Assessing the Effectiveness of Fault-Proneness Prediction Models Across Software Systems

Roy de Wildt
roydewildt@gmail.com

August 10, 2016, 85 pages

Supervisors: Jurgen Vinju
Rinse van Hees
Host organisation: Info Support B.V.

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Contents

Abstract 4

1 Introduction 5
1.1 Problem Statement 5
1.2 Initial Study 6
1.3 Literature Overview 7
1.4 Research Questions 8
1.5 Contributions 9
1.6 Recommendations 10
1.7 Outline 10

2 Background 12
2.1 Fault Detection 12
2.1.1 Issue-trackers 12
2.1.2 Selection bias 12
2.2 Fault Distribution 12
2.2.1 Pareto principle 12
2.2.2 Gini Coefficient 13
2.3 Software Measurement 14
2.3.1 Chidamber-Kemerer metrics 14
2.3.2 Li-Henry metrics 15
2.3.3 Briand et al. coupling metrics 16
2.3.4 Benlarbi-Melo polymorphism metrics 17
2.3.5 Khoshgoftaar reuse metrics 18
2.3.6 Relationship between product metrics and fault-proneness 18
2.4 Principle Component Analysis 19
2.4.1 Rotations and loadings 20
2.5 Logistic Regression Models 20
2.5.1 Model comparison 21
2.5.2 Stepwise selection 21
2.5.3 Model measures 21
2.5.4 Model validation 22
2.6 Outlier Detection 23
2.6.1 Mahalanobis distance 23

3 Fault Distributions in Software Systems and the Pareto Principle 24
3.1 Introduction 24
3.2 Description of Study Setting 25
3.2.1 Systems 25
3.2.2 Measurement instruments 26
3.2.3 Variables 27
3.3 Data Analysis Methodology 27
3.4 Analysis Results 28
3.5 Conclusion and Discussion 28

3.5.1 Threats to validity 29
3.5.2 Future research 29

4 Reassessing the Applicability of Fault-Proneness Prediction Models Across Software Systems 31
4.1 Introduction 31
4.2 Description of Study Setting 32
4.2.1 Systems 32
4.2.2 Variables 33
4.3 Data Analysis Methodology 34
4.3.1 Descriptive statistics 34
4.3.2 Outlier analysis 34
4.3.3 Principle component analysis 34
4.3.4 Prediction model construction 35
4.3.5 Model evaluation 36
4.3.6 Hypothesis testing 36
4.4 Analysis Results 37
4.4.1 Descriptive statistics 37
4.4.2 Outlier Analysis 37
4.4.3 Principal component analysis 37
4.4.4 Prediction model construction 38
4.4.5 Model validation 39
4.4.6 Hypothesis testing 40
4.5 Conclusion and Discussion 41
4.5.1 Threats to validity 42
4.5.2 Future research 43

5 Improvements to Regression-Based Fault-Proneness Prediction Models 44
5.1 Introduction 44
5.2 Description of Study Setting 46
5.2.1 Systems 46
5.2.2 Variables 46
5.3 Data Analysis Methodology 47
5.3.1 Descriptive Statistics 47
5.3.2 Outlier analysis 47
5.3.3 Prediction model construction 47
5.3.4 Model evaluation 47
5.3.5 Hypothesis testing 47
5.4 Analysis Results 48
5.4.1 Descriptive Statistics 48
5.4.2 Outlier analysis 48
5.4.3 Prediction model construction 48
5.4.4 Model evaluation 49
5.4.5 Hypothesis testing 50
5.5 Conclusion and Discussion 52
5.5.1 Threats to validity 53
5.5.2 Future research 53

6 The Influence of Environmental Factors on Fault-Proneness Prediction Models 54
6.1 Introduction 54
6.2 Description of Study Setting 55
6.2.1 Systems 55
6.2.2 Variables 57
6.3 Data Analysis Methodology 58
6.3.1 Model construction 58

6.3.2 Model validation . . . 58
6.3.3 Hypothesis testing . . . 58
6.4 Analysis Results . . . 59
6.4.1 Hypothesis testing . . . 61
6.5 Conclusion and Discussion . . . 62
6.5.1 Threats to validity . . . 63
6.5.2 Future research . . . 63

Bibliography **64**

A Tooling **67**
A.1 File Selection . . . 67
A.2 Lines of Code Counter . . . 67
A.3 Git Crawler . . . 68
A.4 Byte-Code Metric Suite . . . 69
A.5 Fault Prediction Metric Suite . . . 70
A.6 System Overview Measures . . . 73

B Preliminary Study: Fault Distribution Histograms **76**

C Replication Study: Principal Component Analyses **78**

D Factor Study: Pools **81**

E Factor Study: Selected Variables **83**

Abstract

Fault-proneness prediction models uses techniques from machine learning and statistics in order to indicate parts of the software system that are likely to be fault-prone. These models can be used to improve the fault discovery strategy by pointing out fault-prone parts of the software system. If accurate enough, they could be applied in the industry to reduce the resources needed for reviewing or testing software systems. A number of researchers succeeded in building accurate fault-proneness prediction models which were trained and validated using a single-system. However, there is little known about the effect on fault-proneness prediction model accuracy when applied across systems; let alone the factors that influence the model's accuracy.

In this study, we assess the effectiveness of regression-based fault-proneness prediction models applied across systems. Three axes are proposes along which fault-proneness prediction models could be improved: (i) By changing the included fault-proneness predictors; (ii) by tuning or altering fault-proneness prediction modelling techniques; (iii) or by considering the context in which the fault--proneness prediction model operates. 13 commercial systems active in the financial sector were used to test the effect of these improvements.

Based on our discoveries, it seems to be possible to build effective fault-proneness prediction models that could be applied across systems; our best cross-system fault-proneness prediction models obtained on average an accuracy of 92%. However, building stable fault-proneness prediction models is far from straightforward, there is still little known about which factors influence the prediction models and which predictors actually correlates with class fault-proneness. In our opinion, we do not think that (regression-based) fault-proneness prediction could effectively applied in practice at this point.

Chapter 1

Introduction

1.1 Problem Statement

The claim that a large part of the faults reside in a small part of a software system [1, 2] triggered many researchers to find ways to discover parts of the system that are likely to contain faults. A number of these researchers focused on building fault-proneness prediction models that are able to separate fault-prone modules from non fault-prone modules [3]. These models can be used to improve the fault discovery strategy by pointing out the modules that are likely to contain a larger part of the faults. If accurate enough, they could be applied in the industry to reduce the resources needed for reviewing or testing software systems.

A number of studies claimed they build highly accurate fault-proneness prediction models, using regression analysis techniques and product measures, that find between the 70% and 90% of all the fault-prone modules with an accuracy around 80% [4, 5, 6, 7, 8]. However, these models are build and verified on the same software system, assuming the fault-proneness prediction model will only be used on that particular system. This assumption affects the quality and the generalisability of the model: (i) The quality is affected by the limited data a single system is able to provide for training the model. If there is too little data available for the model to be trained with, it will influence the prediction accuracy negatively by overgeneralising the training data. As a result, the current system must be mature enough before any prediction model can be used with reasonable accuracy, and even then the data is still limited to a single system. (ii) The generalisability is affected by only considering the current system in the validation process. The validation of the prediction model will only provide insights in the accuracy of the model regarding that particular system, but it tells nothing about the performance of the model on other systems. As a result, the model is tied to that system and cannot be used with confidence on other systems. Also, because the model needs to be rebuild for each system, the model is unable to evolve over time.

The latter two limitations could be resolved by using fault-proneness prediction models that are trained and validated using multiple systems. By using more than one mature system to build the fault-proneness prediction model, it should solve the constraint on the training data and makes it possible to apply the model new or small software systems. Moreover, if the factors that influence the quality of a prediction model are known and can could be controlled, that model could be reused on similar systems with reasonable accuracy (the ratio of true positives and negatives to all predictions). However, there is little known about the effect on the model's accuracy when applied across systems; let alone the factors that influence the model's accuracy (e.g. does team composition influences prediction accuracy).

A realistic condition when applying fault-proneness prediction models in practice is that they could be used across systems and trained using multiple-systems datasets. If not, models will be usable on a single system of substantial size. If it is the case that fault-proneness prediction models are incapable of predicting accurately across systems, it will have serious ramifications for the utility of fault-proneness prediction models in practice.

1.2 Initial Study

Briand et al. [4] stated that previous studies on fault-proneness prediction modelling "*can be characterized as feasibility studies*", this is because these studies were not applied under "*realistic conditions*". In their opinion "*the purpose of building such [fault-proneness prediction] models is to apply them to other systems*", for this reason they assessed the applicability of fault-proneness models across object-oriented software projects.

Briand et al. build a fault-proneness prediction model based on a medium size Java system and applied the model to a different Java system developed by the same team (only different project manager), using similar technologies (OO-design, and Java), and in a similar environment. The systems varied in coding standards and design strategies. The two systems used were *Xpose* and *Jwriter*. *Xpose* is an application for displaying and showing XML documents and has 144 Java classes with a total of 1.774 methods. Although this application was developed after *Jwriter*, it was used to build the model because it was the the larger of the two systems. *Jwriter* is a component that provides basic word processing capabilities and has 68 classes with a total of 933 methods. This component was used to validate the model on.

The model of Briand et al. used a set of measurements extracted from a static analyser, called Jmetrics [9]. The static analyser measured a subset of coupling measures described in [10, 11], a set of measures related to polymorphism [12], a subset of of the Chidamber & Kemerer OO metric-suite [13], and some simple size measures based on counts of class methods and attributes. The data about faults found in the field by customers were collected and used as input for the prediction model, and to verify the model's prediction accuracy.

For the analysis of the data, a subset of the measurements was used. The metrics in this subset were selected using a mixed stepwise selection procedure. Also, a Principal Component Analysis (PCA) was used to find groups of measurements that measure the same underlying concept. Briand et al. conducted three regression analyses: an univariate regression analysis, and two multivariate analysis:

- The univariate regression analysis was used for each individual measure against the dependent variable – fault/no-fault to detect if the measure is an useful predictor of fault proneness.
- A logical regression analysis with raw metrics as its input and the assumption of a linear relationship with the dependent variable was used to select a subset of variables that tend to explain the fault-proneness for that system.
- A logical regression analysis in combination with the MARS basic functions, assuming a more complex relationship between the independent variables and the dependent variable, was used to create a composite function of measurements that tend to explain the fault-proneness for that system best.

For a more detailed description of the data analysis methodology, Briand et al. refers to a previous study [14].

The fault-proneness prediction models were validated in terms of precision, percent of correctly classified faulty classes out all faulty classes identified by the model; and recall, percent of correctly classified faulty classes out all faulty classes in the system. The models were evaluated using v-cross-validation and cross-system validation techniques.

The results indicate that "*a model built on one system can be accurately used to rank classes within another system according to their fault proneness*" but that "*applying the models across systems is far from straightforward*". Their two multivariate regression fault-proneness prediction models had "*completeness and correctness values of about 60% for both models*". To obtain these values they had to adjusted the cut-off values to 0.22 for the linear model and 0.06 for the MARS model. Briand et al concluded that both the multivariate models perform better than chance and models based on size measures. They speculated that changes in the distribution of measures and system factors (e.g. experience, design method) affects fault-proneness prediction when used between systems.

This thesis continues the work done by Briand et al. on cross-system fault-proneness prediction models. We will replicate their study and verify the results. Finally, new research goals will be added based on the outcomes of the replication study.

1.3 Literature Overview

Various methods are used in research for building fault-proneness predictions models [3], including: genetic programming [15], neutral networks [16], case-based reasoning [17], fuzzy logic [18], Dempster-Shafer networks [19], decision trees [20], Naive Bayes [21], and regression analysis [5, 6, 22, 23, 7, 24, 8].

In order to replicate the study done by Briand et al. [4], we focus regression-based fault-proneness prediction models. The following paragraphs present the work of other researchers on logistic regression based fault prediction. An overview of the literature can be found in Table 1.1. The table presents the study, the precision and recall of their best performing prediction model, and the predictors used by the model. We refer to Chapter 2 for more information on precision, recall, and predictors.

Denaro et al. [5] used logistic regression to relate software measures to class fault-proneness of homogeneous software products. They also promote the use of cross-validation techniques to validate prediction models that use small datasets. The predictors used in their best prediction model were: *eLOC*, *Comm*, *Lines*, *FP*, *LFC*, *EXEC*. This model was able to find 89% of all faults in the system with an accuracy of 77%. They concluded that it is possible to build statistical models based on historical data for estimating fault proneness of software modules.

Khoshgoftaar et al. [22] developed a fault-proneness prediction model based solely on process-history variables like module age (*Age*), new modules (*IsNew*), and changed modules (*IsChg*). They investigated if a module its history prior to integration could help predict the likelihood of fault discovery during integration and testing. They used logistic regression to build the classification model with a cost-weighted classification rule. Their model found on average 79 % of all faults in the system with an accuracy of 65%. They drew the following conclusions: (i) Modules that had faults in the past are likely to have faults in the future. (ii) Unplanned requirements changes result in faults. (iii) Faults are more likely when code is changed. (iv) Software-quality models can be useful to help target reliability improvement.

Nagappan and Ball [23] presented an empirical approach for early prediction of pre-release defect density based on the defects found using static analysis tools. They showed that there exist a strong positive correlation between static analysis defect density and the pre-release defect density determined by testing; with *static analysis defect density* the number of defects found per KLOC, and *pre-release defect density* as the number of defects per KLOC found by other methods before the component is released. Their model used the input of their in-house fault detection tools, *Prefix* and *Prefast*, as predictors and classified 83% of the components correctly (the model's precision and recall were omitted to protect proprietary information).

Schneidewind [7] investigated logistic regression as a discriminant of software quality. He used Logistic Regression Functions (LRFs) and Boolean Discriminant Functions (BDFs) to predict the probability of the occurrence of discrepancy reports (*drcount*; reports of deviations between requirements and implementation). He used two unrelated systems, one system for validation and another system for application. He concluded that very high quality classification accuracy can be obtained while reducing the inspection cost incurred in achieving high quality. His best model, using a combination of LRFs and BDFs and *C*, *S*, *E1*, *E2*, *N*, *L* as predictors, classified modules with at least a *drcount* of 1 from modules with no *drcount* with an accuracy of 98.75%.

Ostrand and Weyuker [24] summarized their work of ten years of software fault-proneness prediction research. Their model, called the *standard model*, is a negative binomial regression model that predicts faults in a release of a system based on the predictors: *LOC*, *Faults in Release N-1*, *Changes Releases N-1 and N-2*, *Files status*, *File-age*, *File-type*. The model predicted the top 20% of most fault-prone files over releases of nine industrial systems. Averaging over the releases of nine systems, their *standard model* was able to correctly identify files that accounted for between 75% and 93% of the actual defects. They also found that in spite of the differing functionality of the systems, development and testing personnel, corporation that wrote and maintained them, development methodologies, and level of maturity, their standard model always behaved very well (these conclusions were drawn from observations made between releases of a same system). Finally they concluded that Negative Binomial Regression performed better than other models, Recursive Partitioning, Random Forests, and Bayesian Additive Regression Trees.

Munson and Khoshgoftaar [8] used discriminant analysis as tool for detection of fault-prone programs. They used mostly size measures as predictors: *PROCS*, *COM*, *LOC*, *BLNK*, *ELOC*, *VG1*,

VG2, *N1*, N2, *n1*, *N*, *V*, *E*. They state that linear regression models are of limited value for the detection of fault-prone modules. They also investigated multivariate regression analysis techniques and argue that the distribution of faults are heavily skewed in favour of programs that have no or a small number of faults. Instead they used the discriminant analysis which was able to correctly identify 75% of the modules. They concluded the following (i) There is a relationship between program faults and certain orthogonal complexity domains (ii) That predictive models could possibly be used for the determination of program faults and program modifications. (iii) The complexity metrics are strongly intercorrelated which could lead to unreliable predictive quality of models used.

Briand et al. [14] explored the relationship between existing object-oriented coupling, cohesion, and inheritance measures and the probability of fault detection in system classes during testing. They used a dataset consisting out of eight systems developed by students as part of an assignment and found that their best model classified 92% of the classes as fault prone with a precision of 78%. *NOP*, RFC_1_L, *NMI*, *FMMEC*, *NIH-ICP_L*, *CLD* were the predictors used in their fault-proneness prediction model. Beside the size of a class, the frequency of method invocation and the depth of inheritance hierarchies seems to be the main driving factors of fault-proneness.

Table 1.1: Literature overview of regression-based fault-proneness prediction modelling

	Predictor-set	Recall	Precision
Briand et al. [4] †	*NIP*, *OCMIC*, *OCMEC*	±60%	±60%
Briand et al. [14]	RFC_∞, *NOP*, RFC_1_L, *NMI*, *FMMEC*, *NIH-ICP_L*, *CLD*	92%	78%
Ostrand and Weyuker [24]	*LOC*, *Faults in Release N-1*, *Changes Releases N-1 and N-2*, *Files status*, *File-age*, *File-type*	-	75%-93%
Denaro et al. [5]	*eLOC*, *Comm*, *Lines*, *FP*, *LFC*, *EXEC*	89%	77%
Khoshgoftaar et al [22]	*IsNew*, *IsChg*, *Age*	79%	65%
Nagappan and Ball [23]	*PREfix tool*, *PREfast tool*	-	83%
Schneidewind [7]	*C*, *S*, *E1*, *E2*, *N*, *L*	-	99%
Munson and Khoshgoftaar [8]	*PROCS*, *COM*, *LOC*, *BLNK*, *ELOC*, *VG1*, *VG2*, *N1*, N2, *n1*, *N*, *V*, *E*	75%	82%

† initial study (see Section 1.2)

1.4 Research Questions

This thesis continues the work on cross-system fault-proneness prediction models done by Briand et al [4]. The goal of this thesis is to asses the effectiveness of fault-proneness prediction models when used across systems. Formulating this goal lead to the following research question. This question stands central in our research.

> *RQ1: Is it possible to build an effective fault-proneness prediction model that could be applied across systems?*

To answer our main research question (RQ1), we will start by investigating the effectiveness of fault-proneness prediction models in general. Fault-proneness prediction models depend on the assumption that faults are inequality distributed over the system. If this assumption does not hold, then fault-proneness prediction models are not useful because every class in the system is equally fault-prone. The following research question will be used to learn more about fault distributions in software systems and will aid the assessment of the practical effectiveness of fault-proneness prediction models.

> *RQ2: Are faults within a software system unequally distributed over its classes?*

Next, we will replicate the research of Briand et al. [4], and reassess the applicability of fault proneness models across software projects. The initial study is described in Section 1.2.

RQ3: Can fault-proneness prediction models effectively be used across systems developed by the same team?

In our replication study, we proposed three axes along which we think cross-system fault-proneness prediction models could be improved. Improvements along two of the axes requires optimizations to the prediction model itself. The goal of this study is to alter the model construction method as proposed by Briand et al., in order to increase the accuracy of the current regression-based fault-proneness prediction models.

RQ4: Can Briand et al. their cross-system fault-proneness prediction model be improved?

Finally, we will investigate which factors influence fault-proneness prediction models when they are applied across systems. Briand et al. suspects that system factors like experience and design method affect the prediction models. But other factors could be thought of like team composition, process protocols and standards, and system type and function.

RQ5: Which factors influence the fault-proneness prediction model's accuracy when used across systems?

To answer the main research question, we will use the findings from our preliminary research to scope the domain in which the fault-proneness prediction models could be effectively used. Next, we use the improved model construction process to build our best fault-proneness prediction models, and apply those on datasets with idealistic similarities.

- If we are not able to construct a fault-proneness prediction model with a reasonable accuracy under these circumstances, then changes are that fault-proneness prediction models are not effective across systems and thereby answering our main research question. If this tends to be true, then we strongly advise against using regression-based prediction models for fault prediction in practice.
- If we are able to create one or more fault-proneness prediction models with reasonable accuracy, then we conclude that it is possible to build cross-system fault-proneness prediction models and that additional research is required to fully answer the main research question. If this is the case, we advice that future research focusses on the factors whom influence the fault prediction models.

1.5 Contributions

Contribution #1: Java metric-suite

In effort to replicate Briand et al. their study [4], a metric-suite was build to replace the non-public and closed source JMetrics metric-suite. Our metric-suite contains a subset of Briand et al. their cohesion metrics [25], Benlarbi & Melo their polymorphism measures [12], all metrics from the Chidamber & Kemerer metric suite [13], all metrics proposed by Li & Henry [26], and some size metrics. Besides from the byte-code metrics, an extension of the tool contains also process metrics: change metrics based on Koshgoftaar et al. [22] their change measurements, a subset of Ostrand & Weyuker their 'optimal' *standard model* measures [27], and a class-author count. The implementation of the metrics are described in detail (see Appendix A) and the tool is open-source and publicly available[1].

Contribution #2: Empirical knowledge

One part of this thesis was to replicate the study done by Briand et al. regarding cross-system fault-proneness prediction models [4]. We successfully replicated their study by using both the initial study and their exploratory study [14]. The validation data is based on 4 observations, using 6 large active commercial systems. The results provide a small contributes to the body of empirical knowledge.

[1] https://github.com/scrot/mt

Contribution #3: Prediction model improvements

We proposed three improvements regarding regression-based fault-proneness prediction models. The improvements cover three axes: (i) altering the collection of predictors to choose from; (ii) the modelling technique by considering the state of the system during measurement; (iii) improve the model by controlling the context of the systems. The first two improvements were validated using 6 large active commercial systems, the latter improvement was validated using 13 commercial systems. Two out of the three improvements (ii and iii) drastically increased the accuracy of the fault-proneness prediction models. The average accuracy of the 100 prediction models that were build using 13 commercial systems was 92%; with an average precision and recall of 92% and 93%, respectively. This is an increase in accuracy of 36% compared to the fault-proneness prediction models from the replication study; which had an average accuracy of 56%, an average precision 76%, and an average recall of 60%.

Contribution #4: Factor analysis

In the final part of this thesis we did extensive research into the influence of environmental and system related factors on fault-proneness prediction models. The study was supported by 13 large commercial systems. Although, we did not found factors whom influence prediction models, we provided evidence that people, technology, or process related factors are not likely to influence the fault prediction model.

1.6 Recommendations

Our recommendations are based on the results from the preliminary research (see Chapter 3), the replication study (see Chapter 4), the improvement study (see Chapter 5), the factor study (see Chapter 6), and the knowledge obtained from the literature study (see Section 1.3).

Based on our discoveries, it seems to be possible to build effective fault-proneness prediction models that could be applied across systems; All our cross-system fault-proneness prediction models obtained on average an accuracy of 92%. However, building stable fault-proneness prediction models is far from straightforward, there is still little known about which factors influence the prediction models and which predictors actually correlates with class fault-proneness. In our opinion, we do not think that (regression-based) fault-proneness could effectively applied in practice at this point. More research is necessary in order to build effective and reliable prediction models.

During our research, we improved commonly used methods, which are often used in regression-based fault-proneness prediction modelling. Comparing the average model accuracy from the replication study with the average model accuracy from our factor study, an improvement of 35% was observed. In our opinion, the increase was mainly caused by three things:

- *The used predictor-set.* The predictors used in the fault-proneness prediction models must be simple and have a clear relation to fault-proneness. Also, prevent using predictors that measure the same underlying concept and make sure that they measure various aspects of a class (e.g. include process measures and product measures).
- *State-aware measurements.* To correctly measure the properties of a fault-prone class, the state of the whole system considered. Preferably, revert the system-state to a version which contains a particular fault. The measurements will represent fault-prone classes more accurately and will result in better fault-proneness prediction models.
- *System-pools as dataset.* Build models using system pools, that are collections of systems which are taken from a similar context. The model is more likely to fit its function to generalizable fault-prone patterns rather than to the patterns of single system.

1.7 Outline

This thesis is divided in the following chapters: Chapter 2 provides information on the relevant topics; Chapter 3 is the summary of the preliminary research; In Chapter 4 we summarize the replication

study; Chapter 5 we propose our improvements to the prediction model construction method; Chapter 6 contains the exploratory study regarding factors whom might influence the prediction model.

Chapter 2

Background

2.1 Fault Detection

The interpretation of a fault specifies what the fault-proneness prediction model actually predicts. In this thesis we consider a fault the cause of an error that might lead to a system failure and could cause part of the system state to behave differently than expected [28].

2.1.1 Issue-trackers

Discovered faults could be stored in an issue-tracker system. An issue-tracker system is a database that contains issue-reports with information on faults that are currently in the system or faults that were fixed in the past. These reports could be classified into categories: Fault-reports, enhancements to the system, updates of the documentation, improvements of current system functionality, code re-factorings, and others (see Herzig et al. [29] for more information on issue-report categories). A class contained a single fault if it is changed by a commit that resolves an issue in the issue tracker. Only the fault-reports in the issue-tracker are considered.

2.1.2 Selection bias

"*Selection bias refers to systematic differences between baseline characteristics of the groups that are compared*" [30]. When issue-reports are used to extract system faults, the resulting dataset will be affected by selection bias. This is because the issue-reports could be incorrect; Herzig et al. [29] found that 33.8% of al bug reports are misclassified. Also, the issue-reports could be incomplete because of faults that are not (yet) in the issue-tracker. As a result, the conclusions drawn from these datasets are subjective.

2.2 Fault Distribution

The distribution of faults tells something about how faults are spread out over a system. The inequality of fault distributions is one of the reasons to build fault(-proneness) prediction models. Important to notice is that fault distributions are in most cases estimations of the actual fault distributions, this is because they are subjected to selection bias. The fault distributions in this thesis are therefore biased and the conclusions drawn are subjective. Two measures of inequality are used in this thesis and described in this section: Pareto principle and the Gini-coefficient.

2.2.1 Pareto principle

The Pareto Principle describes the notion of the vital few, where a small part of the observations are responsible for a large part of the effect [31].

It seems like the distribution of faults follows the Pareto Principle. Studies from different environments over many years confirmed the claim that a large number of the defects are caused by a

part of the system. In the Software Defect Reduction Top-10 List [1], Boehm and Basili state that "*About 80% of the defects comes from 20% of the modules and about half the modules are defect free*". Fenton and Ohlsson hypothesized that "*A small number of modules contain most of the faults*" [32] and found evidence for this hypothesis. The latter study is replicated with the same results [33, 34]. Weyuker & Ostrand also found strong evidence for the same statement [35].

Although the support for the Pareto Principle indicates an unbalanced distribution of faults within components, it could be easily explained by the fact that the code within the modules are also unequally distributed. In other words, the small number of modules that contain most of the faults also makes up most of the system's source-code. Fenton & Ohlsson tested the hypothesis "*If a small number of modules contain most of the faults [...] then this is simply because those modules constitute most of the code size*" [32] and found no evidence that supported the hypothesis, they even found strong evidence of a converse hypothesis. Also, the replication studies found no support for this hypothesis [33, 34]. However, there are studies that confirmed this hypothesis [36, 37].

The results of the studies related to the Pareto Principle and fault distribution in software systems are summarized in Table 2.1. The table contains the studies and the results of the two hypotheses: "*Few modules contain most faults*" and "*Few faulty modules constitute most of the size*". These hypotheses are the same as in Fenton and Ohlsson their study, but it does not separate faults found post-release and faults found pre-release. During the analysis we did not take the definition of a fault in consideration.

Based on the literature from Table 2.1, it seems that it is true that few modules contain most of the faults (most distributions are around 20-80, with a lowest extreme of 20-60). However, there is some support for the hypothesis that few fault modules constitute most of the total system size.

Table 2.1: Fault distribution literature

	Few modules contain most faults	Few faulty modules constitute most of the size
Fenton & Ohlsson [32]	Confirmed (20-60; 10-80; 10-100)	No support (20-30; 100-12; 60-6)
Andersson & Runeson [33]	Confirmed (20-63; 20-70; 20-70)	No support (20-38; 20-25; 20-39)
Grbac & Runeson [34]	Confirmed (20-67; 20-66; 20-77; 20-63; 20-80)	No support (20-32; 20-29; 20-22; 20-26; 20-23)
Munson et al. [8]	Confirmed (20-65)	-
Ohlsson & Alberg [6]	Confirmed (20-60)	-
Compton & Withrow [36]	Confirmed (12-75)	Confirmed (12-63)
Kaaniche & Kanoun [37]	Confirmed (38-80)	Confirmed (38-54)
Weyuker & Ostrand [35]	Confirmed (20-83; 20-83; 20-75 ;20-81; 20-93; 20-76)	-

2.2.2 Gini Coefficient

The Gini-coefficient is a measure of statistical dispersion and commonly used to measure inequality [38]. The Gini-coefficient measures the inequality among values of any frequency distribution. The value ranges between 0 and 1 where 0 implies complete inequality and 1 complete equality.

A graphical representation of the Gini-coefficient is shown in Figure 2.1 and can be calculated as follows [39]:

$$G = A/(A + B) \tag{2.1}$$

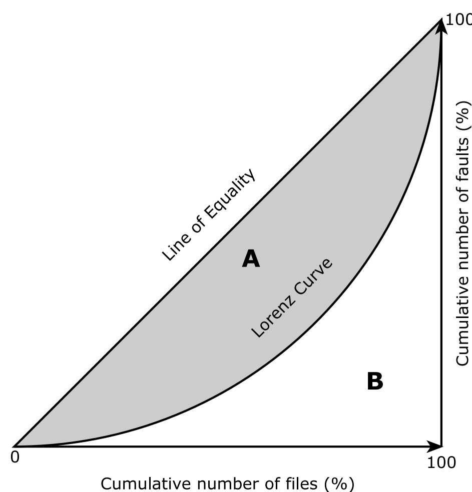
$A + B$ is the area under the line of equality and A can be calculated using Equation 2.2. Where A is the lowest and B is the highest value of the variable y, and $F(y)$ is the cumulative distribution of

y.

$$A = \int_{a}^{b} f(y)[1 - f(y)]dy \tag{2.2}$$

To relate the Gini-coefficient to the Pareto Principle, we will use a simplified version of the 20-80 rule. Instead of the fault-proneness prediction following a smooth distribution, we say that 20% of the system components contain 80% of the faults and that these faults are distributed equally over these 20%. The remaining 20% faults are also equally divided over the rest of the system components. The space between the equality line and the Lorenz curve is equal to $(0.5 - (0.8 + 0.8 + 0.4))/0.5 = 0.6$.

Figure 2.1: Gini coefficient (diagram based on [40])



2.3 Software Measurement

As predictors for the fault-proneness prediction models we will use a subset of product and process measures. Based on a subset of these metrics and their relation with fault-proneness, the fault-proneness prediction model may or may not classify components as fault prone.

A metric captures information about an attribute or entity [41]. A software metric captures a certain aspect of a software system. The goal of a considerable number of software metrics is to provide insight in the quality of the source-code of the system but there are also metrics that measure different aspects of the system. Based on the measurement goal of a metric, software metrics could be categorized into roughly two groups [42]:

- *Product metrics*, also known as quality metrics, measure the properties of the system itself. The product metrics includes reliability metrics, functionality metrics, performance metrics, usability metrics, and style metrics.
- *Process metrics*, also known as management metrics, measure the properties of the process which is used to obtain the software. It includes cost metrics, efforts metrics, advancement metrics, and reuse metrics.

The following subsections cover the research related to product metrics and process metrics.

2.3.1 Chidamber-Kemerer metrics

Chidamber and Kemerer proposed a metric suite consisting out of six object oriented design metrics based on the ontology of Bunge and validated them using Weyuker's proposed set of measurement principles [13]. The metric-suite holds the following product metrics:

- *Weighted Methods per Class* (WMC). the sum of the complexity of all methods within a class. For a class C_1, with methods $M_1...M_n$ defined in C_1 and $c_1...c_n$ the complexity of these methods:

$$WMC = \sum_{i=1}^{n} c_1 \tag{2.3}$$

  This metric is a predictor on how much time and effort is required to develop and maintain classes. In case of inheritance, these methods could also impact the sub-classes.

- *Depth of Inheritance Tree* (DIT). The depth of inheritance of a class. In cases of multiple inheritance, the maximum length from a node to the root of the tree. Classes that are deep in the inheritance tree are likely to inherit a greater number of methods, making it more complex to predict its behaviour.

- *Number of Children* (NOC). Number of immediate sub-classes subordinated to a class in the class hierarchy. A large number of child classes imply more code reuse, higher change of improper abstraction of the parent class, and more method testing on the parent class. The number of children also give an idea of the potential influence a class has on the design.

- *Coupling between Object Classes* (CBO). Number of other classes the class is coupled to. High coupling of classes tend to result in low modularity preventing code reuse. It also affects encapsulation of the classes making the code more sensitive to changes in other parts of the design, making maintenance more difficult.

- *Response for a Class* (RFC). The number of all methods that can be invoked in response to a message to an object of the class or some method in the class.

$$RFC = |RS| \text{ where RS is the response set of the class.} \tag{2.4}$$

  The response set for the class can be expressed as the equation below, where $\{M\}$ is the set of all methods in the class and $\{R_i\}$ is the set of methods called by method i.

$$RS = \{M\} \cap_{alli} \{R_i\} \tag{2.5}$$

  This metric indicates the complexity of a class and the effort to test or debug a class. A high number of possible methods that can be evoked in response to a message complicates testing and debugging because a higher level of understanding is necessary.

- *Lack of Cohesion Methods* (LCOM). The count of the number of method pairs whose similarity is null minus the pairs whose similarity is not null. For a class C_1, with methods $M_1...M_n$ and instance variables $\{I_j\}$ used by method M_i. Let $P = \{(I_i, I_j)|I_i \cap I_j = \emptyset\}$ and $Q = \{(I_i, I_j)|I_i \cap I_j \neq \emptyset\}$. If all n sets are $\emptyset$ then let $P = \emptyset$.

$$LCOM = \begin{cases} |P| - |Q|, & \text{if } |P| > |Q| \\ 0, & \text{otherwise} \end{cases} \tag{2.6}$$

  The measure of cohesion gives an indication of the complexity of a class. Low cohesion means that a class does more than one thing making it more difficult to understand.

2.3.2 Li-Henry metrics

Li and Henry revised and extended the Chidamber and Kemerer metric suite [26]. They removed the CBO metric which measures the non-inheritance related coupling and added three coupling metrics: *coupling through inheritance*, *coupling though message passing*, *coupling though abstract data types*; a class increment metric, number of local methods (NOM); and two size metrics: lines of code of a class (SIZE1), and the number of class properties (SIZE2).

Objects could be coupled to each other through certain communication mechanisms. There are three form of coupling: coupling through inheritance, coupling through message passing, and coupling through data abstraction.

- *Coupling through Inheritance* (CTI). Measured using depth of inheritance (DIT) or number of children of a class(NOC). Inheritance promotes reuse but also creates the possibility of violating encapsulation and information hiding. Incorrect use of inheritance or improper design may introduce extra complexity to a system, making it more fault-prone.

- *Coupling through Message Passing* (CTM). Measured using the MPC metric, that is the number of send statements defined in a class. Message passing occurs when an objects needs some service that another object provides. The number of messages send from a class may indicate how dependent the implementation is on other classes. One needs to keep the classes the program depends on in mind, increasing the complexity of the class. Also, if the other class contains a bug it is probable it could affect the classes that depend on it (e.g. when a class throws a null-pointer exception, the dependent classes must deal with this exception accordingly).

- *Coupling through Abstract Data Types* (CTA). Measured using the DAC metric, the number of abstract data types (ADTs) in a class. A class can declare a variable that has the type of the ADT (e.g. extensions and implementations of the ADT). This type of coupling may cause violation of encapsulation if private properties could be accessed directly, which could result in faults. Moreover, the more ADTs a class has the more complex the coupling of that class.

Measures related to class interface increment:

- *Number of local methods* (NOM). The number of local methods. Gives an indication of the complexity on an interface of a class. This metric may indicate the operation property of a class; the more methods a class has, the more complex the class its interface.

Measures related to size:

- *Class lines of code* (SIZE1). measured as number of semicolons in a class. The size of a procedure or function could be an indication of the complexity of a class. Also a class is more likely to contain bugs if it contains more code.

- *Number of class properties* (SIZE2). The sum of the total number of attributes and the total number of local methods. This gives a more high level indication of the size of a class. The higher the measure the more likely it is that the class contains faults.

2.3.3 Briand et al. coupling metrics

Briand et al. devised a suite for measures to quantify the level of class coupling [25]. The suite includes different measures of OO specific coupling mechanisms. They analysed the relationship between the measures and the probability of fault detection across classes. The results show that some of the coupling measures may be useful as early quality indicators of OO design. Moreover, they found that the measures are conceptually different from the Chidamber & Kemerer metric suite and could be used to complement the CK-metric suite.

The metric suite contains 18 metrics: $IFCAIC$, $ACAIC$, $OCAIC$, $FCAEC$, $DCAEC$, $OCAEC$, $IFCMIC$, $ACMIC$, $OCMIC$, $FCMEC$, $DCMEC$, $OCMEC$, $OMMIC$, $IFMMIC$, $AMMIC$, $OMMEC$, $FMMEC$, $DMMEC$. The coupling metrics are counts for interactions between classes and distinguish the relationship between the classes, the locus of impact, and the type of interaction. The acronyms for the metrics indicates what interactions are counted:

- The first or first two letters indicate the relationship between the classes, that is the relation of an arbitrary class C_i to the considered class C. The following relationships are taken into account: A, coupling to ancestor class; D, coupling to descendant classes; F, coupling to friend classes; IF, coupling to inverse friend classes; and O, any other coupling relationship.

- The next two letters indicate the type of interaction: CA, there is a class-attribute interaction between class C and C_i if C has an attribute of type C_i; CM, there is a class-method interaction between class C and C_i if C has a method with a parameter of type C_i; MM, there is a method-method interaction between class C and C_i if C invokes a method of C_i, or if a method of class C_i is passed as parameter to a method of class C.

- The last two letters indicate the locus of impact: IC, import coupling, the measure counts for a class C all interactions where C is using another class; EC, export coupling, counts interactions where class C is the used class.

2.3.4 Benlarbi-Melo polymorphism metrics

Benlarbi & Melo defined an empirically investigation into the quality impact of polymorphism on OO design [12]. They described two aspects of polymorphism: static polymorphism, polymorphism based on compile time linking decisions (e.g overloading functions); and dynamic polymorphism, polymorphism based on run-time binding decisions (e.g. virtual functions). They validated their measures by evaluating their impact on class fault-proneness. They found that their measures measure on a different orthogonal dimension than size measures and that they are significant predictors for fault-proneness.

The metric suite they devised constitutes out of 6 metrics:

- *OVO*. Overloading in stand-alone classes. Measures the number of methods that are overwritten in the same class, that are all methods with the same method name but different arguments. The metrics is calculated using the following equation:

$$OVO(C) = \sum_{f_i \in C} \mathit{overl}(f_i, C) \tag{2.7}$$

  Where $\mathit{overl}(f_i, C)$ is an operator which returns the number of times the function member name f_i is overloaded in class C.

- *SPA*. Static polymorphism in ancestors. Measurement of the unique class couples of which their methods statically overloads one another and where of one of the classes is an ancestor of the other. The measure is calculated using the following equation.

$$SPA(C) = \sum_{C_i \in Ancestors(C)} \mathit{SPoly}(C_i, C) \tag{2.8}$$

  Where $\mathit{SPoly}(C_i, C)$ is a function which returns the number of statically polymorphic functions that appear in C_i and C. Static polymorphic functions are functions that have the same name but a different signature. $\mathit{Ancestors}(C)$ returns the set of distinct ancestors of class C.

- *SPD*. Static polymorphism in descendants. Measurement of the unique class couples of which their methods statically overloads one another and where of one of the classes is a descendant of the other. The measure is calculated using the following equation.

$$SPD(C) = \sum_{C_i \in Descendant(C)} \mathit{SPoly}(C_i, C) \tag{2.9}$$

  Where $\mathit{SPoly}(C_i, C)$ is a function which returns the number of statically polymorphic functions that appear in C_i and C, and *Descendant* returns the set of distinct descendants of class C.

- *DPA*. Dynamic polymorphism in ancestors. Measurement of the unique class couples of which their methods dynamically overloads one another and where of one of the classes is an ancestor of the other. The measure is calculated using the following equation.

$$DPA(C) = \sum_{C_i \in Ancestors(C)} \mathit{DPoly}(C_i, C) \tag{2.10}$$

  Where $\mathit{DPoly}(C_i, C)$ is a function which returns the number of dynamically polymorphic functions that appear in C_i and C. Dynamic polymorphic functions are functions that have the same name and the same signature. $\mathit{Ancestors}(C)$ returns the set of distinct ancestors of class C.

- *DPD*. Dynamic polymorphism in descendants. Measurement of the unique class couples of which their methods dynamically overloads one another and where of one of the classes is an ancestor of the other. The measure is calculated using the following equation.

$$DPA(C) = \sum_{C_i \in Descendant(C)} DPoly(C_i, C) \tag{2.11}$$

  Where $DPoly(C_i, C)$ is a function which returns the number of dynamically polymorphic functions that appear in C_i and C, and *Descendant* returns the set of distinct descendants of class C.

- *NIP*. Polymorphism in non-inheritance relations. The measure of unique class pairs that dynamically or statically overload their methods and the relation between them is neither ancestor or descendant. The measure is given in the following equation:

$$NIP(C) = \sum_{C_i \in Others(C)} SPoly(C_i, C) + DPoly(C_i, C) \tag{2.12}$$

  Where $SPoly(C_i, C)$ is a function which returns the number of statically polymorphic functions that appear in C_i and C, $DPoly(C_i, C)$ is a function which returns the number of dynamically polymorphic functions that appear in C_i and C, and $Others(C)$ returns the set of distinct classes that are neither ancestors or descendants of class C. NIP is not actual polymorphism but could be a potential for human confusion.

2.3.5 Khoshgoftaar reuse metrics

Koshgoftaar et al. developed a fault-proneness prediction model based solely on process-history variables [22]. This research is based on a preliminary study where they showed that reuse indicators can improve classifications models for identifying fault-prone modules. They used three process metrics for the purpose of measure reuse: module did not existed in previous versions (IsNew), module was changed since last version (IsChg), and the age of a module (Age).

$$IsNew = \begin{cases} 1 & \text{If module did not exist in ending} \\ & \text{version of prior build} \\ 0 & \text{Otherwise} \end{cases} \tag{2.13}$$

A module is considered reused if it had existed as part of a previous build. If a module required no code change, it was reused as an object.

$$IsChg = \begin{cases} 0 & \text{If no changed code since prior build} \\ 1 & \text{Otherwise} \end{cases} \tag{2.14}$$

They argue that modules with a long history may be more reliable and therefore expected to contain less faults. The definition of a module's age is the number of builds it has been through.

$$Age = \begin{cases} 0 & \text{if module is new} \\ 1 & \text{If module was new in the prior build} \\ 2 & \text{Otherwise} \end{cases} \tag{2.15}$$

2.3.6 Relationship between product metrics and fault-proneness

Briand et al. [14] explored the relationships between software measures and the quality of object-oriented systems. They looked at the relation of coupling, cohesion, and instance measures and the probability of fault detection in system classes during testing. They argued that size of classes, frequency of method invocations, and depth of inheritance hierarchies might be the main driving factors of fault-proneness.

Briand et al. hypothesized that "*a class with high import coupling is more likely to be fault-prone than a class with low import coupling*". This is because a class with high import coupling relies on many external services, which all have to be understood. The challenge of understanding all the services, and the increased likelihood of misunderstanding or misuse could result in more fault-prone classes. The result of the univariate logistic regression analysis provides strong support for this hypothesis. Most relationships between the import coupling measures and fault-proneness were significant; method invocation seems to have the highest impact on fault-proneness.

Regarding the export coupling measures, Briand et al. hypothesized that "*a class with high export coupling is more likely to be fault-prone than a class with low export coupling*". An export-coupling class has many other classes that rely on it. Failures are therefore likely to be traced back to a class with export-coupling, this makes the class more fault-prone. There is no evidence for this hypotheses; only the OCAEC measure was significant. A class that is used by many other classes does probably not relate to fault-proneness.

As third hypothesis was formulated that "*a class with low cohesion is more likely to be fault-prone than a class with high cohesion*". "*Low cohesion indicates inappropriate design*", and therefore a class with low cohesion would be more fault-prone. The univariate logistic regression analysis of the cohesion measures showed that only the LCOM3, Coh, and ICH were significant, but these are unlikely to measure cohesion according to Briand et al. They concluded that there was weak support for the cohesion hypothesis.

For the relation between depth measures (DIT, AID) and fault-proneness, Briand et al stated that "*a class situated deeper in the inheritance hierarchy is more likely ot be fault-prone than a class situated higher up in the inheritance hierarchy*". Classes situated lower in the class hierarchy are more likely to be inconsistent in correctly extending or specializing the ancestor classes, and would therefore be more fault-prone. This hypothesis is supported, the DIT and AID measures were all significantly related to class fault-proneness.

Regarding the relationship between ancestor measures (NOA, NOP, NMI) are fault-proneness, Briand et al. hypothesized that "*a class with many ancestors is more likely to be fault-prone than a class with few ancestors*". They state that the larger the number ancestors a class is concerned with, the larger the context needed to understand what the class represents. Such a class is more likely to be fault-prone. The hypothesis is supported; all ancestor measures were significant.

For the relation between descendant measures (NOC, NOD, CLD) and fault-proneness, Briand et al stated that "*a class with many descendants is more likely to be fault-prone than a class with few descendents*". As with high export-coupled classes, classes with many descendants have large influence on the system because many classes rely on that class. "*The class has to serve in many different contexts, and is therefore more likely to be fault-prone*". There is support for the hypothesis, however their impact of fault-proneness is smaller compared to the depth measures or ancestor measures.

For polymorphism measures (NMO, SIX), the hypothesized that "*The more use of method overriding is being made, the more difficult/complex it is to understand or test the class*". The result is a class that is likely to be fault-prone. The univariate regression analysis results provided evidence for this hypothesis; both the NMO and SIX measures were significant.

Finally, they stated that "*the larger the class, the more fault-prone it is*". This is because the class contains more information. This hypothesis has weak support, only the NMA metric was significant.

2.4 Principle Component Analysis

Principle Component Analysis (PCA) is a multivariate statistical technique for analysing data where observations are described by several inter-correlated dependent variables and was first formalized by Hotelling [43]. The technique is used to extract important information from the dataset and represent it as a set of new orthogonal variables called Principle Components (PCs). There a multiple ways of performing a PCA, we adopted the method as described by Smith [44]. For more information on the Principal Components Analysis see Abdi & Lynne [45].

Principle Components are obtained as linear combinations of the original values, these values are called factor scores. For finding the principle components, the PCA starts with a first PC that has the largest possible variance, the second PC is computed under the constraint of being orthogonal

to the first PC and to have again the largest possible variance. The other principle components are calculated in a similar fashion as the latter component.

Taking a correlation matrix as input, the unit eigenvectors and their accompanying eigenvalues can be calculated. These eigenvectors are perpendicular to each other and reveal the patterns in the data if there are any. For choosing the first PC the eigenvector with the highest eigenvalue is chosen. The second PC is the eigenvector with the second highest eigenvalue, and so on; the result is a feature vector. On this point, one could choose to ignore vectors that have an eigenvalue below a certain threshold, this is called dimensionality reduction. The final step is to multiply the transposed feature vector with the transposed input matrix.

2.4.1 Rotations and loadings

An Interesting observation could be made when looking at the loadings of the variables of the PCs, those are the correlations between the PC (the eigenvector) and that of the original variable. Component loadings are analogous to correlation coefficients, squaring them gives the amount of explained variation and tells something about how much of the variation in a variable is explained by the component. If a variable has a high loading then it is strongly correlated with the PC and therefore measures along that dimension.

To simplify the loadings and make them more interpretable the PCs are often rotated. A rotated PC is called a Rotated Component (RC). The RCs are extracted by rotating the axes to align them with the eigenvectors. Two types of rotations are possible: orthogonal rotations (e.g. varimax) that assume the factors are not correlated, and oblique rotations (e.g. promax, oblimin) that allow for correlation.

2.5 Logistic Regression Models

Logistic regression is one of the most frequently used regression methods in data analysis concerned with describing the relationship between a response variable and one or more explanatory variables. The application of the logistic regression model is to find the best fitting and easily interpretable model that describes the relationship between an outcome (dependent or response) variable and a set of independent variables (predictor or explanatory variables). Logistic regressions is frequently used when the outcome variable is dichotomous, or binominal (e.g. dead or alive, passed or failed) [46].

The equation for calculating the probability of an outcome variable, based on a set predictors is as follows:

$$\pi(\vec{x}) = \frac{e^{(\beta_0+\beta_1 x_1+\beta_2 x_2+\cdots+\beta_p x_p)}}{1+e^{(\beta_0+\beta_1 x_1+\beta_2 x_2+\cdots+\beta_p x_p)}} \tag{2.16}$$

Where π is the probability of the outcome and $\vec{x}$ are the independent variables used in the model. The logistic regression model (see Equation 2.16 has $\beta_0+\beta_1 x_1+\beta_2 x_2+\cdots+\beta_p x_p$ as unknown parameters.

To fit the regression model to a dataset, the values of these parameters must be estimated. In logical regression, this is done using *maximum likelihood*. In short, *maximum likelihood* assigns values to the unknown parameters such that they maximize the probability of obtaining the observed set of data. In order to calculate the *maximum likelihood* the following function is used (the log variant of the equation is more often used due its mathematical benefits, but this variant is more easy to understand):

$$l(\vec{\beta}) = \prod_{i=1}^{n} \pi(\vec{x_i})^{y_i}[1-\pi(\vec{x_i})]^{1-y_i} \tag{2.17}$$

Where $\vec{\beta}$ represents the unknown parameters, the pair (x_i, y_i) is the value of the outcome variable (y_i) and the value of the independent variable (x_i) for an independent observation i, $pi(x_i)^{y_i}$ is the probability of $(Y = 1|x)$ based on all observations that are associated with the outcome variable, and $[1-\pi(x_i)]^{1-y_i}$ is the probability of $(Y = 0|x)$ based on all observations that are associated with the outcome variable. $p+1$ likelihood equations to be solved and are obtained by differentiating the *log-likelihood function* with respect to the $p+1$ coefficients. The *maximum likelihood estimators* of

the parameters are the values that maximizes the equations. Because the equation is not linear the optimization is done iteratively.

2.5.1 Model comparison

Three methods are occasionally used to statistically determine if the independent variables in the model are significantly related to the outcome variable: The likelihood ratio test, Wald test, and scored test. All these tests makes use of the likelihood of the models to assess their fit. We only focus on the log likelihood ratio test, the other methods are described in detail by Hosmer & Lemeshow [46].

The likelihood ratio test (or ls-test) compares two models based on the relative goodness of fit. It assumes that one model is a nested model of the other, meaning that one model is a sub- (or simplified) model of the other. It uses the likelihood ratio, which expresses how many times more likely the data are under one model than the other. A low ratio means that the observed result was less likely to occur under the null hypothesis.

$$G = -2 \ln \left[\frac{(\textit{likelihood without the variable})}{(\textit{likelihood with the variable})} \right] \tag{2.18}$$

The function G is chi-square distributed with p degrees of freedom, the difference in number of parameters between the two models. The model could be expressed in terms of deviance or in terms of fitted model and saturated model.

2.5.2 Stepwise selection

The goal of selection methods is to find the set of variables that result in the best model within the context of the problem. Statistical model building involves seeking the a minimal model that still accurately reflects the true outcome of the data. The reason for finding this model is because it is likely to be numerically more stable (less chance of overfitting) and more easily adopted for use. Moreover, the more variables included in the model, the greater the estimated standard errors become, thus making the model more dependent on the observed data [46].

A popular selection method is stepwise selection. Stepwise selection is used in cases where there are a large number of independent variables of which the association with the outcome variable is not well understood. All stepwise selection methods base their inclusion or exclusion of variables on the outcome of a statistical algorithm that checks the importance of variables. Based on the outcome and a fixed decision rule, variables are included or excluded in the model. For logistic regression models, the importance of variables is tested using one of the three model significance tests: likelihood ratio, score, and Wald test. Two popular types of stepwise selection processes exists, forward and backward (Hosmer & Lemeshow describe both the algorithms in detail [46]).

- *forward stepwise selection.* Forward selection starts with a model that includes the intercept only. Based on a statistical criteria, variables are selected one at a time for inclusion until the stopping criteria is met.
- *backward stepwise selection.* Backward selection includes all independent variables, and variables are deleted based on a statistical criteria until the stopping condition is met.

2.5.3 Model measures

For validating the model itself, several metrics could be used. These metrics are based on two elementary counts:

- *True/false positives.* The count of correctly identified outcome values such that $(Y = 1|x)$ and incorrectly identified outcome values respectively.
- *True/false negatives.* The count of correctly identified outcome values such that $(Y = 0|x)$ and incorrectly identified outcome values respectively.

These counts are used in a confusion matrix and used to describe the performance of a classification model, see Table 2.2. The rows represent the actual cases and the columns the predicted case. The values range from T- to F+ were T/F represents true/false and +/- represents positive/negative.

Table 2.2: Confusion matrix

	Predicted True	Predicted False
Actual True	$T+$	$F-$
Actual False	$F+$	$T-$

Based on the elementary counts, new composite measures could be formed:

- *Accuracy.* Accuracy is defined as the true positives and true negatives identified by the model divided by the all observations. The measure can be obtained using the following equation:

$$Accuracy = \frac{(T+) + (T-)}{All\ Observations} \tag{2.19}$$

  The measure is a ratio of the accuracy of the model. A low accuracy means that most of the predictions of the model are incorrect. If an arbitrary model did 10 predictions in total but only 8 were correct, then the model's accuracy is .8.

- *Precision.* Precision is defined as the true positives identified by the model divided by the total number of actual positives. The measure is expressed by the equation:

$$Precision = \frac{T+}{(T+) + (F+)} \tag{2.20}$$

  The measure is the ratio of all the actual positives the model did find. A low recall indicates that a lot of the actual positives were not detected by the model. If an arbitrary model identified 7 true positives but the actual number of positive cases is 10, then the model's recall is .7.

- *Recall.* Recall is defined as all the correctly identified positive cases divided by the incorrectly identified negatives and the correctly identified positives and given by the following equation:

$$Recall = \frac{T+}{(T+) + (F-)} \tag{2.21}$$

  The measure is the fraction of all correct results returned by the model. If an arbitrary model identifies 2 true positives, 3 true negatives, and the total of cases is 10, then the model's recall is .5.

.

2.5.4 Model validation

For cross-validation validates the model using the same system it was build on. Cross-validation can be done in several ways, [47] studied different types of cross-validation and bootstrap techniques for accuracy estimation and model selection. In his paper he describes several types of cross-validation techniques, including the two most frequently used: holdout and k-fold cross-validation

- *Holdout*, also called test sample estimation, partitions the data into two mutually exclusive subset; the training set and the test set. The training set is usually 2/3 of the data set and the test set 1/3. The smaller the training set, the higher the bias; the smaller the test set, the wider the confidence interval. A drawback of the holdout technique is that is makes inefficient use of the data, a third of the dataset is not used for training. Because the individual systems of our dataset are rather small, we will not use this validation technique.

- *Cross-validation*, also called k-fold cross validation or rotation estimation, randomly split the dataset into k mutually exclusive subsets $D_1, D_2, ..., D_k$ of approximately equal size. The model is trained and tested k times; each time t it is trained on $D \setminus D_t$ and tested on D_t. Kohavi recommends the stratified ten-fold cross-validation method for model selection. This is the one we will use in our research.

2.6 Outlier Detection

Outlier analysis is important in prediction model building, especially if the models that are being build are regression models. Outliers in data can distort the prediction and the effect on accuracy. Detecting outliers in an univariate sense done by considering the Inter Quartile Range (IQR), that is the difference between the 75th and 25th quartiles. A continuous data-point is considered an outlier if the following equation holds:

$$x_i > 1.5 * IQR \tag{2.22}$$

A multivariate outlier analysis is done a bit differently compared to the univariate variant. The difference between the two types of outlier analysis is that the multivariate type measures the distances with respect to the correlation structure. As a result, the exclusion criteria is not based on the distance from the mean but rather the distance from the correlation structure.

2.6.1 Mahalanobis distance

Various distance measures could be used to measure the distance from the correlation structure, one of them is is Mahalanobis distance. The Mahalanobis distance of an observation is the distance between a data-point and the centroid of the dataset in units of standards deviations. It takes the correlation structure of the data as well as the individual scales into consideration. The lower the distance, the closer a data-point is to the multi-dimensional mean (centroid). The distance is measured with respect to the Principal Components, a Mahalanobis distance of 1 is equal to the standard deviation of along single orthogonal dimension; this holds for every orthogonal dimension. The distance of a single data-point is calculated as follows.

$$D(x) = \sqrt{(\vec{x} - \vec{\mu})^T S^{-1} (\vec{x} - \vec{\mu})} \tag{2.23}$$

Where $\vec{x} = (x_1, x_2, \ldots, x_n)$ is a single datapoint its values of the independent variables. $\vec{\mu} = (\mu_1, \mu_2, \ldots, \mu_n)$ are the means of the independent variables. S^{-1} is the inverse covariance matrix of the independent variables.

Outliers could be detected using the Mahalanobis distances by setting the distances of the data-points of gainst the 97,5% Quantile. A data-points that satisfy the following equation are considered outliers.

$$D(x_i) > Q \tag{2.24}$$

where $D(x_i)$ is the Mahalanobis distance of a single data-point and Q is the 97.5%-Quatile of the Chi-Square distribution.

Chapter 3

Fault Distributions in Software Systems and the Pareto Principle

The fault distribution plays an important role in fault-proneness prediction. Without an unequal distribution of faults in a system, one part of the system is not more fault-prone than another. There is much support for the claim that a large part of the faults reside in a small part of the system (see Section 2.2), but this support comes from around 20 systems where most of those systems are closed source or dated. Because of the importance of this claim, we dedicate this chapter to validate if the Pareto Principle holds for the fault distribution in modern software systems. To re-evaluate the claim in a modern context, we will replicate a part of Fenton & Ohlsson's research on faults and failures in a complex software system [32]. In the following section we will describe our research question, study setting, and data analysis methodology. We conclude with the analysis results and discussion.

3.1 Introduction

It is widely accepted that faults are unequally divided over software. Many belief that an average system's fault distribution adheres to the Pareto Principle; many faults reside in a small part of the system. From the eight studies we considered, all of them supported this claim (see Section 2.2). They found distribution between 20-60 and 10-100, where 20-60 means that 20 percent of the modules contains 60 percent of the system's faults.

One of the main remarks regarding the Pareto Principle is that the effect could easily be explained by the inequality in size of the modules. In other words, the module size correlates with module faults, and as a result, the modules that are large in size naturally contains more faults compared to modules that are small in size. However, this effect was not observed in any of the studies of our literature study.

A remark regarding the literature is that they do not clearly mention that their dataset is only an approximation of the actual fault distribution that their conclusions are subjective (see Section 2.1 for more information on selection bias). Note that the conclusions drawn in this study are subjective and that they only tell something about the faults we observed.

Our hypotheses related to the Pareto Principle are based upon hypotheses 1 and 2 of Fenton & Ohlsson their qualitative analysis of faults and failures in a complex system [32], but we did not replicate the hypotheses. Our hypotheses differ in two ways:

- Instead of measuring faults on module-level, we measure faults on class-level. The reason is that the study we will replicate only considers classes. Moreover, Catal & Diri [3] recommends to perform measurements on class-level.
- Instead of separating faults found in pre-release testing and during operation, no such division was made in this study. Systems developed using an agile software development methodology have no strict separation of pre-release and operation faults.

We use hypothetical population means based on the findings from our literature study. The first hypothesis will provide an answer to the question: does the distributions of observed faults in software systems adhere to the Pareto Principle? The 21 fault distribution observations of our literature study have a median of 20-75. Therefore, we expect that an average system has at least an observed-fault distribution of 20-75.

H1: 20% of a software system's classes contains at least 75% of the total observed faults

Secondly, we expect that 20% of the most fault-prone classes in a system does not make up more than 29% of its total size. This value is the median of 13 observations from the literature study.

H2: 20% of a software system's classes that contains most of the observed faults make up 29% of the total system's size at most

Beside the two hypotheses devised by Fenton & Ohlsson, we will also propose a different measure for measuring the inequality of the observed-fault distribution, the Gini-coefficient (see Section 2.2). The Gini-coefficient is more expressive compared to the Pareto Principle because it tells something about the whole distribution instead of a single point in the distribution. We consider a distribution based on the first hypothesis (H1) 20-75. We interpret this rule as a discrete distribution. The Gini-coefficient is calculated as follows:

$$B = \left(\frac{1}{2} * \frac{75}{100} * \frac{25}{100}\right) + \left(\frac{25}{100} * \frac{25}{100} + \frac{1}{2} * \frac{25}{100} * \frac{75}{100}\right) = \frac{25}{100} \tag{3.1}$$

And the Gini coefficient:

$$G = \left(\frac{50}{100} - \frac{25}{100}\right) \Big/ \frac{50}{100} = 0.5 \tag{3.2}$$

Our hypothesis is as follows:

H3: The Gini coefficient of a software system's observed-fault distribution is at least 0.5

The chapter is structured as follows: Section 3.2 describes the setting of the study. Section 3.3 lays out the analysis methodology. In Section 3.4 we present our findings. In Section 3.5 we discuss our findings, draw our conclusions, and layout feature work.

3.2 Description of Study Setting

In this section we describe the systems, and how we collect and filter the dataset. We describe the independent and dependent variables, and how we will analyse the data and answer our hypotheses.

3.2.1 Systems

For all our three hypothesis we will use the same collection of systems. We selected the first 1000 Java systems from GitHub [1] stars. A star is given to a project for each person who marked the project as interesting and wants to follow its activities; a person can revoke its star if he no longer finds the project interesting. As a result, projects with the most stars may indicate the projects (hosted on GitHub) that are deemed to be most interesting by the users of GitHub at that current moment. We assume that these projects are popular, modern, and actively developed and tested systems. A threat to validity is that the selection of the systems is not completely random. In order to select the most popular dataset, we made an ordering of population we could pick from. The result is that the dataset is biased towards the popular projects. However, we think that these ordering causes less bias compared to the constrains we had to set on the data like minimum size, minimum degree of activity, maximum age. Because the star-rating provides an indication of projects that are popular today; we expect them to be active, relevant, heavily used, and of reasonable size.

We retrieve all the project data of the 1000 systems using an automated tool that is part of *gcrawler*. This tool collects up-to-date data of the project characteristics (e.g. starts, forks, description) and

[1] https://github.com

pulls the latest version of the system from GitHub. All the systems are based on the latest commit of the master branch at the time of collection. From the 1000 systems, 934 used the GitHub issue tracking system. The last measured activity of 45 systems was before 2015 (with 1 in 2011, 3 in 2012, 16 in 2013, and 25 in 2014), 130 systems were active till 2015, and 825 systems were still active in 2016. The systems had on average 1916 stars, 609 forks, and were on average 54 megabytes large (for most projects this size is based on the source-code size). According to GitHub, the largest project in the dataset, liferay-portal, is 7.415 megabytes, and the smallest project android-smart-image-view is 57 kilobytes (one project had a size of 0). 933 system used the issue tracking system of GitHub.

The factors of the collected systems are gathered using *ovms* (see Section **??**). Of the selected systems, the source-code and the git-tree are analysed. Not all files are included for analysis; excluded files are non-Java files, generated files, and test files (see Section **??**). Due to the size of our dataset, it is not possible to analyse the systems on class-level, instead we did the analysis on file-level which is still more fine grained compared to an analysis on module-level. A class contains a fault if it is changed by a commit that fixed a fault (see Section **??**). Based on the results, we will exclude some systems based on the number of observed faults they have. All systems with less than 2 faults will be removed from the dataset. We exclude these systems because they will skew the dataset. For example, if a system has zero faults (e.g. because they did not mention the fix in the commit message) the Gini coefficient will be 0, meaning that the fault distribution is perfectly distributed. On the other hand; if a system has a single fault, the Gini coefficient will be 1, which means the faults are extremely uneven distributed. These extreme cases do not give a realistic representation of a fault distribution and are therefore excluded from the dataset. In total, 856 systems were included in the dataset.

3.2.2 Measurement instruments

For building the fault- and code distribution we use the data provided by our own tools *gcrawler* and *xloc*, respectively. The distribution takes a map of classes to counted values and returns map of number of classes to cumulative counted values (with increasing difference). More simply put, the fault distribution is an aggregation of the counted values of classes that are sorted from low to high.

When taking a percentage of the distribution (e.g. 20% of distribution) the percentage is converted to the total number of files that is requested and is round down if the value is not an integer. For example, the distributions has a total of 101 classes a 20% is requested, 20% of 101 classes is 20.2; so one wants the total number of faults that reside in the first 20.2 classes. This is not possible and the number of faults of the first 20 classes is returned instead.

The function of the distributions are always discrete and a composition of linear functions because the space between the classes is always a positive integer, therefore we can exactly calculate the Gini-coefficient (see Figure 3.2.2). We calculate the Gini-coefficient by taking a distribution as described in the previous paragraph. The space between x and $x-1$ can be calculated as follows (the difference between $x-1$ and x is always 1):

$$S(x) = 1/2 * (y_x - y_{x-1}) * x + y_{x-1} * x \tag{3.3}$$

With y_x the corresponding y of x and y_{x-1} the corresponding y of the predecessor of x. To calculate the total space under the under the Lorenz-curve:

$$B = \sum_{x=1}^{n} S(x). \tag{3.4}$$

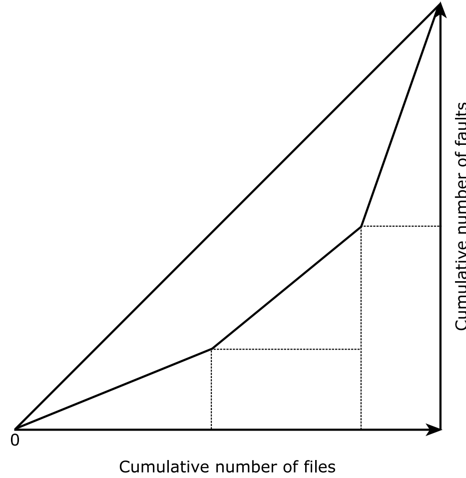
Difference between the space of the line of equality and the Lorenz curve is then calculated as follows:

$$A = 1/2 * \text{total faults} * \text{total files} - B \tag{3.5}$$

Finally the Gini-coefficient can be calculated:

$$G = A/(A+B) \tag{3.6}$$

Figure 3.1: Calculating Gini coefficient (diagram based on [48])



3.2.3 Variables

Independent variables

Of the systems that were included for analysis, we collected the following values to answer the hypotheses: the percentage of faults in the 20% most faulty classes; the percentage of code in the 20% most faulty classes; and the Gini coefficient of the fault distribution. The independent variables are the means of the collected values.

Dependent variables

As dependent variables for the first two hypotheses we used the median of all the observation made in the literature survey regarding the Pareto Principle. For the fourth hypothesis we used a Gini coefficient based on a distribution of 20-75. These values are fixed and mentioned in the hypotheses.

3.3 Data Analysis Methodology

From the data collected by *ovms*, the mean (μ), standard deviation (σ), maximum (Max), and the minimum (Min) are calculated.

Next, the histograms of the Pareto Principle values (x'es of the 20-x are plotted), the percentage of code in the 20% most fault classes, and the Gini-coefficients of the observed-fault distributions are presented to help understand the results of remaining analysis.

The goal of testing the first hypothesis (H1) is to check if on average an observed-fault distribution of a software system adheres to the Pareto Principle. In the literature study, a fault distribution of at least 20-60 is considered a distribution that adheres to the Pareto Principle (with a median of 20-75). To test the hypothesis we will conduct an one-tailed one-sample t-test for the hypotheses (with an alpha value of $\alpha \leq .05$). We choose this statistical test because we are working with a hypothetical population mean and a sample mean, and we want to know about the direction of the difference between those means.

The goal of testing the second hypothesis (H2) is to verify that in general only a small part of the system's code resides in the 20% most faulty classes. In the literature, at most 40% of the system's code may reside in 20% of the most faulty classes to accept the hypothesis (with a median of 29%) This hypothesis is similar to Hypothesis H1, therefore the same test will be used.

The last hypothesis is similar to the previous hypothesis, and tested using the same test. We consider an observed-fault distribution with Gini-coefficient of .50 unequally distributed and therefore use this

value as the hypothetical mean.

3.4 Analysis Results

Out of the 1000 systems, 856 systems were analysed by the *ovms* tool. The other systems caused an error: some projects had a size of zero; some projects had the exact same name as other projects (our tool used project-name as unique identifier); Some project did not had a HEAD commit; and some projects had illegal field values. These projects were automatically excluded from the dataset by the tool. Of the 856 systems none of the systems had less than 4 faults and were all included for further analysis. Histograms of the fault- and code-distributions can be found in Appendix B.

Testing the Pareto Principle on fault distribution. A one-tailed one-sample t-test was conducted to determine if a statistically significant difference ($\alpha \leq .05$) existed between inequality of distribution of the observed faults from our sample of 1000 popular GitHub projects and the observations from our literature study. For the first hypothesis (H1), the null hypothesis was formulated $H_0 : \mu < 75$. The total percentage of faults in the 20% most faulty files in the systems we analysed was more than 75% ($M = 77.27$, $SD = 18.991$) than the systems from our literature study on the Pareto Principle, $t(856) = 3.519, p = .000$. There is significant evidence to reject the null hypothesis, therefore we conclude that the faults we observed are unequally distributed over the system and that at least 75% of the system's faults resides in 20% of its files. Further, Cohen's effect size value ($d = .12$) suggested a small practical significance.

Testing the percentage of code in the 20% most faulty files. A one-tailed one-sample t-test was conducted to determine if a statistically significant difference ($\alpha \leq .05$) existed between the percentage of code in the 20% of the most faulty files from our sample of 1000 popular GitHub projects and the observations from our literature study. For the second hypothesis (H2), the null hypothesis was formulated $H_0 : \mu > 29$. The total lines of source code in the 20% most faulty files in the systems we analysed was less than 29% ($M = 25.89$, $SD = 24.608$) than the systems from our literature study on the Pareto Principle, $t(856) = -58.383$, $p = .000$. There is significant evidence to reject the null hypothesis, therefore we conclude that the distribution of observed faults could not be explained by the size of the files and that at most 29% of the total lines of source code of the system system resides in the 20% of the most faulty files. Further, Cohen's effect size value ($d = .13$) suggested a small practical significance.

Testing the fault distribution inequality. A one-tailed one-sample t-test was conducted to determine if a statistically significant difference ($\alpha \leq .05$) existed between the fault distribution inequality from our sample of 1000 popular GitHub projects and the observations from our literature study. For the third hypothesis (H3), the null hypothesis was formulated $H_0 : \mu < .5$. The Gini coefficient of the systems we analysed was more than .5 ($M = .54$, $SD = .119$) than the systems from our literature study on the Pareto Principle, $t(856) = 9.822, p = .000$. There is significant evidence to reject the null hypothesis, therefore we conclude that the distribution of observed faults is an unbalanced one and that the Gini coefficient of the distribution of observed system faults is at least .5. Furthermore, Cohen's effect size value ($d = .34$) suggested a moderate practical significance. The results are summarized in a histogram (see Figure **??**). Instead of frequency, the probability density is shown, values on the x-axis are allocated to 60 bins, the kernel density function is plotted over the histogram to give a more clear view of the shape (not restricted to the number of bins).

3.5 Conclusion and Discussion

One of the objectives of this preliminary research was to find out if the Pareto Principle could be applied to the fault distribution in software system. If it is the case that software systems adhere to the Pareto Principle, it means that an useful ordering can be made in such a way that a large part of the faults can be discovered. This assumption is one of the main reasons why researcher invest time and resources in fault-proneness prediction modelling. Our results provides strong support for the claim that distributions of faults in software systems adhere to the Pareto Principle; on average 20% of the classes contained 77% of the faults. Moreover, we found evidence that the most faulty classes do not contain most of the system's code; only 26% of the code resided in the 20% most faulty classes.

Besides replicating Fenton & Ohlsson their qualitative analysis [32], a new metric was used for measuring the inequality of fault distributions based on the measure of economic inequality, the Gini coefficient [38]. The results using the Gini coefficient measure were in line with the $20 - n$ measure and supported the claim that fault are unequally distributed over the system. We prefer the use of the Gini-coefficient to measure fault-distributions over using the $20 - n$ rule. One limitation of the $20 - n$ rule can be seen in the histogram in Appendix B. A lot of observations are in the right most bar, meaning that a lot of the systems in the dataset had 100% of the faults in at most 20% of the modules. At this point, information is lost, it could be that the faults are located in 20% of the classes, in 1% of the classes, or anything in between. Not only in the latter case is information lost, by fixing one of the variables to 20%, the information of fault distribution itself is lost. The Gini coefficient does not has this limitation because it does not fixate on a single variable.

3.5.1 Threats to validity

A threat to the internal validity resides in the fault classification process. The classification of faults is subjective and depends on the way faults are reported and how they are collected again. Moreover, a fault is not always a fault, even if it is reported as a fault. Contrariwise, it could be the case that a fault is never formally reported and could be fixed on the spot. The result is that the observations are biased towards the faults we observed. The hypotheses only tells something about the observed faults. In other words, the outcome to the hypotheses could differ if another fault classifier is used. To mitigate this threat, we tried to create awareness by explicitly refering to the observed faults to press that the classification process is subjective.

Other threats to the internal validity could be caused by the instrumentation used for the calculations and collection of the data. We build the tool explicitly for this research and it has not been validated by an external third party. However, we did test and pilot the tool, but it could be that in edge case scenarios the tool fails. To cope with threats related to instrumentation, we published the source code and the raw output of the analysis of the dataset so it can be validated by external parties. Moreover, the systems in the dataset are all accessible and the exact state of the system can be restored using the provided commit id.

3.5.2 Future research

Most of the current research on fault distribution analyses systems on module-level, we analysed our dataset on a finer level of granularity, on file-level. We think this level of granularity is enough to give an indication of the fault distribution regarding software systems and that these finding do not change significantly if lower level of granularity are used. However, this is a speculation and we could not test this is because of the size of the dataset and the time complexity of analysing fault distribution of class-level. It is interesting to see if this speculation is true because fault-proneness prediction is preferably done on class-level, and knowing the fault distribution on class-level provides the best accuracy [3].

In our research we applied a new measure to represent fault distribution inequality, one that is much more expressive compared to the Pareto Principle ($20-x$ rule). We calculated the accumulated faults with a dx the size of files. Although, this tells something about the inequality of the fault distribution, it does not provide insights in the correlation of the number of faults with the size of these files. A second hypothesis has to be tested to cover the latter. The expressive power of the Gini coefficient could be improved if not the files are accumulated but the source lines of code of the files are accumulated instead. The result is that the area under the Lorenz curve becomes larger (Gini coefficient increases) if the files with the most faults also contain the most lines of code; the area of the Lorenz curve remain unaffected, relative to counting the files, if the lines of code are equally divided over the files. The way the Gini coefficient is affected by the result of the source lines of code within the files is interesting. But in the case if the files with the lowest number of faults contain more lines of code, that is an inverse correlation between lines of code and faults, the area under the Lorenz curve increases (Gini coefficient decreases); this is an unwanted side effect which obscures the actual measurement. Nevertheless, it is still interesting to investigate if the Gini coefficient could not only replace the first hypothesis of this study (H1) but also the second hypothesis (H2).

We used a heuristic way of measuring faults, that is by the means of semantically analysing the commit messages. Because 933 system used the GitHub issue tracker and our semantic analyser detect faults if a commit message formally closes an issue in the issue tracker through a commit, we think that we have most of the faults. However, there is a change of over fitting. Based on the observations by Herzig et al. [29], most of the detected faults are not faults at all. For future research is could be interesting to replicate this study but with a more accurate fault discovery strategy. As suggested by Herzig et al., the discovered faults should be verified manually.

Chapter 4

Reassessing the Applicability of Fault-Proneness Prediction Models Across Software Systems

4.1 Introduction

As shown in our preliminary research (see Chapter 3), the fault distributions of software systems seems to be unequally balanced. Fault-proneness prediction models could be used to exploit this knowledge, and deduce the part of the system that is most fault-prone.

Software fault-proneness prediction models have been put to the test by various researchers and hold promising results. In our literature study on regression-based fault-proneness prediction models, we found that the models were able to find on average 79% of all the faults within a system with a precision of 78% (see Section 1.3). However, almost all of the models from our literature study are probably less useful in practice, because the models were not applied under realistic conditions. These models were validated on the same system as they were build on. A more realistic scenario would be a model trained using one or more systems and used on a set of similar systems. However, there is little knowledge about the effectiveness of fault-proneness prediction models used across systems.

Only two studies from our literature study validated the prediction model accuracy across systems, Schneidewind [7] and Briand et al.[4]. Scheidewind did not focused on cross-system prediction models and paid little attention to that matter. Briand et al. on the other hand, dedicated a significant part of his research to cross-system prediction models, and claimed that they successfully build a cross-system fault-proneness prediction model with reasonable precision and recall (see Section 1.2). However, Briand et al. their study regarding cross-system fault-proneness prediction models was of an exploratory nature and the conclusions were based on a single observation. Therefore, we see the need to validate the findings of Briand et al. and contribute to the body of empirical knowledge of cross-system fault-proneness prediction modelling. A replication study will be conducted; the following hypothesis will be used for this purpose:

> *H1: A fault-proneness prediction model that is trained on a system and validated on another system with the same team composition will have at least an accuracy and recall of 60%.*

The percentage used in the hypothesis is taken from the initial study where the cross-system model obtained precision and recall of both 60%. Briand et al. explicitly mentioned that they kept the team composition constant; this is reflected our hypothesis.

Hypothesis *H1* enables us to compare Briand et al. their findings with ours and draw conclusions based on the outcome. But, it provides no insight on how transferring a model from system A to system B affects model accuracy. Consider the following case: A model has an accuracy of 90% when applied on the same system it was trained on, but when used on another system, the accuracy drops to 65%. The absolute prediction capabilities of the model looks promising, but the accuracy drop

tells otherwise. To also get some insights in the accuracy loss while a prediction model is transferred, we state another hypothesis:

> *H2: A fault-proneness prediction model that is trained of a system and validated on another system with the same team composition has the same prediction accuracy when trained and validated on a single system.*

This chapter is structured as follows: Section 4.2 describes the set up of the study. Section 4.3 lays out the analysis methodology and how the initial study is replicated. In Section 4.4 we present our findings. In Section 4.5 we discuss our findings and draw our conclusions.

4.2 Description of Study Setting

In this section we describe the system selection strategy and which systems we picked for our dataset. Furthermore, we describe the data we collected from the systems and which procedure we used. Because this is a replication study, we try to describe each decision and how it relates to the initial study.

4.2.1 Systems

The initial study used two Java systems: XPosed and JWriter. Those systems are both closed source and could therefore not be used in this replication study. The provided characteristics of the two systems from the original dataset were used to choose similar systems. The systems consisted out of 144 and 68 Java classes respectively, were developed by the same team, and both came from the same company. We will select system with similar characteristics.

In total we collected 10 systems. The systems were build by company that propagates strict protocols and standards regarding software development (the company is kept anonymous on purpose). All systems are still maintained and actively developed during the period of analysis. Most of the systems in our dataset are smaller compared to the systems in Briand et al. their dataset. However, the size of the systems used in the initial study and the size of the systems in this dataset are hard to compare, because we filtered out all the files that are generated or related to testing [1].

Because the dataset will be reused in this thesis and want to minimize observation bias, we randomly selected 6 systems from the data that we will use for replication means. The set of all systems, and the systems selected for this study (labelled with *) are shown in Table 4.1. For each system: the number of non-generated/non-test Java source code files (Files), the number of source lines of code (SLOC), the number of faults (Faults), the fault distribution (FGini), and the team who build the system (Team) are shown in the table.

Table 4.1: Available systems overview

System	Files	SLOC	Faults	FGini	Team
*BIra**	30	3.023	8	.45	C
Doc	83	2.666	86	.41	B
Sec	100	4.511	254	.48	E
IRat	22	781	0	.51	D
*MAdd**	80	4.012	38	.56	D
MApp	15	579	6	.42	D
*MIntO**	46	2.204	93	.45	B
*MIra**	28	1.186	29	.38	D
*MPen**	27	1.217	17	.59	B
*MRep**	85	2.603	72	.44	B

* system included in the single-system dataset

Another dataset must be created that comprises the system pairs that have a *same team composition* relationship. The system pairs in the dataset must be unique couples $((a, b) = (b, a))$ where the largest

[1] On average, the number of files had shrunk with a factor three after the exclusion of test and generated files

system of the pair, in terms of number of files, will be used as training data; the other will be used to validate the prediction model. See Table 4.2 for an overview of all system pairs. The training system (Train), validation (Validate), and the team who build both systems (Team) can be found the table.

Table 4.2: Between-systems overview

System pair	Training	Validation	Team
MAdd-MIra	*MAdd*	*MIra*	D
MRep-MIntO	*MRep*	*MIntO*	B
MRep-MPen	*MRep*	*MPen*	B
MIntO-MPen	*MIntO*	*MPen*	B

4.2.2 Variables

Independent variables

The metric-suite used by Briand et al. was not made available for the public. Moreover, the implementation of the metrics are not described. Therefore, a new metric-suite was build containing the same metrics as used in the initial study, except from a couple of simple size measures. The only difference between our metrics and the metrics used by Briand et al. is at implementation level.

The metric-suite contains a subset of Briand et al. their cohesion metrics [25], Benlarbi & Melo their polymorphism measures [12], all metrics from the Chidamber & Kemerer metric suite [13], all metrics proposed by Li & Henry [26], and some size metrics. The implementation of the metrics are described in detail (see Appendix A) and the tool is open-source and publicly available[2]. In the initial study, Briand et al. uses a total of four size measures (*totatrib*, *totprivatrib*, *totmethod*, *totprivmethod*), we measure *totmethod* using *NOM*.

All the metrics measure at class-level. Inner classes are not treated as individual observations but their measures, methods, and attributes are counted to contribute towards the containing class. Also, faults that traced back to an inner class were assigned to the out most containing class. This way measurement is also used in the initial study.

Dependent variable

We want to evaluate whether the existing design measures are useful for predicting the likelihood that a class is fault-prone; that is if the class contains at least one fault. The outcome value is dichotomous, a class is either fault-prone or it is not.

A class is fault-prone if it at least contains one fault. We collect faults in a class using our own tool, *gcrawler*. The tool analysis git commit messages; if the commit message contains hints of a fault that has been fixed, then we say that all changed classes contain one fault each (see Appendix A).

The latter procedure is not ideal but had to be adopted due restricted data access and the absence of issue-trackers. Take in mind that the faults that were detected using this procedure are probably not all of the faults. Also, there might be some false positives among the observed faults. To roughly check if the detected faults were actual faults, we randomly picked three systems from the dataset. From these systems we collected the first 10 commit messages that contained the word 'fix', 'fixed', or 'fixes' (see Table 4.3). Next, a software architect, that was involved during development of the system, analysed the listed commits and matching changes. He labelled each commit as a commit that fixed a fault or a commit that did not fixed an actual fault (e.g. code refactoring, feature improvement).

[2] https://github.com/scrot/mt

Table 4.3: Commit messages of observed faults in dataset

Commit message	Fault-fixing commit
Fix unittest on maven by setting UTF-8	Yes
Merge branch 'Update_version_and_fix_double_package_name'	Yes
Merge branch 'fixDeserializeBugFortify'	Yes
Merge branch 'Fortify_fixes'	Yes
Fix config docblocs, default	Yes
Added fix for problematic retry of retrieval of requests	Yes
Fixed small bug in Config.java	Yes
Merge branch 'fix-graphite-logging' into develop	Yes
CODE Fixed graphite jndi's.	Yes
Fortify fixes	Yes
CODE Corrected Toolkit dependency versions.*	Yes

Based on the results of this sample, we will make the assumption that our procedure will find actual faults. However, not all faults will be discovered. For example, in Table 4.3 there is one commit message with the word 'corrected' in it (labelled with a star). Our tool does not classify this commit as a fault-fixing commit, but the commit fixes an actual fault according to the software architect who analysed the commit.

4.3 Data Analysis Methodology

In this section we layout the methodology used for analysing the data. The analysis procedure consists of: (i) An analysis of descriptive statistics, (ii) data distribution and outliers analysis, (iii) a principle component analysis, (iv) a multivariate regression analysis, (v) and an evaluation of the prediction model.

4.3.1 Descriptive statistics

For each system, all metrics are calculated. The minimum (Min), maximum (Max), sample mean (μ), Median (Med), and the standard deviation (σ) are collected of each metric. This data will help with the interpretation of the results for upcoming analyses.

4.3.2 Outlier analysis

The outlier analysis is used to spot low variance measures. Outliers are removed from the dataset. The outlier analysis is done following the same two steps as described by Briand et al. in the initial study:

- All measures that do not have more than 5 non-zero data points are removed for analysis. According to Briand et al. these measures do not differentiate classes very well and therefore are not likely to be useful predictors for fault-proneness.
- Multivariate outliers are removed. To calculate the distance of the data point in this multi dimensional space the Mahalanobis distance from the sample space centroid is calculated. If the distance of the data point from the centroid is too large, it is removed from the dataset. In the initial study there is no mentioning of the cut-off value, also it is unclear if the Mahalanobis distance or the Jackknife distance was used (see Section 2.6). In this study, for each observation in each system, the outlier distance is measured using squared robust Mahalanobis distances, the outliers were detected using the 97,5%-quantile.

4.3.3 Principle component analysis

The Principal Component Analysis (PCA) is a method of analysis which involves finding the linear combination of a set of variables that has maximum variance and removing its effect; repeating this

successively. PCA is used for finding metrics that are likely to measures the same underlying concept. The result of this analysis is used as input during construction of the prediction model.

For identifying the Principal Components (PCs) and the variables with high loadings we will use varimax rotated components, following the method described by Briand et al. (see Section 2.4 for more information). Rather than selecting all n PCs, we consider only the PCs whose eigenvalue is larger than 1.0.

Briand et al. suggested that it would be interesting for a replication study to see which dimensions would also be observable in other systems, and to find possible explanations for differences in the result. They expected to see consistent trends across systems for the strong PCs which explain a large percentage of the dataset variance. We will compare the findings of Briand et al. their PCA with ours and reflect on their expectations.

The Principal Component Analysis will be done using R and the *principal* function from the *psych* package[3]. Varimaxi rotations will be applied, requesting the maximum number of principal components. We only consider the Principal Components that have an eigenvalue larger than 1.0.

4.3.4 Prediction model construction

In this study, a logical regression model will be used to classify classes as fault-prone or not fault-prone. Logistic regression is a standard technique based on maximum likelihood estimation for estimating the likelihood that an event will occur (see Section 2.5). The selection of metrics described in Section 4.2 will be used as predictors and the binomial value, fault-proneness, will be used as outcome variable [4].

The goal is to build a prediction model with the best accuracy possible using a minimum number of predictors. The strategy to achieve this goal consists of two parts:

- *Minimize the number of independent variables in the model.* Too many independent variables negatively affects the estimated standard error of the model's prediction and makes it more dependent on the data. To minimize the number of independent variables in the model, Briand et al. used forward selection with significance levels for entering and exiting the model of $\alpha_{enter} = .05$ and $\alpha_{exit} = 0.10$, and tested the significance of a variable by using a log-likelihood ratio test. Instead of using only forward selection we use both forward and backward selection. This is because Briand et al. also intended to use both strategies, but could not fit the backwards selection function to the data. Moreover, we will use a AIC test instead of the log-likelihood ratio test to compare the models; both tests produce the same results only log-likelihood is only valid for nested models, whereas AIC has no such restrictions. The lower the AIC the lower the loss of information, relative to the actual model.

- *Reduce multi-collinearity in the model.* Reduce the number of predictors which are highly correlated. The result is that the model is more easy to interpret. Briand et al. used the predictors their eigenvalues from the Principal Component Analysis as a conditional number and excluded all predictors whom's conditional number exceeded 30. However in the initial study,no significant difference was observed as result of preselecting predictors using PCA as a heuristic. Briand et al. chose not to use PCA to preselect the predictors at all. Based on this decision, we will also not use the PCA to exclude variables.

The result of the model selection procedure is a logical regression model with the best AIC score. This model is used for further analysis.

On implementation level, we will make use of the software platform for statistical computing and graphics, named *R*[5]. The model is build using *glm*[6] and is used to fit the logistic model. The 'binominal' family is used to describe the link function and error distribution. A binary value to indicate fault-proneness (the class contains at least a single fault or non at all) is used as outcome

[3] https://cran.r-project.org/web/packages/psych/index.html

[4] In the initial study, linear regression and logical regression are mixed up. However, these methods differ significantly from each other and only one of the two is actually used in the study. Based on their exploratory study [14] where is referred to by the initial study and the scale of their dependent variable, we assume they used logical regression instead of linear regression.

[5] https://www.r-project.org

[6] https://stat.ethz.ch/R-manual/R-devel/library/stats/html/glm.html

variable. All other variables of the dataset are used as independent variables. For the stepwise model selection method we used *stepAIC*[7] from the *MASS* package. The optimal model is selected using the forward selection strategy and the generalized Akaike a Information Criterion (AIC) for a fitted parametric model. AIC is a relative and generalized method of comparing fitted parametric models and is similar to the log-likelihood ratio. *stepAIC* takes a parametrized *glm* as input.

4.3.5 Model evaluation

In the initial study, the model's prediction capabilities were measured in terms of precision and recall. We included another measure, called accuracy, because this measures tells more about the overall effectiveness of the model.

- *Precision.* Precision is defined as the number of classes correctly classified by the model divided by the total number of classes that are classified as fault-prone by the model. A low precision means that a lot of fault-prone classes identified by the model are false positives.
- *Recall.* Recall is defined as the number of faults in classes classified as fault-prone divided by the actual number of faults in the system. A low recall indicates that a lot of faults were not detected by the model.
- *Accuracy.* Accuracy is defined as the number of correctly classified fault-prone and non fault-prone classes divided by all classes. A low accuracy means that the model incorrectly classifies classes as fault-prone or not fault-prone.

To perform these measures, the trained models must be applied to a new dataset and the results compared to the actual observations. To answer Hypothesis *H1*, only the cross-system precision and recall scores are needed, and could be obtained using a cross-system validation technique. To answer Hypothesis *H2*, the cross-system precision and recall scores as well as the single-system precision and recall scores are needed. To obtain the two measures, two techniques will be used (for more information about those techniques see Section 2.5):

- *Single-system validation.* To establish a base rate, the prediction capabilities of the fault-proneness prediction models are validated using a single-system, that is the same system it was trained with. For this purpose we use k-fold cross-validation. This method builds k models, each model is trained using $k - 1/k$ of the data and validated on $1/k$ of the data; the sets are mutually exclusive. On implementation level we will use R and the *cv.binary* function from the *DAAG* package[8] to perform the single-system validations. The function uses k-fold cross-validation, we configured the function to use 10 folds and to take a binary *glm* function as input.
- *Cross-system validation.* To uncover the prediction capabilities of the fault-proneness prediction models across systems, we will use cross-system validation. The model is trained using system A and validated using system B; the pairs used for the cross-system validation can be found in Table 4.2. Only the systems that occur in the cross-system dataset as validation system are included for analysis in the single-system group. On implementation level we will use R and the *prediction.glm* function from the *stats* package[9]. As the prediction model output scale (type) we choose the response scale, this is to correctly handle dichotomous outcome variables.

4.3.6 Hypothesis testing

In order to answer our research question we will test the hypotheses stated in Section 4.1:

- Hypothesis *H1.* In order to test our first hypothesis for statistical significance, we conducted two one-tailed one-sample t-tests, using an alpha value of $\alpha \leq .05$, on the model validation results of the cross-system dataset. The first t-test will be used to check the significance of the

[7] https://stat.ethz.ch/R-manual/R-devel/library/MASS/html/stepAIC.html
[8] https://cran.r-project.org/web/packages/DAAG/index.html
[9] https://stat.ethz.ch/R-manual/R-devel/library/stats/html/predict.glm.html

hypothesis in terms of precision, and another t-test will be used to check the significance of the hypothesis in terms of recall. Both the null hypotheses need to be rejected in order to reject Hypothesis H1. To analyse the effect size we will use Cohen's d.

- Hypothesis *H2*. To test the second hypothesis, we will use two Weld independent-sample t-tests (with an alpha value $\alpha \leq .05$). The first t-test will be used to test the hypothesis in terms of precision, the other t-test will be used to test the hypothesis in terms of recall. The mean precision/recall of the single-system group is compared to the mean precision/recall of the cross-system group. Only the systems that occur in the cross-system dataset as validation system are included for analysis in the single-system group.

4.4 Analysis Results

4.4.1 Descriptive statistics

Appendix **??** contains the descriptive statistics of the analysed dataset. For each metrics the number of observation (N), mean (Mean), standard deviation (St. Dev.), minimum value (Min), and maximum value (Max) are stated.

In all dataset we see that the use of inheritance is sparse (low mean values of DIT and NOC), this was also observed by Briand et al. As a result, all ancestor/descendant measures (e.g. ACXXC/DCXXC), and polymorphism metrics (e.g. OVO, SPX, DPX) also have low means. Another metric with a low mean is the lack of cohesion measure (LCOM). The reason could be that most classes are cohesive, that is have a negative LCOM value, and that these value are rounded up to zero. All the non-inheritance related measures differentiate the data rather well (e.g. OCXXC, NIP).

4.4.2 Outlier Analysis

Outliers and measures with low variance were identified and removed from the dataset.

- *Exclusion of weak differentiators.* In all projects the following metrics were removed: *NOC*, *LCOM*, *ACAIC*, *DCAIC*, *ACMIC*, *ACMEC*, *DCMIC*, *SPA*, *SPD*, *DPA*, and *DPD*. These metrics do not differentiate the data enough. This is in line with the observations from the initial study. The *DAC* metric was a weak differentiator in two systems and *OCMEC* in one system. All the other measures differentiated the data enough to be included in all the systems and are considered in the selection process. The metrics that were selected are: *WMC*, *DIT*, *CBO*, *RFC*, *DAC*, *MPC*, *NOM*, *SIZE1*, *SIZE2*, *OCAEC*, *OCAIC*, *OCMIC*, *OCMEC* and *NIP*. Note that *DAC* and *OCMEC* are also included. They did not differentiate the data enough in two systems, but for the larger part the systems they did.

- *Exclusion of multivariate outliers.* Out of the six systems only one system was analysed, the rest of the systems were computational or exactly singular and could therefore not be analysed. This is because some variables had near linear dependencies, resulting in the matrix to exceed the reciprocal condition threshold. We would be able to resolve this issue by starting with a PCA and remove co-linear variables, but we choose to do not defer from the method as described by Briand et al. Moreover, the number of observation in some systems are too small for an outlier analysis, making the detection of real outliers more complicated. We ignore two probable outliers that were discovered in the one system that was analysed.

4.4.3 Principal component analysis

The results of the Principal Component Analysis can be found in Appendix C. All measures with a Principal Component that has an eigenvalue larger than .7 are highlighted in the appendix.

In the initial study, Briand et al. found 6 principle components and interpreted them as follows:

- PC1: Class size in terms of attributes and methods.

- PC2: The number of children / descendants of a class.

- PC3: Amount of polymorphism taking place.
- PC4: Export coupling to other classes.
- PC5: Import coupling from ancestor classes.
- PC6: Overloading in standalone classes.

Based on our results of the Principal Component Analysis of 5 systems and the loadings of each measure we found the 5 Principal Components. The components are in descending order based on amount of variance they explain:

- PC1: Import coupling of classes. In 3 out of the 5 systems, class coupling explained most of the variance in the data. In the remaining two systems, the 'coupling' Principle Component had an eigenvalue larger than 1.0. Besides the CBO metric (measures import and export coupling), the metrics with the highest loadings are all import coupling metrics: RFC, MPC, DAC, OCAIC, OCMIC. Import coupling measures that differentiate type of relationships (e.g. attribute-to-class or method-to-class) all seem to measure on the same orthogonal space. In two of the systems, some of size measures (SIZE1, SIZE2, and NOM) were also among the variables of this principal component. Briand et al. mentioned also a recurring relationship between size and import coupling.

- PC2: Class method size and complexity. In 2 out of the 5 systems, size and complexity measures explain most of the variance. In one system, the size and complexity explain the second most of the variance. In the remaining two systems, size and complexity measures are among the Principle Component with an eigenvalue larger than 1.0. Simple size metrics (NOM, SIZE1, and SIZE2), and the complexity measure (WMC) seems to make up this principal component. In one system, NIP had also a high loading in this component.

- PC3: Export coupling to other classes. in all 5 systems, there is at least one Principle Component composed solely out of other-class export coupling metrics (OCAEC, OCMEC) with an eigenvalue higher than 1.0. In one of the systems, the OCAEC and OCMEC measures explain one component, but in all other systems they do not seem to measure along the same dimension.

- PC4: Depth of the inheritance tree (DIT). In all 5 systems, this component mostly explained by the DIT metric, had an eigenvalue larger than 1.0.

- PC5: Polymorphism in non-inheritance class relations. In 4 out of the 5 systems, a Principle Component were only NIP had a high loading was observed among the component with an eigenvalue larger than 1.0.

Comparing the Principal Components found by Briand et al. and the Principal Components found in this study, we see that in both analysis the size metrics measure a similar underlying concept and that this concept is somehow related to input coupling measures. Moreover, in both studies import and export coupling explained a large part of the variance but measures a different underlying concept. Most specialized coupling metrics measure all the same underlying concept and could probably be aggregated. The same holds for the polymorphism measures. Briand et al. their PC2 and PC6 were not observed in our dataset.

4.4.4 Prediction model construction

For each system in Table 4.2, the best logistic regression model was build using mixed stepwise selection based on AIC of the models. An overview of the selected predictors per system and the model's AIC score are shown in Table 4.4.

Briand et al. mentioned that they were unable to use backward stepwise selection because the data could not be fitted. We did not experience that problem and instead of using only forward selection we use both backward and forward selection.

Table 4.4: Dataset overview

System	Selected predictors	AIC
MAdd	WMC, DIT, SIZE2	41.42
MIntO	DIT, CBO, SIZE1, SIZE2, OCAIC	14.77
MIra	OCMEC	4.00
MPen	RFC, DAC, MPC, NOM, OCMIC, NIP	17.82
MRep	MPC, SIZE1, SIZE2	89.22

The set of predictors that resulted in the best fault-proneness prediction model differ from the findings in Briand et al. their research and the other studies in Table 2.1. It seems that a different context result in a different set of fault-proneness predictors. This could oppose a challenge when fault-prediction models are applied across systems, especially when they reside in different contexts.

The inclusion of simple size metrics seems to result in the best possible model. Note that the best possible model does not mean it has good prediction capabilities. In 4 out of the 5 models, a size metric was included in the set of independent variables (NOM, SIZE1, SIZE2). We see the same in other studies [24, 5, 7, 8]. Coupling metrics are also often included in our best possible models (CBO, RFC, MPC, DAC, OCAIC, OCMIC, OCMEC). This is also observed in Briand et al. their studies [14, 4]. Beside the most prominent quality measures, we find an inheritance metric in two models (DIT), a complexity metric in one model (WMC), and a polymorphism measure in one model (NIP).

Because we have no base-line for the AIC scores at this point, and the AIC scores are relative values; we could not compare the information-loss of the models at this stage of our research.

4.4.5 Model validation

For all prediction models the precision, recall, and accuracy were calculated and discussed. The model evaluation phase consists out of two parts: (i) The validation of the prediction model using a single system (single-system validation); (ii) and the validation of the prediction model using another system (cross-system validation).

Single-system validation

For the result of the analysis, see Table 4.5. For each system, the true positives ($T+$), true negatives($T-$), false positives ($F+$), false negatives ($F-$), accuracy, precision , and recall were calculated. Note that the accuracy, precision, and recall are the averages of the k models produced by the k-fold cross validation method. Moreover, the true/false positives/negatives are the aggregated results of the k models; every model predicted $1/k$ of all observations.

Table 4.5: Single-systems overview

System	$T+$	$T-$	$F+$	$F-$	Accuracy	Precision	Recall
MAdd	3	33	2	6	.82	.60	.33
MIntO	22	6	1	0	.97	.96	1.00
MIra	12	5	0	0	1.00	1.00	1.00
MPen	8	14	1	0	.96	.89	1.00
MRep	20	9	6	9	.64	.69	.64

The precision and recall of the single-system prediction models are in line with the observations in the initial study and the results from the literature study. the model constructed using *MIra* was completely accurate, while the prediction model *Madd* was one of the worst models. Interestingly is that they where developed by the same team and were of comparable sizes. This and the dissimilarity of predictors could be indicative of problems with applying model across systems, even with similar teams and system sizes.

Cross-system validation

For the result of the analysis, see Table 4.6. For each system pair (coded as training system - validation system), the true positives ($T+$), true negatives($T-$), false positives ($F+$), false negatives ($F-$), accuracy, precision , and recall were calculated.

Table 4.6: Cross-systems overview

System pair	$T+$	$T-$	$F+$	$F-$	Accuracy	Precision	Recall
MAdd-MIra	1	5	0	11	.35	1.00	.08
MIntO-MPen	8	14	1	0	.96	.89	1.00
MRep-MIntO	8	2	3	4	.55	.80	.55
MRep-MPen	4	2	9	2	.43	.35	.75

If we look at the accuracy of the cross-system prediction models, we see that three out of the four preform no better than random. This contradicts the findings of Briand et al., who found a precision and recall that were notably better than random.

4.4.6 Hypothesis testing

Using the results from the model evaluation phase, we will answer the two hypotheses. The results are discussed in Section 4.5. To aid the hypotheses testing, the descriptive statistics are given in Table 4.7 and Table 4.8. The tables include the number of observations (N), minimum value (Min), maximum value (Max), mean (Mean), and standard deviation (St. Dev.) for the single-system dataset and the cross-system dataset, respectively.

Table 4.7: Descriptive statistics of the single-system prediction models

Statistic	N	Mean	St. Dev.	Min	Max
$T+$	5	13.00	8.00	3	22
$T-$	5	13.40	11.50	5	33
$F+$	5	2.00	2.35	0	6
$F-$	5	3.00	4.24	0	9
Accuracy	5	0.88	0.15	0.64	1.00
Precision	5	0.83	0.17	0.60	1.00
Recall	5	0.79	0.30	0.33	1.00

Table 4.8: Descriptive statistics of the cross-system prediction models

Statistic	N	Mean	St. Dev.	Min	Max
$T+$	4	5.25	3.40	1	8
$T-$	4	5.75	5.68	2	14
$F+$	4	3.25	4.03	0	9
$F-$	4	4.25	4.79	0	11
Accuracy	4	0.57	0.27	0.35	0.96
Precision	4	0.76	0.29	0.35	1.00
Recall	4	0.60	0.39	0.08	1.00

Testing the cross-system prediction capabilities. Two one-tailed one-sample t-test were conducted to determine if the precision and recall of our four fault-proneness prediction models are at least as high as Briand et al. their fault-proneness prediction model ($p \leq .05$). For the first hypothesis (H1), the null hypothesis was formulated as follows:

$$H_0 : \mu_{precision} < .60 \vee \mu_{recall} < .60 \tag{4.1}$$

The average precision of the cross-system fault-proneness prediction models was more than 60% (M = .76, SD = .29), but not significant ($t(3) = 1.122$, p = .172). Cohen's d shows a large effect (d = 2.621). For recall, the mean was 60% (M = .60, SD = .39), the result was not significant ($t(3) = -0.137$, p = .509). The null hypothesis: A fault-proneness prediction model that is trained on a system and validated on another system with the same team composition will have an accuracy and recall less than 60%, is not rejected.

Testing the accuracy loss caused by model transferring. Two Welch independent-sample t-tests were performed to determine if there is a significant difference ($p \leq .05$) between the prediction accuracy of fault-proneness prediction models that are used on a single system and models that are used across systems. For the single-system dataset, we only included the systems that were also used for training the cross-system models. For the second hypothesis (H2), the null hypothesis was formulated:

$$H_0 : \mu_{single\text{-}system} \neq \mu_{cross\text{-}system} \tag{4.2}$$

The difference between the average accuracy of the single-system models (M = .88, SD = .15) compared to the cross-system models (M = .57, SD = .27) was not significant ($t(4.45) = -2.020$, p = .106) and the null hypothesis was not rejected. See Table 4.9 for an overview of the accuracy losses.

Table 4.9: Between-system prediction capabilities losses

Within-system	Between-systems	Accuracy Loss
MAdd	*MAdd-MIra*	.47
MIntO	*MIntO-MPen*	.01
MRep	*MRep-MIntO*	.09
MRep	*MRep-MPen*	.21

4.5 Conclusion and Discussion

This replication study was done to answer one of our research questions (RQ3): Can fault-proneness prediction models effectively be used across systems. Briand et al. concluded that fault-proneness prediction models could indeed effectively be used across systems, but that it is far from straightforward; after our replication study we did not came to the same conclusion. We found no evidence that our fault-proneness prediction model obtained precisions and accuracies significantly higher than .60; most of the models performed not better than random. Moreover, the prediction models lost a considerable part of their accuracy, on average 19.50%, when used across systems. At this point, we conclude that fault-proneness prediction models cannot be effectively applied across systems that only share a common team-composition.

From the results of the Principal Component Analysis, one notable observation was done: import coupling measures seems to measure along the same orthogonal dimension as size measures. One explanation could be that most of the code that is written, in the systems we considered, are calls to other classes or libraries. It could be that these systems are mainly made by tying together existing libraries and that not much code was written from scratch.

One of the observations based on the results of the prediction model construction was that size was included in almost all the best possible models (sometime even more than once); this was also the case in most of studies from the literature study. It seems that the size of the class plays an important roll in the number of faults produces and tends to be a good predictor for fault-proneness. A possible explanation could be that the number of faults increases with the amount of code produced. Another explanation could be because large size and high complexity result in classes that are hard to comprehend, and therefore results in more faults. Note that the complexity of the code is almost never among the predictors of the best models, even though it tends to measure along the same dimension as size measures. Coupling related variables were also often among the variables of our best models and also among the best predictors in the studies of by Briand et al. One possible reason for these type of measures to be good fault-proneness predictors could be because they provide some sort of

indication of how many context a person has to keep track of. Another reason could be that faults in coupled classes triggers a snowball effect. If a class c depends on another class d and class d contains a fault, then it is possible that class c is also affected and needs fixing. Yet another reason could simply be because coupling metrics measure along the same dimension as size measures, which are probably good fault-proneness predictors.

The single-system model validation results were in line with the studies done by Briand et al. and other studies from our literature study. The average accuracy of 5 fault-proneness prediction models, applied on the system they were build on was 88%, with a precision of 83% and a recall of 79%. The cross-system validation results were not in line with the studies done by Briand et al. The 4 fault-proneness prediction models obtained on average an accuracy of 57%, with a precision of 76% and a recall of 60%. Three out of the four cross-system models performed on average not better than random.

Based on the assessment of various cross-system fault-proneness prediction models, we think that there are three important factors that influence the accuracy of fault-proneness prediction models based on regression analysis techniques:

- *Included predictors.* The predictors or independent variables directly influence the logit function regression model. A regression model is fit by applying weight to the available predictors. The number of predictors are important; too much predictors will overfit the data and too few predictors will not be able to discover patterns. Beside the number of predictors, the selected predictors plays an equally import role; if the predictors do not correlate with fault-proneness then the model is not likely to reveal relevant patterns. In other words, an optimal subset of measures that strongly correlates with the outcome variable is a prerequisite for an accurate regression-based prediction model.

- *Model construction method.* During the construction of the prediction models we were confronted with many decisions: How and when to measure the class properties, which variables to select for the model, which systems to train the model on, the values of function parameters (e.g. cut-off value for prediction outcome), and even the shape of the logit function itself (e.g. does the function needs to fit a Poisson, a logical, or a linear distribution; or is the relationship between predictors and outcome variables even more complex, see MARS model used by Briand et al[4]). An alternative approach for constructing the model could lead to a different model with different prediction capabilities.

- *Environmental and system factors.* Looking at the single-system prediction models we observe high accuracies, and but when transferred to another system their accuracies drop considerably. Intuitively, we think that to build effective fault-proneness prediction models that could be applied across systems, environmental and system factors play an important role and needs to be controlled. If we control all factors, it will result in a single-system model, but this is an impossible task. However, if we succeed in controlling the right factors, an large accuracy loss might be prevented.

4.5.1 Threats to validity

The way regression models work poses a threat to external validity. Because, the model chooses its logit function in such a way it fits the training data best, it could be hard to replicate previous studies or generalize findings. The regression model are partly fit to the data, therefore the conclusion drawn from the output of these models are also partly bound to the data. Statements like: "*size measures are good fault-proneness predictors*", are dangerous if they are based solely on the model evaluation results. To deal with this threat, we did not draw our conclusion based on a single model and verified our findings by looking at more than one context. However, we trained every model with the data of a single system instead of using a pool of different systems.

Errors made by the tools we used or misinterpretations of their results posses a threat to the internal validity. The used tools forms a threat in particular because they are developed and validated by the author. To mitigate the risk, we explicitly describe the implementation of the metrics, wrote unit test for all metrics, and published the source code of all tools.

Furthermore, our reasoning is based on the observed faults; the faults discovered by applying a certain strategy. Other strategies could lead to a different set of observed faults and therefore might lead to other observations.

Another threat related to the external validity could be the method of sampling. We took a random sample from a population of specific context. The results we found are all within the boundaries of this context and it could be that our findings can not be generalized.

The rest of the threats are related to the dataset. Some systems in de dataset were relatively small in size; this could hinder the model construction, because it is hard to find relationships with too few observations. We dealt with the small sized systems by using k-fold cross-validation techniques, a technique designed for small datasets. Another issue was the sample size, due the low number of systems we were unable to test our hypothesis for significance and generalize our observations.

4.5.2 Future research

In the discussion, we described three factors that we think could dramatically impact the outcome of prediction models. Alongside these three dimensions, one could try to improve the prediction capabilities of a model. Moreover, it is known that regression-based models are black boxes, it is hard to deduce the factors that influence the model and even more difficult to find out why and how these factors impact the model's accuracy. By considering the factors in isolation and test them in a controlled environment, a lot could be learned about these factors and their relation to fault-proneness. When more is known about these factors and their effect on prediction accuracy, they can be more carefully applied in order to improve models.

In this study we found accuracy rates varying from .35 to 1.00. In the case of our worst models, they performs worse than by chance. For models that perform no better than by chance, we could easily tell that these models have no practical value. But what about fault-proneness prediction models with an accuracy of .60, is this an acceptable lower bound? It could be that a simple fault-proneness selection strategy based on trivial metrics such as size or last-change will easily outperform the much more complex regression model. It is interesting to find out what acceptable lower bound prediction capabilities are and how they would compare to prediction capabilities of trivial fault-proneness models. Briand et al. conducted a cost-benefit analysis that tried to estimate the practical value of the prediction models for example [4].

Although the fault-proneness prediction models perform rather well, even across systems with the same team composition, and there is room for improvement, the question is: are machine based fault-proneness prediction more effective than humans. It is interesting to see the difference in effectiveness of a human based fault-proneness selection strategy and a machine based strategy.

More and more software development is done in an agile manner, where software reviewing and testing is done on a regular basis. Not every class is reviewed as thoroughly as the other classes. The reviewer selects what he thinks are the most important classes and if the code seems good the pull request gets accepted. In this particular case, the selection of possible fault-prone classes is based on human intuition. This strategy is rather unstable depending on the case (e.g. a persons mood or experience in that domain). A machine based strategy might easily outperform humans in terms of stability. The question is: could fault-proneness prediction be used on only the set of classes that were changed.

Chapter 5

Improvements to Regression-Based Fault-Proneness Prediction Models

5.1 Introduction

In our replication study (see Chapter 4), we discuss three axes along which we possibly can improve fault-proneness prediction models for software systems: (i) Altering the fault-proneness predictors; (ii) Changing the fault-proneness prediction techniques; (iii) or by considering the environmental and system factors. In this study will we focus on the first two axes.

Based on our literature study (see Section 1.3) and the replication study (see Chapter 4); we like to suggest some improvements to Briand et al. their prediction model construction method [14, 4]. The two improvements are summarized below.

Predictor-set extension. The problem of most metric-suites from the literature study, is that they mainly consist out of product measures (in their research called quality metrics or design metrics), and therefore are limited in diversity. Moreover, almost all metric-suites contain multiple metrics that measure along the same dimension; introducing ambiguity and making the model harder to interpret. By reducing the number of similar metrics, the models will be easier to understand and will be easier to generalize. For example, three size measures that measure the same underlying concept could be replaced by one measure that describes this concept best. The improvement is twofold: (i) Replace metrics that measures the same underlying concept by one that measures this concept best; (ii) and add process metrics to the predictor-set. Both are described in more detail by the following paragraphs.

The metric-suite used by Briand et al.[4] is a rather large one (containing over 20 metrics), but it is limited in diversity. During the replication of their study (see Section 4), more than half of the metrics were filtered out because they did not explain enough of the variance. Also, the result of the Principal Component Analysis (see Section 4.4) showed that many of the remaining metrics measured the same underlying concept. In our opinion, the result was too homogeneous metric-suite, that in turn resulted in fault prediction models that were more tightly coupled to the individual systems and less stable when used on other systems. We think that one of the solutions to this problem is to combine some strongly related metrics and use that composite metric instead (e.g. not distinguish between ancestor, descendant, and other types of coupling). Based on the observations done in the replication study, we think the aggregation of the ACAIC, DCAIC, OCAIC, ACMIC, DCMIC, OCMIC into the metric XXXIC, still measures the import coupling of a class but is less finer grained. The same holds for the ACAEC, DCAEC, OCAEC, ACMEC, DCMEC, OCMEC metrics. The POLY measure will be an aggregation of the DP, SP, and OVO measure; the metric still measures polymorphism, but again on less fine grained. These composite measures are based on the conclusions drawn during the Principal Component Analysis; where there was only a clear difference between import coupling and export coupling measures but not between ancestor, descendant, or other types of couplings for example. It is likely that the new composite measures will differentiate the data better compared to the individual measures and that the model is more easy to interpret. However, there is a loss of information because

we aggregated the individual measures.

In an effort to make the model more diverse, process metrics could be added to the suite. Process metrics seems to be good fault-proneness predictors when looking at other related studies. For example, three out of the six metrics in Ostrand & Weyuker their 'optimal' *standard model* [27] were process metrics, and Khoshgoftaar et al. based their prediction solely on change metrics and obtained high prediction accuracy [22] .

State-aware measurement. To our knowledge, none of the researchers in the literature study truly consider state during measurement. In all studies we considered, measurements are done on the most recent version of the class. These measurements are then used as predictors for fault-prone classes in fault-proneness prediction models. The problem with this approach is that the measurements does not represent the fault-prone class, but instead a more recent version of that class. The probability that a newer version of the fault-prone class has been changed over time is high, and this probability is even higher if the class contained a fault. It is likely that the newer class does not resemble the fault-prone class any more. An even more problematic consequence would be that the newer version of the fault-prone class is fixed in such a way the measures represent a non fault-prone class instead of a fault-prone class; thus measuring the inverse of what was intended.

Briand et al. [4] also identified this issue and worked around it by not considering all faults but only the faults discovered in a specific release. As a result the difference between the analysed class and fault-prone class differ at most 3 a 4 months, according to Briand et al. This approach is more accurate that an approach which does not consider state, but it does not solve the problem. To solve this problem, we will analyse the fault-prone class instead of the most recent version of the class. More precisely, for each fault we recover the whole system to the latest state that still contains the fault before performing any measurements. The result is a set of measurements that actually represents the fault-prone class and its relationship it had with other classes.

Regarding the effect of the improvements on fault-proneness prediction models, we are mostly interested in how the models will improve when used across systems. Therefore, we will focus solely on the cross-system models and not the single-system models. To test the effect of the improvements on the fault-proneness prediction models, we will take the systems and results from our replication study and use those as base rates. Next, we reapply the improved models and compare the accuracy difference using the following hypothesis:

> *Hypothesis H1: A fault-prediction model with the improved predictor-set will result in more accurate predictions compared to a fault-prediction model that uses the predictor-set as proposed by Briand et al.*

For testing the effect of the second improvement, we will again take the systems and the results from the replication study and reapply the improved models on the systems. We test the effectiveness using the following hypothesis:

> *Hypothesis H2: A fault-prediction model with state-aware measurement will result in more accurate predictions compared to a fault-prediction model without state-aware measurement.*

Finally, both of the improvements are applied. We expect predictions models using both the improvements result in a more accurate model compared to using just a single improvement or none of the improvements. The effectiveness of the improved model is tested using the following hypothesis:

> *Hypothesis H3: A fault-prediction model with the predictor-set improvement and state-aware measurement will result in more accurate predictions compared to a fault-prediction model without these improvements.*

This chapter is structured as follows: Section 5.2 describes the setting of the study. Section 5.3 lays out the analysis methodology and how we will test the improve models. In Section 5.4 we present our findings. In Section 5.5 we discuss our findings and draw our conclusions.

5.2 Description of Study Setting

5.2.1 Systems

For our dataset, we will use the same systems that were used for the cross-system dataset of the replication study. Table 5.1 provides an overview of the systems. For each system: the number of non-generated/non-test Java source code files (Files), the number of source lines of code (SLOC), the number of faults (Faults), the fault distribution (FGini), and the team who build the system (Team) are given. Table 5.1 provides an overview of the system-pairs used to train and test the model. The training system (Train), the validation system (Validate), and the prediction model's accuracy (Accuracy), its precision (Precision), and its recall (Recall) can be found in this table.

Table 5.1: Systems used in the replication study

	Files	SLOC	Faults	FGini	Team
BIra	30	3.023	8	.45	C
MAdd	80	4.012	38	.56	D
MIntO	46	2.204	93	.45	B
MIra	28	1.186	29	.38	D
MPen	27	1.217	17	.59	B
MRep	85	2.603	72	.44	B

5.2.2 Variables

Independent variables

We will use two different set of independent variables, one for each improvement:

- *Predictor-set extension.* A selection of metrics from the metric-suite will be made in the model construction phase that will represent the independent variables. The metric-suite used in this study contains: a subset of Briand et al. their cohesion metrics [25], Benlarbi & Melo their polymorphism measures [12], metrics from the Chidamber & Kemerer metric-suite [13], size metrics, change metrics based on Koshgoftaar et al. [22] their metric-suite, and a subset of Ostrand & Weyuker their 'optimal' *standard model* [27]. We replaced Briand et al. their import/export coupling metrics with the composite measures *XXXIC* and *XXXEC*. Also, Benlarbi & Melo their polymorphism measures, *DP*, *SP*, and *OVO* were replaced by the composite measure *POLY* (see Section 5.1 for more information)
- *State-aware measurement.* The metric-suite used to select the independent variables is exactly the same as the one used in the initial study. The metric-suite used in this study contains: a subset of Briand et al. their cohesion metrics [25], Benlarbi & Melo their polymorphism measures [12], all metrics from the Chidamber & Kemerer metric suite [13], and some simple size metrics. As part of this improvement, we will measure the fault-prone class; the latest version of the class that still contained the fault. Before conducting any measurements on that class, we will not only revert the state of that class but of the complete the system. This is because some measurements also consider class relationships. We will use the 'checkout' and 'reset' functionality of the version control system and the commit-id of the foremost commit whom mentioned that the fault was fixed (for more details see Appendix A).

All the metrics measure at class-level. Inner classes are not treated as individual observations but their measures, methods, and attributes are counted to contribute towards the containing class. Faults that are traced back to an inner class were assigned to the out most containing class.

Dependent variable

We want to evaluate whether the existing design measures are useful for predicting the likelihood that a class is fault-prone; that is if the class contains at least one fault. The outcome value is dichotomous; a class is either fault-prone or it is not.

We collect faults in a class using our *fpms* package. The tool analysis git commit messages, if the commit message contains hints of a fault that has been fixed, then we say that all changed classes contain one fault each (see Appendix A). Take in mind that the faults that were detected using this procedure are probably not all of the faults. Also, there might be some false positives among the observed faults (See Section 4.2 of the replication study for a detailed discussion).

5.3 Data Analysis Methodology

The data analysis methodology follows the same method as described in the replication study. If one of the two improvements require a deviation of the replication method, it will be indicated. An short overview of the methodology is given in this section, for a more detailed description see Section 4.3.

In this section we layout the methodology used for analysis the collected data. The analysis procedure consists out of: (i) An analysis of descriptive statistics, (ii) data distribution and outliers analysis, (iii) a multivariate regression analysis, (iv) an evaluation of the prediction model, (v) and the testing of the hypotheses.

5.3.1 Descriptive Statistics

For each system, all metrics are calculated. The minimum (Min), maximum (Max), sample mean (μ), Median (Med), and the standard deviation (σ) are collected. This data will help with the interpretation of the results for upcoming analyses.

5.3.2 Outlier analysis

The outlier analysis is used to spot low variance measures and originally consists out of two steps: (i) excluding weak differentiators by removing all measures with less than 5 zero data points; (ii) and excluding multivariate outliers with a too large the Mahalanobis distance. In this study we will deviate from the replication study and only perform the first step. This is because detecting multivariate outliers on the dataset is unreliable due to the low number of observations per system (see Section 4.3 for more details).

5.3.3 Prediction model construction

Both improvements will follow the same method for constructing the fault-proneness prediction model as the one used in the replication study. A logical regression model will be build that uses a subset of metrics as predictors and fault-proneness will be used as the outcome variable. The independent variables are selected using a mixed stepwise selection process that compares the models their AIC scores. The multi-collinearity reduction step is skipped for both improvements because is added little value in our replication study.

5.3.4 Model evaluation

The model evaluation for both the improved models is done based on accuracy, precision, and recall. Because we are interested in the cross-system model prediction capabilities, only the cross-systems validation method is used. A small difference between the replication study and this study is that we focus more on the accuracy measure and less on the precision and recall measures. Accuracy implicitly represents both the precision and recall measures but is easier to use when comparing models. For example, comparing a model with a precision of 60% and a recall of 90% with a model with a precision of 70% and a recall of 80% is more complex than comparing two models with accuracies of 70% and 80%.

5.3.5 Hypothesis testing

Before testing the hypotheses, the absolute accuracy values obtained in the model evaluation phase are transformed to relative values. The values are relative towards the base rate values, that are the

accuracy values of the models from the replication study, and represent the difference between those values in terms of accuracy gains. For example, the original model has an accuracy of .70 and the improved model an accuracy of .80, then the improved model has an accuracy gain of .10. A negative gain value is the same as a positive loss value.

After the transformation, we will test both hypothesis using an one-tailed one-sample t-test ($\alpha \leq .05$). Moreover, Cohen'd will be used to measures the effect size.

5.4 Analysis Results

5.4.1 Descriptive Statistics

By looking at the means of the metrics, we see in all systems that number of children, lack of cohesion measure, and polymorphism measure (NOC, LCOM, and POLY respectively) do not differentiate the data very well. For the NOC and LCOM measures, these observations are no different than we saw in our replication study. The composition metric, POLY, does not seem to differentiate the data very much either; even though this metric measures any kind of polymorphism. This could mean that the systems do not make use of the polymorphic aspects of the object oriented language.

5.4.2 Outlier analysis

The outlier analysis of the improvements are given below:

- *Predictor-set extension.* The exclusion of weak differentiators resulted in the removal of four metrics. The NOC and LCOM measures had less than five non-zero data points were removed from all systems; the POLY metric was removed from four systems, and the DAC metric was removed from two systems.
- *State-aware measurement.* The exclusion of weak differentiators resulted in the removal of the LCOM measure in all systems, the NOC measure from four systems, the ancestor and descendent coupling measure from all systems, the static polymorphism measures from four systems, and the dynamic polymorphism from all systems.
- *Both improvements.* LCOM was removed from all systems, the NOC measure from three systems, and the POLY measure from one system.

5.4.3 Prediction model construction

The regression models were build for each system in the cross-system dataset using mixed stepwise selection. The prediction model construction for both improvements were done separately.

predictor-set improvement

An overview of the selected predictors per system and the model's AIC score can be found in Table 5.2 for the predictor-set improvement.

When looking at the selected predictors of the predictor-set improvement model, we see that Changes measure is included as predictor in three out of the five models, and that the Authors metric is included as predictor in two out of the five models. Also, compared to the selected predictors from the replication study (see Table 4.4), we see that the export coupling related metric (XXXEC) is included in two more models.

If we compare the AIC values of the models from the replication study with the AIC values of the predictor-set improvement models, we see that for four out of the five systems the value is lower. In other words, the information loss of these four models is less and are therefore better representations of the actual model. We conclude that the predictor-set improvement models better describes the relation between software measures and faults compared to the original model. However, it does necessarily mean that the model will obtain higher prediction accuracies.

Table 5.2: Predictor-set extension improvement stepwise analysis result

System	Selected predictors	AIC
MAdd	SIZE1, XXXIC, Changes	8.00
MIntO	SIZE2, XXXIC, XXXEC, Changes, Authors	12.00
MIra	XXXEC	4.00
MPen	CBO, RFC, MPC, NOM, SIZE2, NIP	16.77
MRep	WMC, CBO, SIZE2, XXXEC, Changes, Authors	48.97

state-aware measurement improvement

An overview of the selected predictors per system and the model's AIC score can be found in Table 5.3 for the state-aware measurement improvement.

The fact that the selected predictors in these improvement models changed drastically, could indicates that the state-aware measurement affects prediction models.

Table 5.3: State-aware measurement improvement stepwise analysis result

System	Selected predictors	AIC
MAdd	WMC, DIT, RFC, OCAEC, OVO	61.36
MIntO	CBO, SIZE1, SIZE2, OCAIC, OCMIC	14.77
MIra	RFC, DAC, NOM	8.00
MPen	CBO, NOM, SIZE1, SIZE2, NIP	12.00
MRep	WMC, DIT, RFC, NOM, SIZE1, SIZE2, OCIAC	82.88

Both improvements

An overview of the selected predictors per system and the model's AIC score can be found in Table 5.4 for the models that incorporate both the predictor-set and state-aware measurement improvement.

The Changes metric seems to be a strong predictor for fault-proneness, since the metric is included in three out of the five models when using both the improvements. Size related measurements are included in four out of the five models. The number of non-inheritance polymorphism is included in three out of the five models, interesting is that Briand et al [4] also included NIP in their best model. Finally, coupling measures are also among the selected predictors in four out of the five models.

Table 5.4: Both improvements stepwise analysis result

System	Selected predictors	AIC
MAdd	WMC, SIZE2, XXXIC, NIP, Changes, Authors	14.00
MIntO	NOC, CBO, NIP, Changes, Age	12.00
MIra	RFC, DAC, NOM	8.00
MPen	NOM, SIZE1, SIZE2, NIP, Changes	12.00
MRep	DIT, CBO, DAC, NOM, SIZE1, XXXEC, NIP	42.37

5.4.4 Model evaluation

predictor-set improvement

For the result of the analysis, see Table 5.5. For each system pair (coded as training system - validation system), the true positives ($T+$), true negatives($T-$), false positives ($F+$), false negatives ($F-$), accuracy, precision , and recall are calculated.

If we compare the model evaluation results with the results obtained in the replication study (see Table 4.9), we see that three out of the four models in this study have higher accuracies. This suggest that the improvement indeed positively affects the prediction capabilities.

Table 5.5: Predictor-set extension improvement models their prediction capabilities

	$T+$	$T-$	$F+$	$F-$	Accuracy	Precision	Recall
MAdd-MIra	5	3	2	7	.47	.71	.42
MIntO-MPen	6	4	7	0	.52	.42	1.00
MRep-MIntO	11	2	3	1	.79	.86	.86
MRep-MPen	4	4	7	2	.52	.40	.75

state-aware measurement improvement

For the result of the analysis, see Table 5.6. For each system pair (coded as training system - validation system), the true positives ($T+$), true negatives($T-$), false positives ($F+$), false negatives ($F-$), accuracy, precision , and recall were calculated.

The state-aware measurement improvement positively affected all four model when compared to the models used in the replication study. Moreover, three out of the four models obtain higher accuracies compared to the predictor-set improvement. The state-aware measurement seems to have a larger effect on the prediction models than the predictor-set improvement.

Table 5.6: State-aware measurement improvement models their prediction capabilities

	$T+$	$T-$	$F+$	$F-$	Accuracy	Precision	Recall
MAdd-MIra	1	3	2	1	.57	.33	.50
MIntO-MPen	7	10	0	0	1.00	1.00	1.00
MRep-MIntO	9	0	4	4	.79	.91	.84
MRep-MPen	6	2	8	1	.54	.50	.85

both improvements

For the result of the analysis, see Table 5.7. For each system pair (coded as training system - validation system), the true positives ($T+$), true negatives($T-$), false positives ($F+$), false negatives ($F-$), accuracy, precision , and recall were calculated.

The combination of both the predictor-set improvement and the state-aware measurement improvement dramatically increased the fault-proneness prediction models their accuracies for three out of the four system-pairs compared to the models from the replication study. Also, the prediction models build using a combination of both the improvements result in better accuracies compared to the individual improvements for these three system-pairs.

Table 5.7: Models using both improvements their prediction capabilities

	$T+$	$T-$	$F+$	$F-$	Accuracy	Precision	Recall
MAdd-MIra	1	3	2	1	.57	.33	.50
MIntO-MPen	7	5	5	0	.75	.65	1.00
MRep-MIntO	11	3	1	2	.81	.98	.80
MRep-MPen	5	5	5	2	.61	.56	.77

5.4.5 Hypothesis testing

To aid the interpretation of our testing outcomes, the descriptive statistics of the prediction models are given in Table 5.8, Table 5.9, and Table 5.10. The tables include the number of observations (N), minimum value (Min), maximum value (Max), mean (Mean), and standard deviation (St. Dev.).

Testing the prediction accuracy of the fault-proneness prediction models with the predictor-set. To determine if there is a significant difference ($p \leq .05$) between the accuracies of the predictor-set improvement models and the replication models, a Welch independent-sample t-tests

was conducted. For the first hypothesis (H1), the null hypothesis was formulated:

$$H_0 : \mu_{repl} = \mu_{predictor} \tag{5.1}$$

The results of the test indicate that there is no significant difference between the accuracies of the predictor-set improvement models and the replication models ($t(4.592) = .0163$, $p = .987$). The null hypothesis: There is a significant difference in accuracy between the replication models and the predictor-set improvement models, is not rejected. The effect size was calculated using Cohen's d, and indicated a small effect size ($d = 0.023$).

Table 5.8: Descriptive statistics of the predictor-set improvement prediction models

Statistic	N	Mean	St. Dev.	Min	Max
$T+$	4	6.500	3.109	4	11
$T-$	4	3.250	0.957	2	4
$F+$	4	4.750	2.630	2	7
$F-$	4	2.500	3.109	0	7
Accuracy	4	0.575	0.145	0.470	0.790
Precision	4	0.598	0.225	0.400	0.860
Recall	4	0.758	0.247	0.420	1.000

Testing the prediction accuracy of the fault-proneness prediction models with state-aware measurement. This hypothesis was tested in the same way as Hypothesis H1, using the following null hypothesis:

$$H_0 : \mu_{repl} = \mu_{state} \tag{5.2}$$

Based on the test results, there is no significant difference between the accuracies of the state-aware measurement improvement models and the replication models ($t(5.699) = .882$, $p = .413$). The null hypothesis is not rejected. The effect size was calculated using Cohen's d, and indicated a medium effect size ($d = 0.635$).

Table 5.9: Descriptive statistics of the state-aware improvement prediction models

Statistic	N	Mean	St. Dev.	Min	Max
$T+$	4	5.750	3.403	1	9
$T-$	4	3.750	4.349	0	10
$F+$	4	3.500	3.416	0	8
$F-$	4	1.500	1.732	0	4
Accuracy	4	0.725	0.215	0.540	1.000
Precision	4	0.685	0.322	0.330	1.000
Recall	4	0.798	0.211	0.500	1.000

Testing the prediction accuracy of the the fault-proneness prediction models with the predictor-set and state-aware measurement. This hypothesis was tested in the same way as Hypothesis H1, using the following null hypothesis:

$$H_0 : \mu_{repl} = \mu_{both} \tag{5.3}$$

The test result show no significant difference between the accuracies of the state-aware measurement improvement models and the replication models ($t(4.022) = .766$, $p = .486$). The null hypothesis is not rejected. The effect size was calculated using Cohen's d, and indicated a medium effect size ($d = 0.555$).

Table 5.10: Descriptive statistics of the prediction models using both improvements

Statistic	N	Mean	St. Dev.	Min	Max
$T+$	4	6.000	4.163	1	11
$T-$	4	4.000	1.155	3	5
$F+$	4	3.250	2.062	1	5
$F-$	4	1.250	0.957	0	2
Accuracy	4	0.685	0.114	0.570	0.810
Precision	4	0.630	0.269	0.330	0.980
Recall	4	0.768	0.205	0.500	1.000

There is not enough statistical evidence that indicate a significant difference in accuracy between the replication study models and the improvement models. However, the Cohen'd values of the state-aware improvement models and the models using both improvements suggest a moderate to high practical significance [49]; the average accuracy gain of the state-aware improvement and 'both-improvement' models are 15% and 11%, respectively. In Table 5.11 are the individual observations and contains the accuracy gains of the improvement models relative to the values observed in the replication study. The absolute accuracies from the original models used in the replication study (Intercept), the difference between the accuracies of the original models and the predictor-set improvement models could be found in this table.

Table 5.11: Accuracy gains of the improved prediction models

	Intercept	Predictor-set	State-aware	Both impr.
MAdd-MIra	.35	.12	.22	.22
MIntO-MPen	.96	-.44	.04	-.21
MRep-MIntO	.55	.24	.24	.26
MRep-MPen	.43	.09	.11	.18

5.5 Conclusion and Discussion

To answer our research question (RQ2): how could the construction for regression-based fault-proneness prediction models be improved? we suggested two improvements regarding the model construction method used by Briand et al. (i) extending the predictor-set with process measures and simplifying the metric-suite by merging strongly related metrics (predictor-set improvement). (ii) Measuring class properties of the latest class version that still contained the actual fault (state-aware measurement improvement). The predictor-set improvement does not significantly improve the prediction model its accuracy. The state-aware measurement improvement and using both improvements increased the average prediction accuracy by 15% and 11%, respectively. There was no statistical significant evidence that showed that the latter two improvements increased accuracy of the replication models, but the Cohen's d suggested a moderate to high practical significance. Moreover, we observe that the suggested improvements result in higher accuracies except for one system pair. For the three systems, the models that use both improvements perform best in terms of accuracy. To answer research question RQ2, using both the suggested improvements result in fault-proneness prediction models that are more accurate compared to the models build using Briand et al. their method.

One notable observation done during the model construction phase is that we found two reoccurring types of metrics in almost all of the best models, namely size/complexity measures and the coupling measures. The same discovery was done in the replication study (see Chapter 4 for the discussion). Another frequently selected predictor is the Changes metric. This means that frequently changed classes are more likely to be fault-prone compared to less frequently changed classes. This observation could be explained by the trivial fact that faults are more likely to arise in the parts of the code that are changed than in part that are not changed.

A second observation was the inconsistent set of independent variables among the best prediction models, and is one of our biggest concerns. Every model has a different set of independent variables with little similarities compared to the other models. For example, MAdd and MIra are developed

by the same team but do not have one independent variable in common. This could mean that team composition does not result in systems with similar class characteristics. Another explanation could be that the selected predictors are no real fault-proneness predictors. The latter suggest that the models will become (more) unstable when used on other systems.

5.5.1 Threats to validity

In order to make state-aware measurement possible, we had to change the workings of the tool that was used in the initial study. They still measure the same concepts, but it is done in a slightly different way. Two differences in measurement are: (i) The state-aware measurement tool does not take external libraries into consideration, for example the depth of inheritance (DIT) only considers the local inheritance. For example, if *MyList* extends Java's *List*, the DIT of the new method of measurement is 0 instead of $0 + DIT_{List}$; (ii) The state-aware measurement tool does not only consider the classes of a single *jar*, but all non-test/non-generated Java classes that were generated by the Java compiler. For example, a systems compiles two jars. The former method considered only classes in the 'main' *jar*, while the new method considers classes from both *jars* and even the classes that are not included in the *jar*. These changes could introduce new confounding variables and therefore negatively influence the comparison of the replication models and predictor-set improvement models, with the state-aware measurement improvement models and the models that incorporate both improvements.

Other threats to validity are similar to the threats described in the replication study. For more details regarding these threats we refer to Section 4.5.

5.5.2 Future research

The difference between the selected predictors for the prediction models is remarkable. We expected that by using the same team composition, a similar set of metrics would be selected as predictors for the models. Instead, we observed that the selected predictors have little in common when compared the models their prediction-sets. One explanation could be that the team composition has no relationship with class characteristics. We have our doubts about the relationship of some of the predictors and fault-proneness. The only extensive research regarding this relationship, that we found, was done by Briand et al. [14]. In our opinion, the value of replicating Briand et al. their research is high because of the absence of observable patterns regarding the behaviour of faults in software systems. Moreover, Briand et al. states that their research is of an exploratory nature and that an more extensive studies are required to validate their findings.

The state-aware measurement improvement is not yet applied in fault prediction to the knowledge of the author. It could be interesting to see how other prediction models will react when they are 'state-aware'. Moreover, we think that the state-aware measurement really pays of when used on datasets with a large commit history. To follow up on the previous research direction, Briand et al. [14] did not mentioned how they handled state when measuring class properties and did not really consider class-state in their follow-up study [4]. Therefore, a replication of study Briand et al. that explores the relationship between design measures and software quality, but this time considering the state of the classes during measurement could result in interesting discoveries.

The fault discovery strategy could be altered or analysed in more depth. For example, one could abstract data from special bug-tracking databases. This could lead to different (and perhaps more accurate) fault prediction models. Another option would be to analyse the faults is more depth and study the relation of fault prediction models and the relations to certain types of faults. For example, one could manually verify and classify the discovered faults from the bug database and validate the models based on the prediction capabilities of certain types of faults (see the paper of Herzig et al. for more information about fault categories and fault discovery strategies [29]).

Chapter 6

The Influence of Environmental Factors on Fault-Proneness Prediction Models

6.1 Introduction

In our replication study (see Chapter 4), we discuss three axes on which we think fault-proneness prediction models could be improved:

- *Axis I.* Through changing the included fault-proneness predictors.
- *Axis II.* By tuning or altering fault-proneness prediction modelling techniques.
- *Axis III.* By considering the context in which the fault-proneness prediction model operates.

During the improvement study (see Chapter 5), we improved Briand et al. their prediction model by changing the collection of predictors to choose from (Axis I), and by changing the modelling technique such that the state of the system is considered during measurement (Axis II). In this study will we focus on the latter axis, and try to improve the model by controlling the factors that are likely to influence the prediction model (Axis III).

In this study, we hypothesize that context plays an important role in cross-system fault-proneness prediction; and that by keeping the right factors constant, the prediction models can become more stable and effective when applied in practice. In order to test the latter statement, we consider three types of context: people, process, and technology. These types are considered the fundamental elements of an information technology infrastructure [50] and are key components of an information system [51]. Each of these types will be explored in isolation and will help in answering the question: will factors related to that type affect fault-proneness prediction models?

Based on the available information on the systems used in this study, a number of factors were collected and allocated to a context type, for a detailed description see Appendix A:

- *People.* This type contains all the factors related to people whom were involved during development or maintenance. The type provides information regrading the team composition, the architect, the size of development team, the years of experience of team, type of the project team, and the number of teams whom worked on the system.
- *Process.* This type contains all process and managerial aspects. It provides information regarding the budget that was reserved for the system, the product owner, the software methodology that was used, and the number of stories that were available.
- *Technology.* This type contains the system related factors and has information on the type of system, number of files, lines of comments/code, number of faults in the system and changes to the system, the age of the system, and the time it took to develop the system.

This study is of an exploratory nature, there is no known evidence on the effect of factors on fault-proneness prediction models. The factors will be allocated to a type intuitively; the factors are chosen based on the information that is available. As result, the factors might cause an incomplete or inaccurate representation of that context type. Note that the focus of this study is not to point out the factors that influence prediction models but rather to obtain knowledge on the behaviour of prediction models when applied in similar contexts.

The first hypothesis is used to test the effect of people related characteristics on cross-system fault prediction model accuracy:

> *Hypothesis H1: Fault-proneness prediction models that are applied on systems that have many people related similarities differ in accuracy from fault-proneness prediction models that are applied on systems that share little people related similarities*

The second hypothesis is used to test the effect of system related characteristics on cross-system fault prediction models their accuracy:

> *Hypothesis H2: Fault-proneness prediction models that are applied on systems that have many system related similarities differ in accuracy from fault-proneness prediction models that are applied on systems that share little system related similarities*

The third hypothesis tests the effect of process related characteristics on cross-system fault prediction models their accuracy:

> *Hypothesis H3: Fault-proneness prediction models that are applied on systems that have many process related similarities differ in accuracy from fault-proneness prediction models that are applied on systems that share little process related similarities*

The outcome of hypotheses H1, H2, and H3 will not be enough to draw conclusions yet; confounding variables could influence the prediction outcomes. In order to show that the individual system characteristic impacts the prediction model, it must differ significantly from a model based on all systems. Hypothesis H4 will be used for testing this:

> *Hypothesis H4: Fault-proneness prediction models build from a set of systems that share many factors of a certain context type differs from a fault-proneness prediction model that is build from a set of all systems*

Finally, to test if the models that result from a specific context are good enough to be applied in practice, we compare these models to a constant value. Based on the rule of thumb given by Hosmer & Lemeshow [46], we state that the accuracy is reasonable at .60, good at .70, and excellent at .80. We expect that our best model is able to obtain an accuracy of at least .70. The following hypothesis will be used:

> *Hypothesis H5: Fault proneness prediction models applied within an idealistic context are able to predict with an accuracy of at least .70*

This chapter is structured as follows: Section 6.2 describes the setting of the study. Section 6.3 lays out the analysis methodology. In Section 6.4 we present our findings. In Section 6.5 we discuss our findings and draw our conclusions.

6.2 Description of Study Setting

6.2.1 Systems

To minimize the effect of confounding variables it is important to control as much as variables as possible. It is challenging to isolate a single factor that influence the fault model's prediction accuracy. Therefore, the systems in our dataset must be already similar to one another and share common characteristics. To control most of the system their characteristics, we will pick systems that are developed by a single company, whom propagates strict protocols and standards regarding software

development. Moreover, the systems must be developed for a similar business context and in a similar environment. The systems may only slightly differ from each other, preferably on a single aspect only. For these reasons we will not reside to open-source projects. We have little information on most of the open-source projects regarding development environment, standards and protocols, and so on; it will be hard to find a set of similar projects. Instead, we will use commercial systems.

Due to constrains to our analysis tools, not all systems are suitable for our dataset. One constrain is the programming language, we must choose a specific language and keep this constant for all project. The reason for this is because the tools we use for measuring product metrics are language specific. Our language of choice is Java, and was decided by the fact that the company that could provide the dataset builds Java systems.

In total we collected 19 systems. These systems are all developed and used by a firm active in the financial services sector. All systems are still actively developed and used during the period of analysis. Out of the 19 systems, 14 systems were suitable for analysis; the remaining systems suffered from errors or did not meet the criteria. An overview of all systems and a short description is given in Table 6.1.

Table 6.1: Collected systems overview

System	Description
*BIra**	Exposes raw data
*DSto**	Interfaces with a document store
*ESec**	?
*Doc**	Upload documents
*Sec**	Verify documents
*IRat**	Interfaces with old legacy system
kryp	Reactive back-end-service stores large amount of data
*MAdd**	Combines and summarises data from many sources
*MApp**	Combines and summarises data from many sources
MCon	API to move process a step further
*MIntO**	Interfaces with legacy back-end service
MIntR	Manages, validates and distributes user input
*MIra**	Connect and translate back-end-service
*MPen**	Connect and translate back-end-service
*MRep**	Interface with legacy back-end service
Acc	Legacy system
*MDue**	Front-end API
MorM	Combines and summarises data from many sources
Forc	Bridges two protocols
pMorE	Exposes legacy system
Ass	Shared component

* included in the dataset

System pool creation

In order to answer our hypotheses, we will make system-pools. Each pool contains systems whom share similar factors of a specific context type. We use three types of pools, each type representing one of the three context types: people, process, or system. Moreover, each type of pool has an inverse-type, that is a type that is dissimilar on as many factors of a specific type as possible. For example, a pool of the people-type contains systems with similar people factors, the inverse of the people-type pool contains systems with as many dissimilar people factors as possible.

The pools are created using the following strategy:

1. One characteristic, the main-characteristic, is selected that belongs to the pool type in question. This characteristic is chosen in such a way that that the systems in that pool have as many commonalities among the characteristics. The systems that comply to the criteria will form the first sub-pool. This sub-pool will carry the label $TYPE^1$ where $TYPE$ is the name of the

pool-type.

2. At least one other sub-pool must be created. This sub-pool is made out of the systems whom share the same main-characteristic as the first sub-pool; this main-characteristic is mutually exclusive from the other sub-pools. The same conditions hold as for the primary sub-pool; at least 3 systems must be in the sub-pool and the systems must have some commonalities among the other characteristics. In the case of a system that differs too much from the other systems and the sub-pool size consists out of more than 3 systems, the system must be removed. The detection of outliers is subjective. The following sub-pools carry the label $TYPE^n$ where n is the number of the sub-pool.

3. Finally, a special kind of sub-pool must be created, the inverse sub-pool. Based on the first selected characteristic in step 1, systems are chosen that all differ from each other on that specific characteristic. The systems are chosen in such a way that the resulting sub-pool consists of systems that differ on the main-characteristic and on as many other characteristics as possible. This sub-pool is labelled as (e.g. $TYPE^-$).

Selected pools

Three pools are created, more details on the pools can be found in Appendix D.

- One people-type pool was created using team-composition (Team) as the main-characteristic. The PPL^1 only includes the systems developed by team B (see Table D.1). The PPL^2 only includes the systems developed by team D (see Table D.2). The PPL^- is the inverse-pool and is developed by distinct teams (see Table D.3).
- One system-type pool was created using system type (Type) as main-characteristic. The SYS^1 pool only includes server/client systems (see Table D.4). The SYS^2 pool only includes API systems (see Table D.5). The inverse system pool, SYS^-, only includes systems of different types (see Table D.6).
- Finally, a process-type pool was created based on the number of stories (Stories) as main-characteristic. The $PROC^1$ pool only contains systems with less than average stories (average is 20 stories) (see Table D.7). The $PROC^2$ pool only contains systems with more than average stories (see Table D.8). The inverse pool, $PROC^-$, contains a single system of varying number of stories (see Table D.9).

6.2.2 Variables

Independent variables

The metric-suite used in this study contains: a combination of Briand et al. their cohesion metrics [10], a combination of Benlarbi & Melo their polymorphism measures [12], all metrics from the Chidamber & Kemerer metric suite [13], size metrics, change metrics based on Koshgoftaar et al. [22] their metric-suite, a subset of Ostrand & Weyuker their *standard model* [27], and a class-author count. See Appendix A for an overview of the metrics and the tool used to calculate these metrics.

All the metrics measure at class-level. Inner classes are not treated as individual observations but their measures, methods, and attributes are counted to contribute towards the containing class. Also, faults that traced back to an inner class were assigned to the out most containing class.

The measurements are state-aware; the class is measured in the state when it contained the fault. To correctly measure the fault-prone class, the whole system was reverted to the latest state before the fault was fixed. For more information on state-aware measurement and the predictor-set see Chapter 5.

Dependent variable

We want to evaluate whether a subset of the collection of measures are useful for predicting the likelihood that a class is fault-prone; that is if the class contains at least one fault. The outcome value is dichotomous, a class is either fault-prone or it is not.

We collect faults in a class using our *fpms* package. The tool analysis git commit messages, if the commit message contains hints of a fault that has been fixed, then we say that all changed classes contain one fault each (see Chapter A). Take in mind that the faults that were detected using this procedure are probably not all of the faults. Also, there might be some false positives among the observed faults. See Section 4.2 in the replication study for the results of the assessment of our fault detection procedure.

6.3 Data Analysis Methodology

In this section we layout the methodology used for analysis the collected data. The analysis procedure consists out of: (i) prediction model construction, (ii) evaluation of the prediction models, (iii) and finally hypothesis testing. We adapted the data analysis methodology used in the improvement study for constructing and validating the prediction model (see Section 5.3 and Section 4.3 for a more detailed description of the data analysis methodology used in this study). In this study, no outlier analysis is conducted because in our experience the analysis is redundant (e.g. stepwise selection also filters out the weak differentiators).

6.3.1 Model construction

For each sub-pool, a logistic-regression based fault-proneness prediction model will be build. The independent variables are selected using a mixed stepwise selection process that compares the models their AIC scores. Class fault-proneness is used as outcome variable. To map out the effect of confounding variables, we also build a model containing all systems (that are the systems with a star in Table 6.1). This model will be the base rate, and used to determine the actual effect of the context types. First, we expect that the 'all-systems' pool has a low AIC score and many selected predictors. This indicates that the resulting model does not fit the data very well and that the observations differentiate from each other. Secondly, we expect that the AIC scores of the sub-pools $TYPE^n$ are relatively low compared to the AIC score of the inverse sub-pool $TYPE^-$, if the system characteristic type affects the fault-proneness prediction model. This is because it should be less difficult to fit a model to dataset with lot of similarities compared to a dataset with a lot of dissimilarities. Finally, we expect that, if the system-characteristic type affects the model, the sub-pools $TYPE^n$ will have similar independent variables. If this is the case, then this combination of variables are probably good fault-proneness predictors for that specific context.

6.3.2 Model validation

The accuracy, precision, and recall is calculated for each prediction model. The accuracy measure will be used for hypothesis testing and in the discussion. The pool-models are evaluated using the 10-fold cross-validation technique; the same pool will be used for training and validating the model. If a system characteristic type affects fault-proneness prediction models, then we expect to observe three things: (i) The sub-pool models $TYPE^n$ are more accurate that the all-systems pool. If this is the case, we may say that besides the confounding variables whom affected the model, the system characteristic type also has an effect on the prediction model. (ii) The sub-pool models $TYPE^n$ obtain similar accuracies. If this is true, then the system characteristic type is likely to have an effect on prediction models in general rather than for a particular set of characteristics of that characteristic type. (iii) The sub-pool models $TYPE^n$ obtain higher accuracies compared to the inverse sub-pool model $TYPE^-$. If the accuracies are structurally higher for the sub-pools with the similarities compared to a sub-pool without these similarities; then the system characteristic type could actually have an effect on the prediction model.

6.3.3 Hypothesis testing

To answer the Hypothesis H1, H2, and H3 we first test if there is a difference between the accuracies of the sub-pool prediction models, including the inverse sub-pool. If there is no difference between

any of these pools, then it is not needed to test the hypothesis any further and the null hypothesis of the following form will not be rejected.

$$\mu_{type^1} = \mu_{type^2} = \mu_{type^-} \tag{6.1}$$

If there is a difference between one of the sub-pools, then we continue with a post-hoc test in order to identify which pool differ from each other. The null hypothesis will be rejected if and only if the accuracy of the sub-pools models do not differ significantly from each other and the prediction model of one of the two sub-pool differ significantly from the inverse sub-pool prediction model.

A one-way analysis of variance (ANOVA) will be conducted to determine if the means of the accuracies of the sub-pools models are significantly different ($\alpha \leq .05$). To uncover which means are unequal and by how much, we will use the Tukey Honestly Significant Difference (HSD) test. The null hypothesis will be rejected if the ANOVA test is significant and the Tukey post-hoc test indicates that there is only a difference between one of the sub-pools and the inverse sub-pool.

Hypothesis H4 will be tested using the same method as used for hypothesis H1, H2, and H3; only using a different hypothesis. The hypothesis states that there is at least one difference between any of the sub-pools and the all-systems pool. First, we test if there is any significant difference at all between the sub-pools and the all-systems pool; if not then the null hypothesis of the following form is not rejected:

$$\mu_{type^1} = \mu_{type^2} = \mu_{type^-} = \cdots = \mu_{all\text{-}systems} \tag{6.2}$$

If there is a difference between one of the sub-pools, a post-hoc test will be conducted to determine which sub-pools differ from each other. The null hypothesis will be rejected if and only if one of the sub-pool prediction models differs significantly from the all-systems prediction models in terms of accuracy.

Hypothesis H5 will be tested using an one-tailed one-sample t-test ($\alpha \leq .05$). Based on the results of the previous hypotheses, will will pick the pool that resulted in the best prediction models. The models that were generated using this pool will be compared to the constant .70.

6.4 Analysis Results

An overview of the selected predictors per system and the model's AIC scores can be found in Table 6.2. A more detailed table can be found in Appendix E.

Based on the observations shown in Table 6.2, the best inverse sub-pool model fitted the dataset better than the other sub-pools in two out of the three pool-types. This could suggest that more dissimilarity in the dataset result in a better fit of the model and is not what we expected. Secondly, the selected predictors in the sub-pools of the people-type, PPL^n, differs in 10 variables from each other; the selected predictors of the system-type sub-pools differ in 11 variables from each other; and the selected predictors of the process-type sub-pools differ in 7 variables from each other. An explanation for the dissimilarity among the predictor sets could be that any of these factors were not the cause of the discovered faults and are not likely to influence prediction model accuracy.

Looking at the selected predictors, the Changes metric is selected in all models and is a dominant positive variable in all models (see Appendix E); this suggests that a frequently changed class is likely to be fault-prone. The POLY measure is selected in eight out of the ten pools and is a dominant negative variable in almost all the models; this suggests that a class with low polymorphism is likely to be fault-prone. This opposes the conclusion drawn by Benlarbi & Melo [12], where they state that "*polymorphism may increase probability of fault in OO software*". The DAC is selected in six out of the ten pools and is a dominant positive variable in four of the pools (including the all-systems pool); this suggests that a class with high coupling to abstract data types is likely to be fault-prone. Interesting is CBO, which includes the DAC measure, is not a dominant predictor in any of the models. The SIZE1 and SIZE2 measure are included in five out of the ten pools as a combination. The measures do not play a significant role in any of the models. However, an interesting observation is that the SIZE1 measure is always positive while the SIZE2 measure is always a negative factor. In other words, a class with a high number of instructions, and a low number of attributes and local methods is probably fault-prone. This could suggest that the number of methods and attributes do

not result in faults but rather the size of the methods themselves. All other variables do not play a significant role in the larger part of the pools or show contrary results (e.g. positive and negative multipliers).

Table 6.2: Sub-pools stepwise analysis result

	Selected predictors	AIC
PPL^1	DAC, Changes, Age, POLY	80.91
PPL^2	WMC, DIT, DAC, MPC, NOM, SIZE1, SIZE2, XXXIC, POLY, NIP, Changes, Authors, Age	28.00
PPL^-	DIT, DAC, SIZE2, XXXEC, NIP, Changes, Authors	16.00
SYS^1	DIT, NOC, CBO, DAC, SIZE1, XXXEC, POLY, NIP, Changes, Authors	54.53
SYS^2	WMC, NOM, SIZE1, SIZE2, XXXIC, XXXEC, POLY, Changes, Age	123.50
SYS^-	DAC, MPC, Changes, Authors, Age	12.00
$PROC^1$	WMC, CBO, RFC, MPC, NOM, SIZE1, SIZE2, XXXEC, POLY, NIP, Changes, Authors	59.72
$PROC^2$	WMC, NOC, CBO, RFC, MPC, SIZE1, SIZE2, NIP, Changes	20.00
$PROC^-$	CBO, RFC, SIZE2, XXXIC, POLY, NIP, Changes	97.42
All Systems	NOC, CBO, DAC, SIZE1, SIZE2, XXXEC, POLY, NIP, Changes, Age	265.50

The results of the model validations can be found in Table 6.3. For each sub-pool, the true positives ($T+$), true negatives($T-$), false positives ($F+$), false negatives ($F-$), accuracy, precision, and recall were calculated. The values in the table represent the average values of k models, where k is the number of folds used for validation.

The accuracies of the type-related sub-pool models are similar, but are all lower than the inverse sub-pool instead of higher, this observation does not match our expectations. Also, the sub-pool models their accuracies does not seem to differ from the all-systems pool, and probably means that the context type has no effect on prediction model accuracy.

All prediction models obtain accuracies of at least .88; these are the best results we have obtained so far and is higher than the accuracies of cross-system prediction models observed in similar studies (see Section 1.2). One explanation of the high accuracies could be because we use pools of multiple similar systems to train our model instead of using only a single system. Another explanation could be that a lot of the confounding variables were kept constant (see Section 6.2).

All the prediction models are very accurate and that none of the models differ from the models build using the all-systems pool. This could suggest that some confounding variables were at play that positively influenced the fault prediction models. It seems that some factors were kept constant, factors we did not consider.

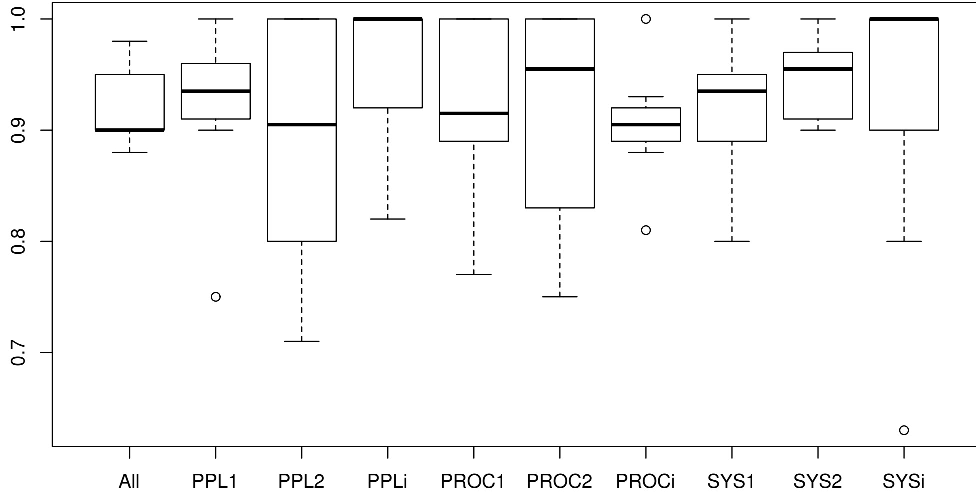
Table 6.3: Sub-pool models their prediction capabilities

	$T+$	$T-$	$F+$	$F-$	Accuracy	Precision	Recall
PPL^1	163	47	7	5	.95	.96	.97
PPL^2	25	56	8	3	.88	.76	.89
PPL^-	67	57	4	1	.96	.94	.99
SYS^1	114	36	6	9	.91	.95	.93
SYS^2	212	90	10	8	.94	.95	.96
SYS^-	50	57	3	3	.95	.94	.94
$PROC^1$	107	29	10	8	.88	.91	.93
$PROC^2$	48	15	5	6	.85	.91	.89
$PROC^-$	122	102	10	13	.91	.92	.90
All Systems	341	209	21	27	.92	.94	.93

6.4.1 Hypothesis testing

A summary of the accuracies of the sub-pools are given in Figure 6.4.1. The individual box-plots represent the accuracies obtained by the prediction models during the k-fold cross validation. The sub-pools are labelled TYPEn, were n is the sub-pool number. The inverse sub-pool is labelled TYPEi. The all-systems pools is labelled 'All'.

Figure 6.1: Parallel boxplot of the sub-pool their accuracies



Testing the effect of the people characteristics on fault-proneness prediction models. An one-way ANOVA was conducted to determine if there is a significant difference between any of the sub-pools of the people type, including the inverse sub-pool ($p \leq 0.05$). For the first hypothesis (H1), the following null hypothesis was formulated:

$$\mu_{PPL^1} = \mu_{PPL^2} = \mu_{PPL^-} \tag{6.3}$$

The results of the ANOVA test indicates that there is not a significant difference between the sub-pool means ($F(2) = 1.844$, $p = .178$). As result, the null hypothesis is not rejected. A post-hoc test was not conducted due the absence of a significant difference.

Testing the effect of the process characteristics on fault-proneness prediction models. An one-way ANOVA was conducted to determine if there is a significant difference between any of the sub-pools of the people type ($p \leq 0.05$). For the second hypothesis (H2), the following null hypothesis was formulated:

$$\mu_{PROC^1} = \mu_{PROC^2} = \mu_{PROC^-} \tag{6.4}$$

The results of the ANOVA test indicates that there is not a significant difference between the sub-pool means ($F(2) = 0.035$, $p = .966$). The null hypothesis is not rejected. A post-hoc test was not conducted due the absence of a significant difference.

Testing the effect of the factors on fault-proneness prediction models. An one-way ANOVA was conducted to determine if there is a significant difference between any of the sub-pools of the people type ($p \leq 0.05$). For the third hypothesis (H3), the following null hypothesis was formulated:

$$\mu_{SYS^1} = \mu_{SYS^2} = \mu_{SYS^-} \tag{6.5}$$

The results of the ANOVA test indicates that there is not a significant difference between the sub-pool means ($F(2) = 0.146$, $p = .865$). The null hypothesis is not rejected. A post-hoc test was not conducted due the absence of a significant difference.

Testing the effect of the confounding variables on fault prediction models. An one-way ANOVA was conducted to determine if there is a significant difference between any of the sub-pools, and the all-system pool ($p \leq 0.05$). For the fourth hypothesis, the following null hypothesis was formulated:

$$\mu_{PPL^1} = \mu_{PPL^2} = \mu_{PPL^-} = \mu_{PROC^1} = \mu_{PROC^2} = \mu_{PROC^-} = \mu_{SYS^1} = \mu_{SYS^2} = \mu_{SYS^-} = \mu_{ALL} \quad (6.6)$$

The results of the ANOVA test indicates that there is not a significant difference between the sub-pools, including the all-system pool, their means ($F(9) = 0.675$, $p = .729$). The null hypothesis is not rejected. A post-hoc test was not conducted due the absence of a significant difference.

Testing the best model's practical application efficiency. An one-tailed one-sample t-test was conducted to determine if the pool that resulted in the best models resulted in an average model with an accuracy significantly higher than .70 ($p \leq .05$). The pool was the PPL^-. The null hypothesis was formulated as follows:

$$H_0 : \mu_{accuracy} < .70 \quad (6.7)$$

the prediction models from the PPL^- pool obtained on average accuracies higher than 70% ($M = .96$, $SD = .06$). The difference is significant ($t(9) = 13.229$, $p = .000$) and the null hypothesis is therefore rejected. The Cohen's d shows a large effect ($d = 15.277$).

6.5 Conclusion and Discussion

To answer research question RQ5 based on the results of this study, we do not think that people, process, or technology related factors influence fault-proneness prediction model accuracies which are applied across systems. However, we think that there are factors, which we kept constant but which where not considered, that have a positive effect on fault-proneness prediction model accuracy.

In this study, we found no evidence that factors related to people, technology, or process affects the accuracy of the fault-proneness prediction models. No significant difference between the accuracies of the prediction models were found that kept a specific context related factors constant compared to the prediction models which did not keep those factors constant. Moreover, none of the models differed significantly in accuracy from the prediction models that was based on all systems.

The results of the model construction showed no overlap in predictors among the pools. Instead, some metrics reoccurred in every model and might tell something about the factors which are related to fault-proneness in this context. Changes was a dominant positive predictor in all models; a frequently changed class is likely to be fault-prone. A trivial explanation could be that faults are not introduced in non-changing classes and are introduced in changing classes (see Chapter 5 for a more elaborate discussion). POLY was a dominant negative predictor; a class with low polymorphism is likely to be fault-prone. This observation is in-line with the object-oriented paradigm and contradicts the conclusion Benlarbi & Melo drew; "*polymorphism may increase probability of fault in OO software*" [12]. DAC was a dominant positive measure; a class with high coupling to abstract data types is likely to be fault-prone. Interesting is that other coupling measures like XXXIC, XXXEC, and CBO (which includes the DAC measure) are not dominant predictors in any of the models.

The average accuracy of all prediction models that we build (10 models for each pool, and a total of 10 pools) was 91.5%. These are the best results we have obtained so far and is much higher than the accuracies of cross-system prediction models observed in similar studies (see Section 1.2). One explanation of the high accuracies could be because we use pools of multiple similar systems to train our model instead of using only a single system. Another explanation could be that a lot of the confounding variables were kept constant (see Section 6.2).

All the prediction models are very accurate and that none of the models differ from the models build using the all-systems pool. This could suggest that some confounding variables were at play that positively influenced the fault prediction models. It seems that some factors were kept constant, factors we did not consider.

6.5.1 Threats to validity

One threat to validity is caused by the k-fold cross-validation technique. The technique splits the data into 30/70 partitions, trains the model on the larger partition, and validates the model on the smaller partition. This method could cause an overlap of classes belonging to the same system in the training and validation set. If one system is significantly larger than the others, the model is similar to a model trained and validated on the same system, with higher accuracies compared to models validated using the cross-system validation technique as possible result.

Other threats to validity are similar to the threats described in the the replication study (Section 4.5) and the improvement study (Section 5.5). For more details regarding these threats we refer to those studies.

6.5.2 Future research

One noteworthy observation was the the high accuracy of the all-systems pool and absence of a difference between the sub-pools and the all-systems pool. This could hint at other factors that influences prediction models that we did not consider during this study. One factor we did not consider was the company from which we took our sample. It is interesting to see if there is an accuracy drop when fault proneness prediction models are build from similar systems build by different companies compared to models build using similar system form the same company. Moreover, the company we took our sample from has strict development protocols and standards, this might also have lead to the effective prediction models in this study.

Another interesting research direction is to find out if a company specific predictor-set could be composed (could be less or more general), that is able to predict with reasonable to good accuracy. Another related direction could be to replicate this research and focus on the variables that are selected during the stepwise selection procedure and see what the dominant variables are. We expect that Changes measure will often reoccur in the selected predictors.

One of or threats to validity is caused by the k-fold cross-validation. Even though the technique takes the error of using partly the same data in consideration it is still not ideal compared to hold-out or cross-system validation techniques. However, the later two techniques require a large dataset in order to produce stable prediction models. To validate the observed accuracies in this study, one could perform a replication but substituting the k-fold cross-validation by a cross-system validation technique.

Two measures beside the Change metric were dominant in most of our systems. However, we could not provide a logical explanation of why these factors are dominant. One could explore these metrics in more detail by splitting up the POLY measure and analyse the measures on a finer scale.

Bibliography

[1] B. Boehm, H. Rombach, and M. Zelkowitz, *Foundations of Empirical Software Engineering*, 2005.

[2] S. Forrest, "What we have learned about fighting defects," *Software Metrics, IEEE International Symposium on*, vol. 0, p. 249, 2002.

[3] C. Catal and B. Diri, "A systematic review of software fault prediction studies," *Expert Systems with Applications*, vol. 36, no. 4, pp. 7346–7354, 2009.

[4] L. C. Briand and W. L. Melo, "Assessing the applicability of fault-proneness models across object-oriented software projects," vol. 28, no. 7, pp. 706–720, 2002.

[5] G. Denaro, G. Denaro, S. Morasca, S. Morasca, M. Pezze, and M. Pezze, "Deriving models of software fault-proneness," *Proceedings of the 14th international conference on Software engineering and knowledge engineering - SEKE '02*, p. 361, 2002.

[6] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches," *IEEE Transactions on Software Engineering*, vol. 22, no. 12, pp. 886–894, 1996.

[7] N. Schneidewind, "Investigation of logistic regression as a discriminant of software\nquality," *Proceedings Seventh International Software Metrics Symposium*, pp. 328–337, 2001.

[8] J. C. Munson and T. M. Khoshgoftaar, "The detection of fault-prone programs," *IEEE Transactions on Software Engineering*, vol. 18, no. 5, pp. 423–433, 1992.

[9] A. Veiga and F. Barbosa, "R.; melo, w," *JMetrics Java Metrics Extractor: An Overview. University of Brasilia, Dep. of Computer Science, Under-Graduating Final Project, Brasilia, DF, Brazil*, 1999.

[10] L. Briand, P. Devanbu, and W. Melo, "An investigation into coupling measures for c++," in *Proceedings of the 19th international conference on Software engineering.* ACM, 1997, pp. 412–421.

[11] L. C. Briand, J. Wüst, S. V. Ikonomovski, and H. Lounis, "Investigating quality factors in object-oriented designs: an industrial case study," in *Proceedings of the 21st international conference on Software engineering.* ACM, 1999, pp. 345–354.

[12] S. Benlarbi and W. L. Melo, "Polymorphism measures for early risk prediction," in *Software Engineering, 1999. Proceedings of the 1999 International Conference on.* IEEE, 1999, pp. 334–344.

[13] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[14] L. C. Briand, J. Wust, J. W. Daly, and D. V. Porter, "Exploring the relationship between design measures and software quality in object-oriented systems," *Journal of Systems and Software*, vol. 51, no. 3, pp. 245–273, 2000.

[15] "Gp-based software quality prediction," *Proceedings of the Third Annual Conference on Genetic Programming*, pp. 60–65, 1998.

[16] M. M. T. Thwin and T.-S. Quah, "Application of neural networks for software quality prediction using object-oriented metrics," *Journal of systems and software*, vol. 76, no. 2, pp. 147–156, 2005.

[17] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai, "Comparing case-based reasoning classifiers for predicting high risk software components," *Journal of Systems and Software*, vol. 55, no. 3, pp. 301–320, 2001.

[18] X. Yuan, T. Khoshgoftaar, E. Allen, and K. Ganesan, "An application of fuzzy clustering to software quality prediction," *Proceedings 3rd IEEE Symposium on Application-Specific Systems and Software Engineering Technology*, no. 561, pp. 85–90, 2000.

[19] L. Guo, B. Cukic, and H. Singh, "Predicting fault prone modules by the dempster-shafer belief networks," in *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on.* IEEE, 2003, pp. 249–252.

[20] T. Khoshgoftaar and N. Seliya, "Software quality classification modeling using the sprint decision tree algorithm," *International Journal on Artificial*, pp. 365–374, 2003.

[21] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *Software Engineering, IEEE Transactions on*, vol. 33, no. 1, pp. 2–13, 2007.

[22] T. M. Khoshgoftaar, E. B. Allen, R. Halstead, G. P. Trio, and R. M. Flass, "Using process history to predict software quality," *Computer*, vol. 31, pp. 66–72, 1998.

[23] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," *Proceedings of the 27th International Conference on Software Engineering*, pp. 580–586, 2005.

[24] 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.

[25] L. Briand, J. Daly, and J. Wust, "A unified framework for cohesion measurement in object-oriented\nsystems," *Proceedings Fourth International Software Metrics Symposium*, vol. 117, pp. 65–117, 1997.

[26] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," pp. 111–122, 1993.

[27] T. J. Ostrand and E. J. Weyuker, "Predicting bugs in large industrial software systems." in *ISSSE.* Springer, 2011, pp. 71–93.

[28] A. Avivzienis, J.-C. Laprie, and B. Randell, *Dependability and Its Threats: A Taxonomy.* Springer US, 2004, no. Topic 3.

[29] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: how misclassification impacts bug prediction," in *Proceedings of the 2013 International Conference on Software Engineering.* IEEE Press, 2013, pp. 392–401.

[30] J. P. Higgins, S. Green *et al.*, *Cochrane handbook for systematic reviews of interventions.* Wiley Online Library, 2008, vol. 5.

[31] V. Pareto, *The Mind of Society Vol.1*, 1935.

[32] "Quantitative analysis of faults and failures in a complex software system," *IEEE Transactions on Software Engineering*, vol. 26, no. 8, pp. 797–814, 2000.

[33] C. Andersson and P. Runeson, "A replicated quantitative analysis of fault distributions in complex software systems," *IEEE Transactions on Software Engineering*, vol. 33, no. 5, pp. 273–286, 2007.

[34] T. G. Grbac, P. Runeson, and S. Member, "A second replicated quantitative analysis of fault distributions in complex software systems," vol. 39, no. 4, pp. 462–476, 2013.

[35] A. Oram and G. Wilson, *Making software: What really works, and why we believe it.* " O'Reilly Media, Inc.", 2010.

[36] B. Compton and C. Withrow, "Prediction and control of ada software defects," *Journal of Systems and Software*, vol. 12, no. 3, pp. 199–207, 1990.

[37] M. Kaaniche and K. Kanoun, "Reliability of a commercial telecommunications system," in *Proceedings of ISSRE '96: 7th International Symposium on Software Reliability Engineering*, 1996, pp. 207–212.

[38] C. Gini, E. Pizetti, and T. Salvemini, "Variabilità e mutabilità(variability and mutability), c. cuppini, bologna, italy, 1912," *Memorie di Metodologica Statistica. Rome, Italy: Libreria Eredi Virgilio Veschi*, 1955.

[39] R. I. Lerman and S. Yitzhaki, "A note on the calculation and interpretation of the gini index," *Economics Letters*, vol. 15, no. 3, pp. 363–368, 1984.

[40] Bluemoose. (2005) Gini coefficient diagram. [Online]. Available: https://en.wikipedia.org/wiki/File:Economics_Gini_coefficient.png

[41] N. Fenton and J. Bieman, *Software Metrics: A Rigorous and Practical Approach*, third edit ed. CRC Press, 2015.

[42] G. Singh, D. Singh, and V. Singh, "A study of software metrics," *IJCEM International Journal of Computational Engineering & Management*, vol. 11, pp. 22–27, 2011.

[43] H. Hotelling, "Analysis of a complex of statistical variables into principal components." *Journal of educational psychology*, vol. 24, no. 6, p. 417, 1933.

[44] L. I. Smith, "A tutorial on principal components analysis introduction," *Statistics*, vol. 51, p. 52, 2002.

[45] H. Abdi and L. J. Williams, "Principal component analysis," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 2, no. 4, pp. 433–459, 2010.

[46] D. W. Hosmer, S. Lemeshow, and R. Sturdivant, *Applied Logistic regression*, 2013.

[47] R. Kohavi *et al.*, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Ijcai*, vol. 14, no. 2, 1995, pp. 1137–1145.

[48] Woodstone. (2014) Gini coefficient for distribution with only two income or wealth levels. [Online]. Available: https://commons.wikimedia.org/wiki/File:Gini_coefficient_for_distribution_with_only_two_income_or_wealth_levels.svg

[49] D. V. Cicchetti, "Guidelines, criteria, and rules of thumb for evaluating normed and standardized assessment instruments in psychology." *Psychological assessment*, vol. 6, no. 4, p. 284, 1994.

[50] A. Cartlidge, A. Hanna, C. Rudd, I. Macfarlane, J. Windebank, and S. Rance, "An introductory overview of itil v3," *The UK Chapter of the itSMF*, 2007.

[51] J. A. O'Brien and G. Marakas, *Introduction to information systems.* McGraw-Hill, Inc., 2005.

[52] D. A. Wheeler, "More than a gigabuck: Estimating gnu/linuxs size," 2001.

[53] D. Spinellis, "Tool writing: a forgotten art?(software tools)," *Software, IEEE*, vol. 22, no. 4, pp. 9–11, 2005.

[54] T. J. Mccabe, "A complexity measure," no. 4, pp. 308–320, 1976.

Appendix A

Tooling

This appendix contains an overview of all tools that were build by the author and used in this thesis. The source code of these tools can be found on GitHub[1].

A.1 File Selection

Before we analyse the systems using any of the tools, we first filter the the files. The filter excludes three types of files: non-Java files, automatically generated files, and test files.

- *Non-Java files.* Because most of the systems in our dataset are Java projects and some tools are language specific (e.g. depend on language specific compilers), we will exclude all non-Java files, that is all the files that do not contain a Java extension *.java*. Even if a file contains embedded Java but has not the right extension, then the the Java snippet is simply ignored. The result is that none of the tools will consider non-Java files.

- *Automatically generated files.* We decided to ignore automatically generated files. This include all files that are automatically generated an are meant to not be edited manually (e.g. classes generated by parser generators). The inclusion of these classes could lead to less accurate predictions because the perceived complexity of the class, for example, could be high but will the class will never contain a fault (assuming the generator work correctly); if a fault is found, then it would be in the source file and not the generated file. The selection tool uses a heuristic for detecting auto-generated files and ignores them if detected. We check the first 15 lines of comment, if in those lines there is any mentioning of 'generated' (case-insensitive) the file will be ignored. This heuristic filters out most of the generated files but not guarantees that all generated files are filtered out.

- *Test files.* We choose to exclude all test classes in the system. Test classes could negatively influence the fault-proneness prediction. When a fault is discovered in a test class then it is highly probable that as a result there are also faults in the parts that the test covers. As a result, the balance of faults is more evenly distributed. Also, the test classes are not a part of the system itself but more a tool to find faults in the system. We label a class as test class if it include one of the following annotations in the first 30 lines: $@Test$, $@Before$, or $@After$. This also include annotations as $@BeforeSuite$ and $@Test(timeout = 1000)$. This check covers all annotations of both the *JUnit* and *TestNG* libraries. This is a heuristic and it is possible that a test related file will slip through the filter.

A.2 Lines of Code Counter

We used our own tool, *xloc*, for counting various types of lines of code. The reason behind this decision is because their is a lack of tools that correctly count the lines of code or could not easily be extended

[1] https://github.com/scrot/mt

to fit our purposes. We considered the following tools.

- *cloc*[2]. A popular open-source tool written in Perl that counts black lines, comment lines, and physical lines of source code in many languages. This tool wrongly classifies code that contains multi-line comment start symbols (for example '/*') within a string and therefore we did not used it.

- *sloccount*[3]. A tool for counting physical Source Lines of Code in a large number of languages of potentially large set of programs. It was used in a paper by Wheeler [52] and counts as far as we know the lines of code of Java projects correctly. The downside is that it is not easy to use as a Java library, the comment lines of code and logical source lines are not collected, and it does not provide capabilities to ignore files except for generated files. However, we did verified our tool with this program to check if the outputs are the same. We tested it on large open-source projects (e.g. intellj, junit4, fdroid, spring) and found almost identical source lines of code for those projects.

- *locMetrics*[4]. A tool that counts the lines of code in various was. For example total lines of code, blank lines of code, comment lines of code, logical source lines of code physical executable source lines of code. The downside is that it is closed source and not usable as a Java library. We also verified our tool with this tool on the large open-source projects and found similar lines of code when we set our tool to not ignoring auto-generated files (some lines where of by one or two).

Our tool counts three types of lines of code: Source lines of code, comment lines of code, and blank lines of code. For the source lines of code, we count the physical source lines of code and not the logical. The difference is that logical counts the semi-columns, open brackets, and close brackets that are not part of a comment; where physical counts all the lines of code that are not comment or blank lines. There for the definition we used as a source line of code is:

A source line of code is a line that is not solely a comment nor is it blank.

In other words, every line that is blank, meaning it contains no characters or only white-space characters, is not a code line. Also, every line that only holds only commentary is neither a code line. In the latter case, when a line contains commentary as well as source code and the source code is not part of the comment than the line is counted as source code and not as comment.

A.3 Git Crawler

We automatically discover faults by the use of our own tool named *gcrawler*. The tool takes a git directory, the folder containing '.git' folder, and optionally a reference to the issue tracking system. It collects information about commits, issues, faults, and authors on class-level. Our tool makes use of several libraries: JGit for crawling local directories; gitlab-api[5] for receiving the issues from the issue tracker system of Gitlab; github-api[6] for receiving the issues from the issue tracker system of Github.

The tool has two methods to discover faults. The first method requires an issue tracking system and formal issue closing through commits, the other method does not require an issue tracking system and classifies faults using semantic analysis of the commits.

- *No issue tracker.* There is not always an issue tracker used for managing issues, we have projects in our dataset that does not use them. Therefore, we have to rely on more unreliable sources. We use the commit messages to identify 'issue commits', by looking if the messages uses one of the following words (case-insensitive): fix, fixes, fixed, close, closed, closes, resolve, resolves, resolved. If this is the case, then the commit is a 'issue commit' and we assume that all classes related to the commit in question contained a fault.

[2] https://github.com/AlDanial/cloc
[3] http://www.dwheeler.com/sloccount
[4] http://www.locmetrics.com/
[5] https://github.com/gmessner/gitlab-api
[6] http://github-api.kohsuke.org

- *Issue tracker.* A map from commit to issues is build for all commits that formally closes an issue[7] and if the issue is labelled as a bug. Next all the files that are changed by one of the commits in the 'issue commit' map are assumed to contain faults.

A.4 Byte-Code Metric Suite

There are a lot of metric-suites available that could be used on Java systems [8]. However most of the tools are not free to use, do not analyse on class-level, or only contains a small and often incomplete set of metrics. One criteria for the metric-suite is that it contains metrics from the literature and not just only simple count metrics. We considered the following free metric-suites:

- *JMetrics*[9]. A metric-suite used by Briand et al. [4]. The tool was developed internally by Oracle Brazil and never published for the public.
- *CCCC*[10]. A tool which analyses and generates a report on various metrics of the code. Metrics supported include lines of code, McCabe's complexity and metrics proposed by Chidamber & Kemerer and Henry & Kafura. This tool didn't had support for annotated types (e.g. '@') and misses metrics lilke RFC and LCOM, therefore it was not usable for our purposes.
- *jDepend*[11]. A metrics suite containing Martin's Software Package Metrics. These metrics are interesting, for example Ca metric could be used on class level instead of package level, but the tool does not provide support for class-level analysis.
- *Metrics*[12]. A metric-suite measuring containing metrics proposed by Henderson, and Martin's Software Package Metrics. However this metric-suite depends on Eclipse 3.1 and misses metrics like RFC and CBO metrics.
- *JMetric*[13]. A metric-suite for analysing Java projects. The metric suite is abandoned since 2000 and supports only up to Java 1.1. Moreover, the tool only provided a very limited set of metrics: statement count, LCOM, and cyclomatic complexity.
- *Dependency Finder*[14]. Contains besides some other features also a metric suite. However, the metrics are mostly simple count measures and package dependency metrics. Therefore this metric-suite was not suited for our analysis.
- *ckjm*[15]. A Java program that calculates all Chidamber & Kemerer metrics (WMC, DIT, NOC, CBO, RFC, and LCOM), Martin's Ca (class-level), and NPM (Number of Public Methods). However, a couple of metrics, NOC and DIT, do not work on modern Java projects. This is because the way to discover super classes was not properly implemented, with as a result that these values were always 0. And some metrics were incorrectly implemented, for example WMC, their implementation did not took the complexity of the methods into consideration and therefore was not different from just counting the number of methods in a class. For these reasons, this tool is not suited for our analysis. However the method of extracting class information (by analysing byte-code using Apache's BCEL library) is efficient and is also used by our tool. More about this tool can be found in the companying paper by Spinellis [53].

The tool makes use of the Apache BCEL library for analysing the byte-code of the systems [16] and is inspired by *ckjm*. Our metric suite extends the Chidamber-Kemerer metric suite with the metrics proposed by Li & Henry. Furthermore, it includes eight of Briand et al. their class coupling metrics

[7] http://docs.gitlab.com/ee/customization/issue_closing.html
[8] http://www.monperrus.net/martin/java-metrics
[9] JMetrics: java metrics extractor [9]
[10] http://cccc.sourceforge.net
[11] http://clarkware.com/software/JDepend.html
[12] http://metrics.sourceforge.net
[13] https://sourceforge.net/projects/jmetric
[14] http://depfind.sourceforge.net
[15] http://www.spinellis.gr/sw/ckjm
[16] https://commons.apache.org/proper/commons-bcel/apidocs

(excluding the friend relation metrics), and Benlarby & Melo their polymorphism measures. The metrics in *bcms* can be found in Table A.1.

A.5 Fault Prediction Metric Suite

For the predictor-set, we extended *bcms* to include some other measures. These measures contains change metrics that are based on Koshgoftaar's change metrics, and the fault count metric. For the collection of these metrics we used our own git crawler, *gcrawler*. An overview of *fpms* is given in Table A.1.

Table A.1: *fpms* overview

Metric	Description and implementation
WMC	Weighted Methods per Class, the count of sum of complexities of all methods in a class. We calculate the complexity by analysing the byte-code, therefore the complexity does not always maps directly to the complexity as observed in the source code. This is because their could be complier optimizations in between, for example conditional operators (e.g. && and ||) are simplified. As complexity measure we use McCabe cyclomatic complexity [54]. We count the method itself and all the branch instructions (labelled as branch instruction by BCEL) except for GOTO instructions, this is because these instructions interfere with Select instructions (e.g. cases of a switch conditional). We also count all the exceptions (including the try/catch) of the method. We do not count the switch statement itself.
NOC	Number of Children, number of immediate subclasses in a hierarchy. The NOC is calculated by counting all the classes that have this class as their super class.
RFC	Response for a Class, this is the union of all methods of a class and all the methods their invocations (see Section 2.3.1 for the formal definition). We analyse the instructions of the byte-code of the class; of each instruction that is an invoke instruction (e.g. static, special, virtual), the method signature is added to the set of response-set; the methods their own signatures are also added to the responses set. A implementation decision we made is that we count the constructor and destructor also as a method, we also take into account their $<$ init $>$ methods. The RFC is the size of the response-set.
CBO	Coupling between objects, the count of the number of other classes it is coupled to. Two classes are coupled when methods in one class use methods or instance variables of another class. We implemented the CBO in a slightly different manner. We not only count the instance variables and methods the class is coupled to but also the invoked method their arguments, local variables of the method, the return type of the method, the exceptions of the method, and the class its interfaces.
DIT	Depth of inheritance, maximum number of steps from the class node to the root of the tree and measured by the number of ancestor classes. DIT is represented by our tool by the number of all ancestors of the class in question. This includes the ancestors of the external classes and classes from the Java library. Because Java doesn't allow multiple inheritance there is only a single depth to consider, this depth is used.

LCOM Lack of Cohesion in Methods, measure of dissimilarity of methods in a class by looking at the instance variables or attributes used by methods. Every variable of the PUTFIELD byte-code instruction is added to the set of the method if and only if this variable is also among the instance variables of the class; this is done for every method in that class. The LCOM is calculated by subtracting number of 'method instance set pairs' that result an empty-set from the 'method instance set pair' that result in a non-empty set. The method instance set pairs are the intersection-set of all method instance sets (see Section 2.3.1 for the formal definition).

MPC Message Passing Coupling, the number of send statements defined in a class. We count every invoke instruction within a method as one message that is being passed.

DAC Data Abstraction Coupling, the number of abstract data types defined a class. We consider the local instances of a class and only count the fields if they are not part of an external library and if the reference class is abstract or an interface.

NOM Number of Local Methods. The number of the local methods in a class are all methods which are accessible outside the class. We count only the public methods, static or not static.

SIZE1 Number of semicolons in a class. Because we want to keep the tool only dependent on the byte code we will not count the number of semicolons (statements) but the number of instructions and the number of variable fields. This count is only a representation of the code size, because there is no one-to-one mapping from statement to instruction (single line statements could translate in more instructions).

SIZE2 Number of attributes and local methods. We count all the methods and all the instance fields of a class.

ACAIC Number of import class-attribute couples where the attributes are ancestor classes. For each class A, we collect all attributes types of the class and count the attributes types that are ancestors of class A.

ACAEC Number of export class-attribute couples where the attributes are ancestor classes. For every class A, if another class B refers to class A from one of its attributes and class B is an ancestor of class A, then we count it as one ancestor export couple.

DCAIC Number of import class-attribute couples where the attributes are descendent classes. For each class A, we collect all attributes types of the class and count the attributes types that are descendent of class A.

DCAEC Number of export class-attribute couples where the attributes are descendent classes. For every class A, if another class B refers to class A in one of its attributes and class B is a descendant of class A, then we count it as one descendant export couple.

OCAIC Number of import class-attribute couples where the attributes are non-ancestor/descendant classes. For each class A, we collect all its attribute types and count only the attribute types that are neither ancestors nor descendants of class A.

OCAEC Number of export class-attribute couples where the attributes are non-ancestor/descendant classes. For every class A, if another class B refers to class A in one of its attributes and they have no descendant or ancestor relation, then we count it as one export couple.

ACMIC Number of export class-method couples where the method arguments are ancestor classes. For every class A, if another class B refers to class A from one of its method arguments and class B is an ancestor of class A, then we count it as one ancestor export couple.

ACMEC	Number of export class-method couples where the method arguments are ancestor classes. For every class A, if another class B refers to class A from one of its method arguments and class B is an ancestor of class A, then we count it as one ancestor export couple.
DCMIC	Number of import class-method couples where the method arguments are descendent classes. For each class A, we collect all method argument types of the class and count the method argument types that are descendent of class A.
DCMEC	Number of export class-method couples where the method arguments are descendent classes. For every class A, if another class B refers to class A in one of its method arguments and class B is a descendant of class A, then we count it as one descendant export couple.
OCMIC	Number of import class-method couples where the method arguments are non-ancestor/descendant classes. For each class A, we collect all its method argument types and count only the method argument types that are neither ancestors nor descendants of class A.
OCMEC	Number of export class-method couples where the method arguments are non-ancestor/descendant classes. For every class A, if another class B refers to class A in one of its method arguments and they have no descendant or ancestor relation, then we count it as one export couple.
OVO	Overloading in stand-alone classes. The number of times a method is overloaded. We do not count the overloaded method itself.
SPA	Static polymorphism in ancestors. The number of methods that are statically overloaded by other classes, where the class in consideration is it ancestor. All method with the same name but a different method signature are counted as static overloaded methods. The metric count is not symmetric, here this metric differs from the original metric which states the class pairs are symmetric. We made this implementation decision to prevent double counting of relations. For example, there is an ancestor relation between class A and class B, then there is also a descendant relation between B and A. But because the relations are symmetric the following sets are formed: $SPA\{(a,b),(b,a)\}$, $SPD\{(a,b),(b,a)\}$, and $SP(a,b),(b,a),(a,b),(b,a)$, which includes unnecessary double counts.
SPD	Static polymorphism in descendants. The number of methods that are statically overloaded by other classes, where the class in consideration is it descendant. All method with the same name but a different method signature are counted as static overloaded methods. The metric count is not symmetric, see SPA.
SP	Static polymorphism in descendants and ancestors. $SPA + PSD$.
DPA	Dynamic polymorphism in ancestors. The number of methods that are dynamically overloaded by other classes, where the class in consideration is it ancestor. All method with the same name and a method signature are counted as dynamically overloaded methods. The metric count is not symmetric, see SPA.
DPD	Dynamic polymorphism in descendants. The number of methods that are dynamically overloaded by other classes, where the class in consideration is it descendant. All method with the same name and a method signature are counted as dynamically overloaded methods. The metric count is not symmetric, see SPA.
DP	Dynamic polymorphism in descendants and ancestors. $DPA + DPD$.
NIP	Polymorphism in non-inheritance relations. We count all non ancestor or descendent class pairs that share a common method name. We excluded $<$ init $>$ methods from this count, because otherwise it results in a relationship between all non-static classes.
Changes	The total number of changes within a class. This metric is the total number of commits that changed parts of the class. Note that a change can be an addition, deletion, alteration, or any combination of those. It is possible that a single commit changed multiple classes, in this case the count of all these classes is increased.

Authors	Number of unique authors that changed the class in the past. For this count we get all the commits that changed parts of the class. From these commits, we collect the full names of the commit authors and count all unique names. We decided to not base our count on e-mail or author-id because it is possible that the same person commits using a different e-mail or id (e.g. work account and home account).
Age	Number of days between the first commit and last commit of that class. For this measure we locate the first commit and the last commit associated with the class in question and calculate the difference in days between the dates that come with those commits. For the purpose of calculating the difference in date, we use a third party library called *joda-time*.
Faults	The total number of changes to a class that are done in the light of resolving a fault. The collection of faults in a class is a subset of the collection of changes to a class. None of the systems makes use of an issue tracker so we use the no-issue tracker implementation of the *gcrawl* tool. Therefore, we consider a change a fault if the associated commit contains one of the following terms within its commit message (case insensitive): fix, fixes, fixed, resolve, resolves, resolved, close, closes, closed (see *gcrawler* for a more detailed description).

A.6 System Overview Measures

Most of the factor measurements are manually obtained. The manual data comes from mining the JIRA- and GitLab systems; and by interviewing the developers, architects, and business people whom were involved during the development of the system. The rest of the data is collected using our own tool called *ovms*. The data from this tool is obtained by analysing the local git and source-code using *gcrawler* and *xloc*. See Table A.2. All measures labelled with '*' are calculated using the *ovms*.

Table A.2: System overview measures

Characteristic	Description
Team	The team that developed and maintained the system. For each system holds that the team that developed it also maintains it. Moreover, the teams are multidisciplinary devOps teams and no divisions are made between teams responsible for developing, reviewing, or testing. It is possible that more than one team developed or maintained the system, in this case the team who worked most on the system is picked.
Size	The number of people in the team. This count includes operations related people and excludes management and business related people. Also the architect and product manager are not included.
Exp.	The average working experience of the people in the devOps team. The years of the team member worked at the company in question is used as the individual years of experience.
Arch.	The enterprise architect of the system. The architect is involved on a high abstraction.
Proj.	The project tag of the system.
Diff.	The number of teams who worked at the system. The count is the number of unique teams that developed or maintained the system.
Type.	Type of system, based on the kind of service it offers (e.g. back-end system, API, front-end)

*Files** — The count of all the files in the root directory of the system. For the system totals, this count includes also build files, configuration files, hidden files, and test files. That is, everything is counted recursively in the root path that is not a directory. For the considered system total this is the count of all files with the extensions of the language under consideration. This measure is automatically calculated.

*SLOC** — The count of all source lines of code of the system. We only count the SLOC of the files of the specified programming language. A line is considered source code if it is not a comment line or a blank line and contains a line-end (see Section **??** for a more precise definition of a source line of code). A file is only analysed if it contains the right extension, meaning that if source code of the specified language is embedded in another file it will not be counted. This measure is automatically calculated.

*CLOC** — The count of all comment lines in the system. This count includes single-line and multi-line comments, as well as other forms of in-lined documentation (e.g. javadoc for Java). When a line contains comments as well as source code then the line will be counted as source code line and not as a comment line. When a line contains comments and blanks, the line is counted as a comment line. This measure is automatically calculated.

*FDist20** — Number of faults in the first 20% of the classes in descending order of faults. For example, the value 80 means that 80 percent of the the total faults resides in 20% of the classes.

*CinF20** — The percentage of code in the fault distribution partition. For example, the value 30 means that a fault distribution partition of 20 percent contains 30 percent of the total code.

*FGini** — The Gini-coefficient of the fault distribution of the system. A value between 0 and 1, where 0 means complete equality and 1 means complete equality.

Budget — The planned cost of building the system. The budget is an ordinal scale where 1 is the cheapest category and 5 the most expensive.

Epics — The number of Epics. This value is taken from a Jira-system

Stories — The number of stories in the Jira-system.

Owner — The product owner of the project. Not all systems have a product owner.

Meth — The development method used (e.g. SRUM or RUP).

*Faults** — The total number of files that were changed by a commit that closed an issue. A fault is found by pattern matching the commits to find issue related commits (see Section **??**); finding the files that were changed in that commit; and finally count each time a file is changed as a single fault. Note that not all faults are found this way, it is possible that one recovers a fault before it ever reaches the issue tracker system or if he closes the issue without mentioning it in the commit message. For the considered system totals only faults are counted if it resides in the considered files with the right extension. This measure is automatically calculated.

*Changes** — The total number of file changes observed in the version control system. This aggregates all the individual files that were changed in a single commit over the total commits in the master branch. There is no difference between a small change to a file and a large one, both are counted as a single file change. When a new file is created or removed, it is also counted as a change. This measure is automatically calculated.

*Authors** — The total number of unique authors that contributed to the system. This measure is the total of unique authors, identified by their full name, that has once committed. For the considered system totals, only the authors were counted if the author's commit changed a considered file.

*Age**	The number of days between the first activity in the version management system and the date the system was analysed. A system's age is therefore represented by the difference in whole days between the first commit in the version management system (see *Development Duration*) and the date the system was analysed.
*Dev**	The number of days there was activity in the version management system of the system in question. This is the number of whole days between the first activity and the last activity. In other words, it is the difference between the first commit and the last commit that was available in online in the version management system. Note, only the commits of the main branch are taken into consideration, so none of the activity on the branches will be visible in this measure. Also, commits that are not necessarily related to code changes are also included, for example the merge-requests related commits. The number of days are the absolute difference between the two dates, also counting holidays, weekends, and so on. This measure is automatically calculated.

Appendix B

Preliminary Study: Fault Distribution Histograms

Figure B.1: Histogram of the percentage of faults in the 20% most faulty classes

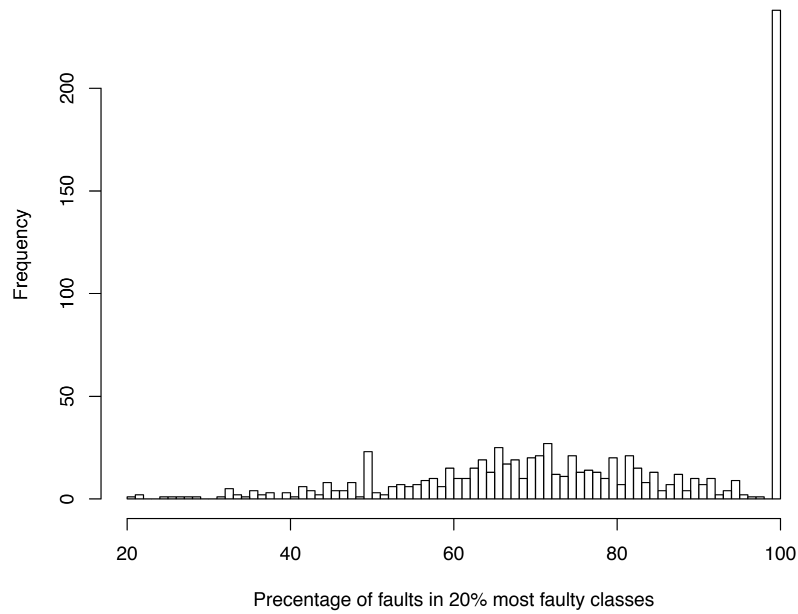


Figure B.2: Histogram of the percentage of code in the 20% most faulty classes

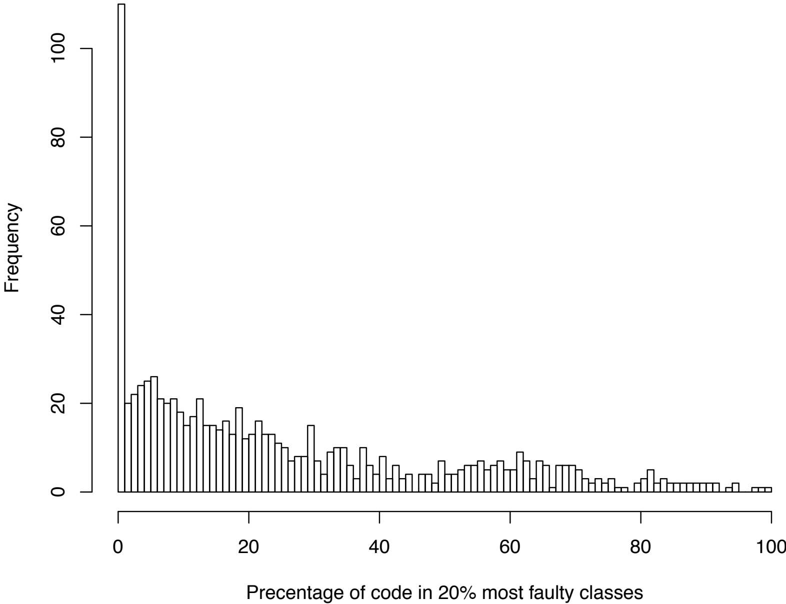
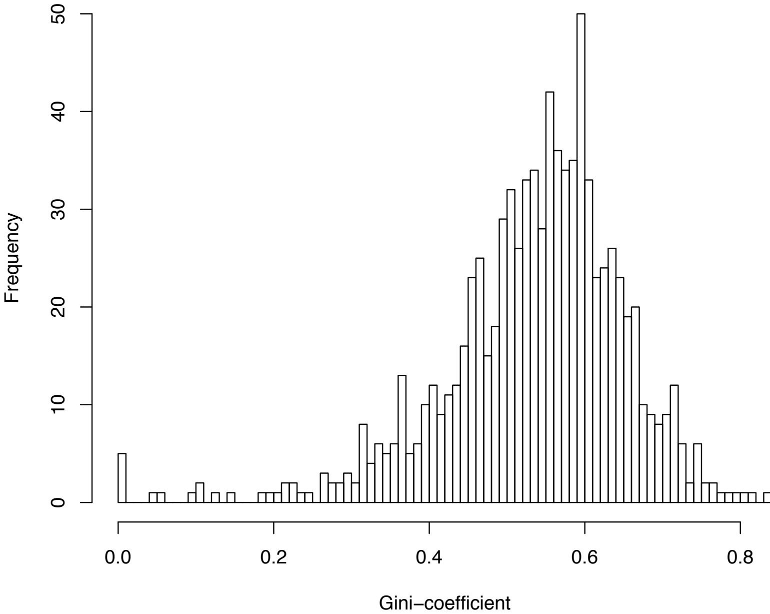


Figure B.3: Histogram of the fault distribution Gini coefficients



Appendix C

Replication Study: Principal Component Analyses

Table C.1: PCA MAdd

	RC1	RC6	RC2	RC7	RC5	RC3	RC4
Eigen	5.04	2.08	1.58	1.40	1.06	1.04	1.02
IPerc	0.34	0.14	0.11	0.09	0.07	0.07	0.07
CPerc	0.34	0.48	0.58	0.67	0.74	0.81	0.88
WMC	0.18	0.62	0.53	0.40	-0.10	0.04	-0.01
DIT	-0.11	-0.11	-0.02	-0.06	**0.98**	-0.06	-0.03
CBO	**0.96**	0.08	-0.06	0.03	-0.06	0.03	0.00
RFC	**0.91**	0.25	0.04	0.10	-0.13	-0.11	0.10
DAC	**0.84**	0.19	-0.14	-0.25	0.02	0.01	-0.05
MPC	**0.82**	0.22	0.26	0.19	-0.09	0.01	0.03
NOM	0.21	**0.91**	0.25	0.20	-0.15	-0.06	0.02
SIZE1	0.65	0.37	0.37	0.35	-0.03	0.01	0.00
SIZE2	0.64	0.63	0.20	0.19	0.06	-0.11	-0.01
OCAIC	**0.84**	-0.04	0.18	0.16	-0.08	-0.02	0.02
OCAEC	-0.03	-0.06	0.10	-0.01	-0.06	**0.98**	-0.10
OCMIC	0.54	0.05	0.12	0.17	-0.09	0.11	0.07
OCMEC	0.05	0.29	**0.92**	0.19	0.00	0.12	-0.07
OVO	0.04	0.01	-0.06	-0.03	-0.03	-0.10	**0.99**
NIP	0.10	0.29	0.23	**0.91**	-0.07	-0.01	-0.04

Table C.2: PCA MIntO

	RC1	RC3	RC6	RC4	RC5	RC2
Eigen	4.42	2.49	1.26	1.16	1.15	1.03
IPerc	0.37	0.21	0.11	0.10	0.10	0.09
CPerc	0.37	0.58	0.68	0.78	0.87	0.96
WMC	0.66	0.59	0.29	0.16	0.13	-0.01
DIT	0.42	0.27	**0.84**	0.04	0.21	0.03
CBO	**0.91**	0.00	0.20	0.24	0.20	-0.03
RFC	**0.81**	0.33	0.30	0.25	0.16	-0.10
MPC	**0.84**	0.22	0.34	0.19	0.16	-0.12
NOM	0.17	**0.94**	0.13	0.20	0.16	0.04
SIZE1	**0.71**	0.48	0.34	0.18	0.23	-0.04
SIZE2	0.38	**0.73**	0.25	0.31	0.29	0.00
OCAIC	0.30	0.32	0.05	**0.89**	0.12	-0.05
OCAEC	-0.06	0.02	0.01	-0.04	0.07	**0.99**
OCMIC	**0.87**	0.31	0.00	0.06	0.14	0.05
NIP	0.28	0.26	0.18	0.12	**0.90**	0.10

Table C.3: PCA MIra

	RC1	RC3	RC7	RC2	RC4	RC5
Eigen	5.13	2.01	1.20	1.14	1.13	1.02
IPerc	0.39	0.15	0.09	0.09	0.09	0.08
CPerc	0.39	0.55	0.64	0.73	0.82	0.89
WMC	**0.90**	0.12	0.25	-0.01	0.12	0.19
DIT	0.20	0.15	**0.87**	-0.03	0.33	0.21
CBO	0.12	**0.85**	0.31	-0.01	0.15	0.33
RFC	**0.82**	0.44	-0.05	-0.19	0.10	-0.02
MPC	**0.91**	0.25	0.01	-0.12	0.17	0.03
NOM	**0.82**	0.05	0.16	-0.15	0.02	0.23
SIZE1	**0.89**	0.17	0.15	-0.12	0.26	0.18
SIZE2	**0.81**	0.20	0.20	-0.24	0.14	0.24
OCAIC	0.33	**0.93**	-0.03	-0.09	0.02	-0.03
OCAEC	-0.25	-0.09	-0.03	**0.95**	-0.09	0.13
OCMIC	0.37	0.23	0.28	0.24	0.11	**0.79**
OCMEC	0.53	-0.01	0.18	0.06	0.16	0.18
NIP	0.26	0.07	0.28	0.10	**0.91**	0.08

Table C.4: PCA MPen

	RC1	RC3	RC2	RC5	RC4
Eigen	4.16	4.08	1.92	1.39	1.05
IPerc	0.30	0.29	0.14	0.10	0.08
CPerc	0.30	0.59	0.73	0.82	0.90
WMC	0.31	**0.93**	-0.02	0.12	0.07
DIT	-0.01	0.04	-0.16	0.06	**0.98**
CBO	**0.87**	0.21	0.14	0.19	0.06
RFC	**0.78**	0.56	-0.07	0.14	-0.03
DAC	0.64	0.11	0.01	0.49	0.15
MPC	**0.89**	0.39	-0.04	0.11	-0.08
NOM	0.13	**0.97**	-0.17	0.01	-0.01
SIZE1	0.49	**0.79**	0.02	0.27	0.07
SIZE2	0.20	**0.94**	-0.11	0.22	0.01
OCAIC	0.48	0.34	-0.19	0.28	-0.09
OCAEC	0.14	-0.11	**0.96**	0.05	-0.04
OCMIC	**0.96**	0.16	0.05	-0.02	-0.01
OCMEC	-0.09	-0.12	**0.92**	0.10	-0.18
NIP	0.15	0.29	0.16	**0.92**	0.05

Table C.5: PCA MRep

	RC1	RC11	RC3	RC4	RC9	RC6	RC2
Eigen	4.25	1.46	1.16	1.14	1.10	1.07	1.02
IPerc	0.30	0.10	0.08	0.08	0.08	0.08	0.07
CPerc	0.30	0.41	0.49	0.57	0.65	0.73	0.80
WMC	**0.92**	0.16	-0.03	-0.02	0.06	0.09	0.14
DIT	0.04	0.13	**0.97**	0.03	0.15	0.01	-0.06
CBO	0.17	0.29	0.22	0.29	**0.84**	0.01	0.03
RFC	0.35	**0.76**	0.26	0.11	0.33	0.02	-0.08
DAC	0.04	0.05	0.03	**0.95**	0.19	-0.06	-0.14
MPC	0.68	0.67	0.06	0.01	0.18	0.01	-0.01
NOM	**0.71**	0.12	-0.01	0.00	0.08	0.05	0.16
SIZE1	**0.88**	0.35	-0.01	0.04	0.14	0.07	0.06
SIZE2	0.62	0.28	0.14	0.13	0.31	-0.03	-0.01
OCAIC	0.32	0.29	0.18	0.20	0.18	-0.06	-0.02
OCAEC	0.16	0.01	0.02	-0.06	0.01	**0.94**	0.26
OCMIC	0.43	0.10	-0.06	0.22	0.16	0.20	0.06
OCMEC	0.21	-0.04	-0.07	-0.15	0.02	0.29	**0.92**
NIP	**0.90**	0.03	0.16	0.10	0.04	0.19	0.15

Appendix D

Factor Study: Pools

Table D.1: People pool PPL_{A1} overview

	Team	Size	Exp.	Arch	Proj.	Diff.
DSto	B	5	3	B	BD	1
Doc	B	5	3	B	BD	1
Sec	B	5	3	B	BD	1

Table D.2: People pool PPL_{A2} overview

	Team	Size	Exp.	Arch	Proj.	Diff.
IRat	D	10	3	D	TRA	2
MAdd	D	10	3	D	EA	2
MApp	D	10	3	D	TRA	2
MIra	D	10	3	D	TRA	2

Table D.3: People pool PPL_{A-} overview

	Team	Size	Exp.	Arch	Proj.	Diff.
BIra	C	2	1	-	TRA	2
DSto	B	5	3	B	BD	1
MAdd	D	10	3	D	EA	2
MDue	F	5	2	E	OPEN	1

Table D.4: System pool SYS_{A1} overview

	Type	Files	SLOC	CLOC	Flt.	Chg.	Age	Dev.
MIntO	SCli	46	2.204	486	93	334	1.102	686
MIra	SCli	28	1.186	228	29	97	255	112
MRep	SCli	85	2.603	507	72	610	883	446

Table D.5: System pool SYS_{A2} overview

	Type	Files	SLOC	CLOC	Flt.	Chg.	Age	Dev.
MAdd	API	80	4.012	740	38	264	305	159
MApp	API	15	579	77	6	36	295	72
MDue	API	33	1.730	185	53	294	189	181
Doc	API	83	2.666	771	86	567	688	616
Sec	API	100	4.511	1147	254	1114	677	547

Table D.6: System pool SYS_{A-} overview

	Type	Files	SLOC	CLOC	Flt.	Chg.	Age	Dev.
BIra	BEnd	30	3.023	854	8	113	213	68
ESec	API	100	4.511	1147	254	1114	677	547
MIntO	SCli	46	2.204	486	93	334	1.102	686

Table D.7: Process pool $PROC_{A1}$ overview

	Budget	Stories	Owner	Meth.
DSto	3	17	No	SCRUM
MRep	2	5	No	SCRUM
MDue	1	18	No	SCRUM

Table D.8: Process pool $PROC_{A2}$ overview

	Budget	Stories	Owner	Meth.
MApp	5	28	Yes	SCRUM
MIntO	5	17	Yes	SCRUM
MIra	5	29	Yes	SCRUM

Table D.9: Process pool $PROC_{A-}$ overview

	Budget	Stories	Owner	Meth.
ESec	4	?	No	SCRUM
MAdd	3	21	Yes	SCRUM
MApp	5	28	Yes	SCRUM
MRep	2	5	No	SCRUM
MDue	1	18	No	SCRUM

Appendix E

Factor Study: Selected Variables

Table E.1: Selected predictors details

Pool	Intercept	WMC	DIT	NOC	CBO	RFC	LCOM	DAC	MPC	NOM
PPL1	-4.842280							1.176630		
PPL2	-11107.38	-2888.17	1855.08					-4658.59	85.56	6268.56
PPLi	-7408.1		831.1					1382.5		
PROC1	-5.8801	0.7968			-0.7209	1.5161			-1.5842	-1.6344
PROC2	-56.56	-138.23		890.59	-209.30	177.61			-72.76	
PROCi	-5.14652				-0.24409	0.18408				
SYS1	-18.25894		9.26508	6.40841	-1.32921			3.70824		
SYS2	-4.788811	-0.210393								0.613347
SYSi	-295.8221							-78.5107	1.3130	

Table E.2: Selected predictors details ctnd.

Pool	Intercept	SIZE1	SIZE2	XXXIC	XXXEC	POLY	NIP	Changes	Authors	Age
PPL1	-4.842280					-1.398302		0.677581		0.006416
PPL2	-11107.38	254.40	-2829.11	137.63		-9661.15	-250.84	3643.22	-6322.07	57.11
PPLi	-7408.1		-422.5		-824.4		279.4	2797.4	-3332.0	
PROC1	-5.8801	0.1904	-1.0041		-0.4643	-3.5089	-0.2876	1.6544	1.3840	
PROC2	-56.56	21.94	-56.57				-14.01	101.53		
PROCi	-5.14652		-0.21287	0.24532		-1.18947	-0.08251	1.25279		
SYS1	-18.25894	0.08785			2.88863	-3.33703	-0.38192	2.09586	1.10703	
SYS2	-4.788811	0.020308	-0.344334	0.177291	-0.343518	-1.574524		0.807736		0.005448
SYSi	-295.8221							75.8287	-70.3464	0.6559
ALL	-3.815097	0.010916	-0.057737		-0.199186	-1.197775	-0.054640	0.727860		0.004274