (very interpretable, easy to build and apply) and stable in terms of cost effectiveness. However, from a general standpoint, if one can pre-determine what percentage of the code will, for example, undergo additional verification and testing, one may be in a position to choose the specific model optimizing CE for this particular percentage. For example, though NN does not fare particularly well in general, it does well for 1%. We can also assess the gain of using a predictive model for a specific percentage in a way which is more interpretable than CE areas. Figure 2, which was presented earlier, corresponds to actual curves for C4.5 on the COS 20 test set. If we take these results as an example, we see that, for example, 10% of the lines of code lead to nearly 60% of the defects. If we now assume we would have a perfect, optimal model to predict the future, we would obtain 100% of the faults for that NOS percentage (see Figure 2). The gain compared to what the average would obtain with random orders (10% of the faults) is substantial, but we also see that there is much room for improvement

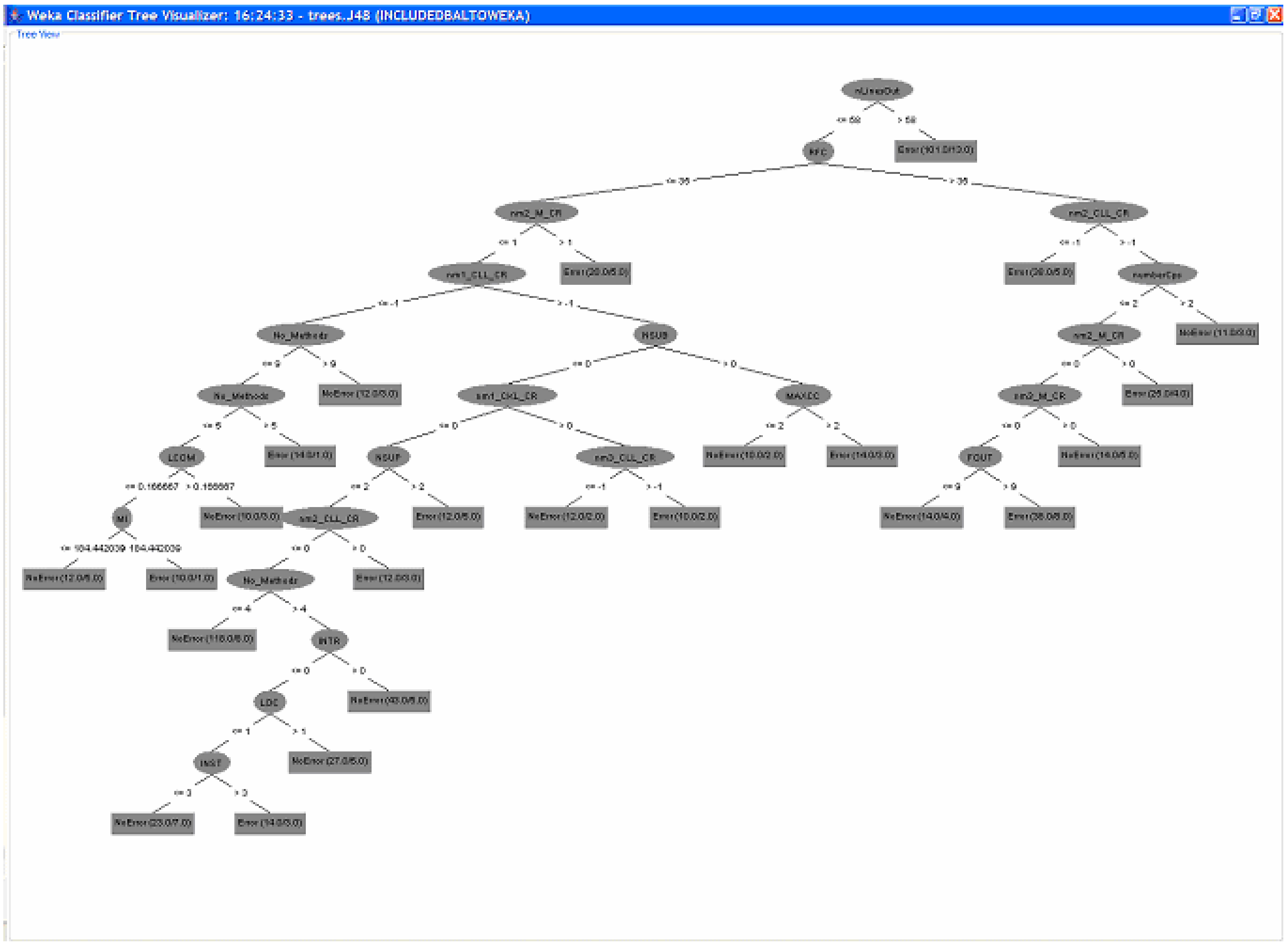


Figure 3 C4.5 Decision Tree

when compared to an optimal order. This was not clearly visible when only considering the confusion matrix precision and recall, or the ROC area.

3.8.3 Variables selected in the C4.5 Tree

The C4.5 decision tree in Figure 3 has 24 leaves, implying that any one prediction will be assigned one of the possible 24 fault probabilities associated with these leaves. The tree makes use of 21 variables out of the 112 originally considered.

From Figure 3, we can see variables that belong to two distinct, broad categories:

- Nine variables relate to the amount of change undergone by classes in the current release or one of the last three releases. However, in some cases this should be carefully interpreted as it may also capture whether the class is new: for example, when nm1_CLL_CR = -1, this means the class did not exist in release n-1.
- Eleven variables relate to the source code properties of the class, such as inheritance, number of methods, cyclomatic complexity, cohesion, and coupling (fan out).

Of course, it is always difficult to interpret such results as many variables are often inter-correlated, as suggested by the fact that CFS only selected 29 variables out of

112 (Section 3.4). But what this variable selection tells us is that both properties of the class source code and change/fault history are useful and complementary predictors. If we compare the variables selected with the ones in [Ostrand et al., 2005], a study which was discussed in Section 3.7, their Binomial regression model also used a code metric: file size in lines of code, and whether the file was changed in the previous release. Their variable coding the age of a file in terms of releases, is coded in our case across many variables (nm* measures in Table 1) capturing whether the class was new in one of the last three releases by assigning a -1 value to variables. The number of faults in the previous release is captured by the nm1_E_CR and nm1_CE_CR measures in Table 1, which separate critical from non critical faults. In addition, we also capture changes and fault corrections in the last three releases and distinguish requirements changes according to their size and complexity (as defined in Section 3.4). We also have many additional structural measures for cohesion, coupling, and many other attributes, some of them being selected in the predictive models as discussed above.

Variables capturing the number of developers involved and their past experience on the COS system (numberPastCR measures) are also considered, but are not selected in the predictive C4.5 model we ended up choosing to focus verification. On the other hand, we do not have a programming language variable in our data set as all our predictions involve Java classes.

3.8.4 Fault-distribution in Fault-prone classes

In order to compare our results with the results of [Ostrand et al., 2005], let us look at the percentage of faults in the top 20% classes, a threshold that these authors indicated was the “knee of the curve” where the number of faults starts to plateau. Though their unit of analysis was a file, we explained above that it should be comparable for Java classes and we therefore compare our results to theirs using their evaluation criterion. We see that if we consider our C4.5 model, the 20% most fault prone classes account for 69% and 71% of faults for the Excluded and COS 20 test sets, respectively. This is significantly less than the 83% average reported in [Ostrand et al., 2005]. However, if we look at other modelling techniques, some of them such as the one combining PART and C4.5 reach 72% and 90%, respectively. But if we look at the percentage of lines of code, 20% of classes correspond to 59% of the code in their study. If we go back to the C4.5 CE analysis in Figure 2, we can see that around 59% NOS we capture around 90% of the faults. Our results in terms are therefore comparable to what Ostrand et al. obtained, but it shows that one must be careful about using size measures such as number of 10 classes or files. The reason why we obtained a slightly larger percentage might be due to the additional variables we consider, but this is hard to ascertain.

4 Practical evaluation of the use of the prediction model to focus testing

This section describes the test-process suggested by us, the actual implementation of the additional unit testing, the data reporting, our cost-effectiveness model, as well as the interview process. Finally the research results are discussed.

4.1 Using prediction models to focus unit testing

Many studies state that a large portion of software development cost is due to fixing reported faults [Khoshgoftaar et al., 1998]. Furthermore, the cost depends on when the faults are discovered and corrected [Bakkelund & Kvam, 2004]: faults should be discovered as early as possible because the cost of fixing them increases over time. Such early identification of faults is likely to increase the efficacy of the testing activity and to improve the overall quality of the evolving product. A challenge related to the above is to choose the right techniques for finding and correcting the faults early and in an efficient manner. We believe that an early identification of fault-prone components will allow an organization to take appropriate action. Explicitly, resources can be allocated to testing and subsequently correcting faults detected in these components in order to reduce the likelihood of software failures in new releases of the evolving software product.

To facilitate this, a strategy of using the predictions in unit testing to focus on fault-prone classes was adapted. Unit testing is typically performed early in the project life cycle and the costs of finding and fixing faults at this stage is considered more cost-effective than in later phases of the project. This proved to be consistent with COS test strategy as well. COS uses a test driven development process, where unit tests are written early in the project life cycle (before anything is implemented and on the basis of the object-oriented analysis artefacts) and this applied nicely our initial strategy.

More specifically, we wanted to ensure that the unit tests of fault-prone classes were reasonably complete, so as to minimize the number of faults that could slip through to subsequent testing phases or production. This was done due to the abovementioned increase in cost in later project phases, but also based on the so-called ripple-effect [Haney, 1972], suggesting a change in one module would necessitate a change in any other module. By making sure of that fault-prone modules work properly in isolation could reduce the likelihood of possible ripple-effects. This assumption is supported by the fact that we, through the model building, established a correlation between changes in a module and its fault-proneness, underlining a possible ripple-effect.

While testing can be used to measure the quality of your software, test coverage can be used to measure the quality of your tests. To analyse test coverage, means to measure how well the test exercise your product. The COS project guidelines specify that 80 percent or more of the code (executable statements) should be covered by tests in a test coverage analysis. In practice this was not achieved due to time constraints. In order to have a reasonable chance of demonstrating benefits of the prediction model, not even 80 percent statement coverage was perceived as sufficient. We suggested that 100 percent loop and branch test coverage [Bullseye] was to be reached for the most fault-prone classes.

Many faults can be associated with loops in practice, e.g. their stopping condition. It is therefore advised to go further in exercising them than just ensuring a loop condition evaluating to true and false as required by branch coverage. The loop coverage measure reports whether you executed each loop body zero times, exactly once, and more than once (consecutively).

To fully exercise a loop it is usually advised to do as follows:

- a test case should bypass the loop (i.e., the loop condition is false to start with)
- a test case should execute the loop once
- a test case should execute the loop a “representative” number of times (consecutively)
- if possible, the loop should be executed a maximum number of times (assuming such a maximum exists)

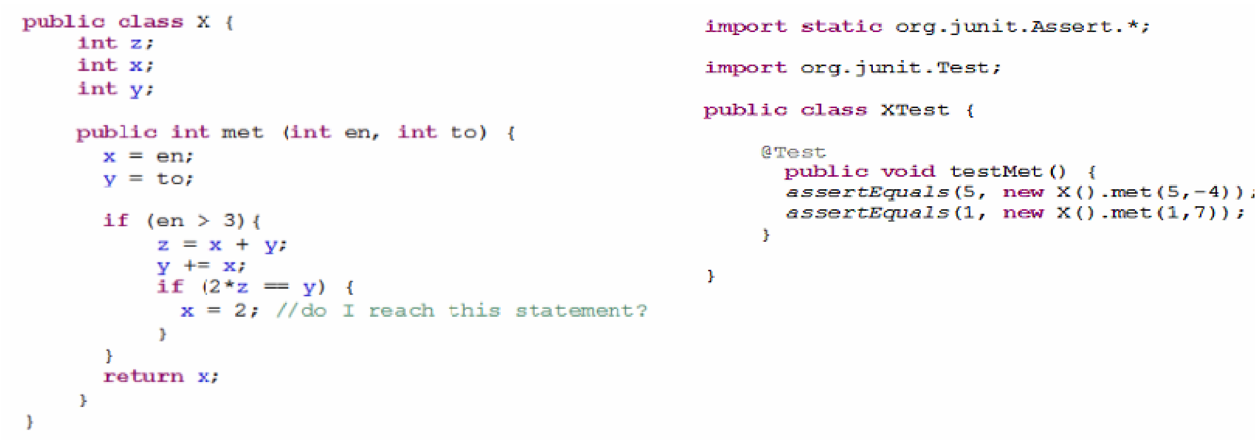
If we take an example of a search in a table:

- we skip the loop if the table is empty
- we find the element we search for in the first position of the table
- we find the element we search for after the first position
- we do not find the element after searching the entire table

The statements that involve conditions in a Java program include *if*, *while*, and *switch* statements. Whereas the former two include exactly one condition, *switch* statements usually include at least two. The branch coverage strategy for testing requires that each of those conditions be exercised by having them evaluated to true in some test case executions and false in others. This coverage criteria is more demanding than simple statement coverage is, for example with the case of an if statement without else block, statement coverage does not always require a condition to be false in one test case execution to cover all statements.

To verify these two test coverage criterions a test coverage tool can be used. Prior to initiating the unit testing based on the predictions, a tool was used at COS that did not report branch coverage properly. As part of our strategy we proposed the use of a new test coverage tool that reported this criterion properly.

When trying to increasing test coverage in practice one typically tries to reach statements of the code not already covered by the initial test cases. This is known as statement coverage and reports whether each executable statement is encountered.
The advantage of this measure is that it can be applied directly to object code and does not require processing source code. The disadvantage of statement coverage is that it is insensitive to some control structures. For example, consider the following Java code fragment and its corresponding test case:

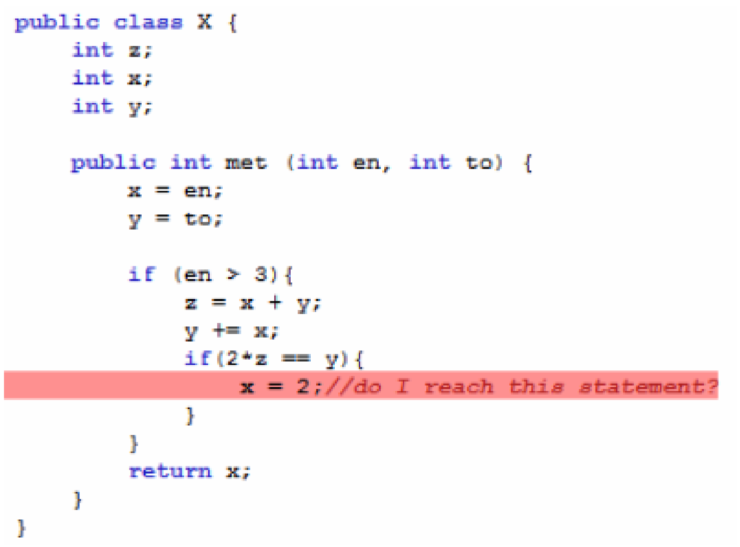


```
public class X {
    int z;
    int x;
    int y;

    public int met (int en, int to) {
      x = en;
      y = to;

      if (en > 3){
          z = x + y;
          y += x;
          if (2*z == y) {
            x = 2; //do I reach this statement?
          }
      }
      return x;
    }
}
```

```
import static org.junit.Assert.*;

import org.junit.Test;

public class XTest {

    @Test
      public void testMet() {
      assertEquals(5, new X().met(5,-4));
      assertEquals(1, new X().met(1,7));
    }

}
```

Figure 4 Java code fragment and corresponding test case

When running the test case, the test coverage report yields that the statement where the comment is placed is not reached illustrated in red.



```
public class X {
    int z;
    int x;
    int y;

    public int met (int en, int to) {
        x = en;
        y = to;

        if (en > 3){
            z = x + y;
            y += x;
            if(2*z == y){
                x = 2;//do I reach this statement?
            }
        }
        return x;
    }
}
```

Figure 5 Test coverage report

This is due to the condition above, if (2*z==y) never evaluates to true. To find test inputs that will execute an arbitrary statement Q within a program source, the tester must work backward from Q through the program's flow of control to input statements.

For simple programs like above, this amounts to solving a set of simultaneous inequalities on the input variables of the program, each inequality describing the proper path through one conditional. Conditionals may be expressed in local variable values derived from the inputs and local variables must be substituted with input variables in the inequalities.

In this example we have the z as local variable and x and y as input variables. Based on analysis of conditional statements, the following two inequalities must be solved:

$x > 3$
$2(x+y) = x+y \Leftrightarrow x = -y$

One possible solution to cover the statement is therefore: $x = 4$, $y = -4$

In practice, the presence of loops and recursion in the code makes it more difficult to solve such inequalities. But the above example is just aiming at illustrating the principles.

When performing the unit test using the predictions, a practical procedure was developed to guide the developers (Appendix A). In short terms the procedure consisted of running the initial test suite and analyzing the results in terms of uncovered code. Then augmenting the test suite to achieve as high branch and loop coverage as possible, re-run it and correct any faults identified.

During a meeting at COS it was discussed whether additional types of verification and validation (V&V) e.g., other kinds of tests, inspections, could be performed as part of the evaluation. Several developers thought this would be useful, and in the long term, we agreed that this probably was a good idea. However, the time allocated to unit testing was limited and we pointed out that it was essential that at least one activity, namely unit testing, was performed well. If we were to allow other kinds of V&V, we would only be able to cover very few fault-prone classes, and furthermore we would not be able to separate the effects of the focused unit testing activity from other kinds of V&V in a reliable way. It is not certain that unit testing is the best application of fault-proneness models. Maybe inspection is a beneficial technique as well. Either way this thesis documents the application of unit testing, and possible research directions in the future could include doing the same for e.g. inspection.

A list of the 100 most fault-prone classes (see Appendix C for the top 30), guidelines explaining coverage criteria (Appendix A) and test report templates (Appendix B) were given to the developers to guide and document the unit testing. The list of fault-prone classes was sorted by their size, as in LOC, if two classes had the same fault-probability the smaller one was placed before the larger one. This was done to achieve a more efficient usage, as in more classes evaluated due to the limited time available. But this was also done due to the fact that our model predicted a relatively small amount of the total number of classes as fault-prone, and evaluating more classes could prove to be beneficial.

4.2The data reporting

During the unit testing, the developers were told to document their increase in coverage by generating a before-and-after test coverage report to be able to observe what had been done. This was done for the entire system upfront of the unit testing and once after it was finished. In addition their IDE supported the generation of test coverage reports during test suite modifications, making them able to monitor their work in progress at code level for one specific class with one corresponding JUnit test class containing the test suite. Obviously this made the additional unit testing more effective, as in the participants not being forced to wait for the entire system to be analyzed to observe results of test suite modifications.

A test report was filled out to report hours used separated into time spent on augmenting the test suite and time used to correct the faults found. Justifications had to be made to explain cases where a class was not chosen to achieve higher branch and loop coverage. In addition, justifications on why not the coverage criteria posed were achieved were recorded. Finally, the numbers of faults found obviously needed to be recorded.

All this information was to be documented in the test report template (Appendix B).

4.3 Cost-effectiveness Model

We developed a cost-effective model (CE model) and discounted potential costs from future test phases into net present values in terms of cost saved.
The rationale of the model is that in each testing phase (System test, System Integration Test, Acceptance Test and Production (after delivery)), the faults found has probabilities of being detected, costs of detection and probabilities of actually being fixed in the phase it was detected. The interviewee then provided estimates for these parameters in the future and the model calculated present cost saved.
Our definition of cost-effectiveness is the ratio *cost saved by the unit testing / cost of unit testing*. The parameters of the CE model are outlined in Table 5.

Table 5 Parameters of the CE model

Parameter	Description
$PD_{x,i}$	Probability of fault *i* being detected in phase *x*
$CD_{x,i}$	Potential detection cost of fault *i* in phase *x*
$PC_{BF,x,i}$	Probability that fault *i* detected in phase *x* will be corrected in Bugfix release *(BF)*
$PC_{NMR,x,i}$	Probability that fault *i* detected in phase *x* will be corrected in Next main release *(NMR)*
$PC_{x,i}$	Probability that fault *i* detected in phase *x* will be corrected in phase *x*
$CC_{x,i}$	Potential correction cost of fault *i* in phase x
$CC_{NMR,i}$	Potential correction cost of fault *i* in Next main release *(NMR)*
$CC_{BF,i}$	Potential correction cost of fault *i* in Bugfix (BF)

Using these parameters, the CE model can be formally expressed as follows:

CE = Cost Saved by additional Unit Testing / Cost of additional Unit Testing
with
Cost Saved by Focused Unit Testing Activity:
$\sum_{i,x}(PD_{x,i} * (CD_{x,i} + PC_{BF,x,i}*CC_{BF,i} + PC_{NMR,x,i}*CC_{NMR,i} + PC_{x,i}*CC_{x,i})$
and
Cost of Focused Unit Testing Activity:
Cost of augmenting the unit test suite, running the tests, checking results and fixing any faults found in the 26 classes selected.

4.4The interview process

It is not always practically possible to collect all the data needed to get a valid result when conducting empirical studies in industry. While conducting empirical studies in industry, other more important activities might occur, forcing the involved parts to focus on these rather than the research, thus degrading the research initiatives and possibly reducing the amount of data collected. In our case, when the cost effectiveness of the unit testing was to be measured, the timeframe was not sufficient enough for us to collect data from all the testing phases in a release.
The amount of data might be too small to be able to draw broader conclusions, rather than just being valid for the specific context and not being useful outside this context.
We wanted to combine actual collected data with expert estimates to make our results more valid and to able to estimate cost-effectiveness in a discounted way using the model described in the previous section.

To get expert estimates, the developers involved in the unit testing where asked to participate in an interview (Appendix D and E). We asked 3 developers about the probabilities for and cost of detecting and fixing faults in the different testing phases, but the first interview was just a pilot serving as an evaluation on our questions and if they captured what we wanted. The results from the last two interviews were used to its full extent. Our assumption was that the costs of fixing faults increased as the project moved forward. For each testing phase we asked for an interval with a practical minimum, a practical maximum and a most likely value for both hours possibly consumed and probabilities for fault detection and correction in the respective phases (Appendix F). Our assumption was that by having estimates of potential hours consumed in the different phases and probabilities for the occurrences of these, we could calculate the cost effectiveness of performing additional unit testing.

More informally we informed that within the boundaries of practical minimum and maximum one would find 9 out of 10 cases, and explained this concept using the visual aid documented in Appendix F. We explained for that a distribution not necessarily had to be normally as in evenly distributed. There are cases where the most likely value could be closer to the practical minimum or maximum and this had to be taken into consideration. An additional visual aid was provided to explain how we were to work with the collected data after the interviews (Appendix G).

It was viewed to us as very important to make the developers aware of *all* the costs associated with each phase. For this reason the test manager was asked what was done in each phase to detect and correct faults. Before each interview started the developers were reminded about this to take into account any hidden costs as well when providing their estimates. A sheet providing an overview of typical activities in each testing phase worked as visual aid to remind them during the interviews.

The concept of bias and more specific the concept of availability, as in people retrieving events from long-term memory with different ease, was considered when asking the experts for estimates. One should be aware of that by asking straightforward, accurate estimates will not be provided accurately. In order to provide accurate estimates, the experts need to remember relevant information regarding the parameters to be estimated [Meyer & Booker, 1991]. Knowing this, we gave the test report and coverage report to the interviewee before we asked the questions and told them to think through the origins of the fault and what was done to fix it. We made the interviewee think aloud.

In addition, the interviewees were made aware of that the answers they provided was to be considered only rough estimates. This was due to another concept of bias, namely the over-confidence, as in the interviewee thinking that by being an expert implies the ability to give exact answers.

4.5Preliminary results and practical model evaluation

In the pre-unit testing phase some issues regarding the validity of the model was emphasized from some members of the development team. It was pointed out that the list of fault-predicted classes was not optimal from a user point of view. More specific it was argued that about 70 percent of the top ranking fault-prone classes had a cyclomatic complexity of 1. Further it was stated a lot of the most fault-prone classes were only classes with set- and get methods or only classes containing constructors calling super constructors being just interface classes or abstract classes. For that reason, performing the activity suggested by us did not make any sense to them. Of the top 20 most fault prone classes 70 per cent of them would, according to COS members, be meaningless to test. We argued that a prior evaluation of the model had been conducted in terms of using one release n and applying the prediction formula on these data and evaluating how well it predicted faults in release n+1. This evaluation stated that by checking 10 percent of the classes, one could detect 60 percent of the total number of faults.

Another argument proposed by the COS team and its members was that the prediction model did not actually managed to state where the errors actually occurred. They argued that they have observed that in their configuration management system Change Requests had been reported as errors, when it for real not was an error.
This type of things would obviously create noise in our statistical model. If they in some cases report error as maintenance or the other way around and it would lead to a worse model than if they where consequent all over in their logging. Actually we use the presence or absence of error to evaluate the classification accuracy of our model. But when these things are reported inconsistent or different between developers our model is build on inconsistent data and therefore contains more noise than it would have if this was consistent.

They finally argued that by using a technique that combined both the prediction results and a so called expert considerations, which can be observed as e.g. domain knowledge in the sense of overview of the systems code and more specific what part of the code that was dead code (or code that really doesn’t do anything just sits there, but removing it would result in faults), would result in violations to the application of model. In other words by skipping a lot of classes one could ruin somewhat the intension of using the model to focus testing. We stated that this did not ruin anything at all, but rather made the use of the predictions better. The optimal way to use the results is to combine the expert considerations and the prediction results to better focus verification.

Two days into the evaluation, a little breakthrough occurred. The change manager informed me that they had found a fault using the technique. They also found faults surrounding the class under scope. He thought the technique was very interesting and if we removed the irrelevant classes then this could be a good way to perform cost effective quality assurance. The motivation among participants in the evaluation was significantly better this day than earlier. The technique had some limitations because of the, in their opinion, irrelevant classes, but the technique was very promising and interesting was the change managers conclusion.

4.6 Results

The practical usage at COS summed up to involve 3 developers using 49.5 hours on augmenting the test suites for fault-prone classes, re-running the tests and correcting the faults found.

Of the 100 top ranking classes 26 was subject of *potentially* receiving higher coverage, as in the participants got this far before the unit testing ceased. Of these 26 classes 7 achieved higher coverage due to participant appraisal, that is. Before the unit testing began, the total test coverage on the classes being subject to the evaluation was 69 percent, underlining discordance with COS policy of test quality (80 percent coverage). Post-evaluation total test coverage was 82 percent being in accordance to COS policy. Average increase in test coverage among evaluated classes was 16 percent, with a standard deviation of 7 percent. 7 faults were detected in 3 classes and the distributions of faults among classes were not one-one. Two converter classes consisted of 1 actual fault and 4 robustness faults, where the first more specific had to do with converting a value and that this converting initially was wrong. The latter 4 robustness faults had to do with that faults occurred when a value was null. These 4 faults were joined to one fault due to the fact that, according to the person who found it, if one of these faults were found, then the rest of them would occur automatically, as in these faults having the same structure. One class predicted as fault-prone actually received extra inspection and modifications in parallel to the evaluation, not based on the predictions, but still supporting the classification accuracy of our model.

Reviewing the results from the evaluation (Appendix H), a trend was observed in that DTO's consistently was rejected as subject for increased test coverage. However this proved to be wrong, observing that two DTO classes actually contained faults.

Skipping classes before achieving 100 percent loop and branch test coverage was justified several times due to huge workload developing stubs. A stub is a partial implementation of a component on which the tested component (here: class) depends upon, enabling it to be isolated from the rest of the system for testing.
If this skipping suggest limitations regarding out initial unit testing strategy or the fact that COS has a low code quality in terms of unit test quality is not clear. Considering that this is a legacy system under maintenance, certainly actions should be taken to assure higher quality of the code, especially considering the recurring problem with bugfix releases yielding a serious root problem. Increasing conformance costs by developing better stubs could reduce non-conformance costs.

Table 6 Results CE calculations

	Fault#1	Fault#2	Fault#3	Fault#4	Sum
Dev. 1	16	4	11.35	13.17	44.52
Dev. 2	14.4	7	49.25	48.12	118.77
Average	15.2	5.5	30.3	30.7	91.7

Our cost-effectiveness model was applied on the data collected and the detailed results are provided in (Appendix I). A summary the results are presented in Table 6. The results state a cost saving of roughly 50 percent by performing the additional unit testing when using only the most likely values as means to calculate cost saved. This is a very conservative measure, corresponding to the so-called mood (most frequent value) as opposed to the mean, because many of the estimates had distributions skewed to the left. In future work of this study and in replicates, using means instead and combining this with Monte Carlo simulation is suggested.

One important reason for the difference between the estimates of Dev. 1 and Dev. 2 in Table 6 was that Dev. 2 thought it was much more likely that Fault#3 and Fault#4 would not be discovered in the later testing phases, and would result in faults remaining undetected until the system was released, in which case it was much more expensive to correct. Otherwise, results from the interviews yielded a fair interrater reliability between the actual interviewees. The majority of the answers were consistent between the two, although they had been interviewed separately and we ensured that the interview questions were not discussed between them upfront. Due to modifications of some uncertainties regarding the questions, the pilot interview differed a lot from the two actual interviews suggesting our modifications had good impact, as to ask the right questions in the right way.

As mentioned earlier other V&V techniques were also discussed and considered prior to the unit testing. As future work, one possibility is to perform inspections on fault-prone classes instead of (or in addition to) unit testing in the next release to evaluate if this is a even more suited technique as in being able to cover more faulty classes with the same amount of effort.

5 Conclusions and Further work

One part of this thesis focused on comparing different data mining and machine learning techniques to build fault-proneness models in a Java legacy system. The usual and general way to compare classifier models is to use criteria based on the confusion matrix; precision, recall, ROC area. Our results show that the modelling techniques do not show, in most cases, practically significant differences in terms of these criteria. However, we believe that using such general evaluation criteria can be misleading as they are not direct, surrogate measures of the cost-effectiveness of using such fault-prone models. Cost-effectiveness models need to be context-specific. In our context, where extra testing is applied to a subset of classes in their decreasing order of predicted fault-proneness, we wanted to detect as many faults as possible while covering the least amount of code possible with our extra testing.

The underlying assumption is that the extra testing effort will be roughly proportional to the size of the code tested. Based on our proposed cost-effectiveness analysis procedure, we concluded that significant differences were indeed visible across modelling techniques, as opposed to what was concluded based on the confusion matrix.

Though we use data coming from one large industrial project, the data was gathered across many releases during which significant organizational and personnel change took place. Furthermore, we do not believe that this project environment has any specificity that would somehow make it substantially different from other object-oriented, legacy Telecom systems with frequent releases and high personnel turnover. This is also supported by the fact that we obtain results that are similar to those of other case studies in the Telecom domain. This is encouraging as this suggests such prediction models could be applicable in a variety of environments.

From a more general standpoint, regardless of how it is defined in any specific context, we recommend to use a specific cost-effectiveness model in addition to standard confusion matrix criteria when building and evaluating fault-proneness prediction models.

Another part of this thesis focused on the practical evaluation of adapting fault-proneness prediction models to focus unit testing. Although only a small part of the system was subject to the evaluation, revealing 7 faults is considered a good indication of the usefulness of fault-proneness prediction models to focus testing, knowing that in a typical release, somewhere between 7 and 83 faulty classes exists among the 2600 core Java classes. The historic data indicated a fault density of between 7/2600 – 83/2600, or between 0.0027 – 0.032 faults per class. In our case 26 classes were considered and 7 faults were found, giving a fault density of 0.27 faults per class. Furthermore, the results of a cost-effectiveness evaluation, using expert estimates to elicit cost savings, suggest a high return of investment from using this technique in unit testing. The evaluation indicated that cost saved was 91.7 hours and time consumed during unit testing was 49.5 hours, indicating more than 50 percent cost reduction due to the focused testing activity.

Another interesting aspect was the clustering of faults surrounding the largest class that has been reported in other research [Ostrand & Weyuker #2, 2004]. Evidently this supports the Pareto Principle, as a in the majority of the faults are grouped together in relatively small parts (classes) in the software.

An unintended result from the practical evaluation was the need for more rigid fault-reporting. Analyzing the change management system for errors during data collection revealed that the routines for reporting errors and what was considered an error were not consistent across the development team.

Further the need for tuning the model proved to be of huge importance. Both to make the model accuracy better and more applicable from a users point of view, but maybe also as a mean to involve the users of the technique more, thus accomplishing better motivation among them, which proved to be a problem in this work.

There are many open questions that need to be addressed before one can generalize the results and adapt the techniques described in this thesis to a wider context. For example, should data be collected for all modules in that system or just the modules modified? Using metrics from the entire system or just part of it, as in changed or new modules, seems to be divided in work in the area. Descriptive statistics before building the actual models could reveal which approach is most suited to follow. Second, what type of metrics should be gathered, in terms if being good predictors of external system qualities? Do some metrics or measures prove to be better predictors than others? An empirical comparable study on quality models concerning this revealed that coupling was a good predictor as well as size, while inheritance and cohesion proved not to be. Third, when building fault-proneness prediction models, should you use a discrete or continuous indicator of faultiness?

Further work consists of ensuring a more robust and automated data collection procedure introducing a relational database with consistence checking supporting quality assurance of the data. In addition different views of the data are to be introduced, making it easier deriving models for, e.g., fault-proneness of change requests or modules of higher abstraction level. A GUI and fully automated data collection would make it possible for people at COS to use this technique as a test support tool without having any knowledge of the statistics involved.

Appendixes

Appendix A: Guidelines

Guidelines to apply the Simula fault-proneness prediction model on the COS project at Telenor

E. Arisholm, L. Briand, M. Fuglerud
Simula Research Laboratory

Context:
The prediction model developed by Simula indicates the classes that are the most likely to contain a fault. A ranking of the COS 22 classes according to the fault-proneness has been provided by Simula to Telenor.

Motivation:
The question is now how to best use such a prediction model to focus testing efforts in the COS project, and in particular for the COS 22 release.
Strategy:
Our initial strategy to use such predictions is simple, though more complex strategies will be considered in the future. As a first step, we want to ensure that the unit test of fault-prone classes is reasonably complete, so as to minimize the number of faults that slip to system test. To evaluate the cost-effectiveness of using the prediction model to focus testing efforts on COS 22, an additional two-day unit testing phase will be conduced just before “code freeze”.

Though, in theory, COS project guidelines specify that 80% of the code (executable statements) or more should be covered, this is not systematically achieved in practice due to time constraints. In order to have a reasonable chance of demonstrating benefits of the prediction model, even 80% statement coverage is probably not sufficient. We suggest that full loop and branch coverage should be reached for the most fault-prone classes according to the guidelines provided below. This strategy furthermore presumes that the developers use the Clover coverage tool to assess branch coverage (Cobertura does not properly report branch coverage).

Note that during the presentation/meeting in December we also discussed whether additional types of V&V (e.g., other kinds of tests, inspections) could be performed as part of this evaluation. Several developers thought this would be useful, and in the long term, we agree that this is probably a good idea. However, given the limited time allocated to the focused testing activity in COS 22, it is essential that we perform at least one activity (unit testing) well. If we allow also other kinds of V&V, we would only be able to cover very few fault-prone classes, and furthermore we would not be able to separate the effects of the focused unit testing activity from other kinds of V&V in a reliable way. Thus, we propose the following procedure for COS 22.

Reaching uncovered statements

To find test inputs that will execute an arbitrary statement Q within a program source, the tester must work backward from Q through the program's flow of control to input statements.

For simple programs, this amounts to solving a set of simultaneous inequalities on the input variables of the program, each inequality describing the proper path through one conditional. Conditionals may be expressed in local variable values derived from the inputs and local variables must be substituted with input variables in the inequalities.
Let's take for example the following program chunk (in C) where we wish to cover a specific statement (as indicated by the comment).

```
int z;
scanf("%d%d", &x, &y);
if (x > 3) {
     z = x+y;
     y+= x;
     if (2*z == y) {
          /* statement to be covered */
…
```

where we have the following variables :
    Local variable: z
    Input variables: x , y

Based on analysis of conditional statements, the two following two inequalities must be solved :

```
X > 3
2(x+y) = x+y ⇔ x = -y
```

One possible solution to cover the statement is therefore: `X=4, y=-4`

In practice, the presence of loops and recursion in the code makes it more difficult to solve such inequalities. But the above example is just aiming at illustrating the principles to be followed.

Branch Coverage
The statements that involve conditions in a Java program include if, while, and switch statements. Whereas the former two include exactly one condition, switch statements usually include at least two. The branch coverage strategy for testing requires that each of those conditions be exercised by having them evaluated to true in some test case executions and false in others. This coverage is more demanding than simple statement coverage as, for example with the case of an if statement without else block, statement coverage does not always require a condition to be false in one test case execution to cover all statements.

Exercising loops
Many faults can be associated with loops in practice (e.g., their stopping condition). It is therefore advised to go further in exercising them than just ensuring a loop condition evaluates to true and false (as required by branch coverage).
To fully exercise a loop it is usually advised to do as follows:

- a test case should bypass the loop (i.e., the loop condition is false to start with)
- a test case should execute the loop once
- a test case should execute the loop a “representative” number of times
- if possible, the loop should be executed a maximum number of times (assuming such a maximum exists)

For example, if we take the example of a search in a table:

- we skip the loop if the table is empty
- we find the element we search for in the first position of the table
- we find the element we search for after the first position
- we do not find the element after searching the entire table

Practical procedure for unit testing in COS22

The test manager ensures that a code “checkpoint” is provided *at least four days before the unit testing phase will start <replace with date>.*

We will then collect change and code data for that checkpoint on the COS 22 release, apply the fault prediction model on the data, and deliver a prioritized list of fault-prone classes *before the unit testing phase starts <replace with date>.*

The two-day testing phase will start on January <*xx*>, before “code freeze”. The process consists of the following activities and deliverables:

The test manager assigns the most fault prone classes, one at a time, to developers on the basis of the prioritized list of fault prone classes.

The developer assigned to the class will:

1. Analyze the branch coverage of their assigned class: Use Clover to identify the branches that are uncovered after executing the existing test suite. The code corresponding to uncovered branches can fall into three categories:
   a. Unreachable (e.g., dead code): No further action is required
   b. Changed functionality: The uncovered code corresponds to new or changed functionality and should be entirely covered by the test suite (see below for further guidelines).
   c. Unchanged functionality: The uncovered code should not be affected by the current release changes. However, one should be very careful that this is really the case. It is not always easy to determine the impact of changes. In this case, if there is any doubt that a change could have an impact, it is better to be conservative and ensure that all code be covered.
2. Analyze the loop coverage of their assigned class: As described above, loops are fully exercised (“covered”) when they are bypassed, executed once, a representative number of times, and possibly a maximum number of times. If some of these options are not possible, a short justification should be provided

by the developer.
3. Augment the class test suite to achieve branch and loop coverage, run it and correct any faults identified.
4. Provide a test report to the test manager. This consists of
    - the tool coverage report after executing the initial (existing) test suite
    - the tool coverage report after executing the final (augmented) test suite
    - a justification in cases where there are still uncovered code (branches, loops)
    - number of additional faults found
    - estimate of time spent (hours) on augmenting the test suite and running the tests
    - estimate of time spent (hours) on correcting the faults

Once the test manager approves the test report, the test manager assigns the next most fault prone class to be tested to the developer.

Appendix B: Test report

Test Report to be used during the additional unit test phase

1. Provide the tool coverage report after executing the initial (existing) test suite. To do this, simply save the report concerning the class under scope as it is before you start modifying the test suite. Name this report <klassenavn>_before.xxx
2.

Class Name:	
Developer:	
Justification if there still exists uncovered code (branches, loops):	
Number of additional faults found:	
Estimate of time spent (hours) on augmenting the test suite and running the tests	
Estimate of time spent (hours) on correcting the faults:	

3. Finally include the tool coverage report after executing the final (augmented) test suite. Name this report <klassenavn>_after.xxx
4. Remember to save this template as <klassenavn>.doc

Appendix C: Fault-prone classes. The 30 sorted classes provided during the additional unit testing

fileName	FaultProb.	Rank	Assigned to:	Not assigned because:
AgreementContainerConverter	0.929	1		
CleanupFilesMDBean	0.929	2		
FileResult	0.929	3		
AgreementDtoStatus	0.929	4		
AgreementMemberDtoStatus	0.929	5		
ChordiantUtil	0.929	6		
ExternalTransactionInitialData	0.929	7		
ProductOfferRuleAssembler	0.929	8		
ModifiableAgreement	0.929	9		
ProductOfferSenderBase	0.929	10		
DealerInfoDto	0.929	11		
DealerIdDto	0.929	12		
AgreementContainer	0.929	13		
ExternalTransactionRepository	0.9	14		
ExternalTransactionLogException	0.9	15		
ExternalTransactionPaymentException	0.9	16		
ExternalTransactionValidationException	0.9	17		
ExternalTransactionException	0.9	18		
CommissionReportService	0.9	19		
ProductOfferValidator	0.871	20		
AgreementRepository	0.871	21		
ProductOfferServiceBean	0.871	22		
EurekaServiceBean	0.871	23		
OrderInformation	0.871	24		
OrderValidationServiceBean	0.871	25		
AgreementServiceBean	0.871	26		
ExternalTransactionDto	0.871	27		
ProductOfferStructToProductOfferInterceptor	0.871	28		
ProductOfferRepository	0.871	29		
SubscriptionFinder	0.871	30		

Appendix D: Interview questionnaire

Questions for fault #

> The information given in this questionnaire will be kept confidential!

Date: ______________________

Name of interviewee: ______________________

Email of interviewee: ______________________

Phone-number of interviewee: ______________________

Background

In the following we would like you to give some information about your experience with respect to the COS system and Telenor

1. How long have you been working with the COS project? ________________ years

2. How long have you been working with Telenor? ______________________ years

Questions regarding the cost of fault correction

1. For this particular fault, in the context of the System testing phase and based on your experience:

   - In which range, according to your experience, is the cost for correcting this fault?

     _______ to _______

   - What would you deem as a most likely cost for correcting this fault?

     ____

2. For this particular fault, in the context of the System integration testing phase and based on your experience:

   - In which range, according to your experience, is the cost for correcting this fault?

     _______ to _______

   - What would you deem as a most likely cost for correcting this fault?

     ____

3. For this particular fault, in the context of the Value chain testing phase and based on

your experience:

- In which range, according to your experience, is the cost for correcting this fault?

  _______ to _______

- What would you deem as a most likely cost for correcting this fault?

  ____

4. For this particular fault, in the context of the Acceptance testing phase and based on your experience:

   - In which range, according to your experience, is the cost for correcting this fault?

     _______ to _______

   - What would you deem as a most likely cost for correcting this fault?

     ____

5. For this particular fault, in the context of Production (the delivered system – in which case the correction is known as a “krisepatch”) and based on your experience:

   - In which range, according to your experience, is the cost for correcting this fault?

     _______ to _______

   - What would you deem as a most likely cost for correcting this fault?

     ____

6. For this particular fault, and assuming that this fault is corrected as part of a bugfix release rather than in any of the abovementioned development phases, based on your experience:

   - In which range, according to your experience, is the cost for correcting this fault?

     _______ to _______

   - What would you deem as a most likely cost for correcting this fault?

     ____

7. For this particular fault, and assuming that this fault is corrected as part of the NEXT main release rather than in any of the abovementioned development phases, based on your experience:

- In which range, according to your experience, is the cost for correcting this fault?

  _______ to _______

- What would you deem as a most likely cost for correcting this fault?

  ____

Questions regarding the cost of fault detection

8. For this particular fault, in the context of the System testing phase and based on your experience:

   - In which range, according to your experience, is the cost for detecting this fault?

     _______ to _______

   - What would you deem as a most likely cost for detecting this fault?

     ____

9. For this particular fault, in the context of the System integration testing phase and based on your experience:

   - In which range, according to your experience, is the cost for detecting this fault?

     _______ to _______

   - What would you deem as a most likely cost for detecting this fault?

     ____

10. For this particular fault, in the context of the Value chain testing phase and based on your experience:

    - In which range, according to your experience, is the cost for detecting this fault?

      _______ to _______

    - What would you deem as a most likely cost for detecting this fault?

      ____

11. For this particular fault, in the context of the Acceptance testing phase and based on your experience:

- In which range, according to your experience, is the cost for detecting this fault?

  _______ to _______

- What would you deem as a most likely cost for detecting this fault?

  ____

12. For this particular fault, in the context of Production (the delivered system) and based on your experience:

    - In which range, according to your experience, is the cost for detecting this fault?

      _______ to _______

    - What would you deem as a most likely cost for detecting this fault?

      ____

Questions regarding the probability of fault detection

Please check that the above probabilities (for min, max, most-likely values, respectively) sum up to 100%.

13. For this particular fault, in the context of the System testing phase and based on your experience:

    - In which range, according to your experience, is the probability of detecting this fault?

      _______ to _______

    - What would you deem as a most likely probability for detecting this fault?

      ____

14. For this particular fault, in the context of the System integration testing phase and based on your experience:

    - In which range, according to your experience, is the probability of detecting this fault?

      _______ to _______

    - What would you deem as a most likely probability for detecting this fault?

____

15. For this particular fault, in the context of the Value chain testing phase and based on your experience:

    - In which range, according to your experience, is the probability of detecting this fault?

      _______ to _______

    - What would you deem as a most likely probability for detecting this fault?

      ____

16. For this particular fault, in the context of the Acceptance testing phase and based on your experience:

    - In which range, according to your experience, is the probability of detecting this fault?

      _______ to _______

    - What would you deem as a most likely probability for detecting this fault?

      ____

17. For this particular fault, in the context of Production (the delivered system) and based on your experience:

    - In which range, according to your experience, is the probability of detecting this fault?

      _______ to _______

    - What would you deem as a most likely probability for detecting this fault?

      ____

Questions regarding the probability of fault correction

Please check that the probabilities for each question (for most-likely values) sum up to 100%.

18. For this particular fault, in the context of the System testing phase and based on your experience:

    - Assuming the fault is detected in this phase, what is the probability of correcting this fault also in this phase?

_______ to _______

- What would you deem as a most likely probability for correcting this fault?

  ____

- Assuming the fault is detected in this phase, what is the probability of correcting this fault in a bugfix release?

  _______ to _______

- What would you deem as a most likely probability for correcting this fault?

  ____

- Assuming the fault is detected in this phase, what is the probability of correcting this fault in the NEXT main release?

  _______ to _______

- What would you deem as a most likely probability for correcting this fault?

  ____

19. For this particular fault, in the context of the System integration testing phase and based on your experience:

- Assuming the fault is detected in this phase, what is the probability of correcting this fault also in this phase?

  _______ to _______

- What would you deem as a most likely probability for correcting this fault?

  ____

- Assuming the fault is detected in this phase, what is the probability of correcting this fault in a bugfix release?

  _______ to _______

- What would you deem as a most likely probability for correcting this fault?

  ____

- Assuming the fault is detected in this phase, what is the probability of correcting this fault in the NEXT main release?

_______ to _______

- What would you deem as a most likely probability for correcting this fault?

____

20. For this particular fault, in the context of the Value chain testing phase and based on your experience:

- Assuming the fault is detected in this phase, what is the probability of correcting this fault also in this phase?

_______ to _______

- What would you deem as a most likely probability for correcting this fault?

____

- Assuming the fault is detected in this phase, what is the probability of correcting this fault in a bugfix release?

_______ to _______

- What would you deem as a most likely probability for correcting this fault?

____

- Assuming the fault is detected in this phase, what is the probability of correcting this fault in the NEXT main release?

_______ to _______

- What would you deem as a most likely probability for correcting this fault?

____

21. For this particular fault, in the context of the Acceptance testing phase and based on your experience:

- Assuming the fault is detected in this phase, what is the probability of correcting this fault also in this phase?

_______ to _______

- What would you deem as a most likely probability for correcting this fault?

____

- Assuming the fault is detected in this phase, what is the probability of correcting this fault in a bugfix release?

  _______ to _______

- What would you deem as a most likely probability for correcting this fault?

  ____

- Assuming the fault is detected in this phase, what is the probability of correcting this fault in the NEXT main release?

  _______ to _______

- What would you deem as a most likely probability for correcting this fault?

  ____

22. For this particular fault, in the context of Production (the delivered system) and based on your experience:

- Assuming the fault is detected in this phase, what is the probability of correcting this fault also in this phase (that is, a “krisepatch”)?

  _______ to _______

- What would you deem as a most likely probability for correcting this fault?

  ____

- Assuming the fault is detected in this phase, what is the probability of correcting this fault in a bugfix release?

  _______ to _______

- What would you deem as a most likely probability for correcting this fault?

  ____

- Assuming the fault is detected in this phase, what is the probability of correcting this fault in the NEXT main release?

  _______ to _______

- What would you deem as a most likely probability for correcting this fault?

  ____

Questions about the consequences and criticality of the fault

For this particular fault, answer the following questions using the following scale,

1 – Strongly agree 2 – Agree 3 – Not certain 4 – Disagree 5 – Strongly disagree

Question:

1.	To fix this fault would be of the highest possible priority	1 ☐	2 ☐	3 ☐	4 ☐	5 ☐
2.	This fault would be very expensive to fix if it were not discovered before delivery	1 ☐	2 ☐	3 ☐	4 ☐	5 ☐
3.	This fault would have caused system failures (e.g., data inconsistencies) in the production system and would have had a critical negative impact on the correct operation of the COS system	1 ☐	2 ☐	3 ☐	4 ☐	5 ☐
4.	This fault would have had a critical negative impact on the stability of the COS system, and would thus have caused system crashes in the production system	1 ☐	2 ☐	3 ☐	4 ☐	5 ☐
5.	System failures (in the production system) caused by this fault would result in substantial costs for the users	1 ☐	2 ☐	3 ☐	4 ☐	5 ☐
6.	System failures (in the production system) caused by this fault would result in substantial costs for the development team	1 ☐	2 ☐	3 ☐	4 ☐	5 ☐

Appendix E: Interview procedure

- Explain the motivation for doing this (read letter of motivation)
- Explain how the CE calculations will work, as discuss potential sources of bias with the interviewee
- For each interviewee, ask first questions regarding those faults that this person found. Then ask the questions for the faults that some of the other developers found (we assume that they have sufficient knowledge of each fault to give reasonable estimates).

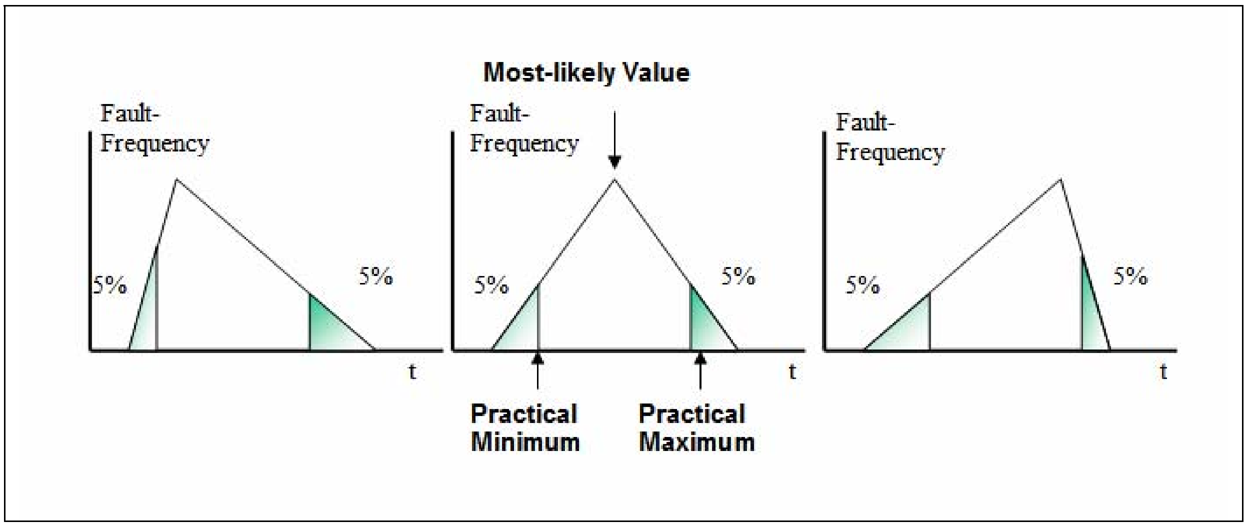
For each fault:
- Present the fault report, coverage report etc to the developer. Let them study it to refresh their memory.
- Read introduction of question.

For questions regarding the fault detection costs:
- Ask what are the typical verification activities that would be performed when the given fault being considered is detected in each of the phases (Integration testing, System testing, System Integration testing, Acceptance Testing, Production). For example, if a fault is detected in the production system (e.g., by causing a failure discovered by a user), there might be a user inquiry to a technical support help desk, who record the problem, then someone determines whether this is a fault, classifies it, etc, etc. Remind the interviewee about other activities that were mentioned in previous interviews or by the test manager, if any. Make a note of the type of activities mentioned.
- Emphasize that we are interested in practical minimum and maximum values. Explain these values by means of the visualization of the response mode.
- Ask: From your experience, in which practical situations would the detection effort be very high? What would be then a practical maximum value for the correction effort of the current fault?
- Ask: From your experience, in which practical situations would the detection effort be very low? What would be then a practical minimum value for the detection effort of the current fault?
- Ask: Is the most likely value closer to the minimum or the maximum? What would you deem to be the most likely value for the current fault?

For questions regarding the fault correction costs:
- Ask what are the typical activities that would be performed when the given fault is corrected in each of the phases (Integration testing, System testing, System Integration testing, Acceptance Testing, Production). For example, is a CR created, how is it allocated to developers, what quality assurance activities take place, etc. Remind the interviewee about other activities that were mentioned in previous interviews or by the test manager, if any. Make a note of the type of activities mentioned.
- Emphasize that we are interested in practical minimum and maximum values. Explain these values by means of the visualization of the response mode.
- Ask: From your experience, in which practical situations would the correction effort be very high? What would be then a practical maximum value for the correction effort of the current fault?
- Ask: From your experience, in which practical situations would the correction effort be very low? What would be then a practical minimum value for the correction effort of the current fault?
- Ask: Is the most likely value closer to the minimum or the maximum? What would you deem to be the most likely value for the current fault?

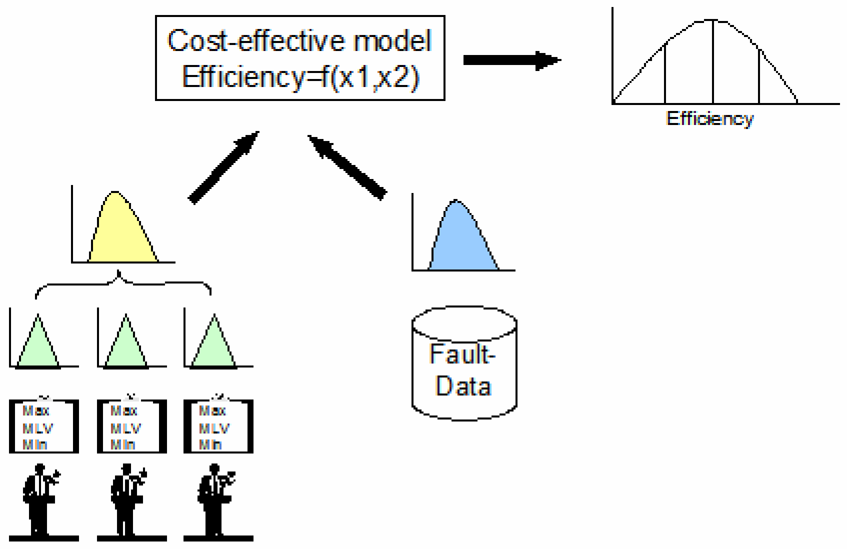
For questions regarding fault probabilities:
- Ask what are the factors that would affect the likelihood of detecting a fault in each of the phases (Integration testing, System testing, System Integration testing, Acceptance Testing, Production).

Appendix F: Visual Aid 1



Appendix G: Visual Aid 2

How do we work with the data?



Appendix H: Quantitative results from the additional unit testing

CLASS	FAULT PROB.	JUSTIFICATION	FAULTS FOUND	COVERAGE BEFORE	COVERAGE AFTER	EFFORT SPENT	% INCREASE IN TEST-COVERAGE
AgreementContainerConverter	0.929		1. A nullPointerException is generated because of the use of the wrong API. This is corrected in a CR.	Conditionals: 62.5 % Statements: 80 % Methods: 85.7 % Total: 77.8 %	Conditionals: 100 % Statements: 100 % Methods: 100 % Total: 100 %	3 hrs.	22.2 %
CleanupFilesMDBean	0.929	Not tested due to huge workload developing mocks to test trivial stuff. Good coverage initially.				2 hrs.	
FileResult	0.929	DataHolder without logic and full coverage.					
AgreementDtoStatus	0.929	DTO, no logic to test.					
AgreementMemberDtoStatus	0.929	DTO, no logic to test.					
ChordiantUtil	0.929	DataHolder without logic to test.					
ExternalTransactionInitialData	0.929	DataHolder without logic and full coverage.					
ProductOfferRuleAssembler	0.929		1. Found due to better unit test coverage. Also found during system test.	Conditionals: 41.7 % Statements: 51.3 % Methods: 50 % Total: 48.4 %	Conditionals: 100 % Statements: 100 % Methods: 100 % Total: 100 %	8.5 hrs.	51.6 %
ModifiableAgreement	0.929	Two exceptions are not tested. To test these would only involve testing the mocking. Not interresting.		Conditionals: 92.9 % Statements: 82.9 % Methods: 100 % Total: 87.3 %	Conditionals: 100 % Statements: 94.3 % Methods: 100 % Total: 96.4 %	2 hrs.	9.1 %
ProductOfferSenderBase	0.929	Test class, shall not be tested					
DealerInfoDto	0.929	DTO, no logic to test					
DealerIdDto	0.929	DTO, no logic to test.					
AgreementContainer	0.929	DataHolder without logic to test.					
ExternalTransactionRepository	0.9	Interface					
ExternalTransactionLogException	0.9	Exception, no logic					
ExtrenalTransactionPaymentException	0.9	Exception, no logic					
ExternalTransactionValidationException	0.9	Exception, no logic					
ExternalTransactionException	0.9	Exception, no logic					
CommissionReportService	0.9	Interface					
OrderInformation	0.871	Did not have time to test more.		Conditionals: 44.9 % Statements: 56.6 % Methods: 61.5 % Total: 53.8 %	Conditionals: 64 % Statements: 75.1 % Methods: 85.2 % Total: 72.8 %	7 hrs.	19 %

ProductOfferValidator	0.871	Test coverage was initially alright (91%). Added a few extra tests, but after a while I got problems with the making of the setup to test some of the exception handling. The class is, in my opinion, too complex at the moment and has a lot to do. Very hard to mock everything. It is possible that some of the exception handling no longer can happen in production.		Conditionals: 87.5 % Statements: 91.6 % Methods: 100 % Total: 91 %	Conditionals: 91.7 % Statements: 94 % Methods: 100 % Total: 93.7 %	4 hrs.	2.7 %
AgreementRepository	0.871	This class was not tested due to the fact that others were going to do large changes to it at the same time the extensive unit testing where done. The people who did changes to the class did not generate any reports before this.					
ProductOfferServiceBean	0.871	Good unit test coverage, only trivial code / empty methods lacking tests.					
EurekaServiceBean	0.871	Huge bean, which's only logic (with a couple of exceptions), is situated in converter classes. For that reason I chose to look into the converter classes (with same fault probability) and skip this class.		Conditionals: 39.7 % Statements: 22.9 % Methods: 25.7 % Total: 24.5 %	Conditionals: 39.7 % Statements: 22.9 % Methods: 25.7 % Total: 24.5 %	3 hrs.	0 %
StructToDtoConverter	0.871			Conditionals: 67.4 % Statements: 82.2 % Methods: 88.2 % Total: 77.7 %	Conditionals: 68.9 % Statements: 83.2 % Methods: 88.2 % Total: 78.8 %	4 hrs.	1.1 %
DtoToStructConverter	0.871		1. A conversion error 4. Minor null-value handling issues This class and the one above (DtoToStructConverter) were first put together in one report. This was due to the fact that the	Conditionals: 80.9 % Statements: 89.8 % Methods: 89.8 % Total: 87.3 %	Conditionals: 85.3 % Statements: 92.9 % Methods: 92 % Total: 90.9 %	16 hrs.	3.6 %

			somewhat belong together. I made a generic class called ConverterTestHelper, that is to be used for all tests of converting (This class is at the moment, as in 3 weeks after the extended unit test phase, being used other places as well. Because this new class uses reflection it will also catch up on future faults that will be introduced in convertings.				

Appendix I: Cost-effective calculations per fault per developer

Name	Dev. 2																								
Exp. COS:	6																								
Exp. Telenor:	6																								
Phase x	PDx,i			CDx,i			PCbf,x,i			CCbf,i			PCnmr,x,i			CCnmr,i			PCx,i			CCx,i			
	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	
S. test	1	1	1	3	6	9	0	0	0	16	18	20	0	0	0	16	18	20	1	1	1	9	10	15	15.2
S. I. test	0	0.1	0	3	6	9	0	0	0	16	18	20	0	0	0	16	18	20	1	1	1	9	10	15	0.8
V. C. test	0	0	0	6	9	12	0	0	0	16	18	20	0	0	0	16	18	20	1	1	1	9	10	15	0
A. test	0	0	0	6	9	12	1	1	1	16	18	20	0	0	0	16	18	20	0	0	0	12	15	18	0
Prod.	0	0	0	9	13	14	1	1	0.7	16	18	20	0.2	0.2	0.3	16	18	20	0.15	0.2	0.25	16	18	20	0
Total Cost Saved																									**16**

Fault #1 CE for developer 1

Name	Dev. 2																								
Exp. COS:	6																								
Exp. Telenor:	6																								
Phase x	PDx,i			CDx,i			PCbf,x,i			CCbf,i			PCnmr,x,i			CCnmr,i			PCx,i			CCx,i			
	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	
S. test	1	1	1	1	2	3	0	0	0	3	4	8	0	0	0	1	2	4	1	1	1	1	2	4	4
S. I. test	0	0	0	2	3	4	0	0	0	3	4	8	0	0	0	1	2	4	1	1	1	1	2	4	0
V. C. test	0	0	0	2	3	4	0	0	0	3	4	8	0	0	0	1	2	4	1	1	1	3	4	8	0
A. test	0	0	0	2	3	4	0	0	0	3	4	8	0	0	0	1	2	4	1	1	1	3	4	8	0
Prod.	0	0	0	8	10	16	0	0	0	3	4	8	0	0	0	1	2	4	1	1	1	6	8	16	0
Total Cost Saved																									**4**

Fault #2 for developer 1

Name	Dev. 2																								
Exp. COS:	6																								
Exp. Telenor:	6																								
Phase x	PDx,i			CDx,i			PCbf,x,i			CCbf,i			PCnmr,x,i			CCnmr,i			PCx,i			CCx,i			
	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	
S. test	0.5	0.6	0.9	2	4	6	0	0	0	4	6	8	0	0	0	4	6	8	1	1	1	4	6	8	6
S. I. test	0.1	0.15	0.2	2	4	6	0	0	0	4	6	8	0	0	0	4	6	8	1	1	1	4	6	8	1.5
V. C. test	0.1	0.15	0.2	4	6	8	0	0	0	4	6	8	0	0	0	4	6	8	1	1	1	6	8	10	2.1
A. test	0	0.05	0.15	4	6	8	0	0	0	4	6	8	0	0	0	4	6	8	1	1	1	6	8	10	0.7
Prod.	0	0.05	0.15	4	6	8	0	0	0.2	4	6	8	0	0	0	4	6	8	0.8	0.9	1	14	16	20	1.05
Total Cost Saved																									**11.35**

Fault #3 for developer 1

Name	Dev. 2																								
Exp. COS:	6																								
Exp. Telenor:	6																								
Phase x	PDx,i			CDx,i			PCbf,x,i			CCbf,i			PCnmr,x,i			CCnmr,i			PCx,i			CCx,i			
	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	
S. test	0	0.01	0.1	3	3	6	0	0.04	0.1	4	5	8	0	0	0	4	5	8	0.9	0.96	1	8	8	16	0.1088
S. I. test	0.8	0.85	0.9	4	5	10	0	0.04	0.1	4	5	8	0	0	0	4	5	8	0.9	0.96	1	8	8	16	10.948
V. C. test	0.06	0.08	0.1	5	6	10	0	0.04	0.1	4	5	8	0	0	0	4	5	8	0.9	0.96	1	8	10	16	1.264
A. test	0.03	0.04	0.1	5	6	10	0	0.04	0.1	4	5	8	0	0	0	4	5	8	0.9	0.96	1	9	10	18	0.632
Prod.	0	0.02	0	5	6	10	1	1	1	4	5	8	0	0	0	4	5	8	0	0	0	16	18	24	0.22
Total Cost Saved																									**13.1728**

Fault #4 for developer 1

Name	Dev. 2																								
Exp. COS:	6																								
Exp. Telenor:	6																								
Phase x	PDx,i			CDx,i			PCbf,x,i			CCbf,i			PCnmr,x,i			CCnmr,i			PCx,i			CCx,i			
	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	
S. test	1	1	1	3	6	9	0	0	0	16	18	20	0	0	0	16	18	20	1	1	1	9	10	15	15.2
S. I. test	0	0.1	0	3	6	9	0	0	0	16	18	20	0	0	0	16	18	20	1	1	1	9	10	15	0.8
V. C. test	0	0	0	6	9	12	0	0	0	16	18	20	0	0	0	16	18	20	1	1	1	9	10	15	0
A. test	0	0	0	6	9	12	1	1	1	16	18	20	0	0	0	16	18	20	0	0	0	12	15	18	0
Prod.	0	0	0	9	13	14	1	1	0.7	16	18	20	0.2	0.2	0.3	16	18	20	0.15	0.2	0.25	16	18	20	0
Total Cost Saved																									**14.4**

Fault #1 CE for developer 2

Name	Dev. 1																								
Exp. COS:	2																								
Exp. Telenor:	6																								
Phase x	PDx,i		CDx,i				PCbf,x,i			CCbf,i			PCnmr,x,i			CCnmr,i			PCx,i			CCx,i			
	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	
S. test	1	1	1	2	3	4	0	0	0	16	20	24	0	0	0	4	4	8	1	1	1	4	4	8	7
S. I. test	0	0	0	2	3	4	0	0	0	16	20	24	0	0	0	4	4	8	1	1	1	4	4	8	0
V. C. test	0	0	0	4	6	8	0	0	0	16	20	24	0	0	0	4	4	8	1	1	1	4	4	8	0
A. test	0	0	0	4	6	8	0	0	0	16	20	24	0	0	0	4	4	8	1	1	1	16	20	24	0
Prod.	0	0	0	8	10	12	0	0	0.2	16	20	24	0	0	0	4	4	8	0.8	0.9	1	16	20	24	0
Total Cost Saved																									**7**

Fault #2 CE for developer 2

Name	Dev. 1
Exp. COS:	2
Exp. Telenor:	6

Phase x	PDx,i			CDx,i			PCbf,x,i			CCbf,i			PCnmr,x,i			CCnmr,i			PCx,i			CCx,i			
	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	
S. test	0.04	0.05	0.06	10	14	18	0	0	0	26	28	34	0	0	0	3	3	4	1	1	1	3	3	4	0.85
S. I. test	0.04	0.05	0.06	18	22	26	0	0	0	26	28	34	0	0	0	3	3	4	1	1	1	6	6	8	1.4
V. C. test	0.04	0.05	0.06	18	22	26	0	0	0	26	28	34	0	0	0	3	3	4	1	1	1	6	6	8	1.4
A. test	0.04	0.05	0.06	24	28	32	0	0	0	26	28	34	0	0	0	3	3	4	1	1	1	26	28	34	2.8
Prod.	0.75	0.8	0.95	24	28	32	0.08	0.1	0.12	26	28	34	0.08	0.1	0.12	3	3	4	0.7	0.8	0.9	26	28	34	42.8
Total Cost Saved																									**49.25**

Fault #3 CE for developer 2

Name	Dev. 1
Exp. COS:	2
Exp. Telenor:	6

Phase x	PDx,i			CDx,i			PCbf,x,i			CCbf,i			PCnmr,x,i			CCnmr,i			PCx,i			CCx,i			
	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	
S. test	0.04	0.05	0.06	8	12	16	0	0	0	24	27	32	0	0	0	1	1	2	1	1	1	1	1	2	0.65
S. I. test	0.04	0.05	0.06	16	20	24	0	0	0	24	27	32	0	0	0	1	1	2	1	1	1	8	8	12	1.4
V. C. test	0.04	0.05	0.06	16	20	24	0	0	0	24	27	32	0	0	0	1	1	2	1	1	1	8	8	12	1.4
A. test	0.04	0.05	0.06	24	28	32	0	0	0	24	27	32	0	0	0	1	1	2	1	1	1	24	27	32	2.75
Prod.	0.75	0.8	0.95	24	28	32	0.08	0.1	0.12	24	27	32	0.08	0.1	0.12	1	1	2	0.7	0.8	0.9	24	27	32	41.92
Total Cost Saved																									**48.12**

Fault #4 CE for developer 2

Bibliography

[Arisholm et al., 2007]
E. Arisholm, L.C. Briand and M.J. Fuglerud, *“Data Mining Techniques for Building Fault-proneness Models in Telecom Java Software”*, Submitted to International Symposium in Software Reliability Engineering (ISSRE), 2007.

[Arisholm et al., 2005]
E. Arisholm, H. C. Benestad, D. Sjøberg,
“How to Recruit Professionals as Subjects in Software Engineering Experiments”, Information Systems Research in Scandinavia (Kristiansand, Norway, 2005).

[Briand & Wüst, 2002]
L. C. Briand and J. Wüst, *"Empirical Studies of Quality Models in Object-Oriented Systems",* Advances in Computers, vol. 59, pp. 97-166, 2002.

[Meyer & Booker, 1991]
M.A. Meyer and J.M. Booker, *“Eliciting and Analyzing Expert Judgment: A Practical Guide”*, Academic Press, Ltd., 1991.

[Ostrand et al., 2005]
T. J. Ostrand, E. J. Weyuker, and R. M. Bell, *"Predicting the Location and Number of Faults in Large Software Systems",* IEEE *Transactions on Software Engineering* vol. 31, no. 4, pp. 340-355, 2005.

[Ostrand et al., #1, 2004]
T.J. Ostrand, E.J. Weyuker, R.M. Bell, *“Where the bugs are”*, International Symposium on Software Testing and Analysis, 2004.

[Ostrand & Weyuker #2, 2004]
T.J. Ostrand, E.J. Weyuker, *“A Tool for Mining Defect-Tracking Systems to Predict Fault-Prone Files”*, Proc. of Int. Workshop on Mining Software Repositories, 2004.

[Ostrand & Weyuker #1, 2002]
T. Ostrand, E. Weyuker *“The distribution of faults in a large industrial software system”*, ACM SIGSOFT Software Engineering Notes, 2002.

[Hall & Holmes, 2003]
M. A. Hall and G. Holmes, *"Benchmarking Attribute Selection Techniques for Discrete Class Data Mining"*, IEEE transactions on knowledge and data engineering vol. 15, no. 6, pp. 14-37, 2003.

[Hall, 2000]
M. Hall, *"Correlation-based Feature Selection for Discrete and Numeric Class Machine Learning"* Proc. Seventeenth Int. Conf. on Machine Learning, pp. 359-366, 2000.

[Freund & Schapire, 1995]
Y. Freund and R. Schapire, *"A Decision-Theoretic Generalization of on-Line Learning*

and an Application to Boosting", Proc. European Conference on Computational Learning Theory, 1995.

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

[Khoshgoftaar et al., 1998]
Khoshgoftaar, T.M. Allen, E.B. Naik, A. Jones, W.D. Hudepohl, J., "*Using Classification Trees for Software Quality Models: Lessons Learned"*, High-Assurance Systems Engineering Symposium, 1998. Proceedings. Third IEEE International, 1998.

[Khoshgoftaar et al., 1996]
Khoshgoftaar, T.M. Allen, E.B. Kalaichelvan, K.S. Goel, N. *"Early quality prediction: a case study in telecommunications".* Software, IEEE, 1996.

[Bakkelund et al., 2005]
D. Bakkelund, K. Kvam, R. Lie, *"Legacy System Exorcism by Pareto's Principle"*, Conference on Object-Oriented Programming Systems Languages and Applications, Companion to the 20th annual ACM SIGPLAN Conference on Object-Oriented programming, systems, languages and applications, p.250-256 (2005).

[Bakkelund & Kvam, 2004]
D. Bakkelund, K. Kvam, R. Lie, *"Cynical Reengineering"*, Presented at the XP 2004 Conference (2004)

[Denaro & Pezzê, 2002]
G. Denaro, M. Pezzê, *"An Empirical Evaluation of Fault-Proneness Models"*, International Conference on Software Engineering, 2002.

[Denaro et al., 1994]
G. Denaro, M. Pezzè and S. Morasca,"*Towards Industrial Relevant Fault-Proneness Models"*, International Journal of Software Engineering and Knowledge Engineering, 1994.

[Gill & Kemerer, 1991]
G. Gill and C. Kemerer. *"Cyclomatic complexity density and software maintenance productivity",* IEEE Transactions on Software Engineering, 17(12):1284–1288, December 1991.

[Witten & Frank, 2005]
Witten and E. Frank, *"Data Mining: Practical Machine Learning Tools and Techniques",* Second edition: Morgan Kaufman, 2005.

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

[Werbos, 1994]
P. Werbos, *"The Roots of Backpropagation: From Ordered Derivatives to Neural*

Networks and Political Forecasting", Wiley, 1994.

[Vapnik, 1995]
V. N. Vapnik, *"The Nature of Statistical Learning Theory"*, Springer, 1995.

[Joachims, 2002]
T. Joachims, *"Learning to Classify Text Using Support Vector Machines",* 2002.

[Shipp et al., 2002]
M. A. Shipp, K. N. Ross, P. Tamayo, A. P. Weng, J.L. Kutok, R. C. Aguiar, M. Gaasenbeek, M. Angelo, M.Reich, G. S. Pinkus, T. S. Ray, M. A. Koval, K. W. Last, A. Norton, T. A. Lister, J. Mesirov, D. S. Neuberg, E. S. Lander, J. C.Aster, and T. R.Golub, *"Diffuse large B-cell lymphoma outcome prediction by gene expression profiling and supervised machine learning,"* Nat. Med., vol. 8, no. 1, pp. 68-74, 2002.

[Melville & Mooney, 2005]
R. Melville and R. Mooney, *"Creating Diversity in Ensembles using Artificial data",* Information Fusion, vol. 6, no. 1, pp. 99-111, 2005.

[Kononenko, 1995]
Kononenko, *"On Biases in Estimating Multivalued Attributes"*, Proc. fourteenth Int. Joint conf. on Artificial Intelligence, pp. 495-502, 1995.

[Mockus & Weiss, 2000]
Mockus, Audris; Weiss, David M, *"Predicting risk of software changes",*
BELL LABS TECH J. Vol. 5, no. 2, pp. 169-180. Apr. 2000.

[Tahat *et al.,* 2001]
L. Tahat, B. Vaysburg, B. Korel, and A. Bader,*"Requirement-Based Automated Black-Box Test Generation",* 25th Annual Int'l Computer Software and Applications Conference, Chicago, Illinois, 2001.

[Slaugther et al., 1998]
Sandra A. Slaugther, Donald E. Harter, and Mayuram S. Krishnan, *"Evaluating the Cost of Software Quality"*, Communications of the ACM 41(8), 1998.

[Haney, 1972]
F.M. Haney, *"Module Connection Analysis—A Tool for Scheduling of Software Debugging Activities"* Proc. AFIPS Fall Joint Computer Conf. pp. 173-179, 1972.

[Basili & Hutchens, 1983]
V. Basili and D. Hutchens. *"An empirical study of a syntactic complexity family"*. IEEE Transactions on Software Engineering, 9(6):664–672, November 1983. Special Section on Software Metrics.

[Frankl & Lakounenko, 1998]
P. Frankl and O. Lakounenko,*"Further empirical studies of test effectiveness"*. ACMSIGSOFT Software Engineering Notes, 23(6):153–162, November 1998. Proceedings of the ACM SIGSOFT Sixth International Symposium on the

Foundations of Software Engineering.

[B.Henderson-Sellers, 1995]
B.Henderson-Sellers. *“Object-Oriented Metrics, Measures of Complexity”.* Prentice Hall, 1995.

[Bullseye]
http://www.bullseye.com/coverage.html

[MKS]
The MKS software (created by the company MKS, previously Mortice Kern System)
http://www.mks.com

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