INAUGURAL – DISSERTATION

zur
Erlangung der Doktorwürde
der
Naturwissenschaftlich-Mathematischen Gesamtfakultät
der
Ruprecht – Karls – Universität
Heidelberg

VORGELEGT VON

Diplom-Informatikerin Timea Monika Illes-Seifert
aus Eisenmarkt (Siebenbürgen)

TAG DER MÜNDLICHEN PRÜFUNG:

JUSTIFIED TEST FOCI DEFINITION
AN EMPIRICAL APPROACH

GUTACHTERIN
Prof. Dr. Barbara Paech

To Dad.

JUSTIFIED TEST FOCI DEFINITION

AN EMPIRICAL APPROACH

Since complete testing is not possible, testers have to focus their effort on those parts of the software which they expect to have defects, the *test foci*. Despite the crucial importance of a systematic and justified definition of the test foci, this task is not well established in practice. Usually, testing resources are uniformly distributed among all parts of the software. A risk of this approach is that parts which contain defects are not sufficiently tested, whereas areas that do not contain defects attain too much consideration.

In this thesis, a systematic approach is introduced that allows testers to make justified decisions on the test foci. For this purpose, structural as well as historical characteristics of the software's past releases are analysed visually and statistically in order to find indicators for the software's defects. Structural characteristics refer to the internal structure of the software. This thesis concentrates on the analysis of bad software characteristics, also known as "bad smells". Historical characteristics considered in this thesis are the software's change history and the software's age. Simple and combined analyses of defect variance are introduced in order to determine indicators for defects in software. For this purpose, the defect variance analysis diagram is used to explore the relationship between the software's characteristics and its faultiness visually. Then, statistical procedures are applied in order to determine whether the results obtained visually are statistically significant.

The approach is validated in the context of open source development as well as in an industrial setting. For this purpose, seven open source programs as well as several releases of a commercial program are analysed. Thus, the thesis increases the empirical body of knowledge concerning the empirical validation of indicators for defects in software. The results show that there is a subset of bad smells that are well suited as indicators for defects in software. A good indicator in most of all analysed programs is the "God Class" bad smell. Among the historical characteristics analysed in the industrial context, the number of distinct authors as well as the number of changes performed to a file proved to be useful indicators for defects in software.

Systematische Auswahl des Testfokus
Ein empirischer Ansatz

Da vollständiges Testen nicht möglich ist, müssen Tester ihre Testaktivitäten auf die Bereiche der Software fokussieren, in denen sie Fehler erwarten. Diese Bereiche bilden den *Testfokus*. Obwohl ein systematischer und auf Fakten basierender Ansatz bei der Auswahl des Testfokus von herausragender Bedeutung ist, hat sich diese Vorgehensweise in der Praxis nicht etabliert. Vielmehr werden die Testaufwände gleichmäßig auf die zu testenden Software verteilt. Das Risiko einer solchen Vorgehensweise besteht darin, dass Bereiche der Software, die tatsächlich Fehler enthalten, zu wenig getestet werden, wohingegen Bereiche, die keine Fehler aufweisen, zu viele Testressourcen verbrauchen.

In dieser Doktorarbeit wird ein Ansatz vorgestellt, der eine systematische und empirisch begründete Auswahl des Testfokus ermöglicht. Um Indikatoren für Fehler in der Software zu finden, werden unterschiedliche Merkmale der Vorgängerversionen der zu testenden Software untersucht. Dabei werden strukturelle und historische Merkmale betrachtet. Strukturelle Merkmale beziehen sich auf den internen Aufbau der Software. Einen besonderen Schwerpunkt dieser Arbeit bildet die Analyse von schlechten Struktureigenschaften, den sogenannten „Bad Smells“. Historische Merkmale umfassen die Änderungshistorie sowie das Alter der Software.

Als Bestandteil des empirischen Ansatzes zur Testfokusauswahl werden einfache und kombinierte Analysen der Fehlervarianz eingeführt. Dabei wird zuerst das Fehlervarianz-Analyse-Diagramm verwendet, um die Beziehung zwischen unterschiedlichen Merkmalen der Software und der Fehler visuell darzustellen. Anschließend werden statistische Verfahren angewendet, um die statistische Signifikanz der visuell erzielten Ergebnisse zu ermitteln.

Ein wesentlicher Bestandteil dieser Arbeit stellt die umfassende Validierung des Ansatzes dar. Hierfür wurden empirische Studien zum einen im Bereich der Open Source Softwareentwicklung und zum anderen in einem industriellen Kontext durchgeführt. Somit trägt diese Arbeit zur Anreicherung der Wissensbasis über empirisch validierte Indikatoren für Fehler in Software bei.

Die Ergebnisse der empirischen Studien zeigen, dass eine Teilmenge der untersuchten Bad Smells als Indikatoren für Fehler geeignet ist. Dabei erwies sich das „Gottklasse“ Bad Smell in allen untersuchten Softwareprogrammen als guter Indikator für Fehler. Unter den historischen Merkmalen haben sich die Anzahl der durchgeführten Änderungen sowie die Anzahl unterschiedlicher Autoren, die Änderungen durchgeführt haben, als die besten Indikatoren für Fehler erwiesen.

ACKNOWLEDGEMENTS

First of all I want to express thousand thanks to Professor Barbara Paech. All this work would not have been possible without her guidance and support. Thanks for the trust she put in me and the patience she had even if I had to suspend the completion of this thesis during my maternity leave. Thanks for always finding time for me whenever I needed it.

I am also grateful to all members of the Software Engineering Group at the University of Heidelberg who have contributed in any way to the development of this thesis. I enjoyed the atmosphere, their friendship, and their support. Thousands thanks for the wonderful time I had at the University of Heidelberg!

Thanks to all colleagues for the fruitful discussions and valuable comments on my research ideas. Thanks to Carsten Binnig, Andrea Herrmann, Lars Borner, and Jürgen Rückert. Alike, I want to thank Doris Keidel-Müller for proofreading my English papers and Willi Springer for his technical support.

Finally, and most deeply, I thank my family, Mum, Bastie and David for their endless love and support throughout my research work. Without their love, and belief in me, I would not have been able to accomplish this thesis.

CONTENTS

1 Introduction 17

1.1 Motivation 18

1.2 Background 20

1.2.1 From data to knowledge 20

1.2.2 Generic approach 22

1.3 Thesis goals 24

1.4 Contributions 25

1.5 Thesis outline 28

2 Basic terms and concepts 30

2.1 Introduction 31

2.2 Software testing 31

2.2.1 Definitions 31

2.2.2 Roles in the testing process 32

2.2.3 Test strategy and test focus 33

2.3 Empirical software engineering 33

2.3.1 Immature status of empirical software engineering 33

2.3.2 Reasons for not conducting experiments 34

2.3.3 Empirical strategies 35

2.3.4 Data collection and analysis in qualitative research 36

2.4 Software measurement 37

2.5 Statistical basics 38

2.5.1 Definitions 38

2.5.2 Sample and population 39

2.5.3 Types of data, measurement scales and operations 39

2.5.4 Summarising data 41

2.5.5 Hypothesis testing 43

2.6 Data visualisation 45

2.7 Chapter summary 46

3 Related work 48

3.1 Introduction 49

3.2 Test strategy 49

3.3 Risk based testing 50

3.4 Generic frameworks for software measurement and quality modelling 51

3.5 Models and indicators for the software's fault-proneness 53

3.6 Chapter summary 57

4 Testing process – A decision based view 59

4.1 Introduction 60

4.2 Decision hierarchy 60

4.2.1 Specification level 62

4.2.2 Test goal level 62

4.2.3 Test approach level 62

4.2.4 Test design level 63

4.2.5 Test realisation level 63

4.2.6 Test run level 64

4.2.7 Test evaluation level 64

4.3 Validation 64

4.4 Related work 64

4.5 Chapter summary 65

5 State of the practice of testing processes – A qualitative study 66

5.1 Introduction 67

5.2 Study goal and research questions 67

5.3 Study design 68

5.3.1 Participants 68

5.3.2 Study process 68

5.4 Results 70

5.4.1 Test process characteristics 70

5.4.2 Documentation characteristics 71

5.4.3 Communication characteristics 73

5.4.4 Experience characteristics 73

5.5 Discussion 74

5.6 Implications 75

5.7 Threats to validity 75

5.8 Related work 76

5.9 Chapter summary 77

6 An empirical approach to the justified definition of test foci 78

6.1 Introduction 79

6.2 Justified test foci definition – An empirical approach 80

6.3 Discussion 89

6.3.1 Characteristics of the approach 89

6.3.2 Social aspects to be considered 91

6.3.3 Other defect types 91

6.4 Chapter summary 91

7 Basic experimental design 92

7.1 Introduction 93

7.2 Basic terms and concepts 93

7.3 Goal definition 95

7.4 Granularity 95

7.5 Defining quality and quality indicators 95

7.6 Software entities 96

7.7 Preparation 98

7.7.1 Computing the number of defects per file in OSPs 98

7.7.2 Keyword definition and validation 99

7.7.3 Algorithm performance 100

7.7.4 Defect correction density 103

7.7.5 Threats to validity 104

7.8 Related work 106

7.9 Chapter summary 108

8 Frequency and Pareto distribution of defects 110

8.1 Introduction 111

8.2 Study design 111

8.3 Results **112**
8.3.1 Exploring the Pareto distribution of defects in files 112
8.3.2 Exploring the Pareto distribution of defects in files across releases 114
8.3.3 Exploring the Pareto distribution of defects in code 115
8.3.4 Exploring the Pareto distribution of defects in code across releases 116

8.4 Related Work **116**
8.4.1 Frequency distribution of defects 117
8.4.2 Pareto distribution of defects in files 117
8.4.3 Pareto distribution of defects in code 121

8.5 Chapter summary **121**

9 Bad smells **123**

9.1 Introduction **124**

9.2 Bad smells and refactoring **126**

9.3 Quality indicators and overall hypothesis **126**
9.3.1 Overall research hypothesis 126
9.3.2 Dependent variable 126
9.3.3 Independent variables 126

9.4 Results **130**
9.4.1 Exploring the relationship between method level bad smells and defects 130
9.4.2 Exploring the relationship between class level bad smells and defects 133
9.4.3 Exploring the relationship between Lack-Of Pattern bad smells and defects 135
9.4.4 Exploring the relationship between package level bad smells and defects 137
9.4.5 Which bad smell is the best indicator for defects in code? 138

9.5 Discussion **139**

9.6 Related work **140**

9.7 Chapter summary **141**

10 Exploring the relationship of a file's history and its defect count **143**

10.1 Introduction **144**

10.2 Study design **145**

10.2.1 Organisation context 145
10.2.2 Software entities 145
10.2.3 Granularity 146

10.3 Quality indicators 146
10.3.1 Dependent variables 146
10.3.2 Research hypotheses 147

10.4 Preparation - Computing the number of defects per file 148

10.5 Analysis and results 149
10.5.1 Exploring the relationship between a file's defect count and the number of authors performing changes to it 149
10.5.2 Exploring the relationship between the frequency of change and the defect count 150
10.5.3 Exploring the relationship between co-changed files and defect count 151
10.5.4 Exploring the relationship between a file's age and its defect count 152
10.5.5 Combined analyses of defect variance 153
10.5.6 Analysis 157

10.6 Discussion 158

10.7 Threats to validity 159

10.8 Related work 160

10.9 Chapter summary 161

11 Synopsis 164

11.1 Summary and conclusions 165

11.2 Future research directions 174

A Appendix **176**

A 1 Validation of the decision hierarchy 176

A 2 Pareto distribution of defects in code 180

A 3 Bad smell detection strategies 181

A 4 Mann-Whitney test for the bad smell FE and GM 185

A 5 Mann-Whitney test for class level bad smells 187

A 8 Mann-Whitney test for "Lack-Of" bad smells 193

A 9 Visual analyses for class level bad smells 198

A 10 Visual analyses for class level "Lack-Of" bad smells 201

A 11 Mann-Whitney test for package level bad smells 203

A 12 Visual analyses for the GP bad smell 205

A 13 Statistical tests for DA 206

A 14 Statistical tests for FC 207

A 15 Statistical tests for CF 207

A 16 Statistical tests for age 207

A 17 Non-parametric tests for combined analyses 208

LIST OF FIGURES **210**

LIST OF TABLES **212**

LIST OF ABBREVIATIONS **214**

PUBLICATIONS **215**

BIBLIOGRAPHY **218**

CHAPTER 1 Introduction

This chapter contains an overall introduction into the topic of the thesis, including its motivation, goals and contributions. It introduces the shortcomings of currently existing approaches for defect prediction as well as the problems encountered in practice when deciding on the test foci. The test foci are those parts of the software that have to be tested due to the expected defects. Furthermore, this chapter gives an overview of the contributions of this thesis to solve the problems identified before.

1.1 Motivation

Spectacular software failures like the crash of the Ariane 5 rocket (Dowson 1997), but also software failures which occur in our daily life show that testing activities are essential in order to detect defects before release. As software quality becomes more and more a competitive factor, i.e. the quality acts as a differentiating factor among competitors, it is essential to find as much defects as possible before release. At the same time, the complexity and the size of today's software increase. Since complete testing is impossible (Myers 1979) and testing resources are limited, it becomes more and more essential for testers to decide which parts of the software are to be tested, i.e. the test foci, and which not.

Despite the crucial importance of a thorough and systematic definition of the test foci, this task is not well established in practice (Illes-Seifert and Paech 2008). Usually, the test effort is uniformly distributed among all parts of the software. A thorough risk analysis by which testers estimate parts of the software which they expect to have defects and which need intensive testing is missing. Another problem often encountered in practice is that the estimation of the faulty parts (i.e. the parts of the software that contain defects) is based on testers' experience instead of on reliable facts. Therefore, the quality of the estimation and in general the quality of the decisions made during the testing process highly depend on the experience of the testers. Though experience is very valuable, it is based on subjective perceptions that do not always correspond to the reality. In addition, this experience is usually not documented and therefore not accessible to novice testers.

In literature, we find potential indicators proposed for identifying faulty parts. In (Kaner, Bach, and Pettichord 2002), a list is presented, containing indicators like new technology, late changes, and distributed teams that give hints on faulty parts of the software. But these indicators are not empirically validated and can vary from project to project so that testers can only use them as a starting point for the definition of test foci in their own context. On the other hand, several (more and more sophisticated) approaches for predicting faulty parts have been presented in literature. The approaches basically differ in the models used for prediction. The proposed models include statistical models, tree based models, analogy based models, or neural networks (Lessmann et al. 2008). Model parameters are often structural code characteristics, for instance the number of lines of code or different other code metrics. Other parameters are process characteristics, like the number of changes performed to a software entity or the number of defects detected in previous releases.

Despite the difference in the proposed models, researchers agree to the fact that there is a need to find indicators for defects in code in order to allocate quality assurance effort appropriately. Nevertheless, there is no empirically validated consensus on the superiority of one modelling method over another (Myrtveit and Stensrud 1999), (Myrtveit, Stensrud, and Shepperd 2005), (Shepperd and Kadoda 2001), (Jiang, Cukic, and Ma 2008), (Holschuh et al. 2009). Results of re-

cent research that tries to make cross-project prediction show that simply using results of one project in another is impossible (Zimmermann et al. 2009). Thus, the selection of the best indicators for defects in code along with the best prediction algorithm can only be made in context. There is no "best" global prediction model. A recent debate shows that there is no consensus on how to evaluate different defect prediction models, i.e. how to assess their performance and to make detailed comparisons of several models (Zhang and Zhang 2007), (Menzies et al. 2007), (Jiang, Cukic, and Ma 2008), (Lessmann et al. 2008).

Beside the issue of the diversity of the proposed models and the problem of not knowing which the best is, there are several other reasons which impede that these approaches are used in practice:

- **Little empirical validation.** Few studies in software defect prediction make use of statistical procedures in order to empirically validate the results (Lessmann et al. 2008). In addition, some empirical studies use small data sets. Therefore, more extensive experimentation is needed instead of presenting new models or model enhancements (Menzies et al. 2007).
- **Focus on more and more sophisticated algorithms instead of on their applicability in practice**. Research focused on presenting more and more complex algorithms for defect prediction without considering their computational efficiency, ease of use and comprehensibility. In order to be applied in practice, defect prediction algorithms have to be, above all, easy to use and to understand. Interrelations encrypted in complex formulae hinder that the nature of the detected relationships is understood (Lessmann et al. 2008). In (Mende and Koschke 2009), the need for new indicators for defects in code instead of presenting new algorithms is advocated.
- **Human in the loop needed**. Prediction accuracy will never reach 100%. Predictors can only be used as indicators and not as "definitive oracles" (Menzies, Lutz, and Mikulski 2003), (Menzies et al. 2007), (Menzies et al. 2008). Consequently, the experience of testers has to be considered. Present approaches neglect this issue. In (Menzies, Lutz, and Mikulski 2003), the authors show that human expertise usually outperforms automatic machine learning algorithms. Nevertheless, in some cases a combination of both human and machine learning is advocated, for instance when the data set is too large or too complex or when expert testers are not available (Menzies, Lutz, and Mikulski 2003).
- **Lack of the awareness for the importance of empirical work.** Practitioners are not trained in the importance of validating their results empirically. For this reason, empirical software engineering methods are not often applied in practice (Juristo and Moreno 2001).

All these issues hinder that the approaches for defect prediction are used by testers. Nevertheless, in practice, large amounts of data are collected but not used in order to gain insight into processes and in order to justify decisions. This is the case for several reasons:

- Often, a tool is used that records large amounts of data but it is not clear which parts of the data are useful and for which purposes.
- The multitude of metrics makes it difficult to aggregate them to a set of a few but meaningful key indicators which would allow an easy interpretation and as a result assess implications and to drive conclusions based on the data.
- Often, the collection of data is not driven by a clearly defined goal that should be achieved when analysing it. In fact, that data are collected which are available or which can be easily recorded by a tool. But that data are frequently not useful or best suited in the specific context.
- In addition, often only a snapshot of the current state is drawn up. But the raw value is mostly not meaningful, for instance it is not evident whether a cyclomatic complexity of 10 is good or bad for a specific program. In fact, monitoring the trend of a metric over time is more purposeful. For example, if the complexity of a piece of code increases abruptly, this could be a hint that substantial changes have been performed and that the particular piece of code is a candidate for code inspections. Therefore, the analysis of past characteristics of the software development eases the assessment of the current status (in reference to the past) and can give hints on how the software will develop in the future.

Consequently, there is a need for an approach that allows testers to make justified decisions based on concrete facts rather than on intuition, i.e. testers should be able to justify decisions concerning the test foci based on the data they usually collect anyway. In addition, there is a high need for extensive empirical studies and for better indicators of software defects. These two issues are addressed in this thesis.

1.2 Background

Mostly, large amounts of data, for instance contained in defect tracking systems (DTS) or test management tools, are available for testers. But the data are useless unless there are transformed into information and knowledge. In this section, the terms "*data*", "*information*", and "*knowledge*" will be introduced. Furthermore, the generic approach to find indicators for defects in software used in literature as well as in this thesis will be presented.

1.2.1 From data to knowledge

The term *"data"* denotes a set of discrete, objective facts or symbols. In an organisational context, it can be seen as a set of „structured records of transaction" (Davenport and Prusak 1998). A defect inserted by a tester into a DTS is a transaction representing data. For instance, this data tells nothing about why the defect occurred or how likely it is that this defect will occur again. Usually, large amounts of data are generated during the lifecycle of software. But without analysing them, they have no value.

Information is data processed to be useful in a specific context. In (Davenport and Prusak 1998), the analogy of information with a message is used, both having a sender and a receiver. Similarly to a message, information has an impact on the receiver's behaviour or judgements when reading it. Consequently, information can be determined to be real information and *not* data only from the receiver's point of view. The sentence "For component X, 10 defects of priority 1 have been detected during the last two months." can be data for someone who does not know what component X is. For a tester, knowing that priority 1 defects are critical defects in the application, the statement represents information. Generally, everything that has not been collected with a purpose in mind and which cannot be interpreted represents data and not information.

Knowledge can be seen as information to which experience, interpretation, and reflection are added. Knowledge can be used in new contexts and situations, for instance when making decisions. For instance, knowing that in a project the number of changes performed to a software entity is a good indicator for its faultiness is useful knowledge when deciding on the testing effort to be allocated to test the software: Components or parts of the software that have been changed frequently would be tested more thoroughly than components that have not been changed at all.

Data can be transformed into information by putting it into context, by building categories, by aggregating or eliminating errors from data, hence, by *understanding relationships* between data. Information, on its part, can be transformed into knowledge by comparing information, deriving consequences on the basis of information, by connecting, communicating, and discussing information, thus by *understanding patterns* in information and data. From data to knowledge, the original facts and symbols become more and more connected, and simultaneously, the understanding increases. Understanding is the process of synthesizing new knowledge from previously stored information and knowledge (Bellinger, Castro, and Mills 2010). The relationships between data, information and knowledge are demonstrated in Figure 1.1.

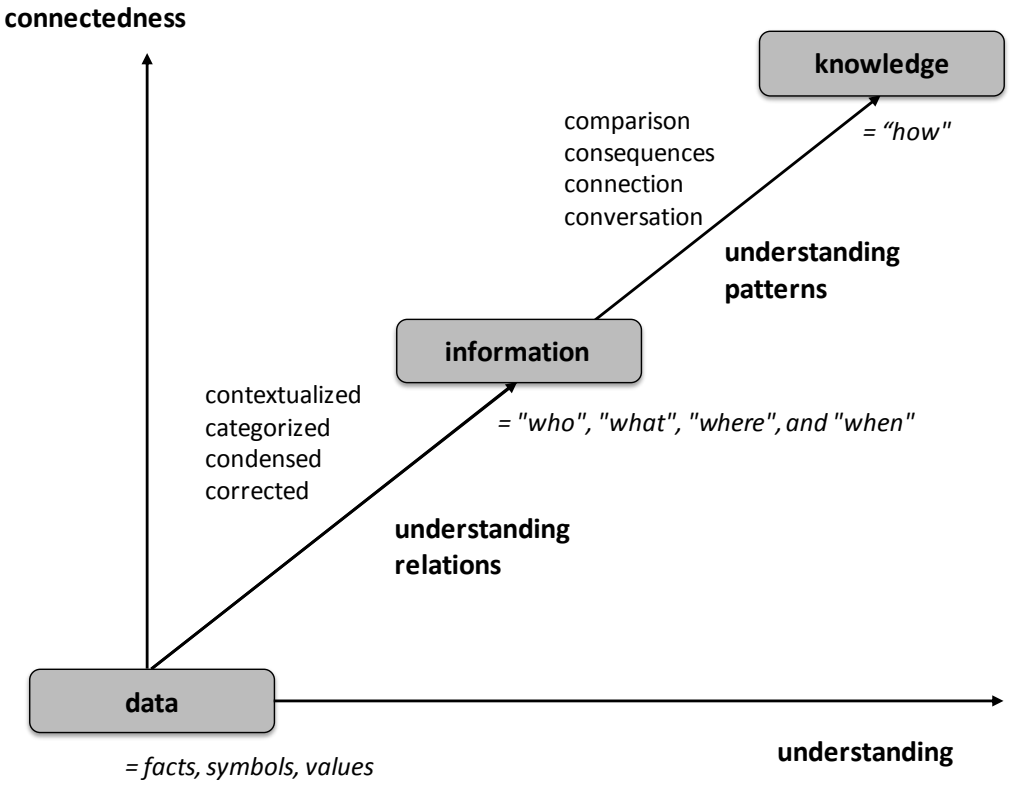


Figure 1.1 - Data, Information, Knowledge
Adapted from (Davenport and Prusak 1998) and
(Bellinger, Castro, and Mills 2010)

1.2.2 Generic approach

A generic goal of all approaches that aim to find indicators for defects in software is to show how the large amount of data available for testers can be selected and analysed in order to obtain knowledge essential to define the test foci.

Which data are available for testers? Basically, a distinction between *quantitative* and *qualitative* data can be made. Qualitative data refers to non-numerical whereas quantitative data refers to numerical data. The number of defects in a file or the number of changes performed to a file represents quantitative data. The statement of a tester, that there is a high defect count in a file is qualitative, because it is not clear what the status "high" stands for. Defining that a "high" defect count is attributed to all files containing more than 5 defects transforms the qualitative statement into a quantitative one.

Similarly to the empirical studies presented for instance in (Fischer, Pinzger, and Gall 2003), (Schröter et al. 2006), (Čubranić and Murphy 2003), (Sliwersky, Zimmermann, and Zeller 2005), (Zimmermann, Nagappan, and Zeller 2008), (Weyuker and Ostrand 2008), in this thesis, data contained in DTSs and in versioning control systems (VCS) as well as the application code itself are considered. Data contained in DTSs and VCSs are collected in order to get information about the software project's history. In this thesis, this information is denoted as project history information as it describes the software's evolution. Information

about the internal structure of the software is obtained by static code analysis. Both project history as well as product information represent quantitative information.

The amount of data which can be collected is nearly unlimited, but only a small sub-set is useful and purposeful. In addition, data collection and analysis is expensive and its interpretation time consuming, thus testers' experience is important and can be used when deciding which data have to be collected. For example, if testers subjectively have the impression that the defect count increases with the number of authors responsible for a software entity, the information about the number of authors that performed changes to a software entity should be collected in order to analyse whether it is actually a good indicator for a file's defect count. Testers' experience can be used during data collection and data analysis. During data collection, experience is important in order to minimise the amount of data to be collected. During data analysis, testers can decide which analyses should be refined based on the results of previous analyses. The use of testers' experience is neglected in research so far when searching for indicators for defects in software. Testers' experience represents "tacit" or *implicit* knowledge that is usually "in the heads" of the testers.

In contrast to implicit knowledge, *explicit* knowledge is based on the analysis of documented data, for example data recorded in VCSs or in DTSs.

The empirical evidence obtained by analysing the project's history and the software's structure (as well as a combination of both) reflects explicit, empirically validated knowledge about relationships between software characteristics and its defects. This knowledge can be used to define the test foci.

Figure 1.2 summarises the generic approach as presented in this section.

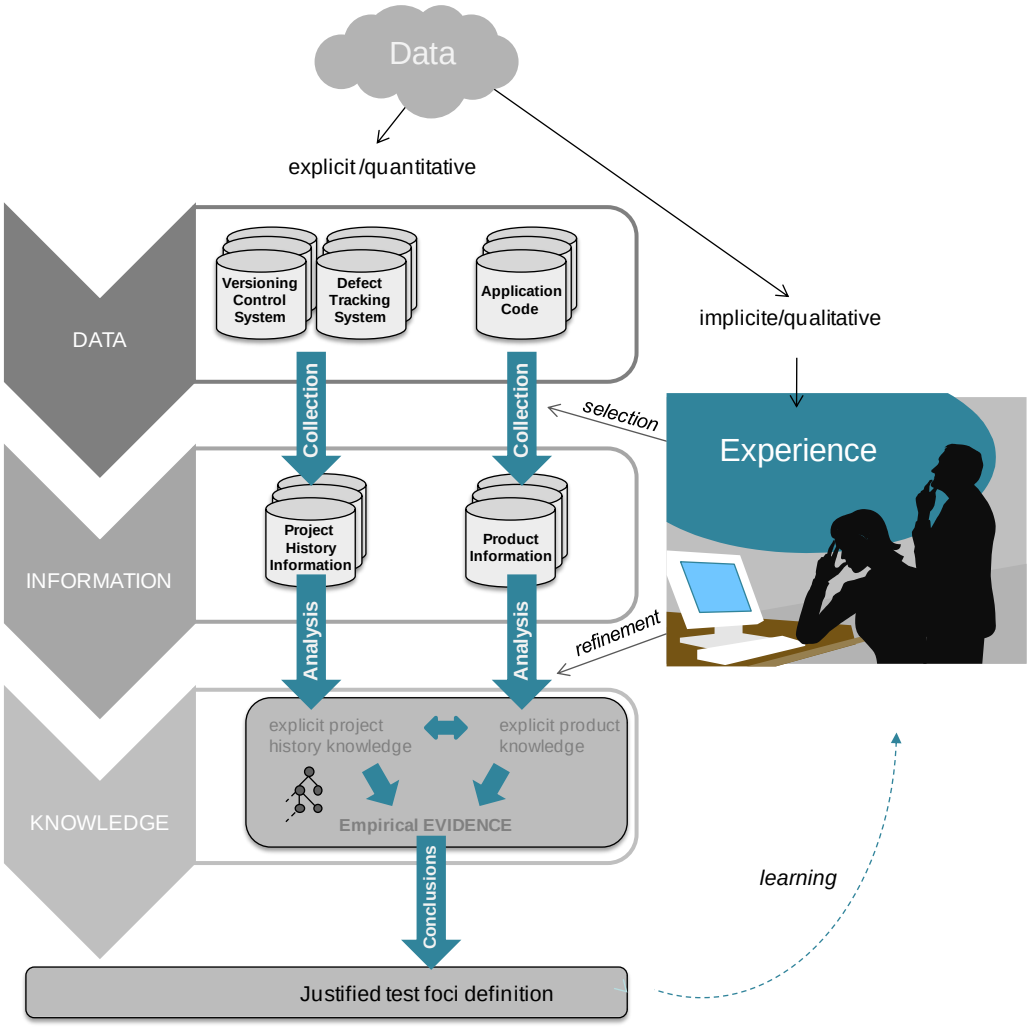


Figure 1.2 - Generic approach

1.3 Thesis goals

This thesis aims to achieve the following two main goals:

GOAL 1: Definition of a lightweight approach that allows testers to find context specific indicators for defects in software and therefore, to identify the test foci by exploring different information sources visually and empirically. Thus, the approach aims to support testers in deriving empirically validated knowledge about indicators for defects in software.

GOAL 2: Increase the empirical body of knowledge, particularly by analysing the relationship between structural and historical characteristics of the software and its defects empirically.

In order to achieve the first goal, the following issues have to be addressed:

- To be able to draw reliable conclusions, statistical procedures have to be integrated into the approach, in order to analyse or to show the statistical significance of the results obtained by analysing the data. Since testers are usually not familiar with statistics and they do not have the time to consult numerous statistic text books, a trade-off has to be made between an

approach that is easy to understand and to apply on the one hand and an approach that uses statistical procedures on the other hand.

- A simple visual representation which abstracts from statistical formulae and procedures can help interpreting and exploring the results quickly.
- Since experience plays an important role for testers when making decisions (Illes-Seifert and Paech 2008), the approach should consider and benefit from this experience. In addition, the approach should give guidance on how to find indicators for software defects *in context*.
- A stepwise refinement when analysing data ensures that first results are obtained quickly. In the initial stage of the analysis, raw tendencies should be derived which can be refined in further steps if necessary.

The second goal of this thesis is to perform extensive experimentation in order to enrich the empirical body of knowledge in the area of indicators for defects in software. For this purpose, the approach has to be evaluated in several contexts and across several releases of open source as well as of industrial programs. In addition, a sub-goal of this thesis is to evaluate new indicators for defects in software empirically.

The following issues are out of the scope of this thesis. First, characteristics of the software's history which are not documented in a VCS are not considered in this thesis. For instance, the number of changes to a requirement is not analysed, because this information is usually not quickly (and automatically) available for testers. In addition, structural characteristics of documents that are not part of the code itself, for instance the complexity of a requirement, are also not considered. The main reason is that in most of the cases, documentation that is not the software's code is informal and therefore, the automatic computation of its structural characteristics is difficult.

This thesis primarily aims to help testers to make justified decisions based on data but not to propose a particular model for defect prediction. In a further step, after indicators for defects in software have been identified, models for defect prediction can be built. Nevertheless, the selection of the most appropriate defect prediction model is not addressed in this thesis.

Beside the number of defects, the quality of the software comprises other characteristics like maintainability or usability. Finding indicators for other quality characteristics is not addressed in this thesis.

1.4 Contributions

In order to define an approach that is suitable to be applied in practice as stated in Goal 1, the testers' needs are analysed in a qualitative study (Contribution 1). The analysis is performed using a decision based framework that structures decisions to be made during the testing process (Contribution 2). The empirical approach for the justified definition of the test foci (Contribution 3) directly supports Goal 1. In order to increase the empirical body of knowledge as stated

in Goal 2, several empirical studies are performed in the context of open source development (Contribution 4) and in an industrial setting (Contribution 5).

In the following, a detailed description of the contributions of this thesis is given.

- **Contribution 1 – Qualitative analysis of the testing process:** In order to understand the needs of software testers, an empirical analysis of the state of the practice is performed that shows strengths and weaknesses of testing processes in practice. For this purpose, experienced testers in several organisations are interviewed in order to determine the most valuable information sources for testers when making decisions during the testing process, i.e. which documents are often used as well as the role of communication and experience. The study shows that testing requires above all domain specific experience. In addition, previously found defects are an important information source for testers. Finally, testers have problems in evaluating the outcome of the testing processes. The main reason for this is that testers do not have approaches that allow making sound and justified decisions concerning the test foci. Without having defined what to test, it is very difficult to evaluate whether the test goals have been achieved. The results of this analysis are used to define a lightweight approach to determine the test foci suitable to be applied in practice.
- **Contribution 2 – Decision based framework for the characterisation of test processes**: In order to analyse the testing processes in practice, a decision based framework is proposed. Software processes often focus on artefacts, activities, and roles, treating decisions to be made during the software development process only implicitly. However, the awareness of these decisions increases their quality by forcing the decision-makers to search for alternatives and to trade off between them. The decision based framework represents a different point of view of the testing process and comprises all decisions made during testing and reflects dependencies between them.
- **Contribution 3 – Empirical approach for justified definition of the test foci**. Decisions in practice are often made based on intuition and subjective appraisal, rather than on facts. This also applies to software testing, particularly to the definition of the test foci. In this thesis, an approach is presented which combines visual analyses and statistical procedures in order to determine those entities that are responsible for defects in software. The approach helps testers to make justified decisions that rest upon statistically validated facts when allocating their limited resources among particular parts of the software. In contrast to sophisticated algorithms presented in literature, this approach is easy to use and to understand and thus, it is applicable in practice. In addition, the approach does not assume a global set of indicators for defects. In fact, testers have to define the most appropriate indicators in their context. Finally, the approach uses testers' intuition in order to select, analyse and to interpret the results.
- **Contribution 4 – Extensive experimentation in the context of open source development.** The need for extensive empirical investigation has been for-

mulated by several researchers, for example in (Juristo, Moreno, and Vegas 2004), (Myrtveit, Stensrud, and Shepperd 2005), (Menzies et al. 2007), and (Lessmann et al. 2008). The second goal formulated for this thesis is to enlarge the empirical body of knowledge with extensive experimentation. For this purpose, the relationship between several project history characteristics as well as structural characteristics and defects in software is analysed empirically both in the context of open source development (Contribution 4) and in an industrial context (Contribution 5).

The main goal of the analysis performed in open source context is to explore the relationship between structural characteristics of software and its defects. For this purpose, seven open source programs across several releases are considered. Particularly, the following empirical analyses are performed in this thesis:

PARETO-Analysis: The Pareto Principle is a universal principle of the "vital few and trivial many". According to this principle, 80% of the consequences originate from 20% of the causes. In this thesis, the Pareto Principle is applied to software testing in order to analyse whether a small part of the software's code is responsible for most of the defects. The Pareto principle is also known as the 80/20 rule. The results show that defects concentrate on a small part of files but they do not concentrate on a small part of the application's code.

BAD SMELLS-Analysis: Bad smells have been introduced as patterns for frequently occurring problems in code (Fowler et al. 1999), i.e. the code might be difficult to understand or might cause high maintenance effort. Thus, bad smells are commonly used as indicators for those parts of the software which should be refactored. Little attention has been paid to analyse the relationship between bad smells and defects in software empirically. Beside the study presented in (Shatnawi and Li 2006), this issue has not been addressed in research so far. Thus, this thesis contributes to analyse new indicators for defects in software. The results show that in fact there are bad smells that actually are good indicators for the software's defects.

- **Contribution 5 – Experimentation in an industrial context.** One main goal of the analysis in the industrial context is to validate the empirical approach. In addition, this analysis aims to explore the relationship between several project history characteristics and defects empirically, the HISTORY-Analysis.

  HISTORY-Analysis: The main assumption of this empirical investigation is that the project's history is a valuable information source when searching for indicators for defects in software. For instance, according to an expression, "many cooks spoil the broth". Is this true for software development, too? This is one of the questions to be analysed in this context. Other characteristics that are analysed include the number of authors performing changes to a software entity, the size of the change as well as the age of a software entity. The results show that the number of changes and the number of distinct authors performing changes to software entities are good indicators for its

defects in the analysed context. The combination of these indicators shows a more precise view. Frequently changed entities by many distinct authors have the most defects.

The results of this empirical study also show that the approach proved of value; the presentation of the results was intuitive and easy to understand by the project team. From the testers' point of view, the empirical study was helpful because the results confirmed their assumptions in large parts and build now the foundation for justified decisions on the test foci.

1.5 Thesis outline

The **thesis is structure**d as follows:

Chapter 2 introduces some basic notions and general concepts used throughout the thesis. It includes basic terms in the area of software testing as well as an introduction to empirical software engineering. In addition, it presents an introduction to basic terms and concepts related to software measurement and statistics.

Chapter 3 aims at giving an overview of related work.

Chapter 4 introduces a decision based framework for characterising testing processes that is used as the basis for the evaluation of testing processes in practice as described in Chapter 5. In this chapter, Contribution 2 of the thesis is detailed.

Chapter 5 presents the results of an interview study conducted with expert testers in order to identify the state of the practice with respect to the most valuable sources of information for testers. Particularly, it identifies which documents are often used by testers as well as the role of experience when making testing decisions. In this chapter, Contribution 1 of the thesis is detailed.

Chapter 6 gives a detailed description of the empirical approach presented in this thesis that helps testers to make justified decisions on the test foci. In this chapter, Contribution 3 of the thesis is detailed.

Chapter 7 introduces the context of the empirical studies, including information on the programs that are analysed and the procedures used for data collection and validation.

Chapter 8 details the results of the analysis of the Pareto principle. In this chapter, Contribution 4 related to the PARETO-Analysis is detailed.

Chapter 9 explores the relationship between *bad* software characteristics (bad smells) and defects in software. In this chapter, Contribution 4 related to the BAD SMELLS-Analysis is detailed.

Chapter 10 presents the results of an empirical study which aims to explore the relationship between a file's history and its defect count. In this chapter, Contribution 5 related to the HISTORY-Analysis is detailed. In addition, this chapter also aims to validate the approach presented in Chapter 6 in an industrial context.

Chapter 11 summarises the results and limitations of this thesis and gives some directions for future research.

CHAPTER 2 Basic terms and concepts

This chapter introduces basic terms and definitions related to software testing, empirical software engineering, software measurement and statistics used in this thesis.

2.1 Introduction

In this chapter, fundamental concepts used in this thesis are introduced. First, basic terms and concepts related to software testing are presented in Section 2.2. At this, a definition of the term "testing" is given along with the discussion of the differences between the similar, but not synonymous terms "defect", "error", and "failure". Section 2.3 motivates the necessity for empirical research in software engineering. In addition, the immature status of experimentation is discussed. Subsequently, basic concepts in empirical software engineering are presented. In the third part of this chapter, in Section 2.4, basic terms related to software measurement are introduced. Basic statistical terms and concepts are presented in Section 2.5 along with frequently used visualisations for data (Section 2.6). Finally, Section 2.7 summarises this chapter.

2.2 Software testing

In the narrow sense, software testing comprises the random execution of software to observe whether it behaves as expected. If this is not the case, correction activities have to be taken. But beyond the simple random execution of test cases, testing involves several other activities performed by several roles in the testing process. In this thesis, the following definition of software testing is used:

Definition 2.1 – Testing

> *Testing* is the process consisting of all life cycle activities concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects (ISTQB 2007).

In Chapter 4, a framework for characterising testing processes is introduced along with a detailed description of the testing process from a decisions based perspective.

2.2.1 Definitions

Often, the term "defect" is used synonymously to the terms "error" and "failure" but there are slight differences.

Definition 2.2 – Failure

> A *failure* is an *observable* deviation of a component or a subsystem from the required or expected function.

Definition 2.3 – Defect

A *defect* is a flaw in a component or system that can cause the component or system to fail to perform its required function (ISTQB 2007). For example, a defect can be an incorrect or omitted statement. Often used synonyms are *fault* or *bug*. A defect can, but must not cause a failure. The terms defect, bug, and fault are used synonymously in this thesis.

Definition 2.4 – Anomaly

Anomalies subsume any kind of deviations of a system or component from expectations based on requirements specifications, design documents, user documents, standards, or from someone's perception or experience (ISTQB 2007).

Definition 2.5 – Error

An *error* is a human action that produces an incorrect result (ISTQB 2007), for instance a defect in the code.

For a defect, three conditions have to be fulfilled in order to expose a failure (Binder 1999):

1) The defect must be *reached*, i.e. the code fragment containing the defect must be executed.
2) The failure must be *triggered*, i.e. the system state and the input data must cause the code fragment containing the defect to produce an incorrect result. Depending on the system state and the input data, a code fragment containing a defect must not expose a failure when executed.
3) The failure must be *propagated*, i.e. the failure must be observable.

Usually, testing activities expose failures that are recorded in defect tracking systems. Developers analyse these records and search for the defect, i.e. the cause of the failure.

Parts of the software with a poor quality are **faulty** if they have defects.

The file *a* is more **fault-prone** than the file *b* if the defect count of the file *a* is higher than the defect count of the file *b*.

2.2.2 Roles in the testing process

Depending on the complexity and size of the software under test, several roles are involved in the testing process. For instance, test managers are involved in test planning and test monitoring activities, test designers are responsible for designing a set of test cases and testers usually execute them. In this thesis, the term "tester" is used to refer to all persons involved in the testing process.

2.2.3 Test strategy and test focus

The test strategy comprises the definition of the high-level approach to testing. It defines *what* should be tested and *how*. In order to decide on *what* to test, testers have to select the **test foci** (i.e. the parts of the software to be tested) and to decide with which **intensity** to test the test foci. In this thesis, an empirical approach to determine the test foci is presented that is based on visual and statistical analyses of project history and product data.

In order to decide *how* to test, testers have to define for example which **testing techniques** have to be used to test the test foci or which is the **ideal order to perform the tests** (e.g. depending on the availability of the components of the software under test). A detailed description of all decisions to be made during test strategy definition is presented in Chapter 4.

2.3 Empirical software engineering

Empirical software engineering research basically consists of tests that "compare what we believe to what we observe" (Perry, Porter, and Votta 2006), or with other words it refers to "matching with facts the suppositions, assumptions, speculations and beliefs that abound in software construction" (Juristo and Moreno 2001).

Empirical software engineering research aims to explore, describe, predict, and explain phenomena in the area of software engineering by using evidence based on observation or experience. It involves obtaining and interpreting evidence for the usefulness of different methods, techniques, tools, and processes, for instance by experimentation, interviews, and surveys, or by the careful analysis of documents or artefacts (Sjoberg, Dyba, and Jorgensen 2007).

Why is empirical software engineering important? First, similarly to "traditional sciences", computer scientists have to observe phenomena, formulate explanations and theories, and test them in order to *understand* the nature of information processes (Tichy 1998) and to understand what makes software good and how to make software well (Fenton and Pfleeger 1998).

Engineers need a proof that a particular approach is really better than another (Juristo and Moreno 2001). Experimentation can help to build a reliable base of knowledge and helps *reducing the risks and the uncertainty* about proposed methods, techniques, tools, and processes. Therefore, experimentation helps determining the effectiveness of proposed approaches (Zelkowitz and Wallace 1998). In addition, experimentation can *accelerate progress* in software engineering research and practice by eliminating inadequate approaches.

2.3.1 Immature status of empirical software engineering

In 1993, Rubin stated: "Little is known [of] the impact of software engineering practices and processes. While much is written about the topic in qualitative terms, little quantitative information is available."

In spite of its importance for practice and research, the status of empirical software engineering is considered as immature. Methods, techniques, tools, and

processes are judged by whether or not people use them (Juristo and Moreno 2001). There is little empirical evidence that a particular practice, tool, or process is better than another.

In (Zelkowitz and Wallace 1998), the authors analyse about 600 papers published from 1985 through 1995 in IEEE Transactions on Software Engineering[1], IEEE Software[2], and the ICSE[3] proceedings according to the amount of empirical validation. They conclude that:

- Many papers (1/3) have no experimental validation at all.
- The most validation of the proposed methods and tools is done by "lessons learned" and case studies (each 10% of all papers).
- Authors often fail to clearly state the value added by the new method or tool they have developed.

The authors in (Lukowicz et al. 1994) performed a similar survey over 400 research articles. They also conclude that the ratio of validated results is a "serious weakness" in computer science research. According to this study, 40% - 50% of articles completely lack of such validation. Related to other disciplines (the authors compare their results with optical engineering), computer scientists validate a smaller percentage of their results.

Two more recent studies (Sjoberg, Dyba, and Jorgensen 2007) and (Wainer et al. 2009) underline the findings obtained by (Zelkowitz and Wallace 1998) and (Lukowicz et al. 1994). They conclude that computer science research has not increased significantly its empirical or experimental component yet.

2.3.2 Reasons for not conducting experiments

Some of the popular fallacies (and rebuttals) when arguing not to perform empirical validation from the researcher's point of view are formulated in (Tichy 1998). Researchers often argue that *the traditional scientific method is not applicable.* But in order to understand the nature of information processes, computer scientists must observe phenomena, formulate explanations and theories, and test them. Another fallacy is that the *current level of experimentation is good enough*. But the results mentioned above underline the "pre-scientific status in software engineering" (Juristo, Moreno, and Vegas 2003). In addition, researchers argue that *experimentation will slow progress.* But the opposite is true. Mature sciences are characterised by using mature empirical knowledge in order to predict results.

From the perspective of practitioners there are also several problems that hinder the application of empirical software engineering methods in practice (Juristo and Moreno 2001). Beside the issue related to the short-term costs, practitioners are not trained in the importance and meaning of empirical studies. They do not understand the importance of empirical work in validating meth-

[1] http://www.computer.org/portal/web/tse/, (June 2011)
[2] http://www.computer.org/portal/web/software/home, (June 2011)
[3] International Conference on Software Engineering, http://www.icse-conferences.org/, (June 2011)

ods, techniques, tools, and processes. In addition, they are not trained in statistical procedures needed to understand the results of statistical analyses.

In research and practice, the short-term **costs and time pressure** prevent seeing the long term benefits of experiments. For the short term, an empirical investigation seems very costly. But selecting and applying the "wrong" method, technique, tool, or process can exceed the costs of an initial empirical validation many times over.

Finally, there are several obstacles to perform thorough empirical validation resulting from the discipline itself, for instance the effect of the human factor or the variety of contexts (Juristo and Moreno 2001). In software engineering, the results depend on the practitioners. Different practitioners will get different results when applying an approach or tool. In addition, there are large differences in contexts and thus, it is difficult to generalise results. Nevertheless, complexity should not lead to neglect empirical work in the field of software engineering, because this is the building block of a mature science.

2.3.3 Empirical strategies

Basically, two main strategies can be distinguished for performing empirical research: qualitative and quantitative strategies.

Quantitative strategies perform (statistical) analyses on numerical data. Quantitative research is an "inquiry into an identified problem based on testing a theory, measured with numbers, and analysed using statistical techniques" (State Justice Institute 1999). Basic statistical definitions and techniques that can be applied in quantitative research are presented in Section 2.5.

Qualitative methods use data in form of text, images, sound drawn from observations, interviews and documentary evidence, and analyse it using methods that do not rely on precise measurement to yield their conclusions. Qualitative methods have originally been introduced by educational researchers and by social scientists in order to study human behaviour like motivation and communication (Seaman 1999). There are also different kinds of methods that can be used to collect and analyse data gathered by qualitative studies.

Usually, software engineering research may incorporate both qualitative and quantitative methods (Seaman 1999). The selection of an appropriate method depends on factors like the problem of interest, resources available, the skills and training of the researcher(s), and the audience for the research. The concept of subjectivity and objectivity is not correlated to either of these types of investigation (Juristo and Moreno 2001). Table 2.1 summarises the differences between qualitative and quantitative research with respect to the inputs, goals, general characteristics, the role of the researcher, and the methods used in the particular research strategy.

In this thesis, both qualitative and quantitative research strategies are applied. The state of the practice concerning the testing processes, as described in Chapter 5, is analysed qualitatively. This research strategy is used, because it helps to get more experienced with the analysed phenomenon. In this case, the overall

goal is to get a deep understanding of the testing process along with the information flow. The empirical studies presented in this thesis that aim to explore the relationship between historical as well as product characteristics of software and its defects are quantitative studies, since they are based on numerical data that are analysed with statistical (and visual) procedures.

	Qualitative Research	**Quantitative Research**
Inputs	text, images, sound drawn from observations, interviews	numerical data
Goal	understand phenomena and different perspectives	find facts, patterns, generalisability, prediction
Characteristics	explanatory discovery oriented	confirmatory verification oriented
The role of researcher	The researcher interacts with those he studies.	The researcher remains distant of what is being researched.
Methods	case studies, interviews, participant observation, document reviews, etc.	statistical analyses, experiments, surveys, etc.

Table 2.1 - Empirical software engineering research
Adopted from (State Justice Institute 1999)

2.3.4 Data collection and analysis in qualitative research

In literature, several methods for collecting and analysing qualitative data have been proposed. Popular methods used for and during data collection are interviews, document analyses, and coding. *Interviews* are particularly useful for getting the story behind a participant's experiences (Kvale 1996). The interviewer can pursue in-depth information around the topic of interest. There are several kinds of *interviews*, for instance face-to-face, telephone, email, or focus groups. *Document analysis* is concerned with the analysis of textual artefacts (e.g. review protocols, project plans) or of visual documents (e.g. photographs, videotapes, art objects, film). The *coding* process aims at assigning "tags" to qualitative data. Coding facilitates the identification of trends and patters and it also can be used to extract quantitative data from qualitative data. Coding should be used throughout the process of data collection. In (Seaman 1999), it is emphasised that coding adds neither objectivity nor accuracy to data.

For data analysis, several methods have been proposed. *Cross case analysis* partitions the data into different categories by using different criteria (Seaman 1999). The main idea is to "look at the data in many different ways" (Seaman 1999). In research, several strategies for partitioning data have been presented, for instance based on particular attributes like the number of people involved, the type of product that has been analysed, etc. Another possibility to categorise data is according to the data source (e.g. interviews, document analysis, etc.) or to compare pairs of cases. The main goal of all data analysis procedures is to

find similarities and differences between the groups identified before (Eisenhardt 1989).

Ensuring validity of the methods used to generate hypotheses and conclusions is one of the most important ways to confirm a qualitative hypothesis or conclusion. One way for assuring validity is the *representativeness* of the experimental subjects/objects. Another way of increasing confidence in conclusions and hypotheses drawn from qualitative data is *triangulation*. Triangulation aims at gathering different types of evidence to support a proposition (Seaman 1999). In Chapter 5, methodological triangulation (uses multiple methods[4]) as well as explanatory triangulation (tries out several explanations for all results) are used in order to assure the validity of the results.

2.4 Software measurement

In this section, basic terms related to software measurement are introduced.

Definition 2.6 – Measurement

Measurement is the process by which values are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules (Morasca 2001), (Fenton and Pfleeger 1998). In case of software measurement, entities are related to software processes or to software products.

Definition 2.7 – Entity

An *entity* is an object (e.g. a piece of software) or an event (e.g. testing phase in the software development process) in the real world.

Definition 2.8 – Attribute

An *attribute* is a property of an entity. For instance, an attribute of a software component is its size; an attribute of the testing process is its duration.

Definition 2.9 – Metric

A *metric* represents a quantitative measure of the degree to which a system, a component, or process possesses a given attribute (IEEE Std 1990). For instance, the size of the software can be represented by the LOC metric, the duration of the testing phase by the time interval between the beginning and the end of the testing activities.

[4] Triangulation can also include the combination of qualitative and quantitative methods (Seaman 1999). One example of combining qualitative and quantitative methods is to statistically validate a hypothesis that has been generated qualitatively.

In (Morasca 2001), a difference between the concept of metric and measure is made. Accordingly, a "measure" is a more general term than "metric".

Figure 2.1 illustrates the relationship between measures, metrics, attributes and entities. A measure can be quantified by several metrics. Metrics assign a value to the attributes of entities in the real world. For instance, the quality of a piece of software can be expressed by its fault-proneness (measure). One corresponding metric to measure the fault-proneness of a file is for instance the number of defects reported for that piece of software after its release. An alternative metric for the fault-proneness of a file is the number of defects reported during system testing. Thus, several metrics can be defined to quantify a measure.

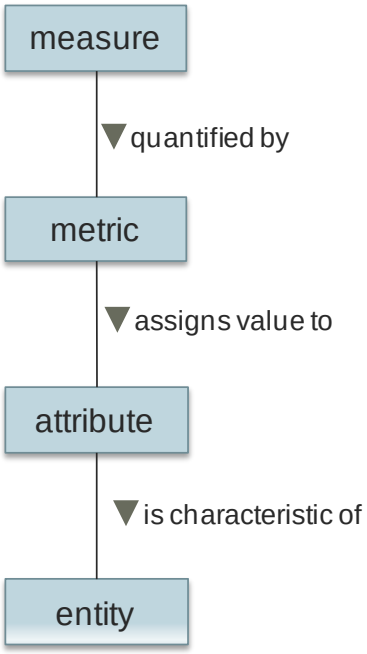


Figure 2.1 - Measures, metrics, attributes and entities

2.5 Statistical basics

This section introduces some basic statistical terms and concepts.

2.5.1 Definitions

Definition 2.10 – Statistic

A *statistic* is a numerical summary of the sample data. Commonly used statistics for quantitative data are for instance means, variances, standard deviations, percentiles, or medians. For qualitative data, proportions or percentages can be used to summarise characteristics of the data (Salkind 2007).

Definition 2.11 – Experimental object

In statistics, the entities on which the empirical study is run are called experimental units or experimental objects. The files of an open source program are experimental objects.

Definition 2.12 – Variable

Each experimental object is measured according to various attributes. Each of these attributes is denoted as a *variable*. For instance, the size or the age of a file is a variable of the experimental object "file". *Independent variables* are those variables that researchers can control and change in the experiment. Independent variables are also called *factors*. D*ependent variables* are those variables that are affected during experimentation. The dependent variable depends on the independent variable.

Definition 2.13 – Observation

The data derived or measured from an object is called *observation*. The concrete value of the variable of an experimental object is an observation. The size of a particular file (e.g. measured by its lines of code) represents an observation.

Definition 2.14 – Treatment

A concrete value of a factor is denoted as treatment.

2.5.2 Sample and population

Definition 2.15 – Population

The population subsumes all data that could be gathered given infinite time.

Definition 2.16 – Sample

A sample is a subset of the population. Since resources and time is limited, experiments use a subset of a larger population and aim to generalise the results obtained to that larger population. If the sample size is large enough, the confidence increases that the results obtained for the sample represent the characteristics of the population (Fenton and Pfleeger 1998).

2.5.3 Types of data, measurement scales and operations

The nature of the data influences the analysis techniques and the operations that can be performed on the data. One basic distinction is whether arithmetic operations can be performed on the data (numerical data) or not (categorical data).

Different types of data have different underlying *scales of measurement*. In particular, following scales can be distinguished (in ascending order of precision and power): nominal scale (for nominal data), ordinal scale (for ordinal data), interval scale (for discrete data), and ratio scale (for continuous data).

- The **nominal scale** is the least powerful of the scale types. The nominal scale represents a list of classes to which objects can be classified, for instance the classification of software defect types into one of the categories: data flow, control flow, etc. Variables on the nominal scale are called nominal variables.
- The **ordinal scale** assigns values to objects based on their ranking with respect to one another. For instance, the ordering of software requirements according to their priority: low, middle, high is an ordinal scale.
- The **interval scale** captures information about the size of the intervals that separate classes. On interval scale, the difference is meaningful, but not the value itself.
- The **ratio scale** preserves ordering, the size of intervals between entities, and ratios between entities. In addition, there is a zero element, representing total lack of the attribute.

Figure 2.2 shows the different types of data along with the corresponding scales of measurement.

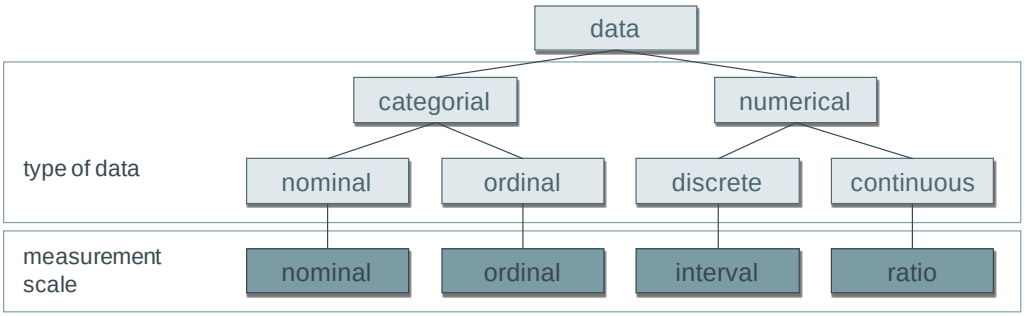


Figure 2.2 - Types of data and measurement scales

There is a hierarchy implied between the several measurement scales. At each level up the hierarchy, the current scale includes all of the qualities of the one below it and adds something new. Figure 2.3 shows the different measurement scales and corresponding operations as well as statistical analyses that are permitted at each level.

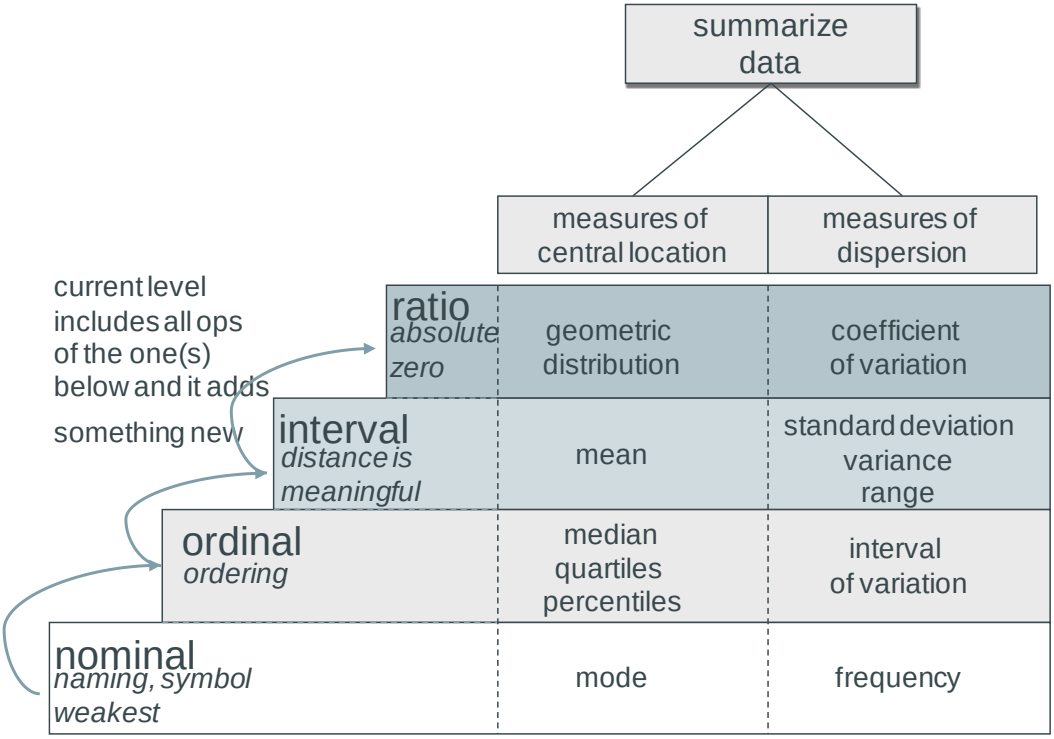


Figure 2.3 - Measurement scales and operations

(Wohlin et al. 2000), (Fenton and Pfleeger 1998), (Siegel and Castellan 1988)

2.5.4 Summarising data

Since data collection produces large amounts of data, it is important to define means by which data can be summarised in order to draw conclusion or to compare different data sets efficiently. Basically, there are two means by which data can be summarised: visual representations (more details in Section 2.6) and statistical measures. *Descriptive statistics* are used to represent quantitative data in a manageable form. These statistical measures form the basis of every quantitative analysis of data.

Measures of central location (also known as measures of central tendency) represent ways of summarising data with reference to its central point. Measures of central location are the mode, the median, the mean, the quartiles and percentiles.

Definition 2.17 – Mode

The *mode* represents the most frequently occurring value.

Definition 2.18 – Median

The *median* of a distribution is the midpoint of the distribution of a variable after the observations have been sorted from low to high. The median can be determined as

$$Median = \frac{n+1}{2},$$

where n is the number of observations.

Definition 2.19 – Mean

The *(arithmetic) mean* is the sum of all the values of a variable divided by the number of observations.

The mean can be determined as

$$Mean = \sum x_i / n,$$

where x_i represents the i-th value of the variable of interest x, and n is the sample size.

Definition 2.20 – Average

The term *average* is used to represent values that indicate the midpoint of a distribution (median or mean).

Definition 2.21 – Percentile/Quartiles

Percentiles are defined as a system of measurement based on percentages, in contrast to the absolute values of a variable (Salkind 2007). *Quartiles* group observations into four equal sized sets according to their rank order. Each of the four sets forms a quartile.

The median splits the data in half, i.e. half of the data is below and half of the data above the median. The median is also called the 50th percentile. Other percentiles can be defined. For instance, the 95th percentile indicates that 95% of the observations are of smaller value and the remaining 5% are larger. The 25th percentile is also called the 1st quartile (lower quartile); the 75th percentile is called the 3rd quartile (upper quartile). Accordingly, the median is denoted as the 2nd quartile.

Measures of variability refer to the spread of values around the central tendency (Wohlin et al. 2000). Frequency, range, variance and standard deviation are measures of variability.

Definition 2.22 – Frequency

Frequency shows the number of observations falling into each of the categories or ranges of values of a variable.

Definition 2.23 – Max, Min, Range, Interval of variation

The *minimum* is the smallest value, the *maximum* is the largest value, and the *range* is the difference between the maximum and minimum. The pair of (Min, Max) of a variable is denoted as *interval of variation*.

Definition 2.24 – Variance

Variance is the average value of the squared difference between an observation and the *population mean*. The *variance* is calculated as:

$$s^2 = \frac{1}{1-n} * \sum_{i=1}^{n} (x_i - \bar{x})^2$$

Definition 2.25 – Standard deviation

The *standard deviation* is defined as the square root of the variance and is calculated as:

$$s = \sqrt{\frac{1}{1-n} * \sum_{i=1}^{n} (x_i - \bar{x})^2}$$

Variance tends to increase with increasing variability around the mean, i.e. large deviations from the mean contribute heavily to the variance because they are squared. The standard mean is therefore a more intuitive measure (Albright, Winston, and Zappe 1999).

2.5.5 Hypothesis testing

Hypothesis testing provides objective rules for determining whether a hypothesis is supported by the data or not. More exactly, in hypothesis testing two competing hypotheses are formulated: *the null hypothesis (H_0) and the alternative hypothesis (H_a)*. The null hypothesis is often the reverse of what the experimenter actually believes. The main goal of hypothesis testing is to reject H_0.

When rejecting or accepting a hypothesis, there is always the possibility of making an error. In hypothesis testing, two types of errors can be made:

- **Type I Error (false positive, alpha α)** means rejecting the null hypothesis when it is true.
- **Type II Error (false negative, beta β)** means accepting the null hypothesis when it is false.

The probability of making a type I error is equal to the **significance level alpha α**. Alpha indicates the probability level the researcher is willing to accept for incorrectly rejecting the null hypothesis. Typical significance levels for alpha are

0.01 and 0.05. For an alpha level of 0.05, the probability of rejecting a true null hypothesis is 0.05.

The **p-value** of a test is the smallest value of alpha for which the null hypothesis would be rejected.

The process of hypothesis testing consists of the following steps:

1. Formulate the null hypothesis and the alternative hypothesis. Choose a significance level for α.
2. Identify the test statistic that can be used to assess the validity of the null hypothesis. Details on criteria are given below.
3. Compute the p-value. The smaller the p-value, the stronger the evidence against the null hypothesis.
4. Compare the p-value with the significance level defined in the first step. If $p \leq \alpha$, the observed effect is statistically significant and the null hypothesis can be rejected.

In the literature, several statistical tests are proposed that can be used to evaluate a hypothesis. A main distinction between the tests is whether a test is parametric or non-parametric.

Parametric tests assume a particular distribution of the underlying data (e.g. normal distribution). **Non-parametric tests** do not make any assumption concerning the distribution of data. Since all tests performed in the empirical studies described in this thesis do not make any assumption on the distribution of the data, non-parametric tests are applied. A second distinction between statistical tests is the type of experiment design with respect to the number of treatments of the analysed factor. Table 2.2 shows different statistical tests for different designs.

Factor with	**Parametric test**	**Non-parametric test**
one treatment		Chi-2
two treatments	t-test, F-test	Mann-Whitney, Chi-2
more than two treatments	ANOVA	Kruskal-Wallis, Chi-2

Table 2.2 - Parametric and non-parametric tests for different designs
adopted from (Wohlin et al. 2000)

In this thesis, two non-parametric tests are applied: the Mann-Whitney test and the Kruskal-Wallis test.

The **Mann-Whitney test** is a non-parametric test that is used to analyse the difference between the mean ranks of two data sets (Wohlin et al. 2000). The null hypothesis is that there is no significant difference between the two data sets. For instance, the Mann-Whitney test can be applied in order to analyse whether a particular bad smell is an indicator for defects in files. In this case, the factor is the bad smell. The treatment is 0 or 1, i.e. 0 if the bad smell applies to a file and 1 otherwise. In order to perform the Mann-Whitney test, the data are classified into two groups: a group containing files for which the bad smell applies and

another group that contains files for which the bad smell does not apply. The null hypothesis is that there is no difference between the two groups; the alternative hypothesis is that there is a difference between the two groups, i.e. the mean defect count (more precise, the mean rank of defect counts) in the group for which the bad smell applies is higher than the mean rank in the "non"-bad smell group.

The **Kruskal-Wallis test** is a non-parametric test, that is used to analyse the difference between the mean ranks of *more than two data sets* (Wohlin et al. 2000). Similarly to the Mann-Whitney test, the null hypothesis is that there is no significant difference between the data sets. For instance, the Kruskal-Wallis test can be applied in order to analyse whether a file's age affects its defect count. The factor is the file's age; possible treatments are "newborn", "young" or "old"[5]. In order to perform the Kruskal-Wallis test, the data are classified into one of the three groups: a group containing newborn files, another group containing young files and a third group containing old files. The null hypothesis is that there is no difference between these groups; the alternative hypothesis is that the mean rank of defect counts differs in the particular groups.

All statistical tests presented in this thesis have been performed with the software product SPSS[6], version 12.

2.6 Data visualisation

Visualisation is a good means of getting a first impression of the data to be analysed. In (Card, Mackinlay, and Shneiderman 1999), (information) visualisation refers to the use of computer-supported visual representations of abstract data to amplify cognition. *Visual representations* use attributes like location, length, shape, colour, size, etc., in order to display information. *Abstract data* usually refer to quantitative data, for instance describing processes or relationships, in contrast to data representing physical objects. Since abstract data have no shape, a mapping to visual representations like shapes, colours, etc. (Few 2009) is necessary. Visualisations are used to *amplify cognition*, i.e. visualisations allow seeing patterns, trends, and exceptions that might be otherwise hard to discover. In addition, visualised data extend our ability to think about information, representing data in ways that human brains can easily comprehend. In this thesis, the histogram is used in order to visualise the mean defect counts in different categories of files, depending on the analysed independent variable.

The histogram shows the distribution of a variable. It is obtained by categorising a variable into classes. Then, for each class, the numbers of observations from the data set which fall into each class are counted. On the y-axis, the frequency of the data in each class is represented by bars. On the x-axis, the classes of the dependent variable are displayed.

---

[5] Detailed definitions for the classification of files in one of the categories "newborn", "young", or "old" are presented in Chapter 10.

[6] http://www.spss.com/de/

The following information can be identified in a histogram:

- centre of the data,
- spread of the data,
- skewness,
- outliers, and
- the presence of multiple modes.

Figure 2.4 shows several histograms with different distributions of the variables.

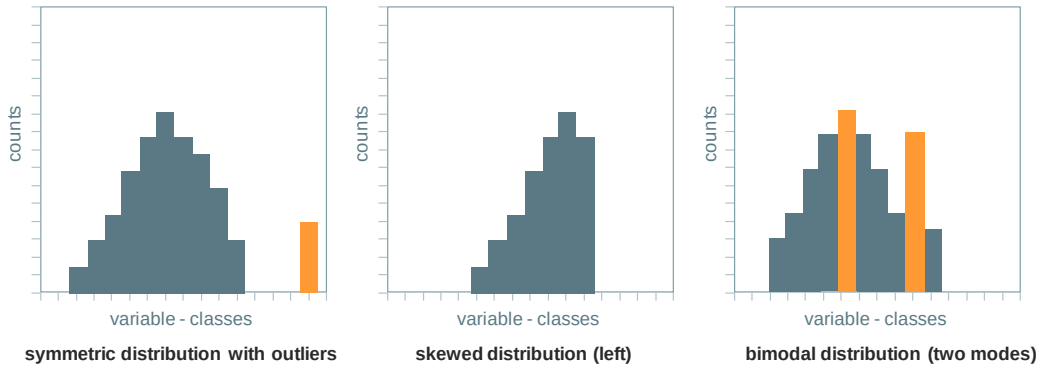


Figure 2.4 - Histograms for different distributions

2.7 Chapter summary

In this chapter, basic terms and concepts related to software testing, empirical software engineering, software measurement, as well as statistics are presented.

Software testing represents a crucial activity in the life-cycle of a software product. But beyond the simple random execution of test cases to see whether the system under test behaves as expected, software testing involves several other activities performed by several roles in the testing process, like the planning and monitoring of testing activities, the definition of the test strategy, or the design of test cases.

Empirical software engineering research is concerned with the analysis of phenomena in the area of software engineering by using evidence, based on observation or experience. Empirical research in the area of software engineering is important because it can accelerate progress by eliminating inadequate approaches. Basically, two main strategies can be distinguished: qualitative and quantitative strategies. Quantitative strategies perform analyses on numerical data and use statistical procedures. In contrast, qualitative strategies aim at understanding a social or human problem from multiple perspectives. These methods use data in form of text, images, or sound drawn from observations, interviews and documentary evidence. Both strategies are important for research and can complement each other. In this thesis, both strategies are used.

For both qualitative and quantitative strategies, methods for data collection and analysis have been proposed in literature.

Software measurement is concerned with the process by which values are assigned to attributes of entities (software processes or software products) in such a way as to describe them according to clearly defined rules. A metric represents a quantitative measure of the degree to which a system, a component, or process possesses a given attribute (IEEE Std 1990).

For analysing quantitative data, *statistical procedures* as well as visual representations can be used. The nature of the data influences the analysis techniques and the operations that can be performed on the data. Different types of data have different underlying scales of measurement. Starting point of an empirical study is the formulation of research hypotheses. Hypothesis testing provides objective rules and statistical procedures for determining whether a hypothesis is supported by the data or not.

Another way of representing data is visualisation. *Visualisations* allow seeing patterns, trends, and exceptions that might be otherwise hard to discover. Visualisation can be used in early stages of the data analysis and complements statistical procedures.

CHAPTER 3 Related work

This chapter presents research related to the topic of this thesis. First, the use of the terms "test strategy" and "test focus" in literature is discussed. Then, related work concerning risk based testing as well as generic frameworks for software measurement and quality modelling are presented. Finally, an overview of models for predicting the software's fault-proneness is given.

3.1 Introduction

In this chapter, topics closely related to the thesis will be presented. The following areas have been considered to be of interest for the research work proposed in this thesis:

1) **Test focus and test strategy definition in literature:** In literature, the terms "test focus" and "test strategy" are used in different contexts. A review of literature with respect to the aspects that are covered by the term "test focus" respectively by the term "test strategy" is presented in Section 3.2

2) **Risk based testing:** Risk based testing is a heuristic generic approach to identify "risky" parts of the software, the test foci, which should be tested (intensively). A discussion of risk based testing approaches is given in Section 3.3.

3) **Generic frameworks for software measurement and quality modelling:** The *Goal Question Metric* (GQM) approach is the most established conceptual framework for software measurement. The approach proposes a hierarchical, goal oriented procedure for the definition of measures and metrics concerning software processes and products. In addition, several frameworks for *quality modelling* have been proposed in literature. These frameworks aim to give a comprehensive view of software quality. Section 3.4 discusses how the empirical approach presented in this thesis relates to the GQM framework as well as to proposed quality models.

4) **Models for the software's fault-proneness:** Research most related to the topic of this thesis concerns the analysis of indicators and models for the software's fault-proneness. A general overview of these models is given in Section 3.5.

3.2 Test strategy and test focus

In this thesis, the definition of the test foci is part of the test strategy, whereas the test strategy indicates the overall approach to testing. In literature, both terms are used in several contexts.

First, the term "test focus" is used to denote parts of the test strategy as defined in this thesis (Kaner, Bach, and Pettichord 2002), (Spillner and Linz 2010), (Koomen and Pol 1999). In (IEEE Std. 1998), the IEEE standard for test documentation, the test plan is a "document that describes the technical and management approach to be followed for testing a system or component". Typical contents identify the items to be tested, tasks to be performed, responsibilities, schedules, and required resources for the testing activity. According to this definition, the test foci represent the items to be tested.

In (Gras, Gupta, and Perez-Minana 2006), the term "test strategy" is used to indicate "high risk areas" in the software. Thus, the term "test strategy" is used synonymously with the term "test focus" as defined in this thesis.

In another group of research work, the term "test strategy" is used to refer to other aspects of software testing that are not related to the test focus definition. For instance, in context of integration testing, "test strategy" refers to the optimal order in which components of the software have to be tested, for instance

in (Briand, Labiche, and Wang 2003), (Jéron et al. 1999). The term "test strategy" is also used in the context of optimizing testing techniques. For instance, in (Cui, Li, and Yao 2009), strategies for the pair-wise[7] test data generation are presented. Finally, in context of product line engineering, the term "test strategy" is used to refer to the approach to test product lines. Basically, the following "test strategies" can be applied: test each product line member separately, or test just product-specific parts and compose them with tested core assets from family engineering.

3.3 Risk based testing

Risk based testing is a generic heuristic approach to identify "risky" parts of the software that should be tested (intensively). Risk is defined as the product of damage and the probability of failure, i.e. the more probable it is that a software entity (e.g. a component) will fail and the higher the damage in case of failure is, the higher is the risk of that component. Risk based testing prioritises testing activities according to the risks assigned to software entities (Schäfer 2004), (Amland 2000), (Bach 2003), (Bach 1999), (van der Aalst 2006), (Pinkster et al. 2004).

The probability that a software component fails mainly depends on its usage frequency and the lack of quality (Schäfer 2004). The higher the usage frequency of a software component, the higher is the probability that a defect will expose a failure. If the component is faulty, the probability is high that the execution of the software will expose a failure (see Figure 3.1 **Fehler! Verweisquelle konnte nicht gefunden werden.**).

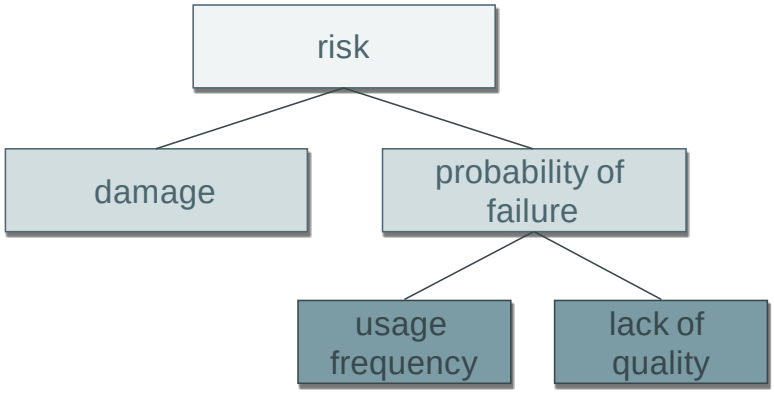


Figure 3.1 - Risk = damage x probability of failure (Schäfer 2004)

In literature, several approaches have been presented, each of them addressing particular aspects of the software's risk as defined in (Schäfer 2004).

- **Heuristics.** This type of research proposes heuristics for faulty parts. These heuristics usually consider *all aspects of risk* as defined by (Schäfer 2004): damage, usage frequency, and lack of quality. In (Schäfer 2004),

[7] Pair wise testing is a test data generation technique that assumes that most of faults are caused by the interaction of two variables. Test cases generated by applying pair wise generation techniques cover all combinations of two pairs of input parameters.

(Amland 2000), (Bach 1999), and (Bach 2003), several heuristics that explore software risks are given; for instance, complex or changed areas, the number of people involved in developing the software, the use of new tools or new technology, time pressure, areas that have been already defective before, geographical spread of the development team, etc., are risks. The main drawback of these heuristics is the lack of empirical validation; they are based on the authors' experience and must be validated in each particular context.

- **Prioritisation of testing activities according to requirements' priority.** This type of research mainly considers the *damage* aspect of the risk definition. The higher the priority of a requirement (from the customer's point of view), the higher is the damage in the case that the realisation of that requirement exposes a failure. In (Srikanth 2006) and (Srivastva, Kumar, and Raghurama 2008), approaches to prioritise testing activities based on requirements' priority are presented.
- **Prioritisation of regression testing activities.** This type of research mainly addresses the *probability of failure* of a previously tested program. Regression testing is performed when the software or its environment changed in order to ensure that defects have not been introduced or uncovered in unchanged areas of the software (ISTQB 2007). In order to reduce the cost of regression testing, software testers prioritise test cases so that those which are more important, by some measure, are run earlier in the regression testing process. One goal of prioritisation is to increase the number of defects detected by tests (Elbaum et al. 2004). There have been various efforts in this area to minimise the number of *existing* test cases to be re-run. Factors often used to prioritise existing test cases are the coverage achieved by the test cases, the defects found when test cases have been executed before, and the costs of re-executing test cases (Elbaum et al. 2004), (Do and Rothermel 2006). Recent research work focuses on the prioritisation of regression test cases based on the priority of the requirements (Jeffrey and Gupta 2008).
- **Estimation of fault-prone parts in software.** This type of research is the most similar to the research presented in this thesis. Research in this area mainly focuses on exploring the relationship between several project history as well as product characteristics and the software's defect count and on estimating fault-prone parts of the software based on different models. An overview of typical models used to estimate the software's fault-proneness is given in Section 3.5.

3.4 Generic frameworks for software measurement and quality modelling

The most established framework for software measurement is the Goal Question Metric approach (GQM), initially developed by (Basili and Weiss 1984). The roots of this approach reach back to the late seventies, where software measurement was in its "primitive stage" (Basili, Caldiera, and Rombach 1994).

As consequence of the weaknesses identified in the approaches at that time, the authors present a goal oriented framework for software measurement. The GQM approach constitutes of a hierarchical framework that can be used to define metrics related to software products and processes. The hierarchy starts with a *goal* that formulates the purpose of measurement, including the object(s) to measure and the viewpoint from which to take the measurement. Then, the goal is refined into several *questions* that characterise the goal. At *metric* level, the metrics to be collected in order to answer the questions are defined.

Table 3.1 summarises the situation until the late seventies and the premises of the GQM approach as described in (Basili, Caldiera, and Rombach 1994).

Immature situation of software measurement until the late seventies as characterised in (Basili, Caldiera, and Rombach 1994)	**Premises of the GQM approach aiming to overcome weaknesses of approaches presented at that time**
Research concentrates on developing new measures and models without clearly defining measurement goals. Popular code size measures (Halstead 1977) or complexity measures (McCabe 1976) originate from this time.	**Premise 1.** What to measure highly depends on the scope and the purpose of the measurement.
Main purpose of the measurement: control product and project level properties.	**Premise 2.** Purposes of measurement are to *understand,* plan and control. The authors emphasise the necessity to *understand* what factors influence product and process quality.
Researchers search for standard sets of models and measures to quantify software processes and products; environmental characteristics and their impact on software products and processes is underestimated.	**Premise 3.** Models and measures for software products and processes highly depend on environmental characteristics.

Table 3.1 - Fundamental ideas of the GQM approach

The empirical approach presented in this thesis uses basic ideas of the GQM approach but it goes beyond it. The usefulness of a *goal oriented* procedure when identifying indicators for defects in software is also advocated in this thesis (Premise 1 of the GQM approach). Furthermore, the approach focuses on *understanding* what factors influence the software's fault-proneness (parallel to Premise 2 of the GQM approach). Research of past years shows that indicators for software defects also highly depend on the *development context* (Premise 3).

The empirical approach goes beyond the GQM approach. First, it proposes detailed visual and statistical procedures to collect, analyse and select justified indicators for defects in software. Second, the empirical approach presented in this thesis integrates statistical analyses to *assess* the usefulness of the metrics empirically. The GQM approach proposes a conceptual framework for the definition of metrics. But in principle, it does not support the limitation and the empirical assessment of the derived measures.

Beside the GQM approach, several conceptual frameworks for defining software quality have been proposed in literature, for instance in (McCall, Richards, and Walters 1977), (Boehm et al. 1978), (Dromey 1996), (ISO/IEC Standard 2001). These frameworks propose a hierarchical decomposition of the software quality, whereas the models basically differ in the factors and the metrics chosen to describe software quality. These frameworks can give guidance when deriving potential indicators for defects in software. But, they do not give advices on how to (empirically) evaluate the usefulness of the derived metrics, that represents the focus of the empirical approach presented in this thesis.

3.5 Models and indicators for the software's fault-proneness

Several (more and more sophisticated) approaches for estimating the fault-proneness of software have been presented in literature. The approaches basically differ in the models and model parameters used for prediction.

The models include *statistical approaches*, e.g. (Zimmermann, Nagappan, and Zeller 2008), *tree based models*, e.g. (Porter and Selby 1990), (Guo et al. 2004), (Khoshgoftaar et al. 2000), *analogy based models*, e.g. (Emam et al. 2001), (Khoshgoftaar, Seliya, and Sundaresh 2006), or *neural networks*, e.g. (Thwin and Quah 2005). Model parameters are often structural code characteristics like the number of lines of code or different complexity metrics. Other parameters are historical characteristics, for instance the number of changes performed to a software entity or the number of defects detected in previous releases. Several approaches combine different kinds of parameters.

Apart from few examples, most of the studies use data collected from commercial products. A small part of research is conducted in the context of open source development, for instance research reported in (Denaro and Pezzè 2002), (Gyimothy, Ferenc, and Siket 2005), and (Kim et al. 2008). Nearly all approaches use structural characteristics of the software as model parameters. In addition, historical characteristics are mostly used in combination with structural characteristics as model parameters. In only few cases, data from academic software developed by students is used (Basili, Briand, and Melo 1996), (Briand, Daly, and Wüst 1998), (Briand et al. 2000). Little attention has been paid on analysing the relationship between bad smells and defects in software empirically. Few studies use visual representation to explore and to analyse the data (Ostrand and Weyuker 2002), (Purushothaman 2005), (Pighin and Marzona 2003), (Andersson and Runeson 2007), (Wu, Wang, and Yang 2008).

Despite the model used, researchers agree to the fact that there is a need to find indicators for defects in software in order to allocate quality assurance effort appropriately. Nevertheless, there is no empirically validated consensus on the superiority of one modelling method over another[8]. A recent debate shows that there is no consensus on how to evaluate different defect prediction models, i.e. how to assess their performance and to make detailed comparisons of several

[8] (Myrtveit and Stensrud 1999), (Myrtveit, Stensrud, and Shepperd 2005), (Shepperd and Kadoda 2001)

models (Zhang and Zhang 2007), (Menzies et al. 2008), (Jiang, Cukic, and Ma 2008), (Lessmann et al. 2008).

Main drawback of the models presented in literature is their complexity that hinders that the nature of the detected relationships is understood. Most approaches neglect criteria like ease of use and comprehensibility that are prerequisites for a model to be applied in practice. In fact, the approaches focus on the model itself without putting its application in a context. Questions like: "Who should use the model and when, during the development process?", "Which steps have to be performed in order to perform efficient analyses on defects in an organisation?" etc. are mostly not considered.

Since prediction accuracy will never reach 100%, prediction models can only be used as indicators and not as "definitive oracles". Therefore, testers' experience is a valuable complementary information source. This fact is also neglected in literature.

Table 3.2 summarises the approaches. The column "Context" indicates whether the research work uses data from commercial systems (COMM), academic systems (ACAD), or from open source programs (OSP) to validate the model. The next columns indicate the parameters used in the corresponding models (H-historical indicators, BS – Bad Smells in code, P – Pareto principle, S – structural characteristics, O – Others). This table aims to give a global overview of the research work related to this thesis.

Detailed discussions of the approaches along with a comparison of the results obtained by the empirical studies presented in this thesis are given in the corresponding chapters. Chapter 8 discusses related work concerning the Pareto principle whereas Chapter 9 presents related work concerning the relationship between bad structural characteristics of software and its defects. In Chapter 10, an overview of empirical studies that explore the relationship between the history of a software entity and its defect count is given.

Reference	**Context**	**Model**	**H**	**BS**	**P**	**S**	**O**
(Adams 1984)	COMM	Statistical procedures			X		
(Andersson and Runeson 2007)	COMM	Statistical procedures (descriptive statistics) and visual representations			X		
(Arisholm and Briand 2006)	COMM	Statistical procedures (logistic regression)	X			X	
(Basili, Briand, and Melo 1996)	ACAD[9]	Statistical procedures (logistic regression)				X	
(Bell 2005)	COMM	Statistical procedures (negative binomial regression model)	X			X	

[9] student programs

Reference	Context	Model	H	BS	P	S	O
(Bell, Ostrand, and Weyuker 2006)	COMM	Statistical procedures (negative binomial regression model)	X			X	
(Binkley and Schach 1998)	ACAD	Statistical procedures (Correlation analysis)				X	
(Briand, Daly, and Wüst 1998)	COMM, ACAD	Statistical procedures: logistic regression				X	
(Briand et al. 2000)	ACAD	Statistical procedures (Principal component analyses, descriptive statistics, logistic regression)				X	
(Briand, Devanbu, and Melo 1997)	COMM	Statistical procedures (logistic regression)				X	
(Cartwright and Shepperd 2000)	COMM	Statistical procedures (descriptive statistics, regression)				X	
(Denaro, Morasca, and Pezzè 2002)	COMM	Statistical procedures (logistic regression)				X	
(Denaro and Pezzè 2002)	OSP	Statistical procedures (logistic regression)				X	
(Emam et al. 2001)	COMM	Analogy-based (case-based reasoning)				X	
(Endres 1975)	COMM	Statistical procedures (descriptive statistics)			X		
(Fenton and Ohlsson 2000)	COMM	Statistical procedures (descriptive statistics) and visual representations			X		
(Graves et al. 2000)	COMM	Statistical procedures (extends linear regression)	X			X	
(Guo et al. 2004)	COMM (NASA)	Tree-based					
(Gyimothy, Ferenc, and Siket 2005)	OSP	Statistical procedures (Linear, logistic regression)				X	
(Hatton 1997)	COMM	Statistical				X	
(Hatton 2008)	SCIENTIFIC	Statistical (descriptive statistics)			X		
(Holschuh et al. 2009)	COMM	Statistical procedures, regression	X	(X)	X	X	
(Kaâniche and Kanoun 1996)	COMM	Statistical (descriptive statistics), visual representations			X		
(Khoshgoftaar et al. 1998)	COMM	Statistical procedures (logistic regression)	X				
(Khoshgoftaar et al. 2000)	COMM	Tree-based	X			X	X[10]
(Khoshgoftaar, Seliya, and	COMM	Analogy-based				X	

[10] execution metrics

Reference	Context	Model	H	BS	P	S	O
Sundaresh 2006)							
(Kim et al. 2008)	OSP	Others (algorithmic)	X				
(Koru, Zhang, and Liu 2007)	OSP	Cox proportional hazards modelling with recurrent events				X	
(Koru et al. 2008)	COMM/ OSP	Statistical procedures				X	
(Koru and Tian 2003)	COMM	Statistical procedures				X	
(Layman, Kudrjavets, and Nagappan 2008)	COMM	Statistical procedures	X			X	
(Mockus, Zhang, and Li 2005)	COMM	Statistical procedures (logistic regression)	(X)				X[11]
(Munson and Khoshgoftaar 1992)	COMM	Statistical procedures (discriminant analysis)				X	
(Nagappan, Ball, and Zeller 2006)	COMM	Statistical procedures				X	
(Nagappan and Ball 2005)	COMM	Statistical (regression)	X				
(Ohlsson et al. 1999)	COMM	Statistical (principal component analysis)	X			X	
(Ohlsson and Alberg 1996)	COMM	Statistical procedures			X	X	
(Ostrand and Weyuker 2002)	COMM	Statistical (descriptive statistics) and visual representations			X		
(Ostrand, Weyuker, and Bell 2005; Ostrand, Weyuker, and Bell 2004)	COMM	Statistical (negative binomial regression)	X			X	
(Pighin and Marzona 2003)	COMM	Statistical (correlation analysis)	X			X	
(Pighin and Marzona 2003)	COMM	Statistical (descriptive statistics) and visual representations	X		X	X	
(Porter and Selby 1990)	COMM (NASA)	Tree-based	X				
(Purushothaman 2005)	COMM	Statistical (descriptive statistics) and visual representations	X				

[11] Hardware configurations, software platforms, amount of usage and deployment issues.

Reference	Context	Model	H	BS	P	S	O
(Schröter et al. 2006)	OSP	Statistical procedures (correlation analysis)	X			X	
(Porter and Selby 1990)	COMM (NASA)	Tree-based method	X			X	X[12]
(Shatnawi and Li 2006)	OSP			X			
(Subramanyam and Krishnan 2003)	COMM	Statistical (univariate/multivariate regression)				X	
(Thwin and Quah 2005)	COMM	Neural nets				X	X[13]
(Weyuker, Ostrand, and Bell 2008)	COMM	Statistical procedures (binomial regression)	X				
(Weyuker, Ostrand, and Bell 2007)	COMM	Statistical procedures (binomial regression)	X				
(Wu, Wang, and Yang 2008)	COMM	Statistical procedures (descriptive statistics) and visual representations	X			X	
(Zimmermann, Nagappan, and Zeller 2008)	COMM/ OSP	Statistical procedures (principle component analysis, correlation analysis)	X			X	
(Zimmermann, Premraj, and Zeller 2007)	OSP	Statistical procedures (principle component analysis, correlation analysis)	X			X	

Table 3.2 - Models and indicators for defects in software

3.6 Chapter summary

In this chapter, research related to the topic of this thesis is presented. It falls into four categories: definition of the terms "test strategy" and "test focus" in literature, risk based testing, generic frameworks for software measurement and quality modelling, and finally, models for estimating the software's fault-proneness.

The terms "test strategy" and "test focus" are used in several contexts in literature. In some cases, test focus is seen as part of the test strategy. In some cases, both terms are used synonymously. But in most of the cases, the term "test strategy" is used to refer to particular aspects of the test strategy that are not related to the test focus.

Risk based testing is a generic heuristic approach to identify "risky" parts of the software that should be tested (intensively). Risk is defined as the product of damage and the probability of failure, i.e. the more probable it is that a software entity (e.g. a component) will fail and the higher the damage in case of failure is

[12]development effort

[13]memory allocation

the higher is the risk for that component. Several authors propose heuristics based on their experience for "risky" parts of the software.

An established framework for software measurement is the Goal Question Metric approach (GQM). This approach proposes a hierarchical framework that can be used to define metrics related to software products and processes. The empirical approach presented in this thesis uses basic ideas of the GQM approach (goal orientation, emphasis on understanding factors that influence defects in software, necessity of defining context specific indicators). But it goes beyond the GQM approach by proposing detailed steps to identify and *validate* indicators for defects in software.

Research most related to the topic of this thesis concerns the analysis of models of the software's fault-proneness. Models can be basically classified into statistical and machine learning models. One of the main weaknesses of the approaches presented in literature is the lack of comprehensiveness of the models used. In addition, the approaches do not consider testers' experience that can be very valuable since 100% prediction accuracy can never be reached.

CHAPTER 4 Testing process – A decision based view

Software processes often focus on artefacts, activities and roles, treating decisions to be made during the software development process only implicitly. However, the awareness of these decisions increases their quality by forcing the decision-makers to search for alternatives and to trade off between them. In this chapter, an overview of the testing process from a decision based view is given. Therefore, a decision hierarchy for the testing process is presented. This hierarchy comprises all decisions made during testing and reflects dependencies between them.

4.1 Introduction

Today's software systems consist of numerous software components; they realise countless requirements and are developed in an industrial environment limited by high time and resource constraints. In order to assess to which extent the software system or its parts fulfil the requirements, testing activities have to be performed. Since complete testing is impossible (Myers 1979), testers are forced to make decisions, i.e. to decide which parts of the software system have to be tested in which way. Usually, these decisions are made implicitly by the corresponding roles and often, the responsible persons are not aware of the decisions they made. However, the awareness of decisions can significantly improve their quality. Making a decision consciously forces the person who has to make this decision to search for alternatives, to establish selection criteria and to trade off between advantages and disadvantages of several alternatives. Consequently, the awareness of decisions leads to better decisions compared with implicit or ad hoc decisions and increases the quality of the testing process.

In this thesis, a decision is defined as follows (Borner, Illes, and Paech 2007a/b):

Definition 4.1 – Decision

> A *decision* denotes a choice consciously or unconsciously made by a person or group of persons. A decision made consciously evolves in the process of discussing possible alternatives and considering existing success criteria.

During the software development process as well as during the testing process, several decisions have to be made. The best alternative has to be selected from e.g. alternative GUI designs, architectural patterns, or testing techniques.

The remainder of this chapter is organised as follows. Section 4.2 introduces the generic decision hierarchy for the testing process, containing decision levels and corresponding decisions. In Section 4.3, the validation of the approach is presented, whereas Section 4.4 gives an overview of related work. A short summary of this chapter is given in Section 4.5.

4.2 Decision hierarchy

The decision hierarchy structures decisions to be made during the testing process. These decisions are assigned to *decision levels* (Borner, Illes, and Paech 2007a/b).

The development of the decision hierarchy involved several steps:

Step 1: First, the tasks and roles proposed in standard textbooks such as (Spillner and Linz 2010) and (Mosley and Posey 2002) have been analysed. The decision hierarchy is mainly based on the fundamental testing process described in (Spillner and Linz 2010) consisting of test planning and specification, test execution, as well as capturing and analysing test results.

Step 2: In a next step, decisions to be made while performing testing tasks have been identified and grouped into seven decision levels. The result is the generic

decision hierarchy illustrated in Figure 4.1. Management decisions and issues like scheduling or training are not addressed here.

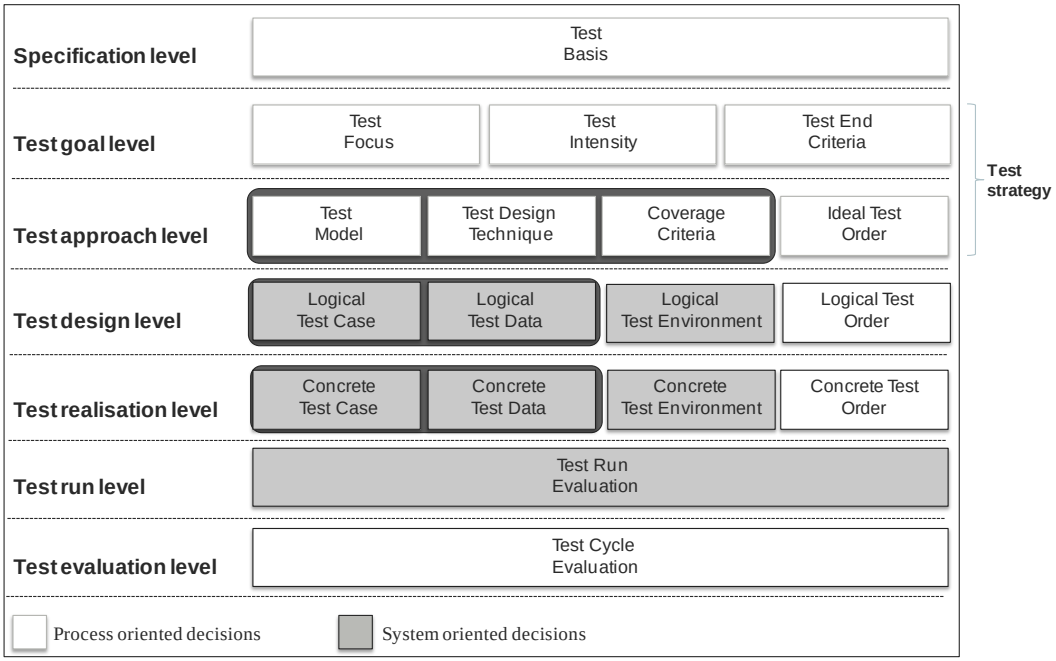


Figure 4.1 - Decision levels and corresponding decisions
(Borner, Illes, and Paech 2007a/b)

The *principles* underlying the decision hierarchy can be defined by the following rules:

- **R1 Decision dependencies:** Decisions at lower levels depend on decisions made on earlier levels. If decisions at top levels are left out, they are implicitly contained in decisions on lower levels. Leaving out a decision decreases the quality of this particular decision as well as the quality of all dependent ones. The goal of making decisions in the proposed order is to facilitate the decision making process.
- **R2 Parallelism:** All decisions on the same level can be done in parallel, i.e. these decisions can be made nearly independently, but they may influence each other. Decisions that influence each other can be combined to *decision bundles*. In **Fehler! Verweisquelle konnte nicht gefunden werden.**, decision bundles are represented by a dark grey box behind the corresponding decision.

The test strategy comprises all decisions of the test goal and test approach level.

Moreover, two different *perspectives* on the decisions can be identified. One perspective contains decisions that influence the testing process (these decisions are process oriented), i.e. which test artefacts will be created. Another perspective contains decisions concerning the system under test (these decisions are system oriented), i.e. how the system will be tested. In the following, the different decision levels are introduced along with the corresponding decisions on each level.

4.2.1 Specification level

The specification level contains decisions that deal with the completeness of the **test basis**. The test basis includes all information needed for a successful start of the testing process. It refers to the documentation on which the test cases are based. Often, the test basis consists of the specification of the software system at different development stages (e.g. requirements specification or system design specification). Defects found in previous releases can also serve as test basis.

At specification level, it has to be decided whether the test basis is complete or not. If important information is missing in the test basis, critical parts of the software can be overlooked and remain untested. Thus, missing information has to be added. The decisions on this level influence all decisions on the lower levels.

4.2.2 Test goal level

Considering that a software project is usually limited in time, not all parts of the test basis can be tested. Therefore, at test goal level it has to be decided which parts of the system have to be tested and which not. For this purpose, it is essential to possess a complete test basis in order to select the critical test objects. All parts of the system that have been selected to be tested are denoted as **test foci**. In this thesis, an empirical approach is presented and validated extensively that guides testers in making justified decisions on test foci.

Besides time pressure within the testing process, another constraint influences the decisions on this level: costs. The cost constraints lead to a limitation of resources needed in the testing process. Therefore, the existing resources have to be split up among several test foci. To concede the correct assignment of resources to the various test foci, it has to be decided which **test intensity (**measured e.g. by man days or funds) a single test focus has to be assigned to.

Test end criteria define conditions that have to be fulfilled to finish the testing activities; for instance, they can give information on the required rate of successful test runs.

4.2.3 Test approach level

The test approach level comprises decisions related to the test design techniques to be used, the test model(s) and its coverage(s) as well as the ideal test order. One decision to be made concerns the **test design technique** which will be used to derive test cases and test data from the test basis. For each test level (system, integration, and unit test level), a countless number of test design techniques can be found in the literature (e.g. in (Beizer 1990), (Binder 1999), (Spillner and Linz 2010), (Myers 1979)). Therefore, existing test design techniques, the defined test foci and test intensities have to be taken into account in order to select the most adequate test design technique(s). In parallel, decisions on the **test model(s)** have to be made. A test model facilitates the derivation of test cases and test data in comparison to the derivation from an informal specification. A state based model or a control flow model are examples of test models. A test design technique influences the selection of the test model and vice

versa. Later in the testing process, the selected test design techniques have to be applied in order to derive test cases and test data to achieve the given test coverage and to fulfil the test **coverage criteria**. The test coverage is an indicator for the number of test cases to be derived. The test design technique influences the decision on coverage criteria and vice versa.

Furthermore, on this decision level an **ideal test order** to test the different test objects has to be specified. The ideal test order represents an optimal order to test the different parts of the system by taking into account the information on the test foci and on test intensity. An example of such an ideal test order could be that all test objects with the highest test intensity should be tested first, followed by the ones with the next lowest intensity and so on.

4.2.4 Test design level

The test design level is the most complex level of the testing process. The main decision on this level is how to test the different test foci, i.e. the selected test objects. Therefore, the given test design techniques are applied to derive **logical test cases**, also called abstract test cases (ISTQB 2007), (Spillner and Linz 2010). A logical test case gives an abstract description of how to test a specific aspect of the objects under test. In parallel to the test case design, it has to be decided which **logical test data** serve as an input for the test objects within the test case. The logical test data represent the abstract description of the data to be sent to and returned by the test object. Both the specification of a logical test case and the required test data, are connected. A logical test case without the required logical test data is not complete and vice versa.

The third decision on this level concerns the definition of the **logical test environment**. The decision comprises the kind of tools as well as software and hardware needed during the execution of the test cases. Similar to the specification of the logical test cases or test data, the description of the logical test environment is also abstract and represents the general requirements on the test environment.

The last decision at test design level discussed here is related to the **logical test order**. This order refines the ideal test order considering dependencies between test cases as well as information about planned test environment factors. Execution efficiency and parallelism are main criteria influencing this decision.

4.2.5 Test realisation level

The test realisation level details the logical representation of the test cases, of the test data, as well as of the test environment. It contains all decisions that influence the execution of a test case. This level contains decisions on the concrete test order, on concrete test cases, concrete test data, and the concrete test environment. Setting up the **concrete test order** means to identify an actual executable test order considering the logical test order and the project environment factors. In parallel, the logical test cases are refined by **concrete test cases.** Thus, information on the specific behaviour of the test case and the test object is added. Concrete test cases contain all information needed to execute the test

case. To complete the specification of a concrete test case, the detailed description of **concrete test data** is needed. Consequently, it has to be decided which concrete "instances" of the logical test data are used in concrete test cases. The decisions on the **concrete test environment** consider the description of the logical test environments and the specification of the logical test cases. The concrete test cases need a corresponding concrete test environment (e.g. the specification of concrete hardware and software needed) in order to be executable.

4.2.6 Test run level

The test run level deals with the evaluation of test run results. After the execution of a test case, the **test run evaluation** decides whether the test run revealed a defect or not. If this is the case, a state (e.g. "open"), a priority (e.g. "critical"), and a weight (e.g. system crash) have to be assigned to the corresponding defect (Spillner and Linz 2010).

4.2.7 Test evaluation level

This level contains the decision whether test activities can be finished. The decisions on the **test cycle evaluation** check whether the test end criteria have been fulfilled and whether every test focus has been tested with the required test intensity. Furthermore, the defects not found within the current test cycle are estimated by using a metric like the defect detection rate. The decision not to finish the test cycle leads to a new iteration of some (or maybe all) of the testing tasks and decisions.

4.3 Validation

The decision hierarchy has been validated in several contexts. First, it has been refined in order to highlight decisions of the system testing process (Borner, Illes-Seifert, and Paech 2007), (Borner, Illes, and Paech 2007). In addition, the decision hierarchy served as the basis for a test process analysis in an industrial case study and as a framework for classifying testing research (Illes and Paech 2006). Furthermore, the decision framework proved of value as a framework for the evaluation of testing tools (Illes et al. 2006). Finally, a qualitative analysis of test processes from the perspective of experienced testers has been performed (Illes-Seifert and Paech 2008). The results of the qualitative study are detailed in Chapter 5. Details on the validation results are summarised in the Appendix A 1.1 - A 1.4 of this thesis.

4.4 Related work

A process model that describes the main phases of the testing process consisting of test planning, test design, test execution, and test evaluation activities has been proposed in (Spillner and Linz 2010). In comparison to the decision hierarchy which explicitly focuses on all decisions to be made during the testing process, the process model described in (Spillner and Linz 2010) does not take decisions into account. The IEEE standard for software test documentation (IEEE Std. 1998) specifies all artefacts to be created during the testing process (e.g. test plan, test design specification, or test case specification). The decisions

made within the testing process are not part of the standard. Another group of related work comprises test process improvement models like TPI (Test Process Improvement), (Koomen and Pol 1999) or test maturity assessment models like TMM (Testing Maturity Model), (Burnstein, Suwannasart, and Carlson 1996), or TMMi (Testing Maturity Model *integration*) (Tmmi Foundation; eds. van Veenendaal, E. 2009). The primary focus of these models is not the test process itself, but the steps for its improvement.

A conceptual framework categorising different decisions made during requirements engineering has been presented in (Paech and Kohler 2003) and in (Aurum, Wohlin, and Porter 2006). These approaches do not consider decisions to be made during other phases of the software engineering process. Furthermore, the system "Sysiphus[14]" supporting the documentation of decisions defined in (Paech and Kohler 2003) has been realised in (Wolf and Dutoit 2004). Additionally, several approaches for the documentation of the decisions made during the software development process have been proposed in (Dutoit et al. 2006). To the best of the author's knowledge, there is no existing research that particularly addresses the decision making process within quality assurance activities.

4.5 Chapter summary

In this chapter, a decision hierarchy that aims to structure the decisions made during the testing process is presented (Borner, Illes, and Paech 2007a/b). A decision based view of the testing process is useful since the awareness of decisions to be made increases the quality of the decisions, by forcing the decision-makers, in this case the testers, to search for alternatives and to trade off between them.

The decisions to be made in the testing process are structured in a hierarchy, i.e. decisions at lower levels depend on decisions made on earlier, "higher" levels. If decisions at top levels are left out, they are *implicitly* contained in decisions on lower levels. Leaving out a decision decreases the quality of this particular decision.

The decision hierarchy proved of value for researchers as well as for practitioners. A detailed application of the framework follows in Chapter 5 where a qualitative analysis of testing processes in industry is described.

Based on the experience in applying this hierarchy in several case studies, it can be concluded that the hierarchy is universal enough to be applied in different contexts. But, it is also specific enough to highlight the similarities and differences of the subject matters. Additionally, the approach is easy to be learned. Thus, students as well as practitioners get familiar with key issues of the testing process without having to get into details. Finally, the hierarchy eases the communication among testers by providing a common terminology.

[14] http://sysiphus.in.tum.de/

CHAPTER 5 State of the practice of testing processes – A qualitative study

During software testing, several decisions have to be made as described in Chapter 4. In this chapter, the results of an exploratory study with expert testers are presented. The main goal of this study is to identify characteristics of test processes in practice along with their strengths and weaknesses.

5.1 Introduction

As seen in Chapter 4, several decisions have to be made during the testing process. In order to conduct all decisions thoroughly, testers need information that is complete and up-to-date, for instance about requirements or project status. The knowledge of testers' information needs allows providing testers with the right information at the right time. Based on this knowledge, test process improvements can be designed and implemented. In addition, approaches that address the gaps identified before can be proposed.

In this chapter, a qualitative study is presented that analyses which documents are frequently used and which roles are consulted when making decisions during testing. In addition, the role of experience needed to make sound decisions is investigated. The results of the study show that (a) experience plays an important role in software testing, (b) the requirements specification and previously found defects are the most important information sources for testers, and (c) testers lack of approaches that alleviate decisions on test goal level (Illes-Seifert and Paech 2008c). The results of this study served as input for the thesis as this thesis addresses a main part of the problems identified in the qualitative study.

The remainder of this chapter is organised as follows. Section 5.2 presents the overall goal of the study along with the research questions. Section 5.3 describes the study design. Section 5.4 presents the findings, whereas Section 5.5 discusses the main results. Section 5.6 shows implications for this thesis resulting from the findings of the study. In Section 5.7, threats to validity of the study are discussed. Related work is presented in Section 5.8. Finally, Section 5.9 summarises this chapter.

5.2 Study goal and research questions

The overall goal of this study is to analyse how testers work and which decisions they make. In addition, this study analyses which of the decisions are made explicitly and which ones implicitly. Particularly, the following research questions are addressed:

Q1: Which documents are frequently used by testers when making which testing decisions? The main assumption of this research question is that documents are an important information source for all participants of the software engineering process, including testers. To know which documents are frequently used by testers is important because quality assurance activities concerning information sources often consulted by testers can be intensified purposefully. In addition, missing information can be identified along with approaches to collect and analyse it appropriately.

Q2: What role does communication play as an information source? The main assumption of this question is that documentation is never completely sufficient as input to the testing process so that details have to be clarified in face-to-face discussions. And even if documentation was complete, communication is often favoured over reading documents.

Q3: What is the role of experience in testing? This is an important question to be analysed, because it is essential to know to what extent and for which decisions testers rely on their experience instead of on documentation. Knowing this enables to decide which activities are suited for test automation or which are suited to be executed by novice testers (because they do not require much experience).

5.3 Study design

5.3.1 Participants

The main criterion for the selection of the participants for this study is their experience in the testing area. As a consequence, all participants have at least three years of experience, and most of the participants have five to ten years of experience. Three participants have even more than ten years of experience. Table 5.1 summarises the characteristics of the participants. The participants are employees of five organisations denoted in the following as organisation A-E. Organisation A and E develop standard software, whereas the other organisations develop individual software. Only organisation C develops software for in-house use. The testers in organisation A work on the same project, whereas the testers in the organisations C and D work on different projects.

5.3.2 Study process

The study is performed as a qualitative study. This research method is used because it helps to gain more experienced with the phenomenon to be analysed. In this case, the overall goal is to get a deep understanding of the testing process along with its information flow.

The study is conducted in form of seven face-to-face interviews and one telephone interview. Three interviewees completed the questionnaire “offline”. The interviews are semi-structured, based on a questionnaire sent in advance to the participants. The interviews took three hours on average.

Data Collection. In the data collection phase, field notes taken during the interviews were coded and stored in a study data base. Coding is a method which assigns values to qualitative statements. This allows the combination of qualitative and quantitative methods for data analysis purposes (Section 2.3.4). During the offline coding process, interviewees were contacted when ambiguities in the data occurred.

To assure the validity of the results, multiple information sources have been used for evidence as recommended in (Yin 2003). Thus, beside interviews, document reviews have been performed (e.g. reviews of test plans, test case specification templates and test case specifications, test protocols, as well as test process descriptions). Furthermore, other information sources have been consulted like internal discussion forums. Another aspect considered to assure validity was the representativeness of the interviewees with regard to their qualification, experience, and testing tasks. All interviewees are experienced testers, three of them with more than ten years of testing experience.

Data Analysis. For data analysis, different qualitative and quantitative analysis methods are used. Quantitative methods are used in order to determine patterns and tendencies in the data, for instance by counting which role is consulted most of all during the testing process. Qualitative methods are used to search for an explanation for these particular tendencies. In this study, cross-case analysis is performed. Cross-case analysis partitions the data into different categories by using different criteria, for example depending on the testing group's organisation as an independent team or not (2.3.4).

Experience (in years)	Role(s)	Main Tasks	Organisation
>10	Test designer	Test planning, Test case design, Manual test execution.	D
>10	Test designer	Test planning, Manual test execution.	D
>10	Test manager	Establishment of a standard testing process including supporting tools.	B
>10	Tester, Test manager	Test planning, Manual test execution.	E
10	Test manager, Quality engineer	Test planning, Test case design, Monitoring system operation.	D
10	Test manager, Test designer	Test management and control, Test case prioritisation, Human resources management and motivation.	A
5	Test manager	Product development, Manual test execution and protocol, Coordination of testing activities Product roll-out (= deployment in the productive environment).	C
5	Test designer	Supports test manager in planning activities, Test case design, Manual test execution and protocol.	A
5	Test designer	Test case design, Execution of test cases, Fault localisation, Regression testing.	D
3	Test automation engineer	Manual test execution and protocol, Test automation: implementation of the test automation framework.	A
3	Test manager	Test planning, Manual test execution.	C

Table 5.1 - Participants' characteristics

5.4 Results

In this section, the analysis of the results of the study is presented. First, details on test process characteristics are given. Then, the documentation, communication, and experience characteristics of the analysed test processes are discussed.

5.4.1 Test process characteristics

Decisions on test basis level. The assessment of the testability as well as of the quality of the input documentation is indicated only by about half of the interviewees. These decisions are mostly made by the testing team.

Decisions on test goal level. Only about half of the interviewees cited that the decision on the test foci is performed. This is also the case with the decision on the test intensity. Nearly all interviewees indicate that the decision on test end criteria is made in their organisation.

Only 4 of the interviewees report that all decisions on test goal level are in the testing team's field of responsibility. Three interviewees even indicate that all test goal related decisions are performed by persons not belonging to the testing team. In this case, the decisions are made by the project manager. In all other cases, test goal related decisions are partially made by the testing team.

In nearly all cases, the decision on test intensity is understood as high level effort estimation rather than a thoroughly assigned intensity to different parts of the software depending on criteria like the expected number of defects or on the criticality of the corresponding software entity. In 5 of 6 cases, the high level decision on the test effort is made by the project manager.

Decisions on test approach level. The systematic definition of the test approach is not well established within the analysed testing processes. Only few decisions are made explicitly. 9 interviewees indicate to decide on the ideal test order. In most of the cases, this decision is a high level decision, in which the test levels (e.g. unit test, system test, user acceptance test), as well as the kind of tests (e.g. regression tests, tests of quality attributes like performance or usability tests) is defined. All other decisions are rarely indicated.

Decisions on test design and test realisation level. Decisions on test steps (as part of the test case specification), on test data, and on test sequences are indicated to be made by nearly all interviewees. These decisions are mostly made by the testing team. Within organisations not having an independent testing team, these decisions are performed by developers (where the "tester" is not the developer of that particular part of the software).

Decisions on test run and test evaluation level. All interviewees report to make decisions concerning the success or failure of particular test runs. The test run evaluation is mostly made by testers, in some cases by the whole testing team. The evaluation of a test cycle is only performed by fewer than half of the interviewees.

Figure 5.1 summarises test process characteristics as indicated by the interviewees. The x-axis contains the decisions whereas on the y-axis, the number of interviewees that indicated to make the particular decision explicitly is shown.

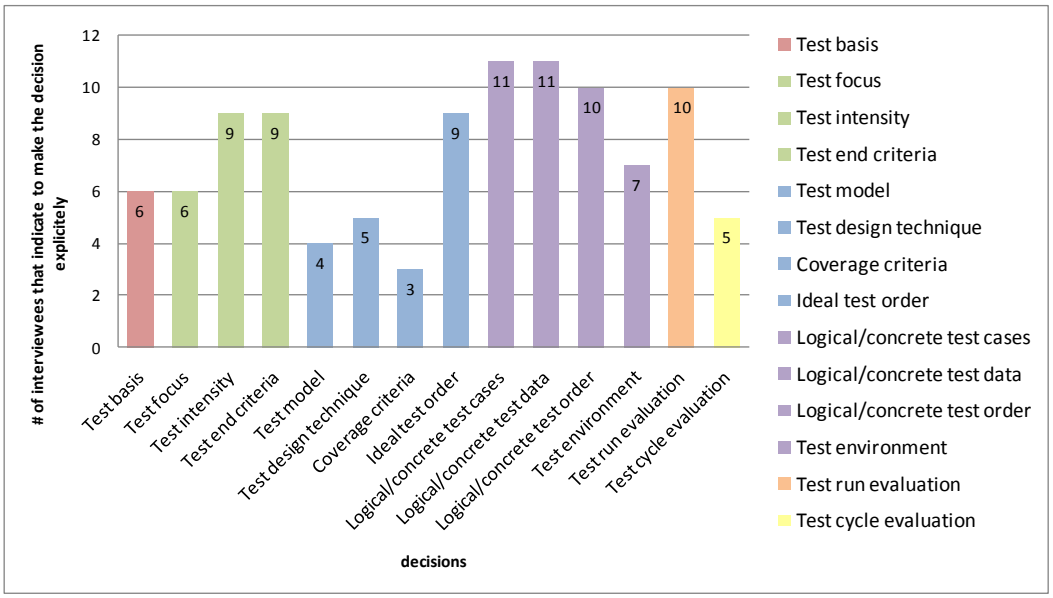


Figure 5.1 - Test process characteristics

5.4.2 Documentation characteristics

The most important documented information sources for testers are past defects and the requirements specification. Figure 5.2 illustrates the documents needed as input during testing as mentioned by the interviewees.

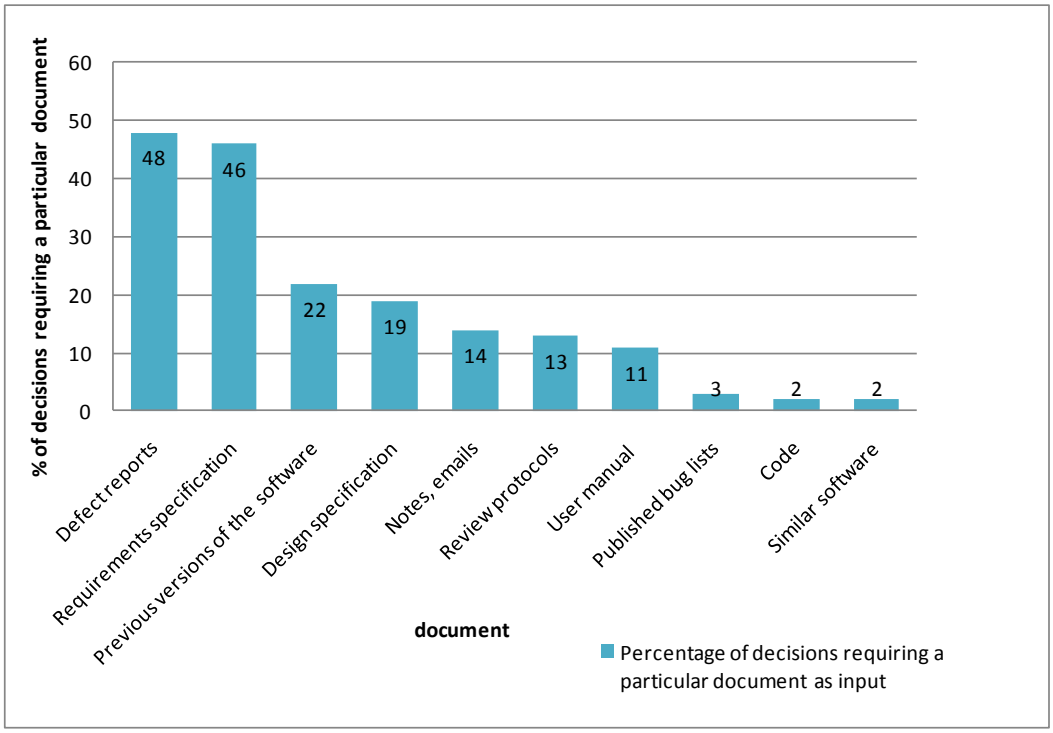


Figure 5.2 - Documentation needs during testing

Learning from defects. Previously found defects are a very valuable information source for testers. For nearly half of the decisions, testers use information on previously found defects. Testers report that previously found defects are good indicators for defects in the software because of following reasons:

(1) Many defects persist across different releases. Two categories of persisting defects are reported by testers: *permanent defects*, that occur across all releases and "*jumping" defects* that regularly "jump" over a constant number of releases.
(2) The correction of a defect introduces more defects.

Knowing potential faulty areas, testers can decide on the test foci. Defects also serve as input for decisions on test design and test realisation level. On the one hand, testers select test cases to be re-executed if they revealed a defect. On the other hand, they develop new test cases on the basis of known defects using the following strategies:

(1) *Intensifying:* Testers investigate the functionality that revealed a failure more intensively and usually vary the test data or the preconditions of the test case.
(2) *Expansion:* Testers search for parts of the software used by the functionality which revealed the defect or for parts of the software that use the faulty functionality.
(3) *Transferring:* Testers search for similar functionality (which could contain the same defect).

The role of the requirements specification. The requirements specification is the most important document for testers. (46% of all decisions need the requirements specification as input; see **Fehler! Verweisquelle konnte nicht gefunden werden.**). On test goal and test approach level, the requirements specification is especially used for decisions concerning the test intensity and the ideal test order, whereas during test design and test realisation, the requirements specification is especially used to decide on test cases (including test data and the logical and concrete test order). In addition, the requirements specification is also used during the evaluation of the test run in two contexts. First, when testers are pressed for time, they report to use the requirements specification as test specification. In this case, decisions on the test design and on test realisation level are made concurrently to the test execution. Second, in case of a failure or of an unexpected behaviour, testers consult the requirements specification in order to analyse if it is actually a defect. All testers emphasise the importance of the requirements specification to be up-to-date and complete.

The role of the user within the testing process. Even though only few of the testers are in direct contact with users of the software they test, the users play an important role during testing. Using documentation produced for and by users, testers can develop more realistic and more relevant test cases. Thus, testers bridge the gap to the customer by using customer problem reports and user manuals in order to develop realistic test scenarios and to define test environments and configurations close to real productive environments. Consequently, this documentation is very valuable when deciding on test data and on

test steps. One interviewee also mentioned to use the user manual to get familiar with the software system.

5.4.3 Communication characteristics

During the testing process, most communication occurs with the requirements engineer and the project manager, followed by the developer. Testers have direct contact with the customer only when the customer is "in-house". Apart from this, there is no direct communication between testers and customers in spite of their request for this type of communication. Figure 5.3 shows communication characteristics of the analysed test processes. The x-axis shows the roles whereas the y-axis indicates the percentage of the decisions made by a particular role.

Most communication is reported to take place when making decisions on test design and on test realisation level. In these cases, the main communication partners mentioned by the interviewees are requirements engineers and project managers. However, when evaluating a test run, there is also a great need for communication, above all, in case of a failure. In this case, the main contact persons are requirements engineers and developers. For decisions on test goal level, communication occurs mostly with the project manager. However, little communication takes place for decisions on test approach level and during test cycle evaluation.

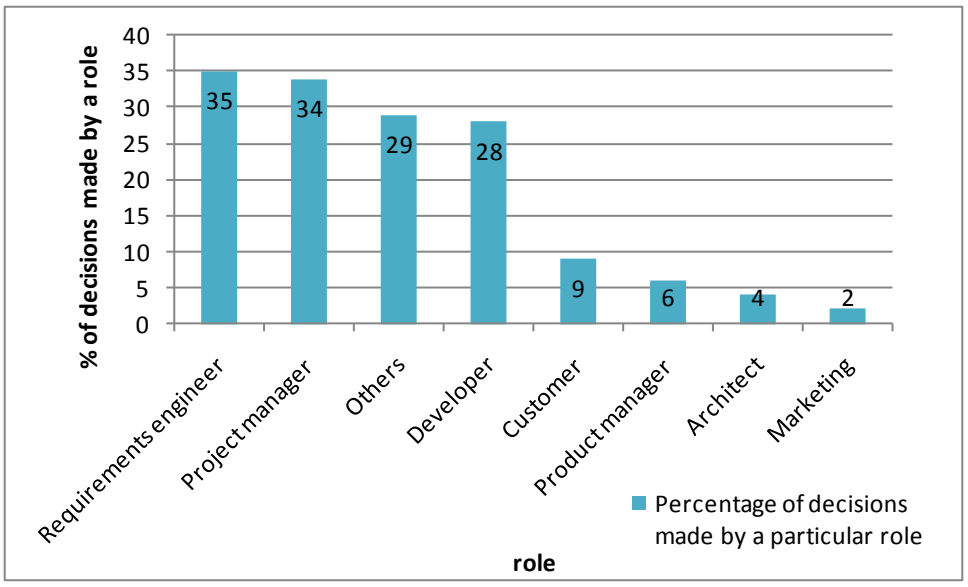


Figure 5.3 - Communication characteristics

5.4.4 Experience characteristics

The interviewees report that most experience is required when making decisions on test goal level and during the evaluation of a test cycle. In addition, a lot of experience is required on test design and test realisation level. In contrast, little experience is required for decisions on test specification, on test approach, and on test run level.

Among the decisions made during the testing process, the definition of test data is stated to be the decision which requires the system specific experience at most. All interviewees indicate that this decision requires very much experience. In addition, this is the only decision which solely requires system specific experience. The definition of the test foci and of the test intensity, as well as the evaluation of the test cycle are also indicated by the interviewees to require high system specific experience.

In general, almost all decisions require more system specific than general experience. The interviewees mention that managerial activities like scheduling, resource planning, and effort estimation require the most general experience.

5.5 Discussion

In the following, the main problems identified in this study as mentioned by the interviewees are discussed.

Testing decisions require system specific experience. Almost all decisions require more system specific than general experience. In addition, testers indicate to rely on their own experience, rather than on experience made by others. For instance, they do not consult published defect lists or other empirical studies.

Testers rely on their own experience more than on test design techniques when making decisions on test approach level. Testers rely more on their own experience than on test design techniques which generate a high amount of test cases and prefer an exploratory-oriented approach. In addition, in case of time pressure, testers deviate from systematic approaches and reduce the set of test cases according to their own experience.

The results of a test cycle cannot be assessed objectively. Surprisingly, testers point out the role of experience in the evaluation of a test cycle. One would expect that the evaluation of the test results requires "only" a decision on the efficiency of the test strategy, i.e. "Have the test design techniques been applied and have the test end criteria been met? Have the test foci been tested with the intended intensity?". But since decisions concerning the test strategy are not well established in testing processes, these decisions have to be made later, namely during the test evaluation.

Poor quality of the documents used as input, especially poor quality of the requirements specification is a major issue during testing. Another problem when making decisions during testing concerns the (poor) quality of the input documents, particularly the lack of quality of the requirements specification. Three participants require more detailed descriptions, particularly concerning pre and post conditions of a requirement, dependencies between requirements, as well as dependencies between the software and its environment (including the software and hardware environment). One of the main reasons for the poor quality of the requirements from the testers' point of view is the lack of involvement in the review process. Only half of the interviewees report that testers are involved in the review process of the requirements specification. In one special case, the requirements specification is not reviewed at all.

High documentation and communication needs during test execution suggest incomplete descriptions of the expected outcome in test case specifications. Reasons for this are either quality deficiencies in the documentation that served as input for decisions on test cases or shortage of time when testers decided on test cases, leading to incomplete descriptions of the expected outcome.

5.6 Implications

Empirical studies are essential in understanding the nature of information processes. This is also the case with the testing process. By this study, previously formulated advices in literature that are not supported by empirical studies could be confirmed. For example, the outstanding role of the requirements specification and of previously found defects for the testing process could be confirmed. This study, however, also allows insights that have not been yet considered in literature, for instance the important role of the user for testers.

This study shows several issues that current research work should address when developing new approaches in the area of software testing.

- **Issue 1:** First, testing requires system specific experience. In addition, testers rely on their experience more than on external facts or information. Thus, testing approaches should consider this.
- **Issue 2:** Test design techniques are not applied "as is" because they generate too much test cases. Approaches to prioritise test cases, and generally approaches that identify the test foci, should consider testers' experience.
- **Issue 3:** Previously found defects are an important information source for testers in order to prioritise testing activities and thus to define the test foci.
- **Issue 4:** Another problem when making testing decisions concerns the evaluation of the outcome of the testing processes. The main reason for this is that testers do not have approaches that allow sound and justified decisions on test goal level. Without having defined the goals of the test, it is very difficult to evaluate whether they have been achieved.
- **Issue 5:** Finally, an issue for testing decisions concerns the (poor) quality of the input documents. Therefore, when defining testing processes in an organisation it should be considered to involve testers in the review process of the test basis.

This thesis addresses the Issue 1 through 4. Issue 5, i.e. the development of high quality documents, is beyond the scope of the thesis and should be addressed by research in the area of e.g. requirements engineering.

5.7 Threats to validity

One threat to validity of this study is the fact that the results may be specific to the particular interviewees. This problem is addressed by selecting very experi-

enced testers for the interviews. Another threat is the ability to generalise the results due to the fact that a small sample has been selected. This issue is addressed by using the following techniques that assure the validity of qualitative studies (Seaman 1999), (Yin 2003):

1) **Diversification**: Diversity with respect to the focus of the activities performed by the interviewees was a key criterion when selecting the participants of the study.
2) **Methodological triangulation**: Different methods to analyse the data have been used (quantitative and qualitative techniques, as described in Section 5.3.2).
3) **Explanatory triangulation** by trying out several explanations for all results in Section 5.4. For example, the result that the requirements specification document is a key information source for testers can be confirmed by several facts. First, asked for main problems in the testing process, almost all interviewees indicate the poor quality of the requirements specification. In addition, asked for required input for different decisions, the interviewees indicate the requirements specification as an important input for almost all decisions. Based on these two facts, the conclusion can be drawn that the requirements specification is an important information source for testers. Nevertheless, organisations with a higher degree of test automation or which use more formal models (e.g. in the embedded area) may show different results.

5.8 Related work

Similar work analysing information gathering strategies of maintainers is described in (Seaman 1999) and in (Tjortjis and Layzell 2001). Most related work focuses on the description of the test process. For instance, the fundamental test process presented in (Spillner and Linz 2010) addresses phases and activities to be passed through when testing a software system. Another group of related work represents test process improvement models like TPI (Test Process Improvement) (Koomen and Pol 1999) or test maturity assessment models like TMMi (Testing Maturity Model Integration), (Tmmi Foundation; eds. van Veenendaal, E. 2009). The focus of these models is not the information flow within the testing process, but the steps for its improvement. None of the references presented above contains empirical studies. The work which is most related to the content of the study presented in this chapter is described in (Dahlstedt 2005). The authors present guidelines for requirements engineering practices that facilitate testing. In contrast to the work in (Dahlstedt 2005) which addresses requirements engineering processes and artefacts, this study has a larger focus including other information sources of the software development project. In addition, this study analyses the role of communication, as well as the role of experience during testing.

5.9 Chapter summary

This chapter details the results of an exploratory study with *expert testers* that has been performed in order to identify characteristics of testing processes in practice along with their strengths and weaknesses (Illes-Seifert and Paech 2008c).

The main results of this study regarding the research questions formulated in Section 5.2 can be summarised as follows:

The requirements specifications as well as defect reports are the documents used most frequently during testing (Question 1). In addition, the requirements engineer and the project manager are roles consulted most frequently by testers (Question 2). Surprisingly, testers mention a high communication overhead during test execution. This fact is an indicator for the poor quality of the requirements specification, confirmed as a major problem during testing by almost all interviewees. Experience plays an important role for testers. The definition of test data as well as decisions on test goal level require by far the most experience (Question 3). At first glance, the latter is unexpected, but since most organisations do not define a test strategy, evaluation is not easy in the absence of operational goals. As expected, test execution requires little experience and is consequently well suited to be automated.

In this study, several issues concerning testing processes in practice have been identified which are largely addressed in this thesis.

CHAPTER 6 An empirical approach to the justified definition of test foci

Decisions in practice are often made based on intuition and subjective appraisal. The "goodness" of a process, method or tool is judged by whether and how many people use it rather than on justified facts (Juristo and Moreno 2001). This also applies to software testing. Testers have to decide which parts of the software to test and how intensively. But these decisions are often not justified by facts and rely on testers' intuition. The empirical approach presented in this chapter proposes a combination of visual analyses and statistical procedures in order to determine indicators for defects in software. Based on these analyses, testers can make justified decisions on test foci (Illes-Seifert and Paech 2010).

6.1 Introduction

The knowledge about particular characteristics of software that are indicators for quality lacks in terms of defects is useful for several roles within the software life cycle. For instance, this knowledge is very valuable for testers, because it helps them to focus the testing effort and to allocate their limited resources appropriately.

Information about the software project can be collected from versioning control (VCS) and defect tracking systems (DTS). These systems contain large amounts of data documenting the evolution of a software product. In practice, this information is often not deeply analysed in order to gain information that facilitates decisions in the present. Information contained in VCSs and DTSs can also be combined. For example, the relationship between historical characteristics (e.g. a file's age that can be determined by analysing the VCS) and software quality (e.g. measured by the defect count that can be determined by analysing the DTS) can be explored. It is very useful to know which particular historical characteristics are good indicators for defects. It helps testers to focus their testing effort appropriately.

The main idea of the empirical approach presented in this chapter is to collect data about the software under test, to analyse it by visual means, and to validate the results by applying statistical procedures (Illes-Seifert and Paech 2010). Based on these analyses, justified decisions can be made. Particularly, the approach uses statistical procedures and visual representations of the data in order to determine those software entities that are responsible for defects in software. For this purpose, *simple analyses of defect variance* are performed in a first step. These analyses explore the relationship between one characteristic of the software (the independent variable) and its defect count. For instance, it can be evaluated whether the software's age is a good indicator for its defect count. Simple analyses of defect variance include visual analyses of the data and statistical procedures that verify the statistical significance of the results obtained by visual analyses.

In a further step of the empirical approach, detailed analyses are performed in order to get more precise results. By *combined analyses of defect variance,* the relationship between several independent variables and a file's defect count is analysed. For instance, an analysis can evaluate whether a file's age *and* the number of changes performed to a file *in combination* are good indicators for defects in software. Similarly to simple analyses, combined analyses of defect variance consist of both visual and statistical procedures.

The advantage of this approach is its applicability in practice. Due to the proposed visual representations, an easy interpretation of the data is possible, thus making this approach an intuitive one. In addition, the approach aims at deriving reliable conclusions from data by requiring statistical tests that support the results derived visually.

The remainder of this chapter is organised as follows. In Section 6.2, the empirical approach is presented in detail, whereas Section 6.3 contains the discussion of the approach. Section 6.4 summarises this chapter.

6.2 Justified test foci definition – An empirical approach

The main assumption of the empirical approach presented in this chapter is that the quality of a decision increases when it is supported by facts rather than relying only on human intuition. Accordingly, the main idea is to collect data about the software under test, to analyse it by visual means and to determine the statistical significance of the results obtained visually by applying statistical procedures. Based on these analyses, justified decisions can be made in practice.

Basically, the approach consists of several steps that can be assigned to one of the phases "planning and design", "data collection", and "data analysis". Starting point of the approach is the definition of the goal that should be achieved by the empirical analysis. This goal has to be detailed in the subsequent phases. Since such an analysis is cost-intensive, it should be clearly stated which benefits are to be expected for each of the stakeholders.

During the **planning and design phase**, the rationale for conducting an analysis is elaborated. This phase includes the definition of how quality will be measured, for instance in terms of the number of defects. In addition, one main goal of this phase is to determine indicators potentially influencing the quality of the software as defined in the step before. Consequently, this phase also includes the definition of measures and corresponding metrics for the identified quality indicators. Finally, the granularity level on which the analyses should be performed has to be defined.

During the **data collection** phase, the software entities to be analysed have to be identified and all activities to prepare the measurement have to be performed. For instance, all necessary tools have to be acquired or developed. Then, the measurement has to be carried out, and the data defined in the steps before have to be collected.

Main goal of the **data analysis** phase is to analyse the data and aggregate the results in order to be able to draw conclusions and to make decisions based on it. In this phase, simple analyses have to be performed in order to evaluate whether the quality indicators proposed in the planning and design phase are actually good indicators for (poor) software quality. Then, detailed analyses have to be carried out in order to refine the results obtained in the step before. Finally, the results of the analyses have to be synthesised in order to be able to draw conclusions and to make justified decision based on the data.

Based on the results of the synthesis, decisions on further process improvements can be made by all stakeholders identified in the goal definition step.

The main steps of the empirical approach are illustrated in Figure 6.1. A detailed discussion of each of the proposed steps is presented subsequently.

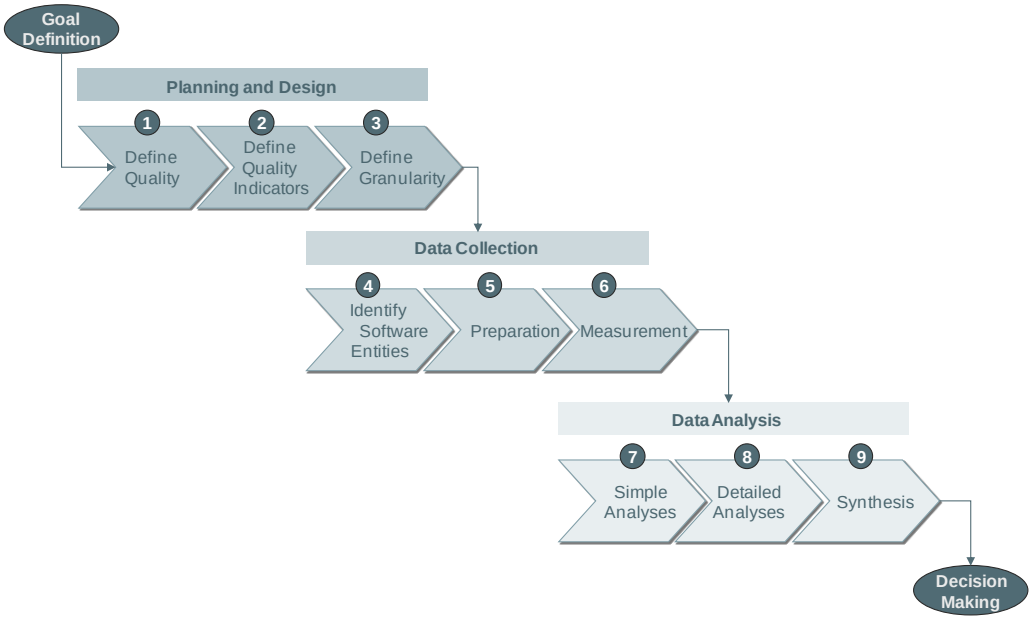


Figure 6.1 - Empirical approach
(Illes-Seifert and Paech 2010)

Goal definition

Since an empirical analysis is time-consuming and expensive, the goals of the analysis have to be clearly stated. In addition, particular benefits for the stakeholders of the analysis should be pointed out. Typical stakeholders for the evaluation of software characteristics influencing its quality (e.g. in terms of its defect count) are the following ones:

- **Testers**: Knowing which particular characteristics of the software lead to more defects, testers can identify these parts of the software and focus their limited resources in testing even those parts.
- **Maintainers**: Information about fault-prone software entities can be used to prioritise maintenance activities.
- **Developers**: Knowing which (bad) characteristics of software lead to defects, the development team can use this information as input to improve the development process.
- **Quality engineers**: Quality engineers can initiate and coordinate the replication of experiments in an organisation-wide context. Thus, an organisation can learn from the past and can guide justified process improvement activities. For instance, if the analysis of several projects shows that the number of authors changing a file is an indicator for defects, it is worthwhile to develop guidelines in which the ownership of code is regulated. In

addition, code-review activities can be focused on those parts of the software that potentially will show defects.

A widely used template for goal definition contains the following elements (Wohlin et al. 2000):

- **Object of study**: Entity that is studied, for instance products, processes, models, metrics, or theories.
- **Purpose**: Intention of the study, for instance to characterise, monitor, evaluate, or predict.
- **Quality focus**: Primary effect under study in the experiment, for instance effectiveness or costs.
- **Perspective:** Viewpoint from which the experiment results are interpreted, for instance from the viewpoint of the developer.
- **Context:** Describes the environment and the circumstances of the study.

Step 1 - Definition of quality

In a first step, the dependent variable of the study has to be defined. The question to be answered in this step is how should quality be expressed, i.e. which measures and corresponding metrics should be used to express (poor) software quality?

The domain of quality metrics is among the most subjective and ambiguous area in the entire literature of software engineering (Jones 2008). Therefore, a critical evaluation of the measures and metrics should be performed. In the empirical studies in which this approach has been applied (Chapter 8 – Chapter 10), quality is expressed in terms of the number of defects that are reported to a file. More detailed analyses that differentiate, for example, between pre-release defects (defects that occurred before release) and post-release defects (defects that occurred after release) as quality measures are also possible. In addition, other quality measures like measures for the maintainability of software[15] can be considered. In this thesis, the focus is on finding indicators for defects in software that allow the selection of test foci.

[15] The (ISO/IEC 2001) standard for software quality defines six quality characteristics and corresponding subcharacterstics for which measures and metrics can be defined.

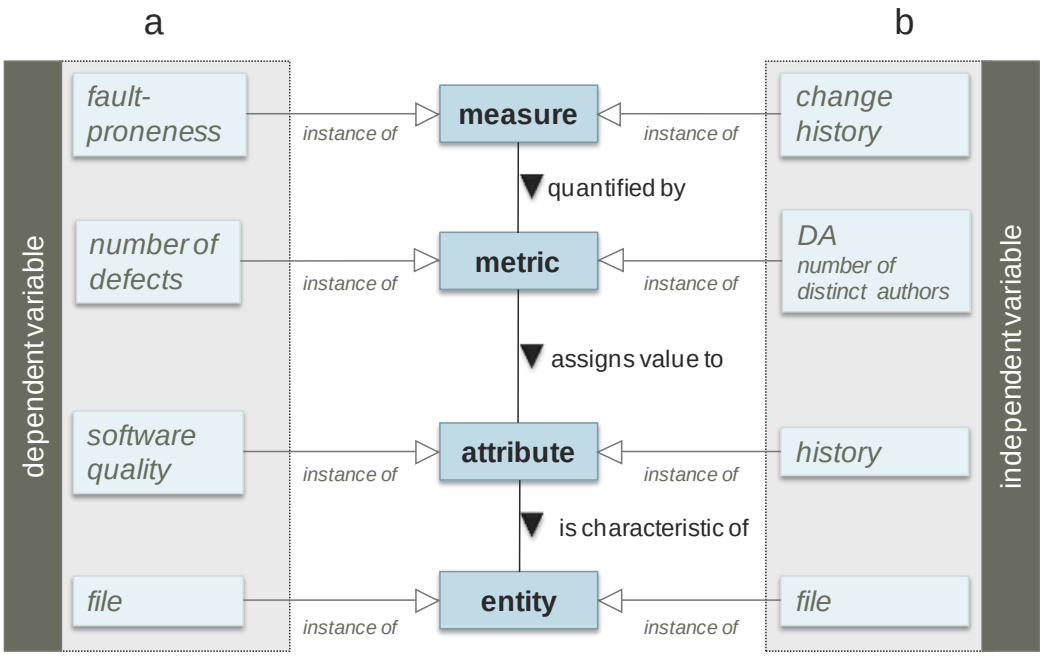


Figure 6.2 - Quality and quality indicators. An example

a) Metrics and measures for the dependent variable *software quality*. Software quality is an attribute of the entity *file*. Software quality can be measured by the *fault-proneness* of a file quantified by the metric *number of defects*. b) Metrics and measures for the variable *history*. History is an attribute of the entity *file* and can be measured by the *change history* of a file that can be quantified by the metric *DA (number of distinct authors that changed that file)*.

Step 2 - Definition of quality indicators

In this step, the independent variable(s) have to be defined. This step includes the formulation of hypotheses on possible indicators for the software's (poor) quality as defined in the step before. Based on these hypotheses, the dependent variables, corresponding measures and quantifying metrics have to be derived. Figure 6.2 b) shows an example for a quality indicator along with corresponding measures and metrics. Accordingly, Figure 6.2 a) shows an example for a possible measure and a corresponding metric for the variable "software quality".

Usually, a lot of metrics can be calculated automatically. Consequently, the selection of the "right" set of variables is not easy. Testers' experience and results from previous analyses can be used as input to define a manageable set of independent variables.

This step also includes the definition of the measurement scale for the variables. The measurement scale determines the operations and statistical procedures that can be applied to the corresponding data (Details on measurement scales are described in Section 2.5.3). Simple and combined analyses of defect variance as proposed in this thesis are performed on categorical data. Numerical data have to be transformed into categorical data.

For all measures and metrics, the following criteria should be considered (Jones 2008), (Ludewig and Lichter 2007):

- **Clear**: The meaning of the information to be collected should be straightforward to developers and managers, so that they can provide accurate and precise answers in little time.
- **Complete but concise**: Only information necessary to collect should be collected.
- **Automated**: If possible, the amount of necessary human intervention should be reduced. Preferable, the collection of metrics should be automated.
- **Non-intrusive**: Data collection should not perturb the software development process.
- **Available**. Data should be available when needed.
- **Repeatable**: Applying the same measurement procedure to measure attributes of a particular entity leads to identical values every time the metric is collected for that entity.
- **Relevant**. The metric should be relevant in the context of the analysis. For instance, if the influence of the software's size on its fault-proneness should be analysed, the LOC metric is a relevant metric to express the software's size. Productivity metrics for the development of that particular software entity are not relevant metrics in *this context*.
- **Economic.** The effort needed to collect the data should not exceed its benefits.

Step 3 - Definition of granularity

In this step, the granularity of the analyses has to be defined. For example, analyses on module, file or package level can be performed. The more fine grained the analyses are, the more precise and differentiated are the results. On the other hand, the more detailed the results are, the higher is the effort needed to synthesise and interpret the results.

Step 4 - Identification of software releases and software entities

In this step, the objects of investigation have to be determined, i.e. all entities for which measurements should be performed have to be identified. The following criteria increase the success and significance of the analyses.

- **Size:** The size of the software or of the analysed components guarantees that the results are statistically significant. A toy project would not lead to statistically significant results.
- **Maturity:** The maturity of software guarantees that effects will have appeared if present.
- **Version controlled source code:** In order to be able to identify different releases of software, the availability of a VCS controlled source code is a prerequisite.

- **Documented history:** The availability of a documented history, mostly in terms of a VCS, is a prerequisite for all analyses concerning the relationship between historical characteristics and software quality.
- **Documented defect history:** In the case that the quality of the software is expressed in terms of the number of defects, the availability of a documented defect history is also indispensable. Usually, the defect history is documented within a DTS.
- **Source code:** The availability of source code is a prerequisite for all analyses concerning the relationship between code characteristics and software quality. In the case that COTS[16] components for which the source code is not available are part of the software to be analysed, structural analyses are difficult.

Step 5 - Preparation

Before any measurement can take place, the instrumentation for the analyses (instrumentation subsumes all instruments needed to perform the analyses) has to be defined, developed or acquired. Tools needed to perform the analyses have to be developed or acquired. Alternatively, existing tools can be adopted for the context of the analyses to be performed. For data collection, tools for extracting information from the VCS and from the DTS, as well as tools for combining information contained in both have to be developed or acquired. In addition, tools for static analysis are needed in the case that the suitability of structural code characteristics as quality indicators has to be analysed. For data analyses, tools supporting statistical analyses will be used. These can be specialised statistic tools or conventional table calculation applications that integrate statistic functionality. In addition, all points of the process at which data should be collected have to be determined along with the persons affected by and responsible for the collection of data.

Personnel have to be familiar with the tools to be used but also with statistical procedures and experimental basics. Thus, training needs have to be identified and the trainings have to be carried out before measurement takes place.

Step 6 - Measurement

Main goal of this step is to collect the data defined in step 1 and step 2 for all identified software entities at the granularity defined in step 3. In addition, collection procedures have to be validated for a randomly selected part of the data.

For instance, in the empirical studies presented in this thesis that have been conducted in the context of open source development (Chapter 8 and Chapter 9), a file's defect count has to be determined retrospectively. For this purpose, information contained in the DTS and the VCS has to be mined and combined. A detailed description of the algorithm is presented in Section 7.7.1. In order to determine and improve the algorithm performance, a validation on a subset of the data should be performed.

---

[16] **C**ommercial **o**ff-**t**he-**s**helf

Step 7 - Simple analyses

In this step, analyses that explore the hypotheses formulated in step 2 have to be performed. In this thesis, two analyses are proposed depending on the level of detail on which the analysis has to be performed: simple and combined analyses of defect variance.

Simple analysis of defect variance

This analysis is used to determine the relationship between one categorial independent variable and a file's defect count, i.e. for the analysis of the variance of the defect count in different categories of the independent variable. A categorial variable classifies the entities according to an attribute (see also Section 2.5.3 for details on types of data). For example, as defined in Chapter 10, the "age" metric of a file classifies the entities with respect to their age into one of the categories: **`Newborn`**, **`Young`** or **`Old`**.

In a first analysis step, the data is displayed in a diagram called *defect variance analysis diagram* (*DVA*). This diagram relates the mean defect count to each of the defined categories as follows: The x-axis contains the category. On the y-axis, the mean defect count in each of these categories is indicated.

For example, Figure 6.3 shows the DVA for the commercial system (CS) used to validate the approach in this thesis (details are presented in Chapter 10). The mean defect count for **`newborn`** files (F-N) is 5.05, for **`young`** files (F-Y) 4.08 and for **`old`** files (F-O) 5.94.

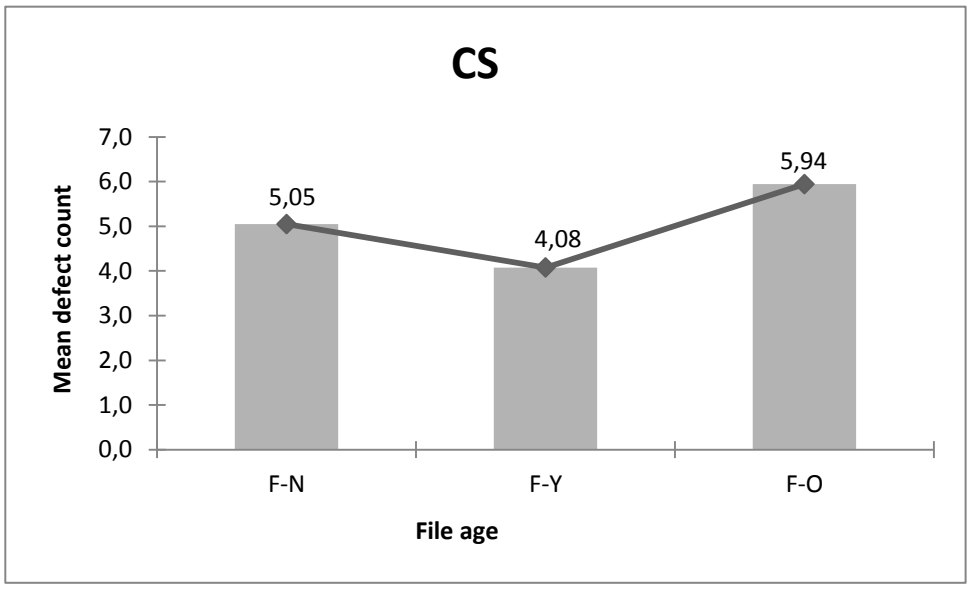


Figure 6.3 - Simple DVA: Mean defect count vs. file age.

In order to obtain *statistical evidence* for the results derived visually, statistical tests have to be performed. The main purpose of the statistical tests is to analyse whether the differences between the groups that have been observed by visual analyses are statistically significant. Two tests can be performed depending on the number of categories of the independent variable:

- **Mann-Whitney test**: Differences between *two* populations can be analysed by the Mann-Whitney test, i.e. if the categorial variable has two categories, the statistical evidence for differences between these two categories with respect to the dependent variable can be analysed (Section 2.5.5).
- **Kruskal-Wallis test**: Differences between *more than two* populations can be analysed by the Kruskal-Wallis test, i.e. if the categorial variable has more than two categories, the statistical evidence for differences between these categories with respect to the dependent variable can be analysed (Section 2.5.5).

Both tests are recommended, because they are non-parametric, i.e. the test does not make any assumptions concerning the distribution of parameters (in contrast to parametric tests). For both tests, the null hypothesis is that the defect count is the same in both/all groups; the alternative hypothesis is that it is not.

In the example, the Kruskal-Wallis non-parametric test has to be applied, because the independent variable "age" defines three categories. In this case, the test is performed in order to analyse whether the differences between **`Newborn`**, **`Young`** and **`Old`** files with respect to their fault-proneness are statistically significant. Based on the DVA, it can be concluded that **`Old`** files are the most fault-prone files because they have the highest defect count followed by **`Newborn`** and **`Young`** files. According to the Kruskal-Wallis test, this observation is statistically significant at the 0.05 level.

Step 8 - Detailed analyses

In order to refine the results obtained by simple analyses, the relationship between two or more independent variables and the dependent variable has to be analysed. Detailed analyses can be performed in order to get more in-depths results. In addition, results that are in contrast to initial expectations (i.e. in contrast to the hypotheses formulated in Step 2) motivate further analyses.

An example for a detailed analysis is the investigation performed for the independent variable "age" (Section 10.5.4). In this case, the simple analysis of defect variance performed in the first step showed unexpected results. In this particular case, **`old`** files proved to be the most fault-prone ones, a result that contradicted the hypotheses formulated in advance. A detailed analysis that examines the relationship between a file's age and the frequency of changes performed to it *in combination* revealed a more precise view. Based on the results of the detailed analyses, it can be concluded that, for instance, **`old`** files that have been changed frequently are the most fault-prone ones. In addition, the detailed analysis reveals that files that have been changed frequently are significantly more fault-prone than files that have not been changed frequently independently of their age.

For detailed analyses, a combined analysis of defect variance is proposed subsequently. Similarly to the simple analysis of defect variance, this analysis combines visual means with statistical procedures.

Combined analysis of defect variance

This analysis is performed in order to explore the relationship between *two* categorial independent variables. If the variables are numerical, a categorisation has to be performed in advance. For instance, files can be divided into two categories with respect to their FC metric (indicating the the number of changes performed to a file) as follows: One group (the stable files) contains all files that have the FC metric lower than average and another group (the unstable files) have the FC value above average. Alternatively, a finer grained categorisation can be defined. But it should be considered that the more detailed a categorisation is, the more time-consuming it is to analyse and interpret the results in practice.

Having two categorial variables, the DVA is used again for the visual analysis. The categories needed for the DVA are obtained by combining the original ones and performing the analysis as described for the simple categorial analysis. For example, an analysis can be performed to determine the extent to which the defect count of a file depends on its age *and* on its stability. Thus, it can be analysed, whether **`old`** files that have been changed frequently (these are **`old`** and **`unstable`** files) are more fault-prone than **`old`** files that have not been frequently changed (**`old`** and **`stable`** files). In this example, the refined categories can be defined as shown in Table 6.1.

		Stability	
		stable	**unstable**
Age	**Newborn**	N-stab	N-unst
	Young	Y-stab	Y-unst
	Old	O-stab	O-unst

Table 6.1 - Category definition matrix for age X stability

The DVA relates the mean defect count to each of the *refined* categories. For instance, for the analysed CS (see Figure 6.4), the mean defect count of **`young`** and **`unstable`** files (**`Y-unst`**) is 7,15.

In order to confirm the results obtained by the visual analysis statistically, the Kruskal-Wallis test has to be applied. Similarly to the simple analysis, the null hypothesis is that there are no differences in the mean defect count among the refined categories, the alternative hypothesis is that there exist differences.

The highest mean defect counts have **`old`** and **`unstable`** files. **`Stable`** files are on average less fault-prone than **`unstable`** files independent from their age. These observations are confirmed by the Kruskal-Wallis test.

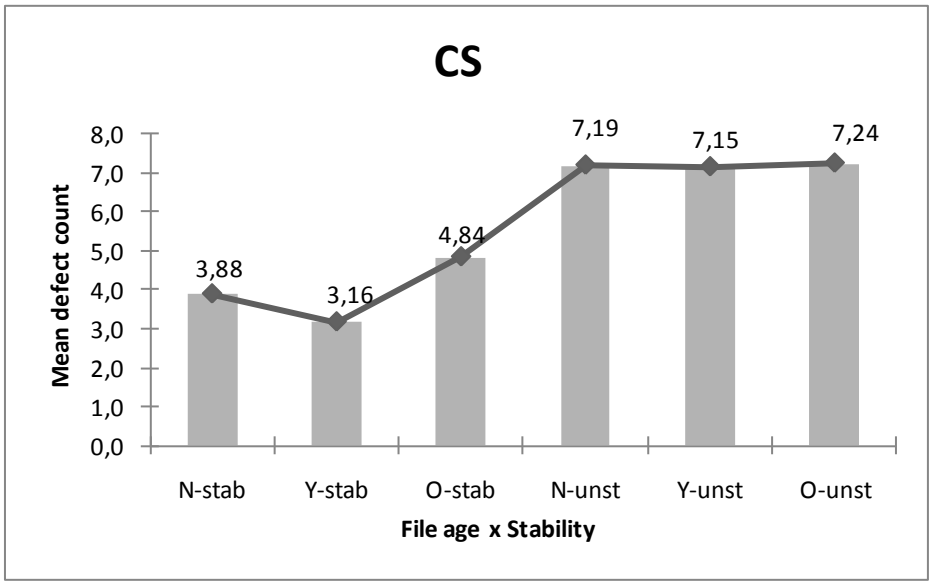


Figure 6.4 - Combined DVA for Ant: Mean defect count vs. file age and stability

Step 9 - Synthesis

In this step, conclusions on the results have to be drawn. The primary goal of analyses in industrial environments is not to validate measures and to build models, but to make the results available to practitioners and to explain interpretations and consequences. Thus, the following questions have to be answered: Which are good indicators for software quality? Which are good indicators for defects in software? Based on the results of the synthesis, it can be decided which measures can be taken to improve the quality and who (i.e. which roles) should be involved in improving it.

The results should be presented in a final report containing recommendations resulting from the results of the analyses that have been performed.

Decision making

Based on the synthesised results and the empirical evidence obtained in the precedent steps, decisions on further process improvements can be made by all stakeholders identified in the goal definition step. For instance, testers can take the results of the analyses as input for the definition of the test foci; quality engineers can decide which organisation-wide insights and process improvement activities can be started, etc.

6.3 Discussion

In this section, characteristics of the approach are reviewed and several aspects to be considered when applying this approach in practice are discussed.

6.3.1 Characteristics of the approach

The approach presented in this chapter has several strengths.

- **Easy to understand**. In order to be applicable in practice, the approach has to be intuitive and easily to understand. For this purpose, the results should not be encrypted within complex formulae but should allow an easy interpretation. For this purpose, visual representations are used to enable a standardised and intuitive interpretation of the results.
- **Basis for justified decisions.** The approach follows statistical procedures. All results obtained by visual means must be validated statistically. Thus, more reliable decisions can be made, because the probability of accidental effects is minimised.
- **Externalises tacit knowledge.** Often, testers have a subjective impression of the factors influencing the software's defects. They know the areas that often lead to "trouble". By following this approach, this tacit knowledge can be justified by statistical means. On the other hand, sometimes, the subjective impression can distort the reality. Thus, this approach helps to minimise subjective distortion.
- **Experience based.** The approach involves testers in the selection of indicators for defects in software. Since usually large amounts of data can be computed, it is essential to select the most appropriate subset of data to be included in further analyses (during the planning and design steps). In addition, testers are involved when deciding which refined analyses should be performed, for instance in order to get more precise results (above all, when performing step 8). Figure 6.5 shows how the generic approach presented in Section 1.2.2 is refined in this chapter.

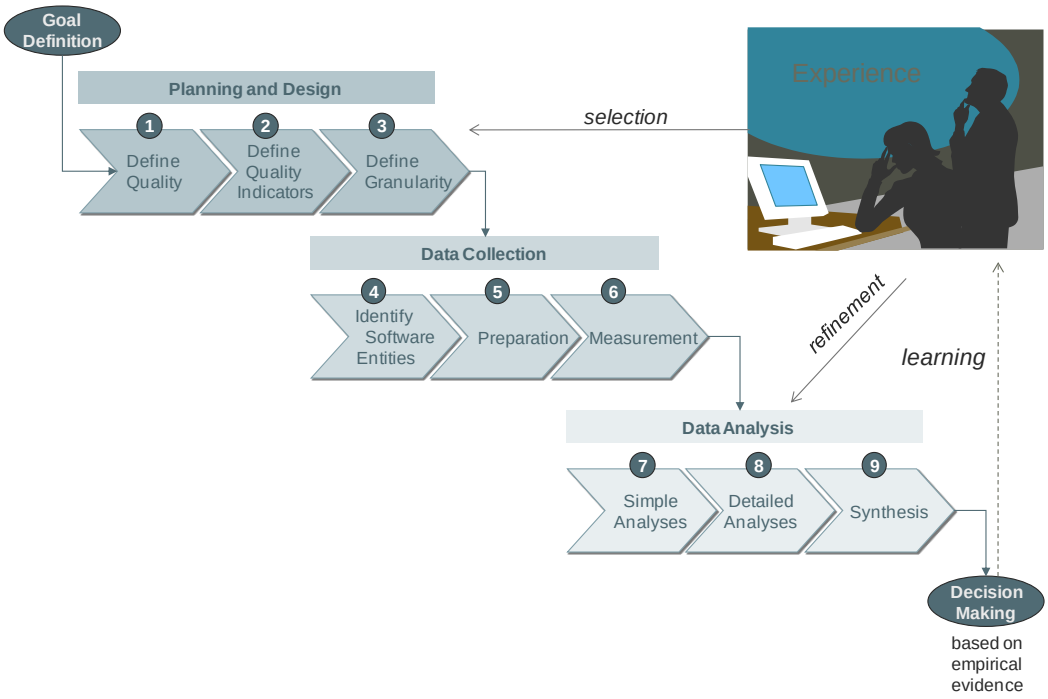


Figure 6.5 - Testers' experience is needed when making decisions on test foci

6.3.2 Social aspects to be considered

Measurement usually takes place in an organisational context. Cultural and social aspects should be considered when starting a measurement program. Thus, measurement should not be applied to judge persons, but to help them. The measurement program should show advantages for all stakeholders. In addition, the transparency of the measurement goals minimises the danger that persons involved do not cooperate in the data collection procedures. Without sensitivity to corporate politics and sociological issues, measurement programs can cause more harm than good and will not achieve their true benefits of revealing areas of strength and weaknesses (Jones 2008). Thus, social aspects of measurement should be considered in advance and persons affected by it should be informed in time.

6.3.3 Other defect types

Care should be taken when measuring software quality in terms of defects in source code. This procedure should not suggest that these are the only defect types that can occur during software development. In fact, it should be clearly stated that there are several other defects types, such as requirements, design or documentation defects that can occur during the software's life cycle that have to be addressed by other complementary approaches.

6.4 Chapter summary

In this chapter, an empirical approach that uses statistical procedures and visual representations of the data in order to determine indicators for the software's quality has been presented. In this thesis, this approach is particularly used to determine indicators for defects in software. The main goal of this approach is to provide assistance to several roles in the software life cycle, for instance to testers, developers, or maintainers in making *justified decisions* based on data (Illes-Seifert and Paech 2010).

After establishing clear goals for empirical analyses to be performed in order to obtain a justified set of indicators for defects in software, several other planning activities have to be performed. These activities include the definition of the granularity on which analyses will be performed (e.g. on file level) and how quality should be expressed (e.g. in terms of a file's fault-proneness expressed by its defect count). In addition, potential indicators for defects in software have to be defined (e.g. structural code characteristics like the size). The measurement activity in which data are collected is then followed by several analyses. For first exploratory analyses, simple analyses of defect variance are proposed. The visualisation occurs in terms of a DVA (defect variance analysis diagram). Detailed analyses are performed in order to get more precise results. By combined analyses of defect variance, the relationship between more indicators and the software's defects is analysed. The last two steps of the approach aim to synthesise the results in order to serve as the basis for a justified decision on test foci and on process improvements.

CHAPTER 7 Basic experimental design

The empirical approach presented in Chapter 6 proposes a combination of visual representations and statistical procedures that help to make justified decisions in practice. In order to validate this approach, several empirical studies are performed. One part of the empirical studies is performed in the context of open source development and another part in an industrial setting. In this chapter, basic information on the empirical studies performed in the context of open source development is presented.

7.1 Introduction

In order to validate the approach presented in Chapter 6, a series of empirical studies are conducted in the context of open source development. For this purpose, seven java open source programs are analysed. In this chapter, details on the context of the empirical studies described in Chapter 8, Chapter 9 will be presented. In addition, the first steps of the approach that are common to all studies are detailed.

The main goal of all empirical studies presented in this thesis is to explore the relationship between characteristics of software and its quality in terms of its defect count. As required by the approach presented in Chapter 6, this generic goal has to be detailed in a first step. The concretisation of the generic goal is shown in Section 7.3. The definition of the granularity of the analysed entities follows in Section 7.4. The dependent and independent variables are presented in Section 7.5. Characteristics of subject open source programs (OSPs) that are analysed are presented in Section 7.6.

A prerequisite of all empirical studies is the computation of the number of defects reported per file. Based on this information, statistical analyses can be performed. For instance, the relationship between the number of defects and other characteristics of files can be analysed.

Usually, defect tracking systems (DTS) do not contain any information related to the location of the defects. Similarly, in versioning control systems (VSC) it is not possible to distinguish between records that have been introduced due to defect correction or due to changes (e.g. by adding new functionality). Consequently, information contained in both VCS and DTS, has to be combined in order to compute the number of defects per file. Usually, VCSs and DTSs do not provide support for the combination of both data sources. In order to overcome this problem, an algorithm is presented that relates the information contained in both systems and computes the number of defects per file. This algorithm and its empirical validation are presented in Section 7.7. Several methods for determining the number of defects per file have been presented in literature. An overview and a discussion about the advantages and drawbacks of each of these methods are presented in Section 7.8.

7.2 Basic terms and concepts

In this section, basic terms and concepts used in all empirical studies are presented.

Definition 7.1 Versioning Control System (VCS)

Versioning Control Systems (VCS) are useful for recording the history of documents edited by several developers. In order to edit a file, a developer has to check out this file, edit it and to commit this file back into the VCS repository. Each time a developer commits a file a message describing what has been changed can be optionally added. CVS[17], ClearCase[18], SourceSafe[19] and SVN[20] are examples for such systems.

Definition 7.2 Defect Tracking System (DTS)

Defect Tracking Systems (DTS) facilitate the recording and status tracking of defects and changes. They often have workflow-oriented facilities to track and control the allocation, correction and re-testing of defects and provide reporting facilities (ISTQB 2007).

Definition 7.3 History Touch (HT)

A *history touch (HT)* is defined as one of the commit actions where changes made by developers are submitted. These changes include modifying, adding or removing files. *Defect-correcting HTs* subsume all those HTs that have been recorded when developers corrected a defect. Accordingly, *non-defect-correcting HTs* subsume all other HTs submitted, for instance in the case that a developer has introduced new functionality or in case of perfective or adaptive maintenance activities.

Definition 7.4 Birth

The *birth* of a file denotes the point of time of its first occurrence in the VCS, i.e. the date, the file has been added to the VCS.

Definition 7.5 Death

The *death* of a file denotes the point of time of its removal from the VCS.

Definition 7.6 Present

Present denotes the point in time where the empirical studies started. All defects and HTs recorded until Present were considered.

---

[17] http://www.nongnu.org/cvs/

[18] http://www-306.ibm.com/software/awdtools/clearcase/

[19] http://www.microsoft.com/ssafe/

[20] http://subversion.tigris.org/

Definition 7.7 System Age

The *system age* is computed as `Present - Birth` of the "oldest" file contained in the VCS.

Definition 7.8 History

The *history* of a file subsumes all HTs that occurred to that file from its birth until present or until its death.

Definition 7.9 Release

A *release* represents a point in time in the history of a project which denotes that a new or upgraded release is available. In the empirical studies presented in this thesis, only final or major releases of the open source programs have been considered.

Definition 7.10 Defect Count

The *defect count* is the number of defects identified in a software entity. In this thesis, for all open source programs, the number of defects of a file is counted.

7.3 Goal definition

The overall goal of all empirical studies is to evaluate which software characteristics are indicators for defects. Using the goal definition template described in Chapter 6.2, the goal can be detailed as follows:

- *Object of study*: Analyse different software characteristics
- *Purpose*: for the purpose of their evaluation
- *Quality focus:* with respect to the efficiency as indicators for the software's defects
- *Perspective*: from the point of view of researchers, testers, and maintainers
- *Context*: in the context of open source development.

7.4 Granularity

Since HTs are performed on file level, the empirical studies presented in Chapter 8 and Chapter 9 are performed on file level, i.e. characteristics of a file are related to the file's defect count. The analysis whether bad smells are good indicators for defects in software is performed on class and on package level.

7.5 Defining quality and quality indicators

In these both steps of the approach, the dependent variable (the software's quality) and the independent variables (quality indicators) have to be defined.

In all empirical studies, the dependent variable is the defect count of a file that occurred between two consecutive releases during its history. Thus, D_{CURRi} de-

notes the number of defects reported for a file *after* release i and *before* release i+1.

The independent variables are structural characteristics of the software that are supposed to indicate defects.

Figure 7.1 illustrates how file characteristics are related to corresponding defect counts for particular releases. A characteristic j in release i of a file f (e.g. the number of changes performed to that file) is related to the defect count reported to that file between release i and release i+1.

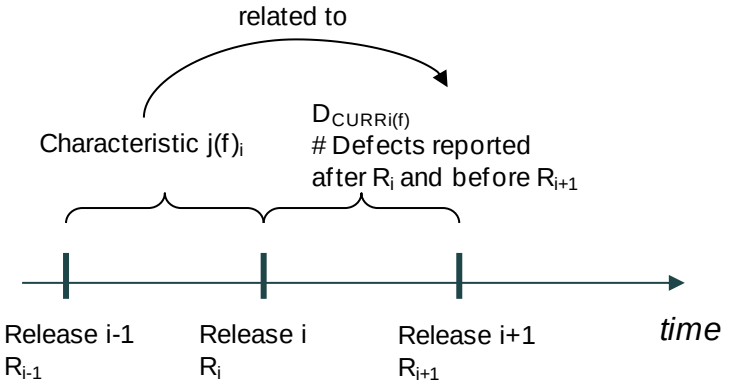


Figure 7.1 - Defect count and characteristics of a file

7.6 Software entities

In this step, the objects of investigation are identified, i.e. all OSPs are determined for which the analysis has to be performed. For this purpose, large repositories for open source programs, mainly SourceForge[21] and Java-Source[22] are searched. As required in Chapter 6, following criteria have been applied when selecting the projects:

a) *size:* The project is of a large size in order to permit significant results. This criterion guarantees that the empirical results are statistically significant.
b) *maturity*: According to this criterion, projects with a number of HTs in the VCS greater than 50.000 have been selected.
c) *Version controlled source code*. In order to extract historical characteristics automatically, only projects for which a well documented history within a VCS have been selected.
d) *Documented history*. For each HT, at least the following information has to be available: author, date, and message.
e) *Documented defect history*: The availability of a DTS is a prerequisite for a project to be considered in the empirical study.

21 http://sourceforge.net/
22 http://java-source.net/

f) *Source code*. Since some empirical studies analyse the relationship between structural code characteristics and defects, the availability of the source code is a criterion for an OSP to be included into the study. In addition, only projects written in Java have been considered. For comparability and generalisability of the results a single programming language has been chosen.

OSCache, a project that does not fulfil the criteria defined above, is included in order to compare the results obtained for all other projects with a smaller, but mature project. This project exists since 2000.

As a result of the search, the following OSPs are used in all empirical studies.

- **Apache Ant** (Ant)[23] (Apache Ant) is a Java application for automating the build process using an XML file where the build process as well as its dependencies can be described.
- **Apache Formatting Objects Processor** (Apache FOP)[24] is a Java application that reads a formatting object (FO) tree and renders the resulting pages to a specified output. Output formats are for example PDF, PS, XML, or PNG.
- **Chemistry Development Kit** (CDK)[25] is a Java library for bio- and chemoinformatics and computational chemistry.
- **Freenet**[26] is a distributed anonymous information storage and retrieval system. Users can use Freenet for instance for publishing websites, communicating via message boards, or for sending emails.
- **Jmol**[27] is a Java molecular viewer for three-dimensional chemical structures. Features include: reading a variety of file types and output from quantum chemistry programs as well as animation of multi-frame files and computed normal modes from quantum programs.
- **OSCache**[28] is a Java application which allows performing fine grained dynamic caching of JSP content, servlet responses, or arbitrary objects.
- **TV-Browser** [29] is a Java based TV guide.

Table 7.1 summarises the attributes of the analysed projects. A * behind the data in the column "Project since" denotes the date of the registration of the project in SourceForge[30]. For the rest, the year of the first HT in the versioning system is indicated. The column "OSP" contains the name of the project followed by the project's latest release for which the metrics "LOC" (Lines of Code) and the number of files have been computed (indicated in the columns 5 and 6). The 3rd and the 4th columns contain the number of defects registered in the DTS and the number of HTs extracted from the VCS. The column "# Analysed re-

23 http://ant.apache.org/
24 http://xmlgraphics.apache.org/fop/index.html
25 http://sourceforge.net/projects/cdk/
26 http://freenetproject.org/whatis.html
27 http://jmol.sourceforge.net/
28 http://www.opensymphony.com/oscache/
29 http://www.tvbrowser.org/
30 http://sourceforge.net/

leases" indicates the number of subsequent releases of the corresponding program that have been analysed, whereas the last column indicates the mean time interval between two consecutive releases.

OSP	Project since	Defect count	# HTs	LOC	# Files	# Analysed releases	Mean time interval between releases (in years)
1. Ant *(1.7.0)*	2000	4.804	62.763	234.253	725	3	1,8
2. FOP *(0.94)*	2002*	1.478	30.772	180.103	902	2	2,8
3. CDK *(1.01)*	2001*	602	55.757	227.037	1.038	3	1,1
4. Freenet *(0.7)*	1999*	1.598	53.887	68.238	464	3	1,5
5. Jmol *(11.2)*	2001*	421	39.981	117.732	332	3	0,9
6. OsCache *(2.4.1)*	2000	2.365	1.433	19.702	113	3	
4. TV-Browser *(2.6)*	2003	190	38.431	169.831	827	3	1,0

Table 7.1 - Subject programs

7.7 Preparation

In this step, the instrumentation of the studies has to be prepared.

In order to analyse the relationship between the defect count and characteristics of files, the defect count per file has to be computed. DTSs contain information on the defects recorded during the lifetime of a project, amongst others the defect ID and additional, detailed information on the defect. But DTSs usually do not give any information on which files are affected by the defect. Therefore, information contained in VCSs has to be analysed and combined with information contained in DTSs. In the following, the procedures and algorithms for extracting the data necessary for the empirical studies are described.

7.7.1 Computing the number of defects per file in OSPs

For this purpose, the information contained in the VCS is extracted into a history table in a data base. Additionally, the defects of the corresponding project are extracted into a defect table of the same data base. Then, a 3-level algorithm is used to determine the defect count per file. At each level, a particular search strategy is applied (Illes-Seifert and Paech 2010).

- **Direct search:** First, a search for messages in the history table containing defect-IDs of the defect table is performed. Messages containing the defect-ID and a text pattern like "fixed" or "removed", are indicators for defects that have been removed. In this case, the number of defects of the corresponding file has to be increased.
- **Keyword search:** In the second step, a search for keywords like "defect fixed" or "problem fixed" within the messages which have not been investigated in the step before is performed.
- **Multi-defects keyword search:** In the last step, a search for keywords which give some hints that more than one defect has been removed (e.g. „two defects fixed") is performed. In this case, the number of defects is increased accordingly.

Figure 7.2 visualises the procedure that is used to compute the defect count per file.

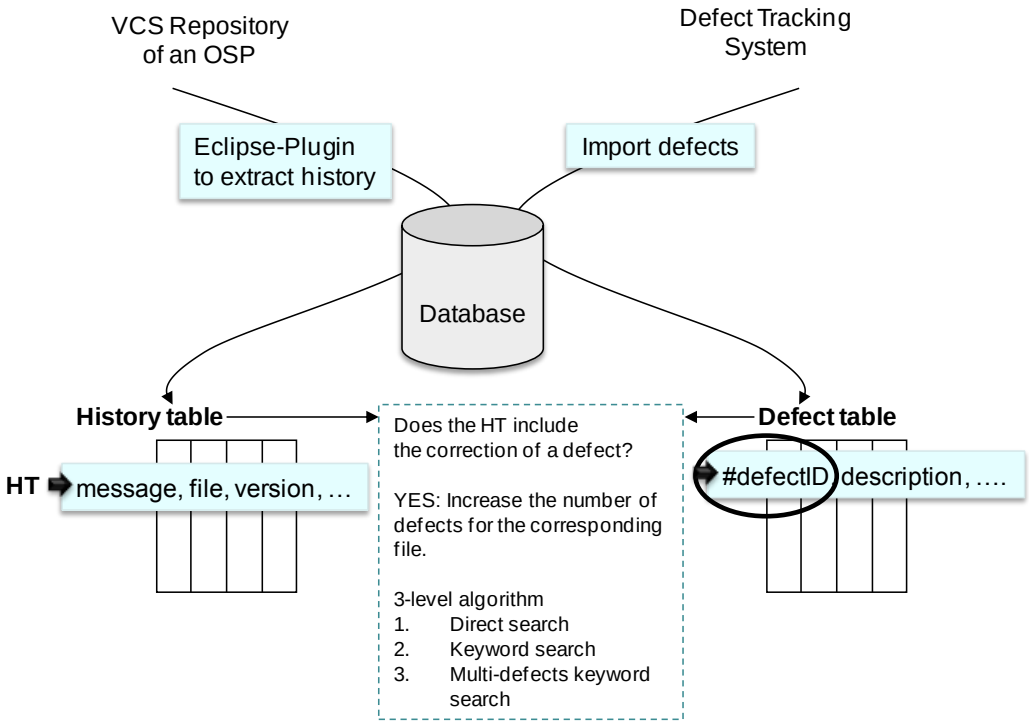


Figure 7.2 - Computing the defect count for files in OSPs

7.7.2 Keyword definition and validation

The definition and validation of keywords is an iterative process consisting of the validation of the direct search, the validation of existing keywords and the search for missing keyword patterns.

Validation of the direct search. The first validation step consists of the analysis whether the HTs found by direct search actually contain an indication that a defect has been corrected. For this purpose, 20% of all HTs found by the first algorithm step have been validated manually. Almost all messaged found in this step (above 99% in all projects) have been classified correctly by the algorithm. One reason for this is that the HT messages are simple, using standard phrases like:

"Bugfix #<BUG-ID>: <What has been done.>"
"Fixed bug related to PR: <Problem Report-ID> submitted by <Submitter>"
"A fix to … PR: <Problem Report-ID> submitted by <Submitter>"
"A bug in …. Bugzilla report <BUGID> submitted by <Submitter>"
"Correction of <What has been corrected>, #<BUGID> …. "
"Fix problem …. #<BUGID> submitted by <Submitter>"
"Fix for #<BUGID>"
"Fix problem with …. #<BUGID>"

Validation of existing keywords. The main goal of this step is to determine whether the HTs identified by the second and third level of the algorithm actually contain an indication that a defect has been corrected. If this is not the case, the corresponding keyword may be too general, ambiguous or incorrect and must be either refined or removed. A total of 10% of the HTs found by the algorithm have been selected randomly and validated in such a way. Incorrect patterns have been removed and ambiguous ones refined.

Searching for missing keyword patterns. The main goal of this validation step is to identify keyword patterns not included in the search so far. For this purpose, HTs containing weak keywords like "fix" or "problem" have been analysed in order to determine missing complex patterns like "error fixed" or "problem corrected".

Finally, HTs that have not been selected by any of the levels of the algorithm have been analysed in order to determine whether some keywords are missing. For each project, 100-200 HTs have been selected randomly and investigated for additional keywords. Only in case of the OSCache project, one additional keyword was found.

7.7.3 Algorithm performance

Formally, determining whether a HT is defect correcting (dc_HT) or not (ndc_HT) is a classification problem. Accordingly, each HT is mapped to one of the element of the set {positive = (dc_HT), negative = (ndc_HT)}. The algorithm represents a classification model that predicts whether an instance is positive or negative. Given a HT, there are four possibilities:

- *true positive* (TP): This is the case if the HT is positive (= dc_HT) and it is classified as positive by the algorithm.
- *false negative* (FN): The HT is positive but classified as negative (= ndc_HT).
- *true negative* (TN): The HT is classified as negative and it is actually negative.
- *false positive* (FP): The HT is actually negative but classified as positive.

In order to determine the overall performance of the algorithm presented in Section 7.7.1, three analyses have been performed: true-positives analysis, anti-pattern analysis, and the overall performance analysis.

For the **true-positives analysis** 10% of all HTs found by the algorithm have been randomly selected and analysed whether the HTs have been correctly classified as dc_HTs. Table 7.2 summarises the results of this analysis. For each project, the percentage of correctly classified dc_HTs (true positives) is indicated. The results show a high classification accuracy with respect to the correctly classified dc_HTs that ranges from 97% to 99%.

	Project	% of correctly classified dc_HTs
1	FOP	0.993
2	ANT	0.974
3	CDK	0.987
4	Freenet	0.997
5	Jmol	0.998
6	OSCache	0.999
7	TVBrowser	0.995

Table 7.2 - Algorithm performance
Percentage of correctly classified HTs out of 10% of all HTs found by the algorithm

For the **anti-pattern analysis**, a set of keyword "anti-patterns" has been defined that indicate a non-defect correcting HT, for instance "initial revision", "refactoring", or "removed warnings". Then, the intersection set of both has been computed: the set of non-defect correcting HTs and the set of defect correcting HTs. All HTs that lie in the intersection can be a sign for an erroneous classification. Table 7.3 shows the results of this analysis. For each project, the percentage of HTs lying in the intersection set is indicated (relative to the total number of dc_HTs identified by the algorithm). The last column indicates the percentage of correctly classified dc_HTs in the intersection set.

	Project	% of the number of HTs in the intersection relative to the number of dc_HTs	Classification accuracy
1	ANT	0.03	0.947
2	FOP	0.06	0.941
3	CDK	0.03	0.947
4	Freenet	0.29	0.994
5	Jmol	0.01	1.000
6	OSCache	0.21	1.000
7	TVBrowser	0.03	0.900

Table 7.3 - Algorithm performance
Antipattern analysis results

This analysis underlines the results obtained by the true positives analysis. The classification accuracy with respect to the correctly classified dc_HTs in the intersection set ranges from 90% to 100%.

In order to evaluate the **overall performance** of the algorithm, 1000 HTs have been randomly selected in each project and analysed whether they were TP, TN, FP, or FN. Then, precision and accuracy have been computed.

The *precision* can be calculated as:

$$\texttt{precision} = \frac{\texttt{TP}}{\texttt{TP} + \texttt{FP}}$$

Thus, the precision indicates the probability that the HT is actually "positive" when the algorithm computes this.

The overall accuracy of the algorithm can be calculated as:

$$\texttt{accuracy} = \frac{\texttt{TP} + \texttt{TN}}{\texttt{TP} + \texttt{TN} + \texttt{FP} + \texttt{FN}}$$

Table 7.4 summarises the precision and accuracy for all projects. The precision of the algorithm is high across all projects. It ranges from 0.917 to 0.985. Thus, it is very probable that a HT is actually positive if this is determined by the algorithm. The overall accuracy is also high and ranges from 0.958 to 0.989. In six cases, the overall accuracy is higher than the precision. In two other cases, both values (precision and accuracy) are quite similar.

Project		Precision	Accuracy
1	ANT	0.945	0.979
2	FOP	0.985	0.983
3	CDK	0.917	0.968
4	Freenet	0.977	0.958
5	Jmol	0.968	0.981
6	OSCache	0.964	0.972
7	TVBrowser	0.969	0.970
	MIN	**0.985**	**0.989**
	MAX	**0.917**	**0.958**

Table 7.4 - Algorithm performance
Overall performance

7.7.4 Defect correction density

At average, 14% of all HTs are defect-correcting HTs. The maximum is 25.7% in case of Freenet and the minimum is 2.9% in case of TVBrowser. Figure 7.3 illustrates for each OSP the percentage of defect-correcting HTs (these are messages that have been found in one of the steps of the algorithm presented in Section 7.7.1). Consequently, most of the HTs have another cause than defect correction (e.g. initial check-in, perfective or adaptive maintenance, etc.).

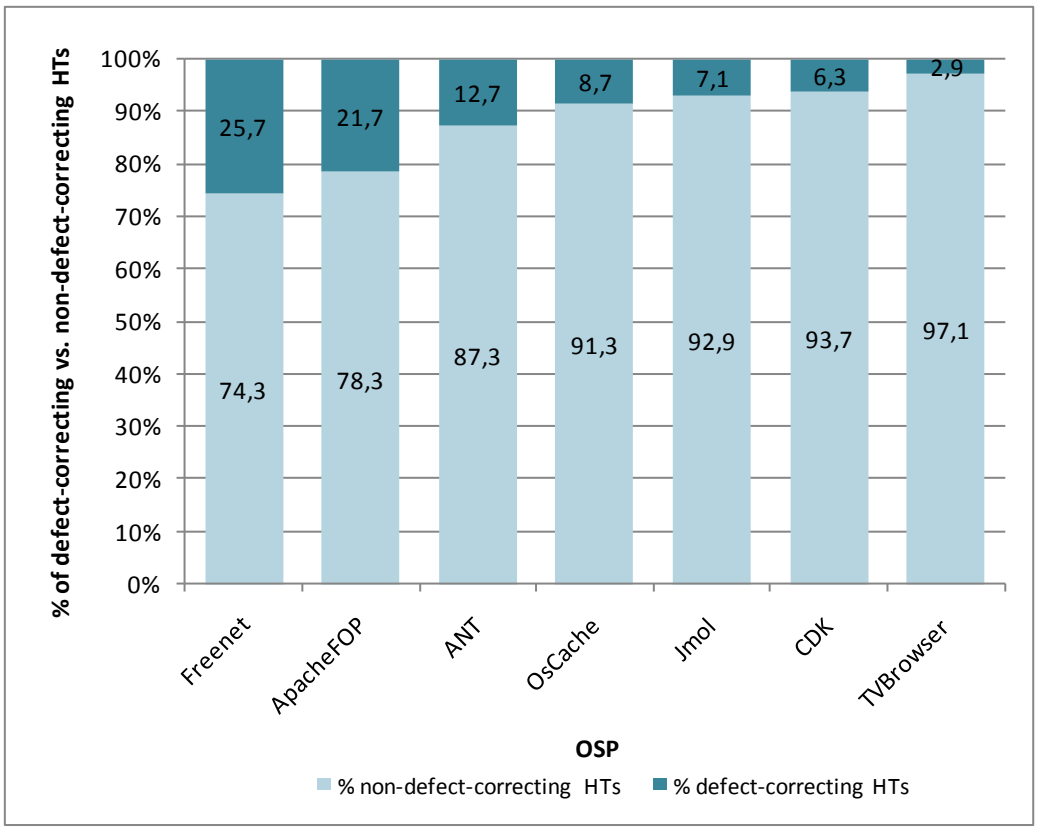


Figure 7.3 - Defect correction density in HTs

In almost all examined programs, the percentage of HTs that are found by direct search makes up the biggest part of all HTs found by any level of the algorithm. For example, in case of Freenet, 19.7% of all HTs (in the history table) contain a reference to a defect ID in the defect table, 1.2% of all messages contain one of the keywords found by direct search, and 4.8% of all messages contain keywords that indicate that more than one defect has been corrected (found by the multi-defect keyword search). Figure 7.4 illustrates for each OSP the percentage of defect correcting HTs per each level of the algorithm (direct search, keyword search, multi-defects keyword search).

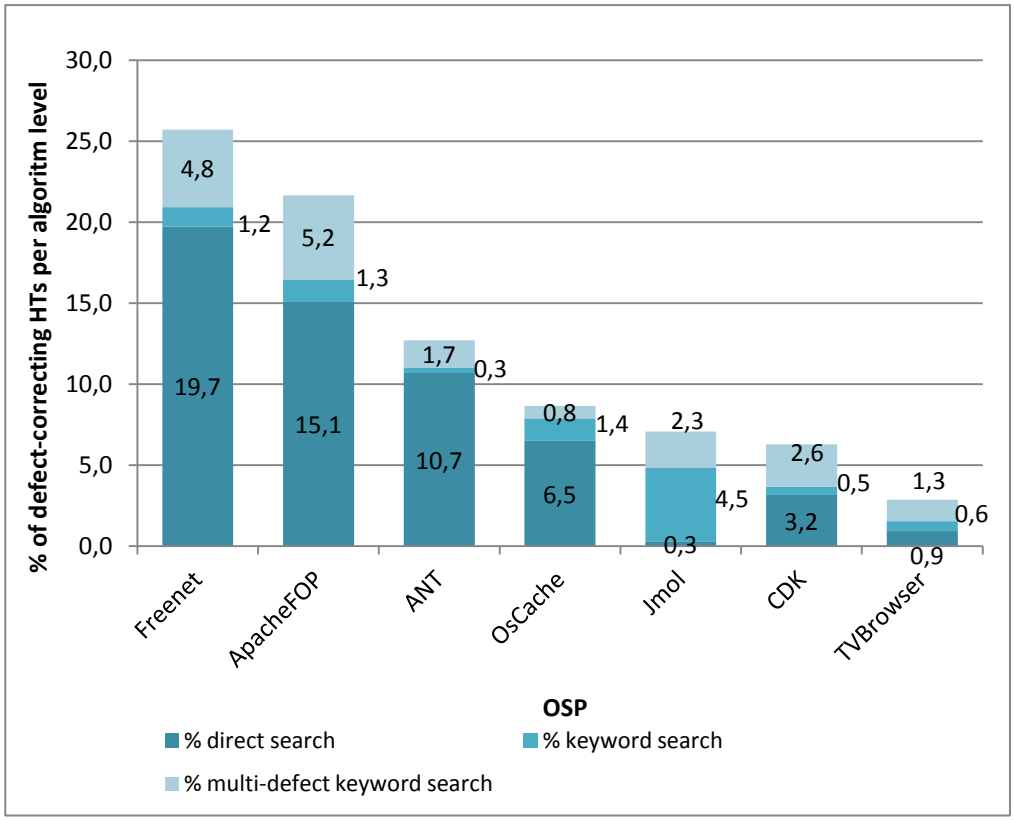


Figure 7.4 - Defect detection per algorithm level

7.7.5 Threats to validity

One threat to validity is that not all developers deliver meaningful messages when they check-in files. Developers, for example, can also check-in files without specifying any reason, even though they had corrected a defect. Thus, the defect count of a file can be higher than the defect count computed by the algorithm. This concern is alleviated by the size of the analysed OSPs.

Another threat to validity is the problem of collective check-ins. Collective check-ins denote check-ins where a set of files is checked in after a developer has removed two or more defects. Suppose that a developer has removed *defect1* in the files A and B and *defect2* in the files B and C. Then, the developer checks in the files A, B, and C with the HT message *"… two defects removed…"* The algorithm presented in Section 7.7.1 would increase the defect count of each of the files A, B and C by two, instead of increasing the defect count in file

A and C by 1 and only of file B by two. Example 7.1 shows an original check-in in the program Ant that contains references to 7 defect IDs. It is not clear if the correction of a defect affected all files that have been checked in conjointly.

Example 7.1 – Collective check-in in Ant. *"Fix label length issues Other fixes unearthed after major refactoring of VSS tasks PR: #11562 #8451 #4387 #12793 #14174 #13532 #14463 Submitted by…"*

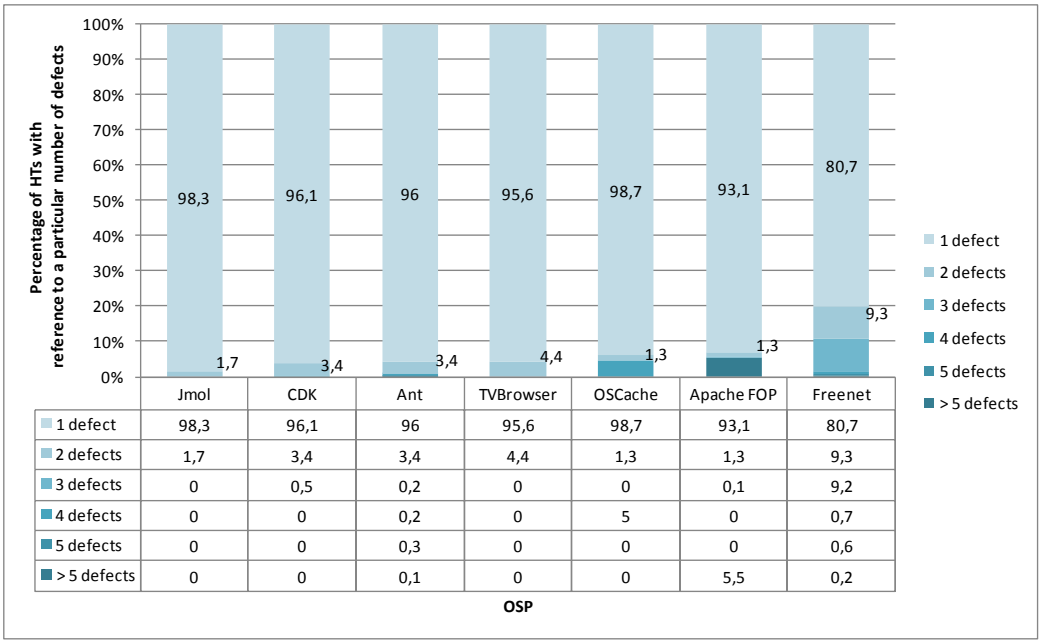
Thus, collective check-ins are a threat to validity and can lead to imprecision in the defect count. The assumption is that such messages are uniformly distributed among all developers and files. Additionally, the average defect count per HT is low in all projects. This fact diminishes the threat to validity.

The average defect count per HT ranges from 1.02 (in case of the Jmol program) to 1.57 (in case of the ApacheFOP program). In case of two programs (Jmol and TVBrowser), the maximum defect count per HT is only 2. Only in case of two programs, Freenet and ApacheFOP, the maximum defect count per HT is above 10. Table 7.5 summarises the average, maximum and the minimum defect count per HT for all programs.

ID	OSP	Average defect count per HT	Maximum defect count per HT	Minimum defect count per HT
1	Ant	1.06	7	1
2	FOP	**1.57**	11	1
3	CDK	1.04	3	1
4	Freenet	1.34	19	1
5	Jmol	**1.02**	2	1
6	OsCache	1.16	4	1
7	TV-Browser	1.04	2	1

Table 7.5 - Average, maximum and minimum defect count per HT

In almost all programs, above 90% of the HTs contain references to only a single defect. Only in case of Freenet, 80.7% of the HTs contain only a single defect. A very low percentage of the HTs contain 2 defects. Apart from ApacheFOP and OSCache, nearly zero percent of the HTs contain more than 3 defects. Figure 7.5 shows the percentage of HTs for each OSP for different defect counts per HT.



	Jmol	CDK	Ant	TVBrowser	OSCache	Apache FOP	Freenet
1 defect	98,3	96,1	96	95,6	98,7	93,1	80,7
2 defects	1,7	3,4	3,4	4,4	1,3	1,3	9,3
3 defects	0	0,5	0,2	0	0	0,1	9,2
4 defects	0	0	0,2	0	5	0	0,7
5 defects	0	0	0,3	0	0	0	0,6
> 5 defects	0	0	0,1	0	0	5,5	0,2

Figure 7.5 - Percentage of HTs for different defect counts per HT[31]

7.8 Related work

A key problem when doing defect prediction is to determine the number of defects that occurred in software entities. In open source as well as in commercial VCSs, there is usually no possibility to distinguish between a HT that reports a defect and a HT that reports any other change performed to the software. There are several methods for assigning the type (defect-correcting, non-defect-correcting) to a HT (Weyuker and Ostrand 2008):

Method 1, classification by explicit link in the VCS: In the case, the VCS provides the possibility to track whether a HT is a defect-correcting or non-defect-correcting HT. This is the easiest possibility to compute the number of defects that occurred in a file. Unfortunately, the most VCSs do not contain such information. Using this method, all kind of defects (pre-release as well as post-release defects) can be considered. The main drawback of this method is that the type of HT (defect-correcting vs. non-defect-correcting) has to be tracked manually at development time and that the quality and completeness of the classification depends on the discipline with which developers track the type of HT they check-in. Despite of these drawbacks, this method is the most reliable of the methods. The authors in (Weyuker and Ostrand 2008) report on using this method to classify a part of the HTs of a voice response system.

Method 2, classification by retrospective manual assignment: In this case, the HT messages are read and manually assigned to one of the types. Since the assignment is performed retrospectively, this method is less reliable than method 1. In addition, due to the effort needed, this method is only applicable for

[31] For the sake of clarity, values below 0.7 are not displayed in the chart. Nevertheless, the values are displayed in the data table below the chart.

small-sized projects. The authors in (Weyuker and Ostrand 2008) report on using this method to classify HTs of a service provisioning system with a small number of HTs.

Method 3, classification by keyword analysis: In this case, the HT messages are automatically analysed for containing keywords (e.g. defect IDs contained in the DTS or keywords like "bug fixed") indicating that a defect has been removed. Main drawback of this method is the possibility of misclassification. In (Fischer, Pinzger, and Gall 2003), an approach for combining data of VCSs and DTSs is presented. The messages recorded in the VCS are searched for defect IDs contained in the DTS using regular expressions. In (Čubranić and Murphy 2003), HTs are searched for keyword patterns like "Fixes bug id" or "id:". In (Śliwerski, Zimmermann, and Zeller 2005) and (Zimmermann, Premraj, and Zeller 2007), the authors combined the defect ID search with the keyword search. In a first step they look for defect IDs contained in the DTS that are referenced in the text of a HT's message. In order to increase the trust level of the results obtained in the first step, the messages obtained in the first step are search for keywords such as "fixed" or "bug". The first step of the algorithm used in this thesis basically corresponds to the approach presented in (Śliwerski, Zimmermann, and Zeller 2005) and (Zimmermann, Premraj, and Zeller 2007). The second step of the algorithm corresponds to the approach described in (Čubranić and Murphy 2003). The multi-defects keyword search has not been considered in literature yet.

Method 4, classification by the number of co-changed files: In this case, the number of files checked-in simultaneously is used as an indicator to differentiate between defect-correcting and non-defect-correcting HTs. If one or two files are changed simultaneously, the HT is supposed to be a defect-correcting HT. If more than two files are changed simultaneously, the assumption is that it is more likely that this change represents a "real" change and thus a non-defect-correcting HT. The analysis occurs automatically by counting the number of co-changed files and by categorising the HTs according to this number. The main drawback of this method is that it has to be analysed empirically if the assumption made applies to the current project. In (Ostrand, Weyuker, and Bell 2005), the authors report on using this classification method for an inventory system.

Method 5, classification by the development stage when a HT has been performed. The assumption behind this method is that changes performed during one of the testing stages following the unit testing phase, for instance integration test, system test, load test, operation readiness test, and user acceptance testing are more likely to be a defect-correcting than a non-defect-correcting HT. This analysis can also be automated by assigning the HT type depending on the development stage it has been reported. One drawback of this method is that only defects reported by the test team can be considered. Defects reported by customers are excluded. In addition, an overlapping of development and testing phases may lead to misclassification of HTs. (Weyuker and Ostrand 2008) report on an empirical study that compares the classification accuracy of this method and of the keyword analysis method. In this special case, they re-

port better results (in terms of misclassification errors) obtained by using the development stage based classification.

Fehler! Verweisquelle konnte nicht gefunden werden. summarises the characteristics of the methods for classification of HTs as defect-correcting and non-defect-correcting HTs. The first column contains the name of the method, followed by the assumptions made (2nd column). In the 3rd column, the degree of automation of the corresponding method is indicated followed by the discussion of the main advantages and the main drawbacks in column 4. Finally, the last column contains the references where the method has been used. Since each of the automatically computed classification of HTs as defect-correcting or non-defect-correcting rely on assumptions, which when violated lead to misclassifications, the manual classifications proposed in Method 1 and Method 2 are more reliable than the others.

Method 3 has been mainly used for analysing open source programs because this is the only possibility to categorise HTs in such projects retrospectively. Usually, neither in case of commercial systems, nor in case of open source programs the type of HT is tracked. In addition, Method 2 can only be applied for very small projects. The projects analysed in this thesis have too many HTs so that a retrospective analysis of *all* HTs is impossible. The assumption about the locality of the defects (Method 4) can vary from project to project. In addition, in open source programs, usually the division of the development process into several sub-phases (e.g. integration testing, system testing) is missing. Consequently, it is hard to apply Method 4 and Method 5 in an open source context.

7.9 Chapter summary

This chapter aims to provide information that builds the context for the empirical studies performed in the context of open source development presented in the Chapters 8 and 9. It includes the definition of basic terms and concepts used in the subsequent chapters. In addition, the first steps of the empirical approach presented in Chapter 6 that are common to all empirical studies are described.

Main goal of the empirical studies presented in this thesis is to explore the relationship between historical and structural characteristics of software and its defect count. In order to analyse whether (bad) structural characteristics can be used as indicators for defects in software, several open source programs are selected and analysed. The following criteria have been applied when selecting the projects: the software's size and maturity, the availability of a version controlled source code as well as of a documented (defect) history.

In the preparation step, the instrumentation of the analyses is defined and developed. Since most of the analyses are performed on file level, the defect count per file has to be determined in this step. For this purpose, information contained in the defect tracking system (DTS) and in the versioning control system (VCS) of each open source program has to be combined because neither the DTS contains information on the location of a defect nor the VCS contains information on the defects for which check-ins (HTs) have been performed.

Method	Assumption	Automation	Discussion	References
Method 1 Classification by explicit link in the VCS	The VCS system provides the possibility to track the type of HT performed (defect-correcting and non- defect-correcting HT).	*Manual* assignment of a HT's type when files are checked-in (at "development time")	1) The quality of the classification depends on the discipline to indicate manually the type of correction done. 2) The most reliable classification method.	(Weyuker and Ostrand 2008)
Method 2 Classification by retrospective manual assignment	Manual assignment is the most reliable classification method.	*Retrospective manual* assignment of a HT's type when files are checked-in (at "analysis time").	Applicable only for small-sized projects.	(Weyuker and Ostrand 2008)
Method 3 Classification by keyword analysis	HTs contain keywords within their messages that indicate whether the HT is a defect-correcting or a non-defect-correcting HT.	*Automated* keyword analysis of the HT messages.	1) Misclassification possible: false positives and false negatives. 2) Usually, the only possibility to classify HTs in OSPs.	(Zimmermann, Premraj, and Zeller 2007); (Śliwerski, Zimmermann, and Zeller 2005); (Fischer, Pinzger, and Gall 2003); (Čubranić and Murphy 2003)
Method 4 Classification by the number of co-changed files	Small changes (affecting 1-2 files) are indicators for defect-correction. Larger changes (affecting more than two files) are indicators for "real" changes (e.g. new functionality).	*Automated* analysis of the number of co-changed files. Empirical evidence required for the analysed program in order to validate the assumption.	Assumption that defects are local does not always apply. It may depend on the program/project if such a classification is appropriate.	(Ostrand, Weyuker, and Bell 2005)
Method 5 Classification by the development stage	HTs performed after "official" unit testing are defect-correcting HTs.	*Automated* analysis of HTs depending on the phase of their check-in.	Concentration on pre-release defects. Defects reported by customers are not considered. Overlapping of development and testing phases may lead to misclassification.	(Weyuker and Ostrand 2008)

CHAPTER 8 Frequency and Pareto distribution of defects

The Pareto Principle is a universal principle of the "vital few and trivial many (Juran and Gryna 1988). According to this principle, the 80/20 rule has been formulated meaning that for many phenomena, 80% of the consequences originate from 20% of the causes. In this chapter, the Pareto Principle is applied to software testing. The following hypotheses are validated:

- ***Pareto distribution of defects in files:*** *A small number of files accounts for the majority of the defects.*
- ***Pareto distribution of defects in files across releases:*** *The Pareto Principle applies to all releases of a software project.*
- ***Pareto distribution of defects in code:*** *A small part of the code accounts for the majority of the defects.*
- ***Pareto distribution of defects in code across releases****: The Pareto Principle applies to all releases of a software project.*

Knowing that the Pareto Principle is valid in a specific project context is useful for testers because they can focus their testing activities on the "vital" 20% of the software.

8.1 Introduction

The Pareto principle, also known as the 80/20 rule, states that a large part of effects (about 80%) come from a smaller part (about 20%) of causes. This phenomenon has been originally analysed by Vilfredo Pareto (Reh, J.F. 2005) who observed that 80% of the income is obtained by 20 percent of the population. Juran (Juran and Gryna 1988) generalised this principle he called the "vital few and trivial many", stating that most of the results in any context are raised by a small number of causes. This principle has been often applied in several contexts, for instance in sales, stating that 20% of the customers are responsible for 80% of the sales volume.

One of the first studies that translated this principle to the software engineering area is reported in (Endres 1975). The author analyses the distribution of defects in an operating system developed at IBM laboratories. The distribution of about 430 defects over about 500 modules has been analysed and confirms the Pareto Principle, i.e. approximately 80% of the defects were contained in 20% of the modules.

Two main hypotheses related to the Pareto Principle form the basis of the empirical study presented in this chapter. A first analysis aims to determine whether a small part of files accounts for the majority of defects. Second, if this is the case, the subsequent question is whether this small part of files also constitutes a small part of the system's code size (Illes-Seifert and Paech 2009).

Knowing that the Pareto principle is valid in the testing context is very valuable for testers because they can focus their testing activities on the "vital few" files accounting for most of the defects. From the research perspective, this study increases the empirical body of knowledge in the area of defect distribution in software. This is one of few studies that focus on the analysis of the Pareto Principle in detail including data from 9 large OSPs.

The reminder of this chapter is organised as follows: The design of the study is described in Section 8.2. In Section 8.3, the results of the empirical study are presented and in Section 8.4, an overview of related work is given. Finally, the summary of this chapter is given in Section 8.5.

8.2 Study design

In this chapter, the following hypotheses related to the Pareto principle are analysed.

- **Hypothesis P1, Pareto distribution of defects in files:** A small number of files accounts for the majority of the defects.
- **Hypothesis P2, Pareto distribution of defects in files across releases:** If the Pareto Principle applies to one release, then it applies to all releases of a software program.
- **Hypothesis P3, Pareto distribution of defects in code:** A small part of the system's code size accounts for the majority of the defects.
- **Hypothesis P4, Pareto distribution of defects in code across releases:** If

the Pareto Principle applies to one release, then it applies to all releases of a software program.

8.3 Results

8.3.1 Exploring the Pareto distribution of defects in files

The first hypothesis related to the 80/20 rule concerns the distribution of defects in files. Generally, most of the files contain few defects. In four programs (ANT, FOP, OSCache, TVBrowser), more than 80% of the files contain no defects. In two programs (CDK, Freenet), about 70% of the files contain no defects. In case of the Jmol program, 64% of the files contain no defects.

All OSPs presented in Section 7.6 are analysed graphically in order to verify the 80/20 rule. Figure 8.1 shows the Alberg Diagram suggested in by Ohlsson and Alberg (Ohlsson and Alberg 1996) for the graphical analysis of the Pareto Principle in the project OSCache. Accordingly, files are ordered in decreasing order with respect to the number of defects. Then, the cumulated number of defects is plotted on the y-axis of the Alberg diagram relative to the percentage of files (plotted on the x-axis). The dotted line shows that 80% of the defects are contained in 11.55% of the files.

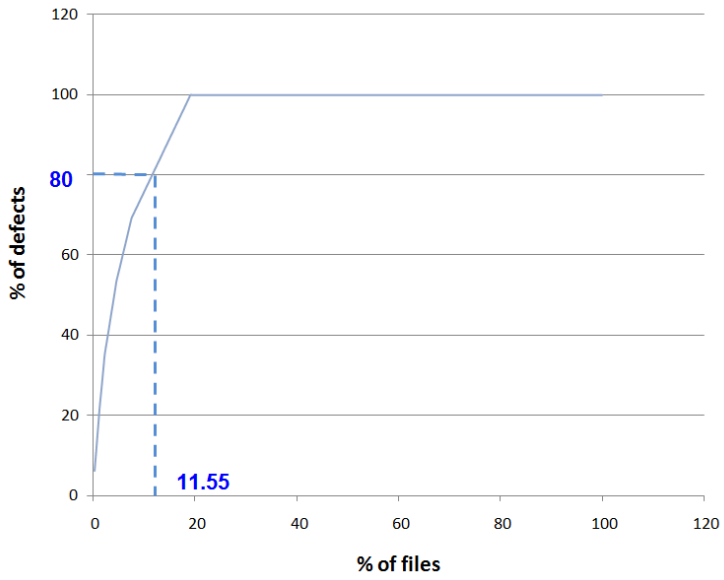
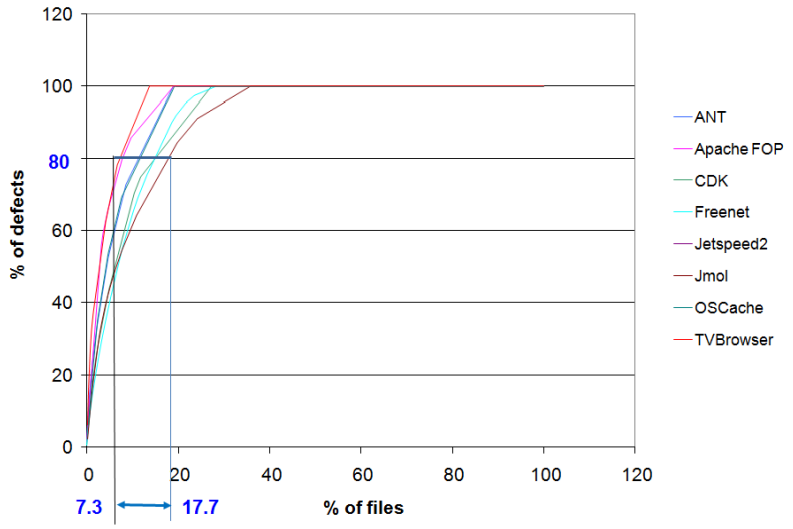
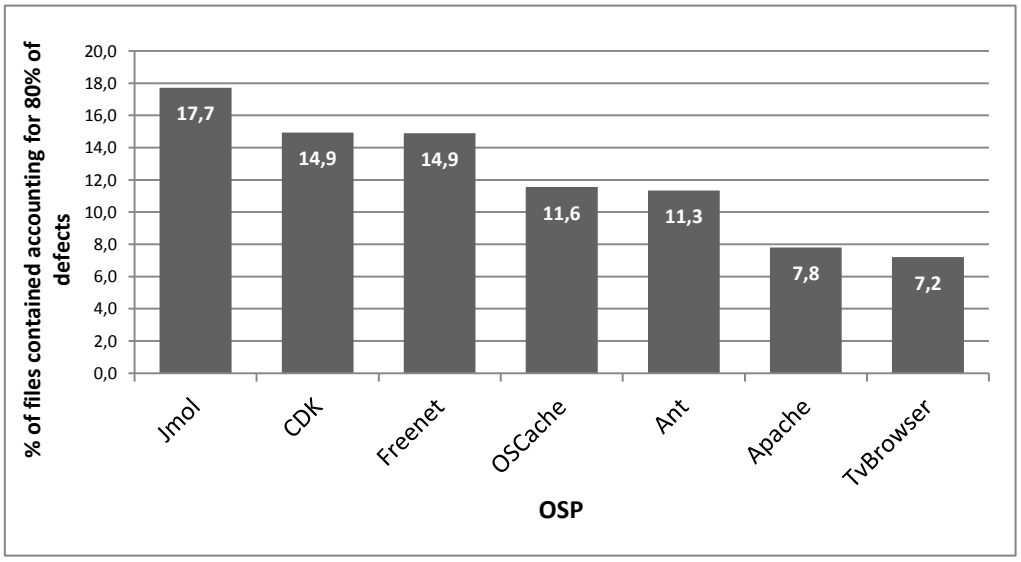


Figure 8.1 - Pareto distribution of defects in files of the OSCache project

Figure 8.2 (a) shows the distribution of defects of all analysed OSPs within one Alberg diagram and Figure 8.2 (b) shows the percentage of files accounting for about 80% of defects as a bar chart.



(a)



(b)

Figure 8.2 - Pareto distribution of defects

(a) Distribution of defects for each OSP in an Alberg diagram; (b) Percentage of defects contained in 80% of the most fault-prone files

Approximately 80% of the defects are concentrated in a range of 7.2% (in case of the TVBrowser program) to 17.7% (in case of the Jmol program) of the files. The TVBrowser program shows the strongest focus of defects on a very small part of the files.

Based on this analysis, Hypothesis P1 can be largely confirmed for OSPs: A small number of files account for the majority of the defects in OSPs.

8.3.2 Exploring the Pareto distribution of defects in files across releases

In order to analyse this hypothesis, the percentage of the files containing 80% of the defects is computed for several releases of the OSPs.

Table 8.1 shows the results. The first column contains the name of the OSP followed by the number of the analysed releases. The next two columns indicate the absolute and respectively the relative number of releases for which about 80% of the defects are concentrated in a small percentage (below 20%) of files. The column "Range" indicates the range for the concentration of defects. For example, the concentration of defects in the CDK program ranges from 7.03% to 19.95% of the files depending on the considered release.

		Pareto distribution holds for …			100% of defects contained in less than 25% of the files. This holds for …	
OSP	Number of analysed releases	Absolute # of analysed releases	Percentage of the analysed releases	Range	Absolute # of analysed releases	Percentage of the analysed releases
1. ANT	3	3	100%	8.59% - 14.06%	3	100%
2. ApacheFOP	2	2	100%	8.87% - 11.93%	2	100%
3. CDK	3	3	100%	7.03% - 19.95%	1	33%
4. Freenet	3	2	66%	14.66% - 36.64%	1	33%
5. Jmol	3	3	100%	10.06% - 22.89%	1	33%
6. OSCache	3	3	100%	8.16% - 13.98%	3	100%
7. TVBrowser	3	3	100%	8.59% - 14.00%	2	66%

Table 8.1 - Pareto distribution of defects in files across releases

For all OSPs, in at least one of the analysed releases, 80% of the defects are contained in less than 20% of the most fault-prone files. For six OSPs, the Pareto Principle holds for all analysed releases. The concentration of the defects ranges from 7.03% to 36.64%. In many releases of the analysed OSPs, a high concentration of defects on a small number of files can be observed. Thus, an additional analysis determines the percentage of files that account for 100%, i.e. for **all** defects in a system. The last two columns in Table 8.1 show the absolute and relative number of releases for which 100% of defects are contained in less than 25% of the files. In two thirds of the analysed releases of the OSPs, 100% of the defects are concentrated in less than 25% of the files.

Based on the results of the analyses presented in this section, it can be concluded that the Pareto Principle largely persists across several releases of a software program. The concentration intensity can vary from release to release.

8.3.3 Exploring the Pareto distribution of defects in code

In order to analyse the Pareto hypothesis for code, the percentage of code that accounts for 80% of the defects contained in the most fault-prone files has been computed. Consequently, this analysis determines whether the small part of the files responsible for most of the defects also represent a small part of the code. The results of this analysis are shown in Figure 8.3. On the x-axis, the analysed releases of the OSPs are indicated. The line chart and the bar chart indicate for each release the percentage of files and the corresponding percentage of code that account for approximately 80% of the defects. For example, in case of the TVBroswer program, release 1.0, **8.65%** of the files that account for 80% of the defects make up **16.28%** of the system's code.

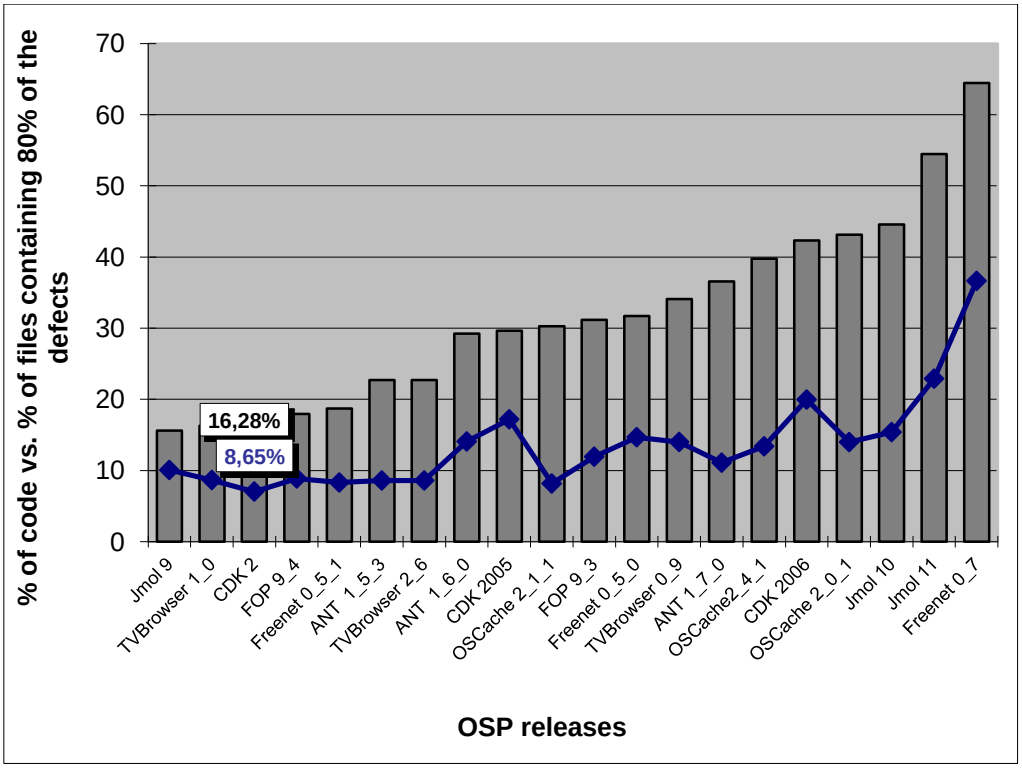


Figure 8.3 - Pareto distribution of defects in code

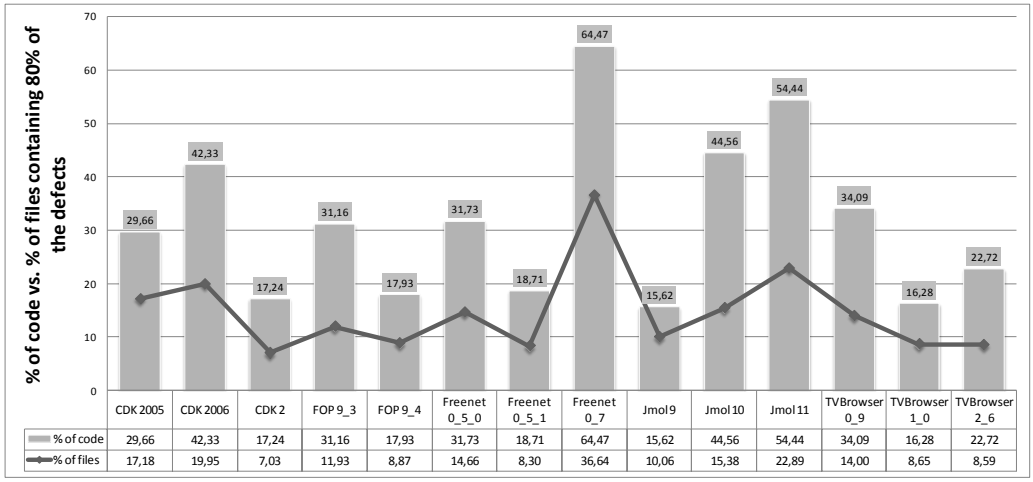
The concentration of the majority of the defects on a small part of the system's code is true only for a small part of the analysed releases. Five releases show a concentration of defects on less than 20% of the code. Most of the analysed releases show a distribution of the defects on about 20% to 50% of the code. For a small part of the releases, the defects are distributed on almost the whole system.

Based on this analysis, the hypothesis P3 has to be rejected. A small part of the code accounts for the majority of the defects only in a few of the analysed cases.

8.3.4 Exploring the Pareto distribution of defects in code across releases

Since the Pareto hypothesis on the distribution of defects in code has been rejected, the hypothesis P4 has to be adjusted. For all cases, in which the Pareto hypotheses could be confirmed: Does the Pareto distribution of defects in code hold for all or at least for the most releases of an OSP? Despite the fact that Hypothesis P3 has been rejected, this research question is important to be analysed. If the hypothesis can be confirmed, it means that for a small part of OSPs, the Pareto Principle is valid and it is worthwhile to perform further analyses in order to determine characteristics of such programs and to find out factors that favour such a distribution.

Figure 8.4 shows the distribution of 80% of the defects in code across releases for all OSPs for which at least one release shows a high concentration of defects on less than 20% of the code. The bar chart shows the percentage of code that contains 80% of the defects and the line chart shows the percentage of files accounting for 80% of the defects. For all programs, only one single release shows a concentration of the defects on a small part of the code. For the other analysed releases, the defects are distributed on about 25% to 65% of the code.



	CDK 2005	CDK 2006	CDK 2	FOP 9_3	FOP 9_4	Freenet 0_5_0	Freenet 0_5_1	Freenet 0_7	Jmol 9	Jmol 10	Jmol 11	TVBrowser 0_9	TVBrowser 1_0	TVBrowser 2_6
% of code	29,66	42,33	17,24	31,16	17,93	31,73	18,71	64,47	15,62	44,56	54,44	34,09	16,28	22,72
% of files	17,18	19,95	7,03	11,93	8,87	14,66	8,30	36,64	10,06	15,38	22,89	14,00	8,65	8,59

Figure 8.4 - Pareto distribution of defects in code across releases

Based on the results of this analysis, the adjusted hypothesis P4 cannot be confirmed. A concentration of most of the defects on a small part of the code in one release does not mean that this concentration will persist in consequent releases.

8.4 Related Work

In this section, related work concerning the frequency distribution of defects as well as concerning the analysed hypotheses is presented. In addition, a discus-

sion of the results reported in literature in comparison with the results obtained by the study reported in this chapter is given.

8.4.1 Frequency distribution of defects

The result that most of the files contain very few defects is supported by other authors in literature.

In (Hatton 2008), the author reports on a study performed on two scientific mature systems: a FORTRAN and a C library. The first program has been developed between 1970 until 2000 and consists of about 3670 routines. The second program is similar to the first one with respect to its specification and has been developed between 1990 until 1999. (Hatton 2008) describes that 78% of the NAG Fortran components have no defects; similarly, 66% of the components in the C library are without defects. In (Kaâniche and Kanoun 1996), the authors analyse data collected from five consecutive releases of a commercial telecommunications system, including two prototype releases. The study analysed the distribution of 1512 defects reported by customers and validation teams on 77 Atomic Components. Atomic components denote parts of the system "fulfilling elementary functions". The authors report that 95% of the analysed atomic components have less than 3 defects/KLOC.

8.4.2 Pareto distribution of defects in files

One of the first studies that published results of an empirical investigation about defect distribution in programs is described in (Endres 1975). The author analysed about 430 defects in 500 modules of the DOS/VS operating system developed in the IBM Böblingen laboratory. The analysis confirmed the hypothesis and showed that 78% of the defects are concentrated in 21% of the modules. The defects included in the study subsume all defects found by system testing, the author denotes as a "formal test period of five months" after unit testing[32]. Another early study has been performed by (Adams 1984) at IBM. His primary findings were that most of the defects lead very rarely to failures in practice[33] and that a very small percentage of the defects (about 10%) are worth fixing because only these lead to serious operational problems.

In (Munson and Khoshgoftaar 1992), the authors analyse data collected from two large commercial systems: a command and control communication system and a medical imaging system. The medical system consists of about 45.000 routines from which 390 routines have been randomly selected for analyses. The command and control communication system consists of two programs having a total of 327 modules. The defects considered in this study comprise

[32] According to the author, the objective of this testing phase was to test the complete system with all its components in as many variations as possible.

[33] Most of the defects have mean times to discovery of hundreds to thousands of months when run on a single machine, a single defect requiring very unusual circumstances to expose a failure.

pre-release and post-release defects together[34]. The authors confirm the Pareto distribution with a 20-65 ratio.

The analyses performed by (Kaâniche and Kanoun 1996) confirm the Pareto distribution of defects: 38% of the Atomic Components contain 80% of the defects.

In (Ohlsson and Alberg 1996), the authors analyse the distribution of defects from two consecutive releases of a telecommunication switching system. The authors analyse data from several subsystems consisting of 20 to 40 modules, each with a size of 1.000 – 6.000 lines of code. The defects included in this study comprise all defects reported during function testing, system testing, as well as during the first months of operation. The analyses confirm the Pareto distribution of faults. 20 percent of the modules are responsible for 60 percent of the defects.

In (Fenton and Ohlsson 2000), a study on the distribution of defects in two releases of a major commercial system developed at Ericsson Telecom AB is reported. For the analyses, 140 respectively 246 modules have been randomly selected; the size of the selected modules ranges from 1.000 to 6.000 LOC. The study considers several types of defects: testing/pre-release defects (reported during function testing and system testing by testers) and operational/post-release defects (reported during operation). The analyses confirm the Pareto distribution in all releases and for all defects types: 20% of the modules account for 60% of the pre-release defects. The Pareto distribution for post-release defects is even stronger. Thus, 10 percent of the modules contain 100% of the defects in release 1 and 80% of the defects in release 2.

In (Ostrand and Weyuker 2002), the authors report on an empirical analysis of the distribution of defects in thirteen releases of a large industrial inventory tracking system. The latest release of the system contains about 2.000 files with a total of 500.000 lines of code. During all releases and all development stages, a total of 4.743 defects were detected, primarily by testing. For each release, the defects were always heavily concentrated in a relatively small number of files. In addition, concentration gets stronger in later releases. Thus, the Pareto distribution is confirmed by this study. The authors additionally investigate whether the Pareto distribution is true for pre-release and post-release defects. In both cases and for all analysed releases, a small number of files accounts for the most part of pre-release as well as of post-release defects.

The study described in (Andersson and Runeson 2007) is a replication of the study presented in (Ohlsson and Alberg 1996). Empirical data from three projects from a large company in the telecommunications domain have been considered. The authors analyse both, the distribution of pre-release and of post-release defects. The analyses confirm the Pareto hypothesis. In all three ana-

[34] Defects recorded during the system integration and test phase and for the first year of program deployment.

lysed projects, about 12 – 20 % of the post-release defects are contained in 80% of the modules. In addition, about 26 – 34% of the modules are responsible for 80% of the pre-release defects.

The results of the studies in literature are summarised in Table 8.2. Beside the study reported in (Ostrand and Weyuker 2002) and in (Kaâniche and Kanoun 1996), all authors concentrate on analysing few or one release of a system. In contrast to the study reported in this chapter, all studies consider commercial software. All authors confirm the Pareto distribution of defects, i.e. most of the defects in commercial software concentrate on a small number of files respective of modules. This is similar to the results obtained by studying OSPs in this chapter.

The type of defects analysed differs from study to study. Roughly, the defect types can be categorised into pre-release and post-release defects. In one study, only one defect type has been analysed (Endres 1975)[35]. Three studies differentiate between pre-release and post-release defects (Andersson and Runeson 2007), (Ostrand and Weyuker 2002), and (Fenton and Ohlsson 2000). All other studies analyse pre-release and post-release defects altogether. When roughly categorising the defects into pre-release vs. post-release defects, the following conclusions can be drawn:

- The Pareto distribution of defects is true for pre-release as well as for post-release defects.
- Authors, who did not make distinction between pre-release and post-release defects, confirm the Pareto distribution of defects, too.
- In the studies that differentiate between pre-release and post-release defects (Andersson and Runeson 2007), (Ostrand and Weyuker 2002), and (Fenton and Ohlsson 2000), the concentration of defects on a small part of modules/files is greater for post-release defects than this is the case for pre-release defects. Ostrand and Weyuker (Ostrand and Weyuker 2002) observe that there is a very small part of defects reported after release and these defects are concentrated in less than 1% of the files. The Pareto distribution for pre-release defects reported in (Ostrand and Weyuker 2002) is similar to the overall Pareto distribution (observed when pre-release and post-release defects have been analysed altogether). The authors also distinguish between early pre-release and late pre-release defects. Defects detected during early pre-release phases accounted for the most part of the defects.

A clear distinction between pre-release and post-release defects is not possible for OSPs. Since the algorithm that computes the number of defects per file presented in Section 7.7.1 considers defects reported *after* release and the number of defects found by direct search makes up the greatest part of all defects, most of the defects considered in the study reported in this chapter are post-release defects. However, there can also be defects (e.g. found by keyword search) that have been detected for example during integration testing. Thus, a clear distinc-

[35] The analyses consider pre-release defects.

tion is not possible. Consequently, the results are comparable to those studies that considered pre-release and post-release defects altogether.

Reference	Characteristics of the analysed projects	Confir-mation?	Relationship	Kind of defects analysed
(Endres 1975)	One release of the operating system DOS/VS.	Yes	21 – 78	pre-release defects (i.e. defects found during system testing)
(Andersson and Runeson 2007)	Three projects from a large company in the telecommunications domain.	Yes	20 – 87 (P1) 20 – 87 (P2) 20 – 80 (P3)	post-release defects (It is not clear, whether post-release defects include the defects reported by the test team only or by the customers, too.)
			20 – 63 (P1) 20 – 70 (P2) 20 – 70 (P3)	pre-release defects
(Ohlsson and Alberg 1996)	Two consecutive releases of a telecom-munication switching system.	Yes	20 - 60	pre-release and post-release defects altogether
(Kaâniche and Kanoun 1996)	Five consecutive releases of a commer-cial telecommunica-tions system.	Yes	38 – 80	pre-release and post-release defects altogether: defects recorded as “Failure Reports” reported from validation teams and from customers
(Fenton and Ohlsson 2000)	Two releases of a major commercial system developed at Ericsson Telecom AB.	Yes	20-60	pre-release and post-release defects altogether: defects reported during function testing and system testing by testers
			10 – 100 1st release 10 – 80 2nd release	post-release defects: defects reported during operation
(Munson and Khoshgoftaar 1992)	Two distinct data sets from large commercial systems: command and control communi-cation system, medical imaging system.	Yes	20-65	pre-release and post-release defects altogether: defects recorded during the system integration and test phase and for the first year of program deployment
(Ostrand and Weyuker 2002)	Thirteen releases of a large industrial inven-tory tracking system.	Yes	10 - 68 10 -100 (for the last four re-leases)[36].	pre-release and post-release defects altogether: all kinds of defects recorded in one of these phases development, unit testing, integration test-ing, system testing, beta re-lease, controlled release, and general release. The Pareto distribution is also true for pre-release and post-release defects

Table 8.2 – Pareto principle, related work

[36] Concentration of defects on a small number of files increases as system matures.

The study presented in (Ostrand and Weyuker 2002) is the only one that analyses the Pareto distribution across several consecutive releases. The authors observe that the concentration of defects on a small part of files becomes stronger when the system matures. This result differs from that obtained when analysing OSPs. In case of the OSPs, the concentration remains low across nearly all releases of the analysed OSPs but the extent to which defects are concentrated on a part of the files varies from release to release.

8.4.3 Pareto distribution of defects in code

Similarly to the results presented in this chapter, there is little evidence for the Pareto distribution of defects in code. The strongest concentration of defects on a small part of code is reported in (Ostrand and Weyuker 2002). Accordingly, 10% of the files that account for a range of 68% - 100% of defects (depending on the analysed release) contain about 35% of the system's code. But the percentage of the code contained in the most fault-prone files always exceeded the percentage of the files that contained the defects. The results reported in (Fenton and Ohlsson 2000), (Andersson and Runeson 2007), and (Kaâniche and Kanoun 1996) do not provide evidence for the Pareto distribution of defects in code as well. This is the case for both, pre-release and post-release defects as reported in (Andersson and Runeson 2007).

The only study analysing the Pareto distribution of defects in code across several releases is reported in (Ostrand and Weyuker 2002). In contrast to a decreasing concentration of defects on a small part of files from release to release, the corresponding percentage of code does not show such a trend. 10% of the most fault-prone files that account for the most of the system's defects make up about 35% of the code. This result is similar to the results obtained by analysing the Pareto distribution in code for OSPs as reported in this chapter.

8.5 Chapter summary

This chapter presents the results of an empirical study on the distribution of defects in software. In contrast to most of the studies considering a small number of commercial systems, this study analyses the distribution of defects in a wide range of open source programs across several releases (Illes-Seifert and Paech 2009). From the research's point of view, this study increases the empirical body of knowledge. Performing a family of similar studies is advocated in order to gain confidence in the results, instead of relying on single studies with specific context (Pfleeger 2005), (Basili and Lanubile 1999).

Two of the initial hypotheses can be confirmed: A small number of files accounts for the majority of the defects (*Hypothesis P1*). This is true even across several releases of software (*Hypothesis P2*). The results widely correspond to the findings reported in literature.

Similarly to the results reported in literature, this study did not find evidence for the initial hypotheses concerning the distribution of defects in code (*Hypothesis P3)*. Defects concentrate on a small part of the files but they **do not** con-

centrate on a small part of the code. One reason for this could be that a considerable part of an application's logic is concentrated on few files that are fault-prone and not well understood. These files are candidates for refactoring and should be considered by maintainers. In addition, unit test coverage criteria should be intensified for those parts of the code responsible for most of the defects. These files should also be higher prioritised during regression testing.

One of the goals of this study is to analyse the Pareto distribution of defects. If confirmed, advices can be given to testers on which parts of the software under test to concentrate their limited resources. Despite of the confirmation of the Pareto distribution for files, defects are not concentrated on a small part of code. Consequently, detecting which 20% of the files account for most of the defects is useful for testers, but not enough to prioritise testing activities because these 20% of the files possibly account for a high part of the code and hence of an application's logic. For this purpose, additional indicators, for instance a file's age or its complexity, should be used in order to give reliable advices to testers on which parts of the software testing activities should be focused. Algorithms like those presented in (Kim et al. 2008) that determine the most fault-prone files are only useful when considering the amount of code covered by these files.

CHAPTER 9 Bad smells

Bad smells have been introduced as patterns for frequently occurring problems in code (Fowler et al. 1999)*, i.e. the code might be difficult to understand or might cause high maintenance effort. Thus, bad smells are commonly used as indicators for those parts of the software that should be refactored. In this chapter, the empirical approach presented in Chapter 6 is applied in order to explore the extent to which bad smells can be used as indicators for defects in code.*

9.1 Introduction

As software degenerates when it evolves, continuous refactoring activities are essential in order to be able to efficiently maintain large software systems. "Bad smells", also called "code smells", have been firstly introduced by Fowler and Beck (Fowler et al. 1999) as patterns for bad design and bad programming practices resulting in frequently occurring problems in code. Often, these parts of the code are used for detecting refactoring opportunities in software (Mens and Tourwe 2004). Consequently, bad smells serve as indicators for those parts of the software with high impact on its quality in terms of flexibility, maintainability or readability.

Little attention has been paid to analyse the relationship between bad smells and defects in software empirically. In general, only a few empirical studies have been conducted to examine the effects of bad smells (Zhang et al. 2008).

The overall goal of this empirical study is to explore the relationship between (bad) structural product characteristics and software quality in terms of defects. Using the template presented in Section 6.2, this goal can be refined as follows.

- *Object of study*: Analyse different **bad structural** software characteristics
- *Purpose*: for the purpose of their evaluation
- *Quality focus:* with respect to the efficiency of showing correlations with a software's defect count
- *Perspective*: from the point of view of researchers, testers, maintainers, and quality engineers
- *Context*: in the context of open source development.

Knowing that particular bad smells are indicators for defects in code is valuable for different roles in the software development process. *Testers* can focus the testing effort and to allocate their limited resources appropriately. *Quality engineers* can initiate improvements of the development process. For instance, they can develop guidelines that assist developers and software designers in avoiding coding style that leads to defects. *Maintainers* have additional support in selecting parts of the software that should be refactored.

The refactoring process consists of six steps (Mens and Tourwe 2004):

(1) identification of what should be refactored, (2) selection of the refactoring to be performed, (3) guarantee that the refactoring does not change functionality, (4) application of the refactoring, (5) test the refactoring, and (6) modify all artefacts that are affected by the refactoring to provide consistency. Knowing which particular bad smells affect the software's defect count, helps maintainers in step 1, i.e. they get decision support in prioritizing refactoring activities.

In (Marinescu 2002), a set of well known bad smells have been quantified in terms of so called detection strategies. Accordingly, a strategy is the "*quantifiable expression of a rule by which design fragments that are conforming to that rule can be*

detected in the source code". For example, the quantification of the rule to detect the bad smell "God Method" is expressed by the following rule:

GodMethod (m_i)=

a) LOC (m_i) ε {TopValues(20%)} ∧ LOC (m_i) > 70 ∧

b) (NOP (m_i) > 4 ∨ NOLV(m_i) > 4) ∧

c) MNOB (m_i) > 4

LOC (Lines of Code)
Number of lines of code in a method m_i, including comments

NOP (Number of Parameters)
Number of parameters of the method m_i

NOLV (Number of Local Variables)
Number of local variables declared in method m_i

MNOB (Maximum Number of Branches)
Maximum number of if-else/case branches in method m_i

Accordingly, the God Method bad smell indicates methods that tend to centralize a class' functionality, becoming more and more complex and difficult to understand and to maintain. Methods conforming to this detection strategy have the following characteristics and will be classified as "God Method":

a) The corresponding method is large (expressed by the LOC metric) AND
b) it has a long parameter list or many local variables (expressed by the NOP metric) AND
c) it has many local variables declared (expressed by the NOLV metric) AND
d) it is complex, in terms of high number of branches (expressed by the MNOB metric).

In this chapter, the results of an empirical study are presented that explores the relationship between bad smells and defects in open source programs. Particularly, popular bad smells have been identified in the code of several java programs following the detection strategies presented in (Marinescu 2002). Then, the relationship between bad smells in code and defects is analysed visually and statistically.

The reminder of this chapter is organised as follows. Basic terms are introduced in Section 9.2 whereas in Section 9.3, the overall hypothesis as well as the dependent and independent variables are presented. Section 9.4 summarises the results of the empirical study. The discussion of the results is presented in Section 9.5. An overview of the related work is given in Section 9.6. The summary in Section 9.7 concludes this chapter.

9.2 Bad smells and refactoring

Bad smells represent a metaphor originally introduced by (Fowler et al. 1999) that describes patterns for re-occurring problems in code due to bad design and bad programming practices. These patterns have been originally proposed to identify code that needs to be refactored.

Refactoring aims to restructure an existing body of code, i.e. the internal structure is altered without changing its external behaviour (Fowler et al. 1999), (Chikofsky and Cross II 1990)[37]. For this purpose, a series of small behaviour preserving transformations are performed, each of these transformations is called a refactoring. A sequence of transformations aims to produce significant restructuring of code. A good indicator to start refactoring is when code starts to "smell" (Fowler et al. 1999). Benefits of undertaking refactoring include reduced complexity and increased readability, extensibility, modularity, reusability, maintainability, and efficiency (Mens and Tourwe 2004).

9.3 Quality indicators and overall hypothesis

In this section, details on the empirical study are presented.

9.3.1 Overall research hypothesis

The main goal of this empirical study is to analyse the relationship between bad smells and defects in software. The overall research hypothesis is that a software entity for which a bad smell applies is more fault-prone than a software entity for which a bad smell does not apply. The rationale behind this hypothesis is that code for which a bad smell applies is difficult to understand, it is too complex, inadequately subdivided or redundant. Thus, changing or introducing new functionality is expected to introduce defects.

9.3.2 Dependent variables

The dependent variables in this study are the defect count of a file (DCF) and the defect count of a package (DCP). The *defect count of a package* DCP is calculated by summing up the defect counts of all files that contain classes belonging to that package.

9.3.3 Independent variables

The independent variables are bad smells that apply or that do not apply to a software entity. In this chapter, bad smells are defined on different abstraction levels: method, class and package level. In the following, for each bad smell, the description and the corresponding research hypotheses are formulated.

---

[37] In (Chikofsky and Cross II 1990), refactoring is defined as “the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system’s external behaviour (functionality and semantics).”

Method level bad smells

Description. Two method level bad smells are considered in this study: Feature Envy (FE) and God Method (GM). The *Feature Envy* bad smell indicates that a method seems more interested in a class, particularly in the data of that class other than the one the method is in (Fowler et al. 1999). Thus, the corresponding method seems to be misplaced. The *God Method* bad smell indicates that too much functionality is centralised in that particular method (Fowler et al. 1999), (Riel 1996).

Since it is not possible to track back the defect count on method level in OSPs, method level bad smells have to be aggregated to class level. For this purpose, this study explores whether a class is more fault-prone than another if it contains at least one method for which that particular bad smell applies.

Hypotheses. The research hypotheses are presented in Table 9.1. The first column contains the null-hypothesis and the second column the alternative hypothesis. For each hypothesis, the formalised hypothesis is indicated in italic face.

Null-Hypothesis N_i	**Alternative Hypothesis A_i**
N-FE: A file with at least one method for which the FE bad smell applies is no more fault-prone than files without at least one method for which the FE bad smell applies. $DCF^{38}_{FeatureEnvy} = DCF_{\neg FeatureEnvy}$	**A-FE:** A file with at least one method for which the FE bad smell applies is more fault-prone than a file that has no methods with this bad smell. $DCF_{FeatureEnvy} > DCF_{\neg FeatureEnvy}$
N-GM: A file with at least one method for which the GM bad smell applies is no more fault-prone than files without at least one method for which the GM bad smell applies. $DCF_{GodMethod} = DCF_{\neg GodMethod}$	**A-GM:** A file with at least one method for which the GM bad smell applies is more fault-prone than a file that has no methods for which this bad smell applies. $DCF_{GodMethod} > DCF_{\neg GodMethod}$

Table 9.1 - Hypotheses for method level bad smells

Class level bad smells

In this chapter, the following class level bad smells are analysed:

- Data Class (DC). The *Data Class* bad smell indicates data containers with lack of responsibility in terms of functional behaviour.
- God Class (GC). The *God Class* bad smell indicates that a particular class centralises too much of a system's functionality.
- Shotgun Surgery (SS). The *Shotgun Surgery* bad smell indicates that every time a change is made to a class, a lot of little changes have to be made to other classes, too.
- Refused Bequest (RB). The *Refused Bequest* bad smell is an indicator for the lack of improper OO-Design, since subclasses use only parts of the members of their ancestors.

[38] Defect count of a file

- Misplaced Class (MC). The *Misplaced Class* bad smell indicates that a class is on wrong place since there are more dependencies to classes in other packages than to classes in the package the class is contained in (Marinescu 2002). This bad smell violates the principles of package cohesion as described in (Martin 2000).

Table 9.2 contains the hypotheses formulated for class level bad smells.

Null-Hypothesis N_i	**Alternative Hypothesis A_i**
N-GC: A file with at least one class for which the GC bad smell applies is no more fault-prone than files without at least one class for which the GC bad smell applies. $DCF_{GodClass} = DCF_{\neg GodClass}$	**A-GC:** A file with at least one class for which the GC bad smell applies is more fault-prone than files without at least one class for which the GC bad smell applies. $DCF_{GodClass} > DCF_{\neg GodClass}$
N-DC: A file with at least one class for which the DC bad smell applies is no more fault-prone than files without at least one class for which the DC bad smell applies. $DCF_{DataClass} = DCF_{\neg DataClass}$	**A-DC:** A file with at least one class for which the DC bad smell applies is more fault-prone than files without at least one class for which the DC bad smell applies. $DCF_{DataClass} > DCF_{\neg DataClass}$
N-SS: A file with at least one class for which the SS bad smell applies is no more fault-prone than files without at least one class for which the SS bad smell applies. $DCF_{ShotgunSurgery} = DCF_{\neg ShotgunSurgery}$	**A-SS:** A file with at least one class for which the SS bad smell applies is more fault-prone than files without at least one class for which the SS bad smell applies. $DCF_{ShotgunSurgery} > DCF_{\neg ShotgunSurgery}$
N-RB: A file with at least one class for which the RB bad smell applies is no more fault-prone than files without at least one class for which the RB bad smell applies. $DCF_{RefusedBequest} = DCF_{\neg RefusedBequest}$	**A-RB:** A file with at least one class for which the RB bad smell applies is more fault-prone than files without at least one class for which the RB bad smell applies. $DCF_{RefusedBequest} > DCF_{\neg RefusedBequest}$
N-MC: A file with at least one class for which the MC bad smell applies is no more fault-prone than files without at least one class for which the MC bad smell applies. $DCF_{MisplacedClass} = DCF_{\neg MisplacedClass}$	**A-MC:** A file with at least one class for which the MC bad smell applies is more fault-prone than files without at least one class for which the MC bad smell applies. $DCF_{MisplacedClass} > DCF_{\neg MisplacedClass}$

Table 9.2 - Hypotheses for class level bad smells

Lack-Of-Pattern bad smells

In (Marinescu 2002), a new type of bad smells, the "Lack-of-Patterns", is introduced. This bad smell category contains design flaws that result when not using an appropriate design pattern. The assumption is that missing a design pattern leads to bad design and consequently to more defects in the corresponding software entities. The following "Lack-of-Pattern" bad smells are analysed in this chapter.

- Lack of Bridge (LoB). The *Lack of Bridge* bad smell indicates that the "Bridge" design pattern is missing. Thus, abstraction and implementation are not decoupled allowing to be varied independently.
- Lack of State (LoSta). The State design pattern allows an object to alter its behaviour when it's internal state changes. The *Lack of State* bad smell indicates that the "State" design pattern is missing.

- Lack of Strategy (LoStr). The *Lack of Strategy* bad smell indicates that the "Strategy" design pattern is missing. Algorithms and the clients that use them are not decoupled and thus cannot be changed independently.
- Lack of Visitor (LoV). The Visitor design pattern abstracts functionality that can be performed on the elements of an object structure. The *Lack of Visitor* bad smell indicates that the "Visitor" design pattern is missing.
- ISP Violation (ISP). The *ISP Violation* bad smell does not indicate that a design pattern has not been applied. It indicates that an OO principle, the Interface Segregation Principle, as introduced by R. Martin (Martin 1996), has been violated. The ISP principle deals with the problem of non-cohesive interfaces where parts of the interface can be grouped by the member functions. Thus, groups of clients use different function groups offered by the interface.

Table 9.3 contains the hypotheses formulated for the lack-of-patterns bad smells as well as for the ISP violation bad smell.

Null-Hypothesis N_i	**Alternative Hypothesis A_i**
N-LoB: A file with at least one class for which the LoB bad smell applies is no more fault-prone than files without at least one class for which the LoB bad smell applies. $DCF_{Lack\ of\ Bridge} = DCF_{\neg Lack\ of\ Bridge}$	**A-LoB:** A file with at least one class for which the LoB bad smell applies is more fault-prone than files without at least one class for which the LoB bad smell applies. $DCF_{Lack\ of\ Bridge} > DCF_{\neg Lack\ of\ Bridge}$
N-LoSta: A file with at least one class for which the LoSta bad smell applies is no more fault-prone than files without at least one class for which the LoSta bad smell applies. $DCF_{Lack\ of\ State} = DCF_{\neg Lack\ of\ State}$	**A- LoSta:** A file with at least one class for which the LoSta bad smell applies is more fault-prone than files without at least one class for which the LoSta bad smell applies. $DCF_{Lack\ of\ State} > DCF_{\neg Lack\ of\ State}$
N-LoStr: A file with at least one class for which the LoStr bad smell applies is no more fault-prone than files without at least one class for which the LoStr bad smell applies. $DCF_{Lack\ of\ Strategy} = DCF_{\neg Lack\ of\ Strategy}$	**A-LoStr:** A file with at least one class for which the LoStr bad smell applies is more fault-prone than files without at least one class for which the LoStr bad smell applies. $DCF_{Lack\ of\ Strategy} > DCF_{\neg Lack\ of\ Strategy}$
N-LoV: A file with at least one class for which the LoV bad smell applies is no more fault-prone than files without at least one class for which the LoV bad smell applies. $DCF_{Lack\ of\ Visitor} = DCF_{\neg Lack\ of\ Visitor}$	**A- LoV:** A file with at least one class for which the LoV bad smell applies is more fault-prone than files without at least one class for which the LoV bad smell applies. $DCF_{Lack\ of\ Visitor} > DCF_{\neg Lack\ of\ Visitor}$
N-ISP: A file with at least one class for which the ISP bad smell applies is no more fault-prone than files without at least one class for which the ISP bad smell applies. $DCF_{ISP\ Violationr} = DCF_{\neg ISP\ Violation}$	**A- ISP:** A file with at least one class for which the ISP bad smell applies is more fault-prone than files without at least one class for which the ISP bad smell applies. $DCF_{ISP\ Violation} > DCF_{\neg ISP\ Violation}$

Table 9.3 - Hypotheses for lack-of-pattern bad smells

Package level bad smells

In this chapter, two package level bad smells are analysed: God Package (GP) and Wide Subsystem Interface (WSI). Similarly to the God Method and God Class bad smells, the *God Package* bad smell indicates that a package centralises

too much of a software's functionality. The *Wide Subsystem Interface* indicates that the interface of a package is wide, leading to a tight coupling to other packages. Table 9.4 contains the hypotheses for package level bad smells.

Null-Hypothesis N_i	**Alternative Hypothesis A_i**
N-GP: A package for which the GP bad smell applies is no more fault-prone than packages for which the GP bad smell does not apply. $DCP_{GodPackage} = DCP_{\neg GodPackage}$	**A-GP:** A package for which the GP bad smell applies is more fault-prone than packages for which the GP bad smell does not apply. $DCP_{GodPackage} > DCP_{\neg GodPackage}$
N-WSI: A package for which the WSI bad smell applies is no more fault-prone than packages for which the WSI bad smell does not apply. $DCP_{WideSystemInterface} = DCP_{\neg WideSystemInterface}$	**A-WSI:** A package for which the WSI bad smell applies is more fault-prone than packages for which the WSI bad smell does not apply. $DCP_{WideSystemInterface} > DCP_{\neg WideSystemInterface}$

Table 9.4 - Hypotheses for package level bad smells

9.4 Results

In this section, the results of the study are presented.

9.4.1 Exploring the relationship between method level bad smells and defects

In order to analyse the first hypothesis A-FE, the Mann-Whitney non-parametric test is performed. For this test, the data in each project are divided into two groups: one group consisting of files that contain at least one method for which the FE bad smell applies (FE-group) and a second group that consists of files that do not contain methods for which the FE bad smell applies (¬FE-group). Differences between two populations can be analysed with the help of Mann-Whitney test (Section 2.5.5).

Figure 9.1 shows the DVAs for all projects with significant results in the Mann-Whitney test. For each group on the x-axis (¬FE-group, FE-group), the mean defect count in each of the groups is indicated on the y-axis. The results show that files for which the FE bad smell applies are 1.9 times (ANT 1.5.3) to 6.1 times (OSCache 2.4) more fault-prone than files for which the FE bad smell does not apply. Detailed results of the Mann-Whitney test can be found in Appendix A 4.

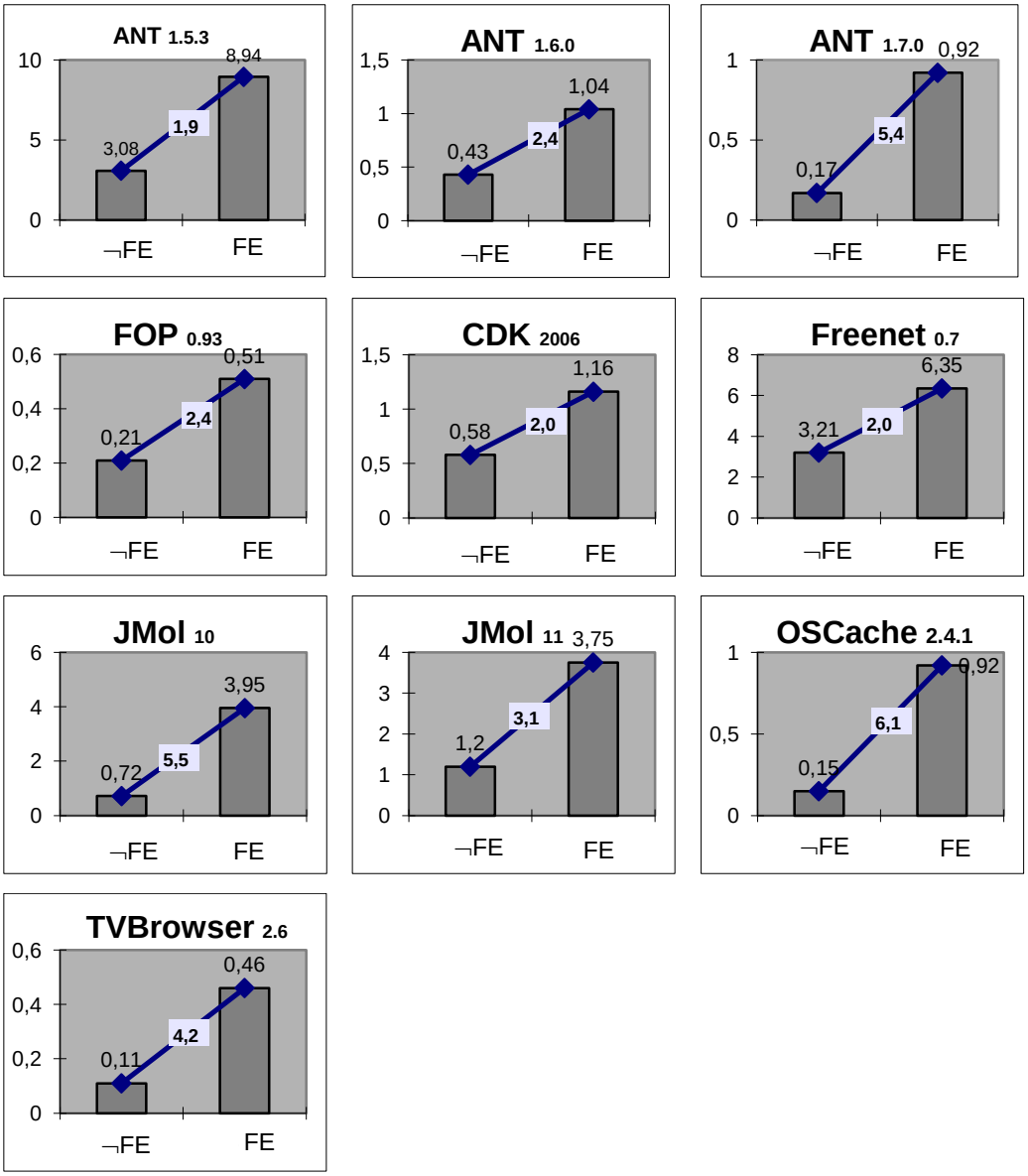


Figure 9.1 - DVA: Visual mean defect count analysis for the FE bad smell

Figure 9.2 shows the aggregated results of the bad smell analyses performed on method level. Fourteen of the analysed releases contain at least one method for which the FE bad smell applies. The results show that for ten of these fourteen releases, files containing methods for which the FE bad smell applies are more fault-prone than the other files. Consequently, the null hypothesis N-FE can be rejected to some part and the alternative hypothesis A-FE accepted.

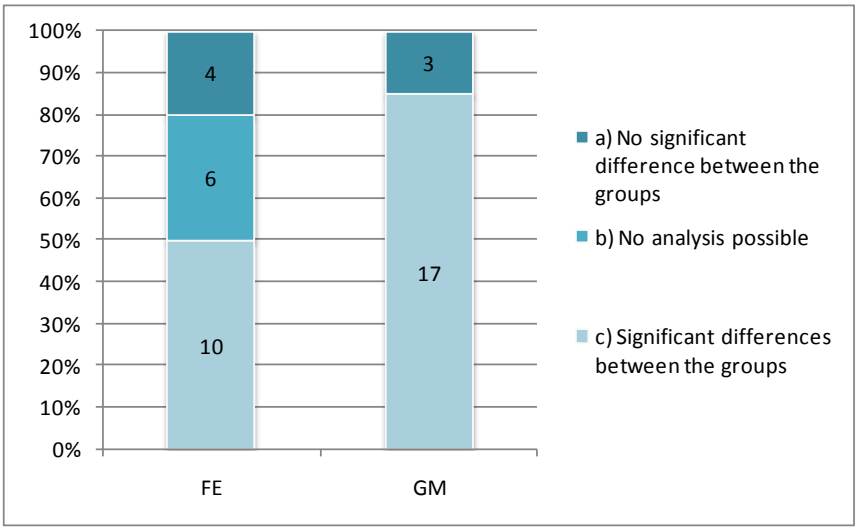


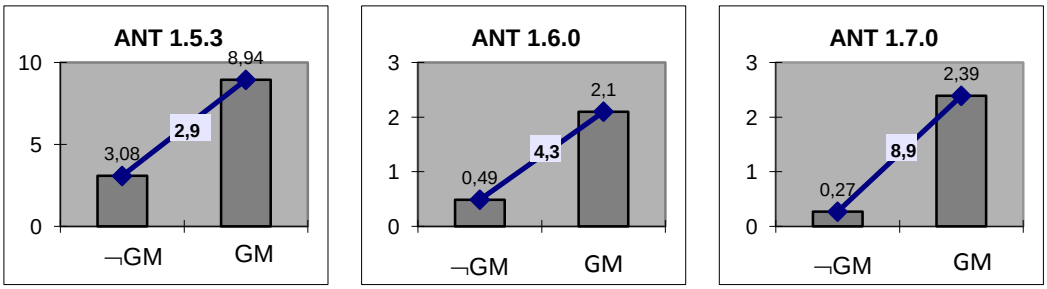
Figure 9.2 - Method level bad smell analysis

Accordingly, it can be concluded that a file with at least one method for which the FE bad smell applies is more fault-prone than a file that has no methods with this bad smell.

In order to analyse the second hypothesis A-GM, the Mann-Whitney non-parametric test is performed again. Similarly to the procedure applied to analyse the A-FE hypothesis, the data in each project are divided into two groups: one group consisting of files that contain at least one method for which the GM bad smell applies (GM-group) and a second group that consists of files that do not contain any methods for which the GM bad smell applies (¬GM-group).

All analysed releases contain at least one method for which the GM bad smell applies. The results show that for seventeen releases, files in the GM-group are more fault-prone than files in the ¬GM-group.

Figure 9.3 shows the DVAs for all projects with significant results in the Mann-Whitney test. For each group (¬GM-group, GM-group), the mean defect count is indicated on the y-axis. Thus, files in the GM-group are 2.1 times (Freenet 0.7) to 8.9 times (ANT 1.7.0) more fault-prone than classes in the ¬GM-group. Detailed results of the Mann-Whitney test can be found in the Appendix A 4.



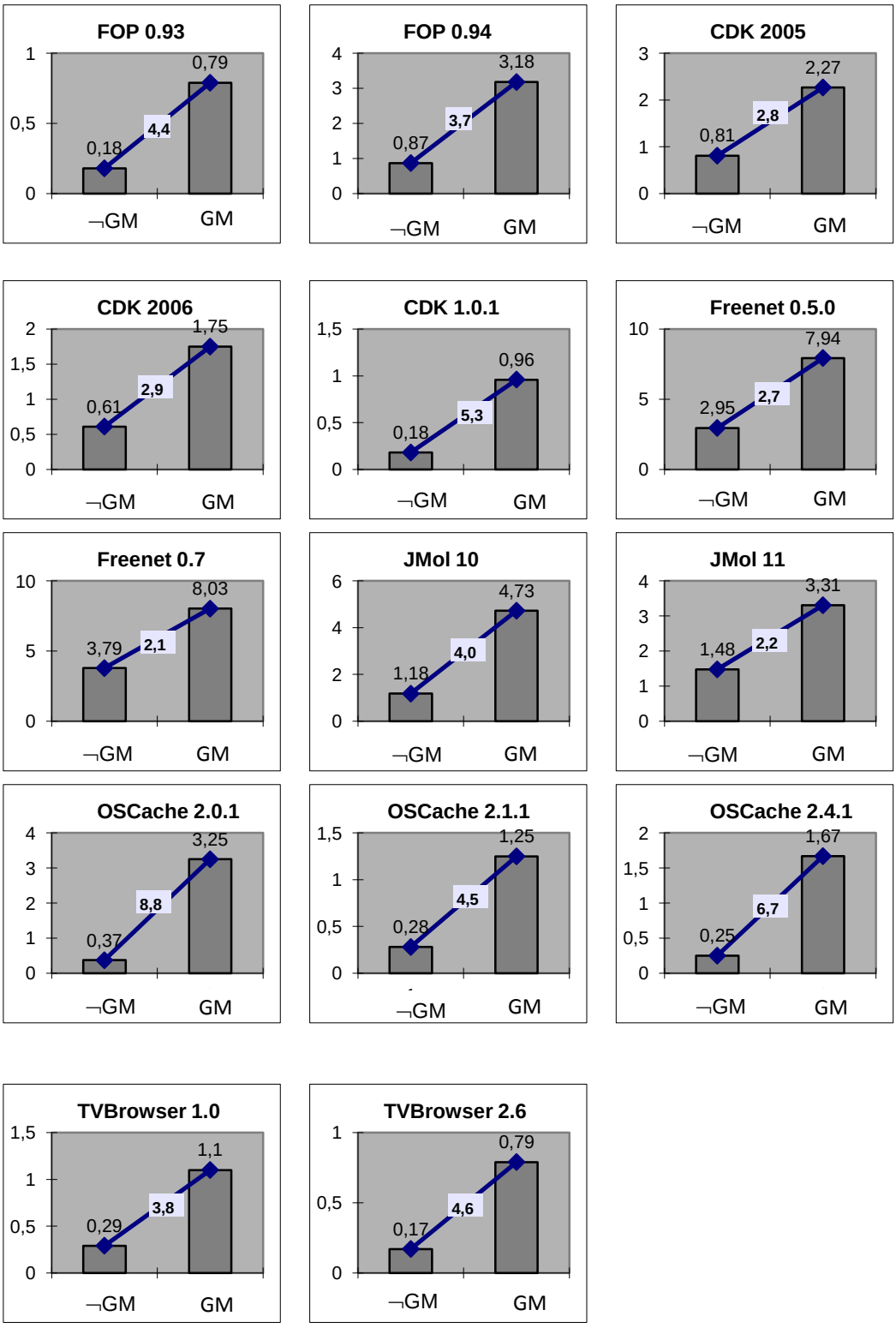


Figure 9.3 - DVA:Visual mean defect count analysis for the GM bad smell

Consequently, it can be concluded that a file with at least one method for which the GM bad smell applies is more fault-prone than a file that has no methods with this bad smell.

9.4.2 Exploring the relationship between class level bad smells and defects

In order to analyse the class level hypotheses, the Mann-Whitney test is performed again. For this purpose, the data in each project are divided into two groups: one group consisting of files for which a particular *class* level bad smell applies (BS_i[39]-group) and a second group that consists of files for which a particular class level bad smell does not apply ($\neg BS_i$-group). Figure 9.4 shows the aggregated results.

On the x-axis, the analysed bad smells are indicated. The y-axis shows the number of programs for which one of the following results applies:

- *a)*: There are no significant differences in terms of defect count between the BS_i-group and the $\neg BS_i$-group.
- *b)*: There are no classes for which the corresponding bad smell applies. As a result, the statistical test cannot be performed.
- *c)*: There are significant differences in terms of defect count between the BS_i-group and the $\neg BS_i$-group.

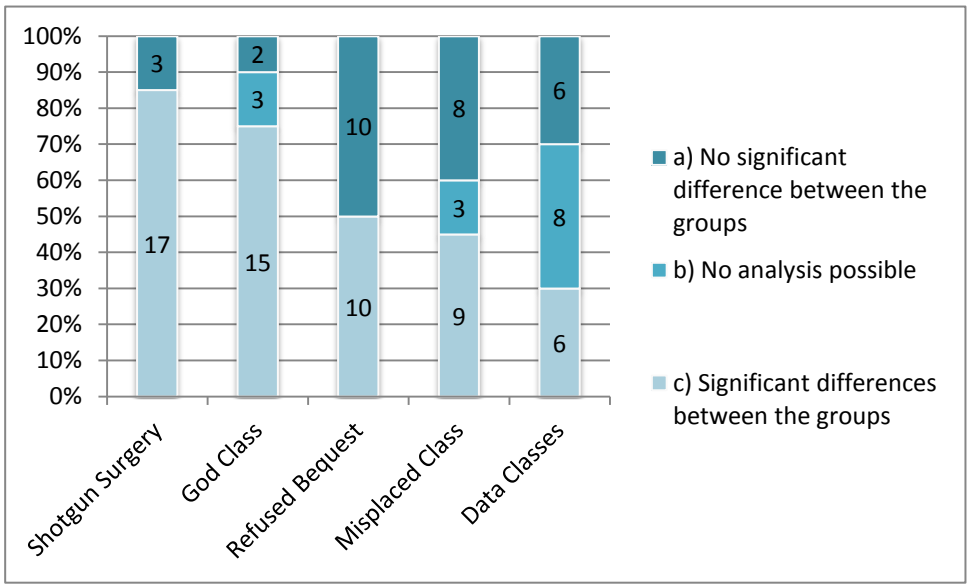


Figure 9.4 - Class level bad smell analysis

The results show that in seventeen respectively fifteen of the analysed releases, files that contain classes for which the SS and GC bad smells apply are more fault-prone than files that do not contain classes for which these bad smells apply.

For the RB bad smell, this is the case for ten of twenty analysed releases. Nine of the analysed releases show significant differences between files in the MC-group and files in the ¬MC-group. In case of the DC bad smell, six releases

[39] $BS_i \in \{GC, DC, SS, RB\}$

show significant differences between the two groups. The detailed results of the corresponding Mann-Whitney test are shown in the Appendix A 5.

The visual analysis shows that files for which the SS bad smell applies are 1.7 (Freenet 0.5.0/0.7) to 8.74 (OS-Cache 2.0.1) more fault-prone than files for which the SS bad smell does not apply. Similarly, files for which the GC bad smell applies are 2.9 (Freenet 0.7) to 9.3 (ANT 1.7.0) more fault-prone than files for which the GC bad smell does not apply. The detailed results of the visual analyses are shown in Appendix A 7.

Based on the results of the statistical tests and of the visual analyses, the following conclusions can be drawn:

- The null-hypotheses N-SS and N-GC can be largely rejected and the corresponding alternative hypotheses accepted.
- The null-hypothesis N-RB can be rejected to some part and the corresponding alternative hypothesis accepted.
- The null-hypotheses N-MC and N-DC can be accepted and the corresponding alternative hypotheses rejected. Files containing classes for which the MC or DC bad smells apply are no more fault-prone than files that do not contain classes for which that bad smells apply. Consequently, the corresponding null hypotheses have to be accepted.

9.4.3 Exploring the relationship between Lack-Of Pattern bad smells and defects

In order to analyse the lack-of-pattern bad smell hypotheses, the Mann-Whitney test is performed. For this purpose, the data in each project are divided into two groups: one group consisting of files for which a particular lack-of-pattern bad smell applies (BS-l_i[40]-group) and a second group that consists of files for which a particular lack-of-pattern bad smell does not apply (¬BS-l_i-group). Figure 9.5 shows the aggregated results.

[40] BS-l_i ε {LoB, LoSta, LoStr, LoV, ISP}

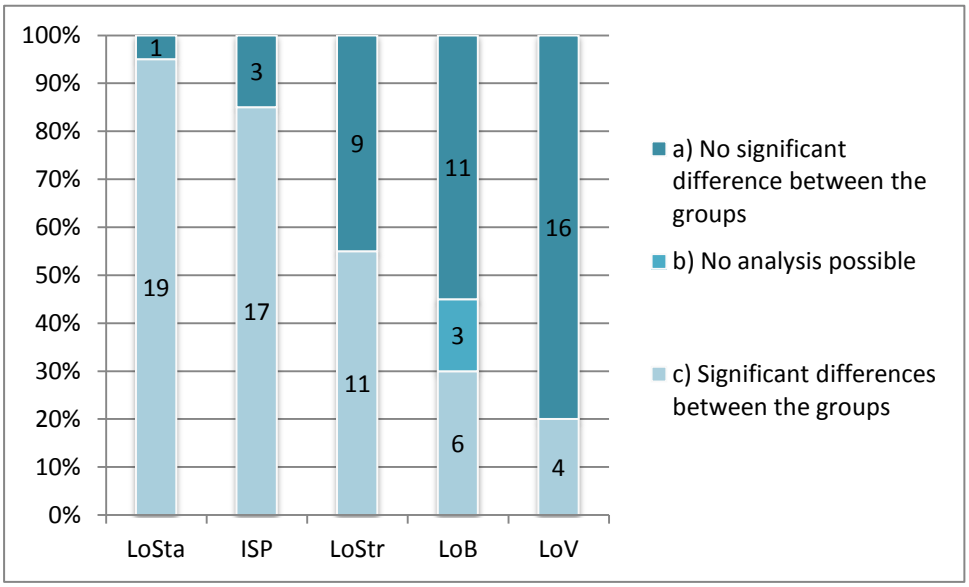


Figure 9.5- Lack-of Patterns

The results show that in nineteen of the analysed releases, files containing classes for which the LoSta bad smell applies are more fault-prone than files that do not contain classes for which this bad smell applies.

For the ISP violation bad smell, this is the case for seventeen releases. About half of the analysed releases show significant differences between files containing classes for which the LoStr bad smell applies and files that do not contain classes for which this bad smell applies. For the rest of the analysed bad smells, less than half of the releases (six in case of the LoB bad smell and four in case of the LoV bad smell) show significant differences between the two groups, one for which the corresponding bad smell applies and one for which the bad smell does not apply. The detailed results of the Mann-Whitney test are shown in the Appendix A 6.

The visual analysis for the LoSta bad smell shows that files containing classes for which the LoSta bad smell applies are 2.2 (FOP 0.94) to 11.5 (TV-Browser 0.9) more fault-prone than files that do not contain classes for which the LoSta bad smell applies. In case of the ISP bad smell, the visual analysis shows that files containing classes for which the ISP bad smell applies are 2.0 (FOP 0.93) to 6.4 (TV-Browser 2.6) more fault-prone than files that do not contain classes for which the ISP bad smell applies. The detailed results are shown in Appendix A 8.

Based on the results of the statistical tests and of the visual analyses, the following conclusions can be drawn:

- The null-hypothesis N-LoSta and N-ISP can be largely rejected and the corresponding alternative hypothesis accepted. Files containing classes for which the LoSta/ISP bad smell applies are more fault-prone than files that do not contain classes for which the LoSta/ISP bad smell applies.

- The null-hypothesis N-Str can be rejected to some part and the corresponding alternative hypothesis accepted.
- There is little statistical evidence that files containing classes for which the LoV or the LoB bad smell apply are more fault-prone than files that do not contain classes for which the bad smells apply. Thus, the null-hypotheses N-LoV and N-LoB can be accepted and the corresponding alternative hypotheses rejected.

9.4.4 Exploring the relationship between package level bad smells and defects

In order to analyse the package level hypotheses, the Mann-Whitney test is performed. For this purpose, the data in each project are divided into two groups: one group consisting of *packages* for which a particular package level bad smell applies (BS-p_i[41]-group) and a second group that consists of packages for which a particular package level bad smell does not apply (¬BS-p_i-group). Figure 9.6 summarises the results.

Accordingly, in case of thirteen releases, packages for which the GP bad smell applies are more fault-prone than packages for which this bad smell does not apply. For the WSI bad smell, there is a single release that shows significant differences between the two groups (packages for which the WSI bad smell applies and packages for which the WSI bad smell does not apply). The detailed results of the Mann-Whitney test are shown in the Appendix A 9.

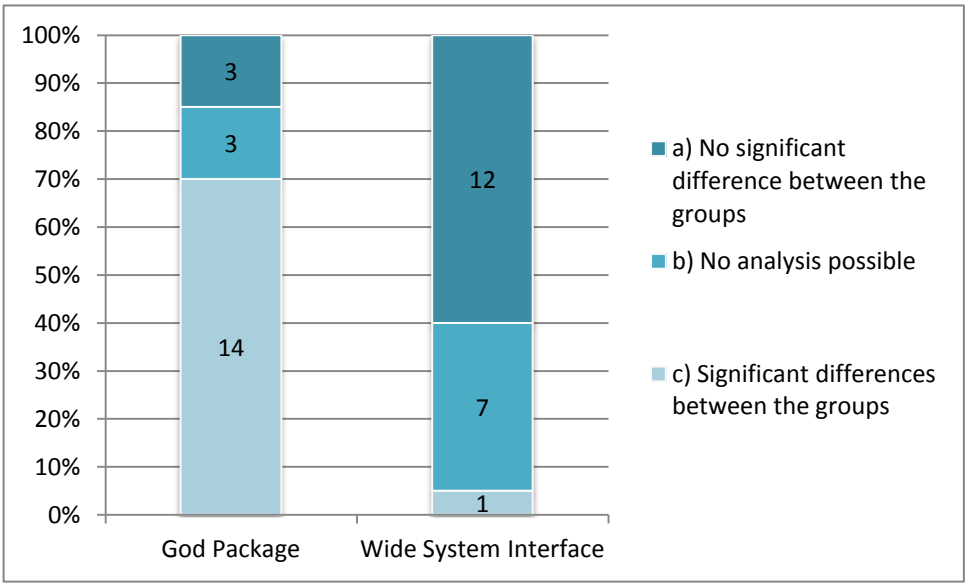


Figure 9.6 - Package level bad smell analysis

The visual analysis shows that packages for which the GP bad smell applies are 1.7 (FOP 0.94) to 20.3 (CDK 2005) more fault-prone than packages for which the GP bad smell does not apply. The detailed results of the visual analyses are shown in Appendix A 10.

[41] BS-p_i ε {GP, WSI }

Based on the results of the statistical tests and of the visual analyses, the following conclusions can be drawn:

- The null-hypothesis N-GP can be largely rejected and the corresponding alternative hypothesis accepted. Packages for which the GP bad smell applies are more fault-prone than packages for which the GP bad smell does not apply.
- There is no evidence that packages for which the WSI bad smell applies are more fault-prone than packages for which this bad smell does not apply. Consequently, the null-hypothesis N-WSI can be accepted and the corresponding alternative hypothesis rejected.

9.4.5 Which bad smell is the best indicator for defects in code?

In order to answer this question, two factors are defined and compared across all analysed projects:

F_{BS} The BADSMELL-factor F_{BS} indicates the proportion between the mean defect count of files that contain classes for which a bad smell applies and the mean defect count of files that do not contain classes for which that particular bad smell applies.

F_{AVG} The AVERAGE-factor F_{AVG} indicates the proportion between the mean defect count of files containing classes for which a bad smell applies and the average defect count in *all* classes.

In order analyse which bad smell is the best indicator for defects in code, the following questions have to be answered:

a) Which bad smells show significant differences between files for which a bad smell applies and files for which that bad smell does not apply *in most of the programs*?

b) Which bad smell has the highest F_{BS}?

c) Which bad smell has the highest F_{AVG}?

d) Which bad smell has the highest factors (F_{AVG}, F_{BS}) across all projects and all bad smells?

(a) The following bad smells show significant differences in the analyses performed in Section 9.4 in 15 to 19 releases: God Method (GM), God Class (GC), Shotgun Surgery (SS), Lack of State (LoSta), and ISP violation (ISP). These bad smells are considered in the following in order to answer the questions (b) – (d). For this purpose, the factors F_{AVG} and F_{BS} have to be compared. The results are shown in Table 9.5. The columns indicate the F_{BS} and the F_{AVG} factors per bad smell. The last two columns indicate the maximum F_{BS} respectively F_{AVG} computed per release.

	GM (17)		SS (17)		GC (15)		LoSta (19)		ISP (17)			
OSP	F-BS	F-AVG	F-BS	F-AVG	F-BS	F-AVG	F-BS	F-AVG	F-BS	F-AVG	MAX F-BS	MAX F-BS
Ant 1.5.3	2,9	2,7	2,3	2,0	3,4	3,2	2,7	0,3	2,8	2,6	3,4	3,2
Ant 1.6.0	4,3	3,7	2,5	2,2	4,2	3,8	3,9	2,0	3,3	2,9	4,3	3,8
Ant 1.7.0	8,9	6,8	4,7	3,4	9,3	6,9	8,8	4,1	5,8	4,6	9,3	6,9
FOP 0.93	4,4	3,0	1,9	1,6	3,3	2,2	3,9	1,2	2,0	1,8	4,4	3,0
FOP 0.94	3,7	3,1			4,3	3,8	2,2	1,9	2,6	2,3	4,3	3,8
CDK 2005	2,8	2,5	2,8	2,4	4,9	4,7	2,3	2,3	2,7	2,5	4,9	4,7
CDK 2006	2,9	2,5	3,7	2,9	5,8	5,4	3,2	2,7	2,8	2,6	5,8	5,4
CDK 1.0.1	5,3	3,7	2,5	2,2	4,3	4,2	4,3	1,1	3,4	3,1	5,3	4,2
Freenet 0.5.0	2,7	2,5	1,7	1,5	4,5	4,3	2,5	1,9	2,1	2,0	4,5	4,3
Freenet 0.5.1			3,4	2,8	10,5	9,5	5,3	4,8	5,2	4,8	10,5	9,5
Freenet 0.7	2,1	2,0	1,7	1,5	2,9	2,8	2,8	2,0	2,0	1,9	2,9	2,8
Jmol 9			2,5	2,0					3,0	2,6	3,0	2,6
Jmol 10	4,0	3,4			4,7	3,3	4,2	3,3			4,7	3,4
Jmol 11	2,2	1,9	3,0	2,3	5,0	3,3	3,1	2,1	3,1	2,5	5,0	3,3
OSCache 2.0.1	8,8	6,6	8,7	5,4			9,5	4,9	**2,3**	5,7	9,5	6,6
OSCache 2.1.1	**4,5**	**4,0**	2,2	**2,0**			**2,4**	**2,6**			4,5	4,0
OSCache 2.4.1	6,7	5,0	8,0	5,3			5,1	6,6	6,3	4,8	8,0	6,6
TVBrowser 0.9							**11,5**	7,0	5,8	3,9	11,5	7,0
TVBrowser 1.0	3,8	3,3			7,8	7,0	2,6	2,2			7,8	7,0
TVBrowser 2.6	4,6	3,5	6,0	3,5	8,3	6,4	6,9	**3,3**	6,4	4,6	8,3	6,4

Table 9.5 - Which bad smells are the best indicators for defects in code?
A comparison of the FBS and FAVG factors

In twelve releases, the F_{BS} is the highest for the *GC bad smell* (question b); in three cases, for the GM bad smell. For four releases, this factor could not be computed as there are no classes for which the GC bad smell applies. Similar results are obtained for question (c) and (d). In thirteen releases, the F_{AVG} is the highest for the *GC bad smell.* In addition, the *GC bad smell* has the highest F_{BS} factor across all projects and all bad smells (9.5 in case of the Freenet 0.5.1 release). The highest F_{AVG} show the LoSta and the GC bad smells. It can be concluded that the GC bad smell and the LoSta lack-of-pattern show the strongest association with a software entity's defect count.

9.5 Discussion

The results show that some bad smells can indicate defects in code. The strongest associations show the "god - *" bad smells, on all analysed levels (god method, god class, and god package). This bad smell indicates a high centralisation of too much of the application's logic into one entity. Thus, the entities for which the "god - *" bad smell applies are too large and complex, not easy to understand and to maintain, and lead to a high defect count.

Another bad smell that proved to be a good indicator for defects in code is the "Shotgun Surgery" bad smell. This bad smell refers to the lack of locality when making changes. When changing the entity, these changes are not local and affect too many parts of the application and lead to defects. The main reason is that when performing changes, not all parts that are affected are considered, leading also to failures.

On class level, so called "Lack-Of" patterns introduced by (Marinescu 2002) have been analysed with respect to their association with the software's defect count. "Lack-Of" patterns are design flaws that result when not using an appropriate design pattern. Apart from the Lack-Of-State and the ISP bad smell, entities for which the lack-of pattern applies are not more fault-prone than entities for which that pattern does not apply.

9.6 Related work

Most of research related to the study presented in this chapter, analyses the relationship between OO metrics and defects, e.g. in (Chidamber and Kemerer 1994), (Szabo and Khoshgoftaar 1995), (Basili, Briand, and Melo 1996), (Fenton and Ohlsson 2000), (Gyimothy, Ferenc, and Siket 2005), (Subramanyam and Krishnan 2003), (Briand, Daly, and Wüst 1998), (Briand, Daly, and Wüst 1999), (Briand et al. 2000), (Cartwright and Shepperd 2000), (Emam et al. 2001), (Emam, Melo, and Machado 2001), (Zimmermann, Premraj, and Zeller 2007), (Nagappan, Ball, and Zeller 2006). Apart from the study reported in (Shatnawi and Li 2006), little attention has been paid to analyse the relationship between bad smells and defects in software empirically. In general, only a few empirical studies have been conducted to examine the effects of bad smells (Zhang et al. 2008).

In the following, a detailed comparison of the study presented in this section and the study reported in (Shatnawi and Li 2006) is discussed. Both studies consider open source programs. A basic difference is the magnitude of the study presented in this chapter. Whereas the study in (Shatnawi and Li 2006) analyses a single OSP across three releases, the study presented in this section considers seven OSPs across twenty releases. In addition, Shatnawi and Li (Shatnawi and Li 2006) consider only a part of the bad smells analysed in this section. Table 9.6 summarises the results of both studies. The first column indicates the bad smell. In the second and third column, the results of both studies are compared. A "+" indicates that in the corresponding study a positive association between the bad smell and the defect count has been observed. A "-" indicates that no association has been observed (respectively that a very small part of the analysed releases show statistical significant differences between entities for which the corresponding bad smell applies and entities for which that bad smell does not apply). A "~" indicates that an association has been observed in some releases of the analysed OSPs. The third column also indicates the number of releases for which the association has been observed. A blank cell indicates that the corresponding association has not been analysed.

Bad smell	(Shatnawi and Li 2006)	This study
Method Level		
God Method	+	+(17/20)
Feature Envy		~(10/20)
Class Level		
Data Class	-	- (6/20)
God Class	+	+(15/20)
Shotgun Surgery	+	+(17/20)
Refused Bequest	-	~/(10/20)
Misplaced Class		- (9/20)
Lack-Of-Patterns		
Lack of Bridge		- (6/20)
Lack of State		+(19/20)
Lack of Strategy		- (11/20)
Lack of Visitor		- (4/20)
ISP Violation		~(17/20)
Package Level		
God Package		+(13/20)
WSI		- (2/20)

Table 9.6 – Bad smells and defects - (Shatnawi and Li 2006) vs. results of this thesis

Comparing the results of the bad smell analysis in (Shatnawi and Li 2006) with the results of this study

The results of both studies are similar. In both studies, the bad smells "God Method", "God Class", and "Shotgun Surgery" show a positive association with the defect count. In addition, none of the studies confirms a positive association of the bad smell "Data Class" and the defect count. The RB bad smell shows in half of the analysed releases a positive association with the defect count; Shatnawi and Li do not confirm any statistical significant association for the RB bad smell.

Based on the results of both studies, it can be concluded that there are some bad smells that are useful as indicators for defects in software.

9.7 Chapter summary

In this chapter, the results of an empirical study exploring the relationship between bad smells and defects are presented. There are several bad smells that are good indicators for the software's defects: On method level, the "God Method" bad smell, on class level, the "God Class" and the "Shotgun Surgery" bad smell, and finally on package level, the "God Package" bad smell.

On class level, so called "Lack-Of" patterns introduced by (Marinescu 2002) have been analysed. The results show that the "Lack of State" and the "ISP violation" patterns are good indicators for a class' defect count. For all other "Lack-Of" patterns, only a small part of the analysed releases show significant differ-

ences between parts of the software for which a particular pattern applies and parts of the software for which this pattern does not applies.

The God Class (GC) and the Lack of State (LoSta) bad smells proved to be the "best indicators" for a software entity's defect count. On average, entities for which the GC bad smell applies are six times more fault-prone than entities for which the GC bad smell does not apply.

CHAPTER 10 Exploring the relationship of a file's history and its defect count

In this chapter, the relationship between several historical characteristics of files and their defect count is explored. For this purpose, the empirical approach presented in Chapter 6 using statistical procedures and visual representations of the data is applied in an industrial context in order to determine historical indicators for a file's defect count. The results show that files that have been changed by a number of authors above average are more fault-prone than files that have been changed by a number of authors below average. In addition, the number of changes performed to a file is also a good indicator for its defect count. In contrast to initial expectations, the hypothesis concerning a file's age as well as the hypotheses concerning the number of co-changed files could be confirmed only partly.

10.1 Introduction

The primary goal of the empirical study presented in this chapter is to apply the approach presented in Chapter 6 in an industrial context. In addition, it aims to analyse the relationship between a file's history and its defect count. The main assumption is that the history of a software entity influences its defects, i.e. there are several historical characteristics that are indicators for defects in code. For instance, according to an expression, "Many cooks spoil the broth". Is this true for software development, too? Does the number of authors that change software entities influence its defect count? This is one of the questions analysed in this chapter.

The goal of this study can be refined as follows:

Object of study: Analyse different historical characteristics
Purpose: for the purpose of their evaluation
Quality focus: with respect to their efficiency as indicators for defects in software entities
Perspective: from the point of view of practitioners and researchers (above all, testers and quality engineers)
Context: in the context of commercial development.

A main weakness of testing processes as indicated by testers in their organisations is the lack of a systematic approach when defining the test foci. Knowing which particular historical characteristics are good indicators for a file's defect count is useful for *testers*, because they can focus their testing activities on those parts of the software. *Quality engineers* can initiate process improvement activities. For instance, if the number of authors that change a file proves to be a good indicator for defects, process guidelines should be developed that recommend not to share large parts of the code by many developers. In addition, code review activities can be prioritised. Parts of the software that have been changed by many developers would be candidates for such code reviews.

From a *researcher's point of view* it is important to know which historical characteristics are indicators for a file's defect count because (1) it increases the empirical body of knowledge in this area and (2) it enables to develop methods and concepts that consider the results of the empirical study. For instance, researchers can propose development processes that avoid characteristics that lead to poor software quality.

Particularly, the following questions are addressed in this chapter:

- Is the number of authors performing changes to files an indicator for a file's defect count?
- Is the number of changes an indicator for a file's defect count?
- Is the number of co-changed files an indicator for a file's defect count? Co-changed files are those files that are simultaneously changed, for instance because of a defect correction.

- Is the file's age an indicator for its defect count?

The reminder of this chapter is structured as follows. Section 10.2 describes the study design including the context of the organisation and the goals of the study. In addition, Section 10.2 gives further details on the study (the analysed software entities and the granularity level). In Section 10.3, quality indicators are introduced whereas Section 10.4 describes how the number of defects per file is determined. The results of the study are presented in Section 10.5 and discussed in Section 10.6. Threats to validity are described in Section 10.7, an overview of related work is given in Section 10.8. The summary in Section 10.9 concludes this chapter.

10.2 Study design

In this section, details on the study design are given including the context of the organisation, the main characteristics of the software that is analysed, as well as the granularity of the study.

10.2.1 Organisation context

The organisation in which the study is performed provides real-time system solutions. Testers are organised within an independent testing group, whereas the ratio of testers to developers is 1:4. The testing process is basically organised according to the fundamental testing process as described in (Spillner and Linz 2010).

10.2.2 Software entities

According to the criteria defined in Section 6.2, a software system with the following characteristics is selected for this study:

- *Size:* The system consists of about 180.000 LOC and 1.550 files.
- *Maturity:* The system matured over several years (since 2002).
- A *documented history* is available within a VCS for the source code.
- The *defects* are separately tracked within a commercial defect tracking system.

The core development team consists of five developers; the extended team consists of about twenty developers. Table 10.1 summarises key characteristics of the system, in the following denoted as CS (commercial system).

Commercial system	Since	Defect count[42]	LOC[43]	# Files	# Analysed major releases	Mean time interval between releases (in years)
CS	2002	817	179.075	1.550	4	1,3

Table 10.1 - Characteristics of the analysed CS

10.2.3 Granularity

All analyses in this study are performed on file level, i.e. characteristics of files are related to their defect counts.

10.3 Quality indicators

10.3.1 Dependent variables

In this thesis, the change history as well as the file's age are considered as historical characteristics. (Illes-Seifert and Paech, 2010), (Illes-Seifert and Paech, 2008a/b).

The **change history** of a file comprises the number, size and author(s) of the HTs performed to that file. The following three change history characteristics are considered in this thesis:

- **DA (Distinct Authors):** Number of distinct authors that performed HTs to a file between two consecutive releases.
- **FC (Frequency of Change):** Number of HTs performed per file between two consecutive releases.
- **CF-SUM/AVG (Co-Changed Files):** Total/average number of files that have been conjointly checked in with a file between two consecutive releases.

`Fluctuating` files have been changed by a number of distinct authors above average and **`non-fluctuating`** files have a DA metric that is below average.

`Stable` files have an FC metric below average; **`unstable`** files have an FC metric above average.

Another dependent variable considered in this chapter is a file's **`age`**. According to their age, files are classified into one of the following categories[44]:

- **`Newborn`:** A file is newborn at its birthday.
- **`Young`:** `< 0.5 *` **`SystemAge`**[45] `AND not` **`Newborn`** (all files that are not older than the half of a system's age and that are not classified as **`Newborn`**)

[42] 20% of the files contain about 60% of the defects.

[43] The table presents data from the last analysed release.

[44] The classification of class hierarchy histories presented in (Girba, Lanza, and Ducasse 2005) has been adopted.

[45] See also Section 7.2.

- **`Old:`** `>= 0.5 *` **`SystemAge`** (all files that are older than or equal to the half of a system's age).

Table 10.2 summarises the dependent variables used in this study.

ID	Description
DA	**Distinct Authors** Number of HTs per file performed between two consecutive releases.
FC	**Frequency of Change** Number of HTs per file performed between two consecutive releases.
CF- SUM/AVG	**Co-Changed Files** Total/average number of files that have been conjointly checked in with a file between two consecutive releases.
Age	**`Newborn, young, old`**

Table 10.2 - Independent variables

10.3.2 Research hypotheses

In the following, the research hypotheses are presented.

- **H-CS-1:** Fluctuating files have on average more defects than non-fluctuating files. The rationale behind this hypothesis is that shared responsibility leads to defects since no single person has an overview on the (effects of) changes.
- **H-CS-2:** Unstable files have on average more defects than stable files. The rationale behind this hypothesis is that a large amount of changes indicates that particular parts of the problem are not well understood and often need rework resulting in fault-prone files.
- **H-CS-3:** Files with the CF-metric above average have a higher defect count than files with a CF-metric below average. The rationale behind this hypothesis is that a local change, affecting just a few files, will cause fewer defects than changes affecting more files.
- **H-CS-4:** A file's age is an indicator for its defect count. Particularly, the following sub-hypotheses can be formulated:
    - **H-CS-4.1:** **`Newborn`** and **`young`** files are the most fault-prone files. The rationale behind this hypothesis is that **`newborn`** and **`young`** files represent new features that might be not well understood and consequently more fault-prone than **`old`** files.
    - **H-CS-4.2:** `Old` files have the lowest defect count. The rationale behind this hypothesis is that **`old`** files represent stable functionality which matured over years so that most of the defects have already been removed.

10.4 Preparation - Computing the number of defects per file

In this section, the procedure to determine the number of defects per files is described.

Starting point of the analyses are two systems: the DTS, mainly used by the testers to track the defects and the VCS, mainly used by developers. Each time a developer checks in a set of files as a result of a defect correction, an email generator is activated that generates information about the check-in process and sends this information to other developers and testers (registered to receive this information). The email contains a subject including the corrected defect(s), optionally an informative text (entered by the developer), and a list of the files that have been checked in. Thus, testers, developers, and project managers get the information that a defect has been corrected via email. Unfortunately, the information about the corrected defect(s) and the affected file(s) is not stored in the VCS. Figure 10.1 shows how defects are communicated to the project team.

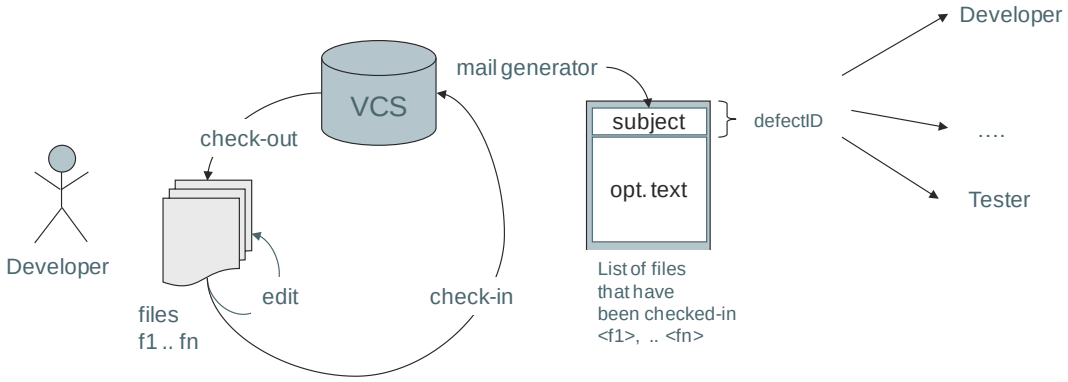


Figure 10.1 - Communication of defects to the project team

In order to determine the defects that occurred in particular files, the email repository has to be analysed. For this purpose, a parser has been developed that analyses the information of the email repository. If the subject of an email contains one or more defect IDs contained in the DTS, the email is further parsed in order to get the list of the files affected by the correction of that particular defect. Consequently, for each defect, a list of affected files can be determined. The date on which the email is sent is used to assign the defect correction activity to a particular release.

98.6% of the defect IDs found in the email repository could be assigned to defects in the DTS. Few defect IDs could not be assigned, i.e. these defects could have been deleted from the DTS or the developer gave a wrong defect ID.

10.5 Analysis and results

10.5.1 Exploring the relationship between a file's defect count and the number of authors performing changes to it

On average, 1.25 distinct authors performed HTs to a file. The minimum count of distinct authors is 1, whereas the maximum count is six authors. Table 10.3 summarises basic statistical characteristics.

Min	Max	Mean	Median	Standard deviation	Variance
1	6	1.25	1	0.939	0.882

Table 10.3 - Descriptive statistics for DA

In order to analyse the relationship between the number of distinct authors and the defect count, simple analyses of defect variance are conducted. For this purpose, the data are divided into *two* groups: one group contains files that have been changed by a number of authors above average (**`fluctuating`** files) and a second group containing files that have been changed by a number of distinct authors below average (non-**`fluctuating`** files). In each group, the mean defect count is computed.

Figure 10.2 shows the corresponding DVA. Accordingly, **`fluctuating`** files are more fault-prone than **`non-fluctuating`** files (factor 2)**.** Non-**`fluctuating`** files have a mean defect count of 4.66, **`fluctuating`** files 9.20.

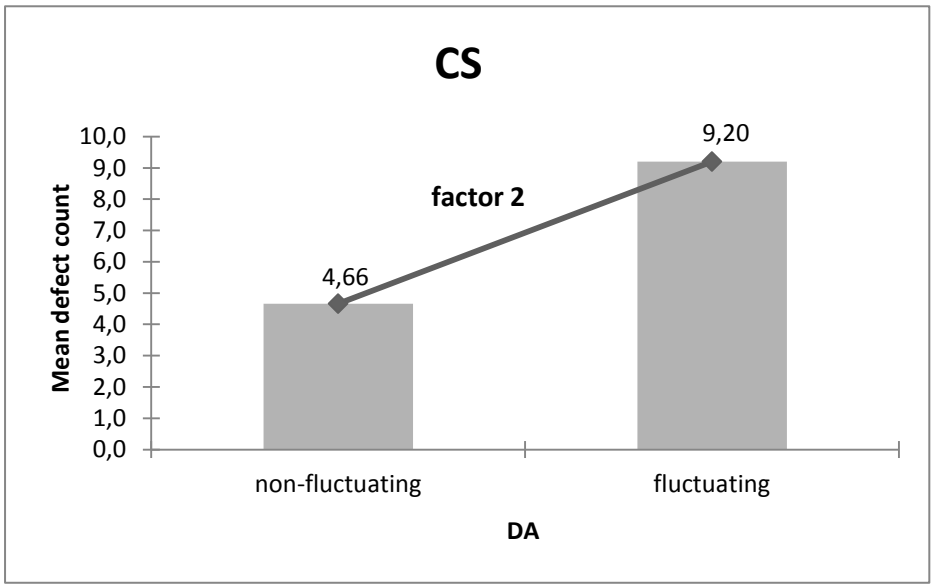


Figure 10.2 - DVA for DA

The Mann-Whitney test shows that there is statistical evidence that fluctuating files are more fault-prone than non-fluctuating files at the 0.02 significance level (Appendix A 11).

The same observation can be made when detailing the categorisation into the following groups.

- Group 1: contains files that have been changed by maximum one author.
- Group 2: contains files that have been changed by two authors.
- Group 3: contains files that have been changed by three or more authors.

A file that is changed by three or more distinct authors is approximately 2.2 times more fault-prone than a file that is changed only by a single author. Figure 10.3 shows the corresponding DVA.

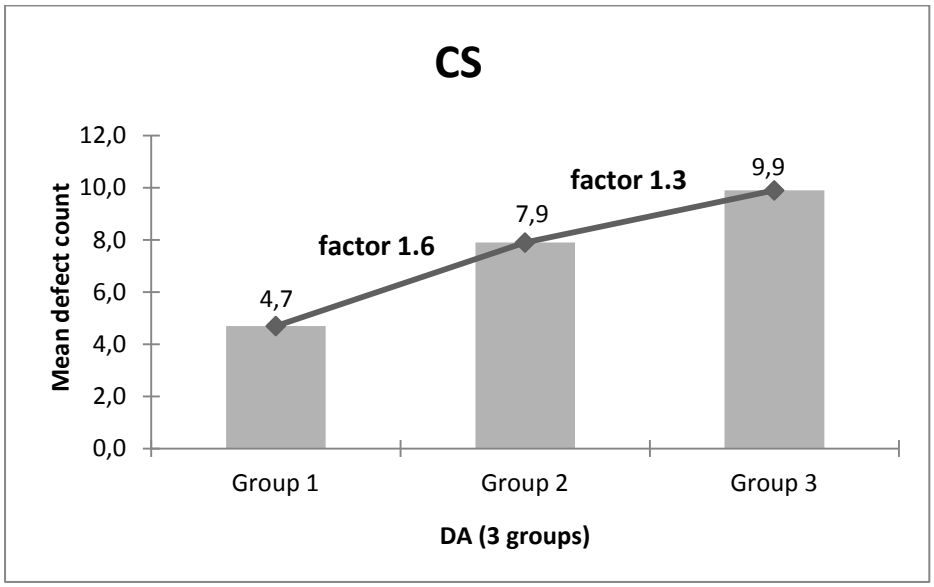


Figure 10.3 - DVA for DA (three groups)

The most fault-prone files are those in Group 3 followed by Group 2 and finally Group 1. The Kruskal Wallis test shows that there is statistical evidence for this observation at the 0.02 significance level (Appendix A 11).

The results obtained by the analyses of defect variance are confirmed by statistical means. The Mann-Whitney test (performed in the first case) as well as the Kruskal-Wallis test (performed for the refined categories) show that the observations made by visual analyses are statistically significant.

Based on these analyses, it can be concluded that there is statistical evidence from the data that fluctuating files have on average more defects than non-fluctuating files in the analysed context. Consequently, the initial hypothesis H-CS-1 can be confirmed.

10.5.2 Exploring the relationship between the frequency of change and the defect count

In order to analyse the relationship between the frequency of change and the number of defects, simple analyses of defect variance are conducted again. For this purpose, the data are divided into two groups, one group containing **`stable`** files and another group containing **`unstable`** files. In each group, the

mean defect count is computed. Figure 10.4 shows the corresponding DVA. Accordingly, **`unstable`** files have 1.8 times more defects than stable files.

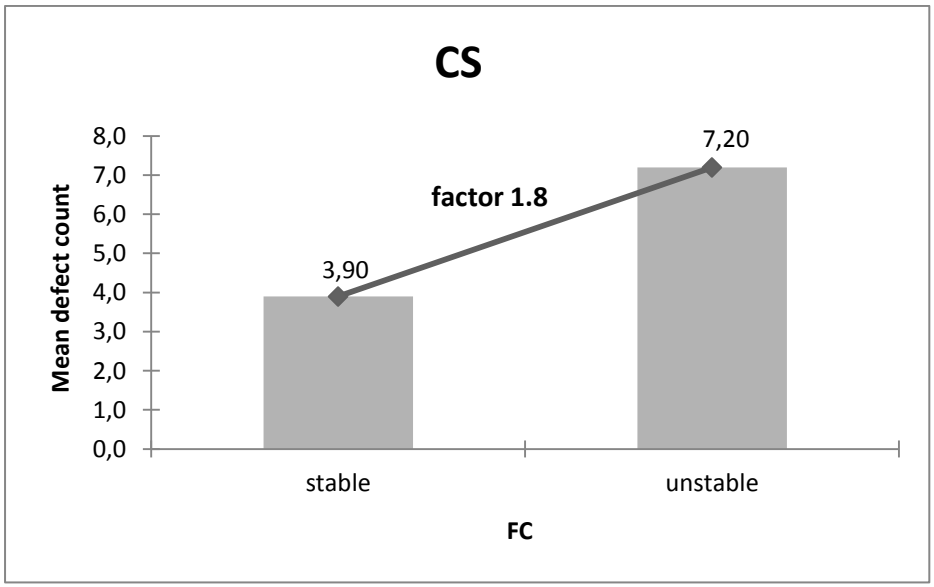


Figure 10.4 - Mean defect count for stable vs. unstable files

The Mann-Whitney test also confirms that there is statistical evidence that unstable files are more fault-prone than stable files at the 0.01 significance level (Appendix A 12).

Based on the results of the analyses, it can be concluded that there is evidence from the data that unstable files have a higher defect count than stable files so that the hypothesis H-CS-2 can be confirmed.

10.5.3 Exploring the relationship between co-changed files and defect count

In order to analyse the relationship between the number of co-changed files and defects, simple analyses of defect variance are performed. For this purpose, the data are divided into two groups: one group contains files that have been conjointly checked in with a number of files above average, and a second group containing files that have been conjointly checked in with a number of files below average. Figure 10.5 shows the DVAs for the CF-SUM and the CF-MAX metrics.

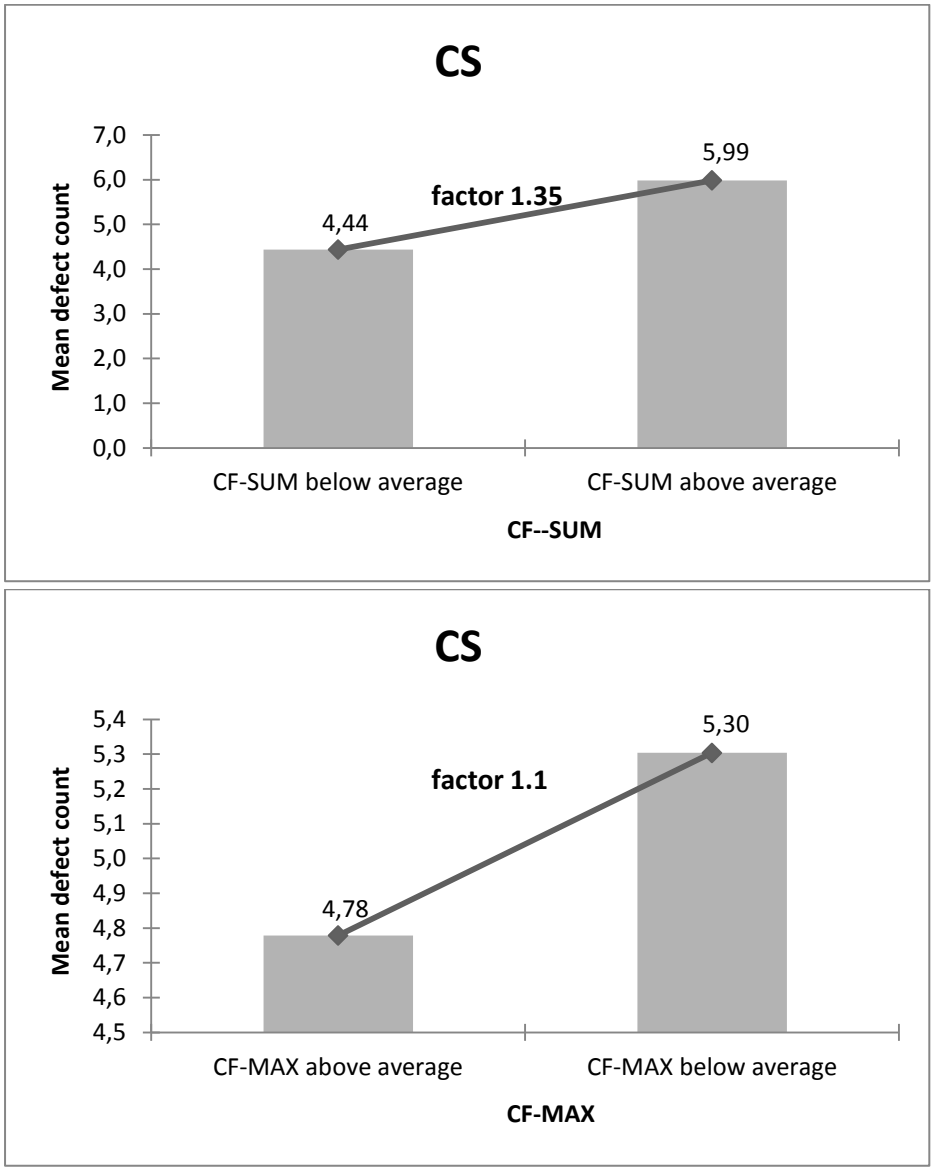


Figure 10.5 - DVA for CF-SUM and CF-MAX

Accordingly, files with a high CF-SUM and CF-MAX metric are more fault-prone than files with a low CF-SUM and respectively with a low CF-MAX metric But the Mann-Whitney test shows that this observation is not statistically significant (Appendix A 13).

It can be concluded that there is no evidence from the data that the number of co-changed files is a good indicator for the file's defect count so that H-CS-3 has to be rejected.

10.5.4 Exploring the relationship between a file's age and its defect count

In order to analyse the relationship between a file's age and its defect count, the data are grouped into three categories: **`newborn`**, **`young`** and **`old`** files. Then, a simple analysis of defect variance is performed in order to answer the question:

Have **newborn** and **young** files on average a higher defect count than **old** files?

Figure 10.6 shows the DVA for the categories: **newborn** (F-N), **young** (F-Y), **old** (F-O). Accordingly, **old** files are about 1.5 times more fault-prone than **young** files. Similarly, newborn files are about 1.3 times more fault-prone than **young** files.

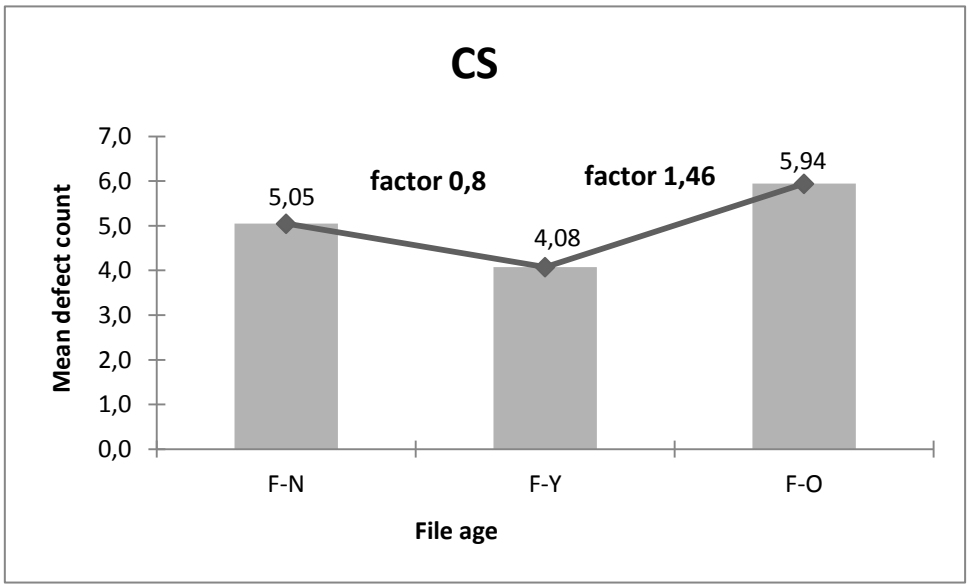


Figure 10.6 - Simple DVA for ANT: Mean defect count vs. file age

The most fault-prone files are **old** files, followed by **newborn** and **young** files. The Kruskal-Wallis non-parametric test shows that this observation is statistically significant at the 0.05 significance level, i.e. based on the data it can be concluded that the differences between the analysed groups are statistically significant (Appendix A 14).

*It can be concluded that there is a slight but statistically significant difference between newborn, young and old files with respect to their mean defect count. Accordingly, the research hypothesis **H4** can be confirmed to some part, a file's age is an indicator for its defect count. In addition, the research hypotheses **H4.1** and **H4.2** must be rejected. Newborn and young files are not the most fault-prone files.*

10.5.5 Combined analyses of defect variance

For more detailed results, further analyses are performed. For this purpose, the initial categories are refined in order to answer the following questions:

1. To what extent does the defect count of a file depend on its stability AND on its fluctuation?
2. To what extent does the defect count of a file depend on its age AND on its fluctuation?
3. To what extent does the defect count of a file depend on its age AND on its stability?

In order to answer the first question, a detailed analysis of the relationship between a file's fluctuation and its stability is performed. For this purpose, the initial categories are refined in order to analyse to what extent the defect count of a file depends on its stability AND on its fluctuation. For example, this combined analysis addresses the question to what extent **non-fluctuating** files that have been changed frequently (these are **non-fluctuating** and **unstable** files) are more fault-prone than **non-fluctuating** files that have not been changed frequently (**old** and **stable** files). Consequently, the mean defect count is related to each of the refined categories presented in Table 10.4-Category definition matrix for stability x fluctuation.

		Stability	
		stable	unstable
Fluctuation	Fluctuating	**F-stab**	**F-unstab**
	Non-fluctuating	**nF-stab**	**nF-unstab**

Table 10.4- Category definition matrix for stability x fluctuation

Figure 10.7 shows the DVA for the refined categories. The lowest mean defect count have **stable non-fluctuating** files (**stab-nF**), the highest **unstable non-fluctuating** files. **Unstable fluctuating** files are about three times more fault-prone than **stable non-fluctuating** files.

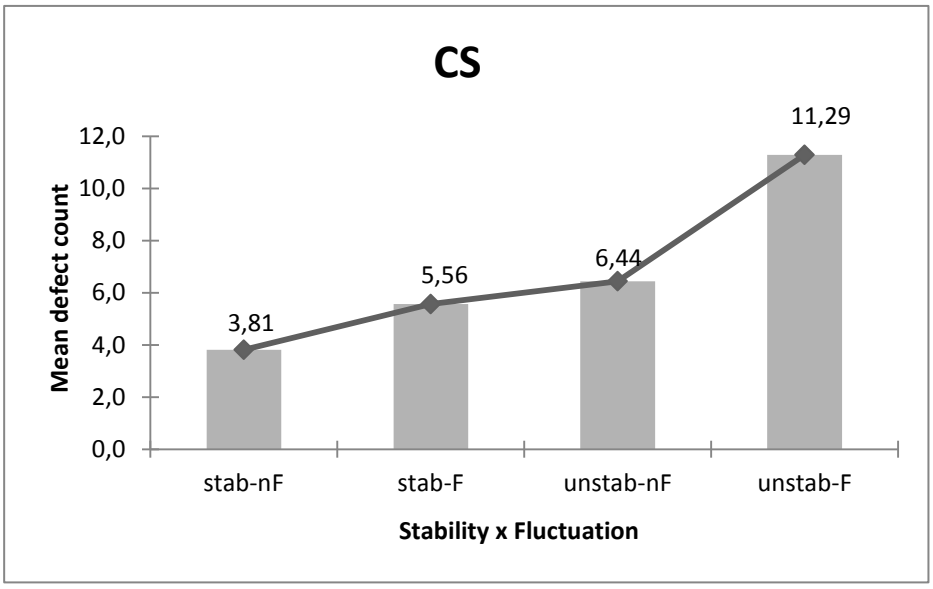


Figure 10.7 - Combined DVA: Mean defect count vs. stability x fluctuation.

stb-nf (61% of all files), stabF (3.2%), unstab-nF (29.6%), unstab-F (5.5%)

The most fault-prone files are the **unstable fluctuating** ones (**unstab-F**) followed by unstable **non-fluctuating** and **stable-fluctuating** files. The Kruskal-Wallis test shows that there is statistical evidence for this observation at the 0.01 significance level (Appendix A 15).

Unstable fluctuating files make up 5.5% of all files, i.e. a very little part of the files are about three times more fault-prone than **stable non-fluctuating** files that make up about 60% of all files.

To answer the second question, the mean defect count is related to each of the refined categories, presented in the matrix in Table 10.5.

		Fluctuation	
		fluctuating	**Non-fluctuating**
Age	**newborn**	**N-F**	**N-nF**
	young	**Y-F**	**Y-nF**
	old	**O-F**	**O-nF**

Table 10.5 - Category definition matrix for age x fluctuation

Figure 10.8 shows the DVA for the refined variables resulting from the age x fluctuation matrix. **Fluctuating old** files are about 1.7 times more fault-prone than **old non-fluctuating** files. Similarly, **young fluctuating** files are about 2.2 times more fault-prone than **young non-fluctuating** files. **Newborn fluctuating** files are about 2 times more fault-prone than **newborn non-fluctuating** files. The most fault-prone files are **old fluctuating** files. These files are about 2.5 times more fault-prone than **young non-fluctuating** files (files with the lowest defect count).

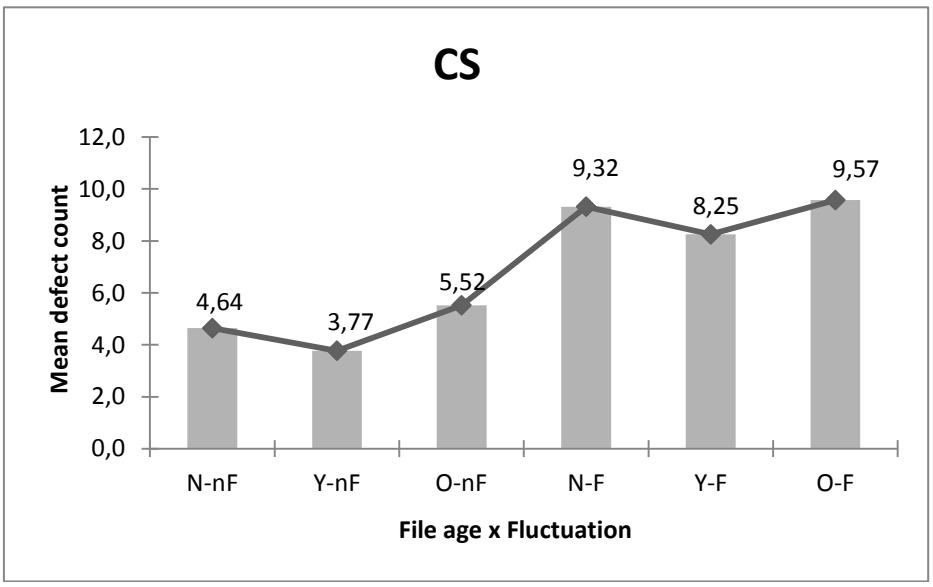


Figure 10.8 - Combined DVA: Mean defect count vs. file age x fluctuation

N-nf (45.7% of all files), Y-nf (21.7%), O-nf (23.9%),
N-F (4.3%), Y-F (1.9%), O-F (2.8%)

Thus, **old fluctuating** files as well as **newborn fluctuating** files are slightly more fault-prone than **young fluctuating** files. Among the **non-fluctuating** files, the most fault-prone files are the **old** ones followed by the **newborn** and **young** ones. The Kruskal-Wallis test shows that there is statistical evidence for these observations at the 0.01 significance level (Appen-

dix A 15). The results also show that a file's fluctuation is a better indicator for its defects than a file's age. This result can be underlined by the fact that 9% of the files (the non-fluctuating files) are about two times more fault-prone than non-fluctuating files independently of their age.

In order to answer the third question, the mean defect count is related to each of refined categories presented in Table 10.6 and then, the DVA is built.

		Stability	
		stable	unstable
Age	newborn	N-stab	N-unst
	young	Y-stab	Y-unst
	old	O-stab	O-unst

Table 10.6- Category definition matrix for Age X Stability

Figure 10.9 shows the DVA for the refined variables resulting from the age x stability matrix. Accordingly, **old unstable** files are about 1.5 times more fault-prone than **old stable** files. Similarly, **young unstable** files are about 2.3 times more fault-prone than **young stable** files. **Newborn unstable** files are about 1.5 times more fault-prone than **newborn stable** files.

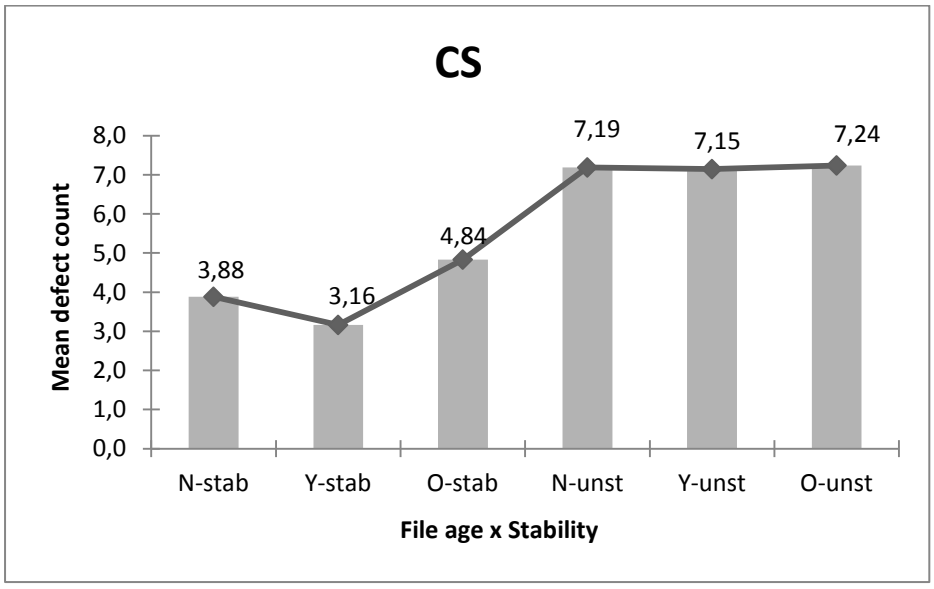


Figure 10.9 - Combined DVA: Mean defect count vs. file age X stability

N-stab (32% of all files), Y-stab (18%), O-stab (15.5%),
N-unstab (17.6%), Y-unstab (5.3%), O-unstab (12.3%)

Basically, there is no difference between unstable files in the different "age categories". Among the **stable** files, **old** and **newborn** files are more fault-prone than **young** files. **Unstable old** files are more fault-prone than **stable old** files, **unstable young** files are more fault-prone than **stable young** files and finally, **unstable newborn** files are more fault-prone than **stable newborn** files. The Kruskal-Wallis test shows that there is statistical evidence for these observations at the 0.01 significance level (Appendix A 15).

About 1/3 of the **newborn** files (the unstable ones) are about 1.9 times more fault-prone than **newborn stable** files. In addition, about 1/4 of the **young** files (the unstable ones) are 2.3 more fault-prone that **young stable** files. About half of the **old** files (the unstable old files) are 1.5 times more fault-prone than **old stable** files.

Nevertheless, the results show that a file's stability is a better indicator for its defects than a file's age.

Considering the variable "age" separately, the results show that there is a difference between newborn, young and old files with respect to their mean defect count. Combined analyses with respect to a file's age and stability show that unstable or fluctuating files are significantly more fault-prone than stable or non-fluctuating files independently of their age. Consequently, it can be concluded that a file's fluctuation and its stability are better indicators for its defects than a file's age.

In addition, combined analyses with respect to a file's fluctuation and stability show that a very little part of the files, the fluctuating and unstable files (5.5%) are significantly more fault-prone than the other files.

10.5.6 Analysis

The results of the analyses show that two of four hypotheses could be accepted. A file's fluctuation is a good indicator for the defect count in the analysed industrial context. One explanation for this observation is that files that are changed by many authors capture too much functionality that is used and changed by many authors. Thus, these files are indicators for bad design leading to a high defect count. A second possible explanation is the lack of responsibility for that particular file that leads to uncoordinated and fault-prone changes so that "too many cooks spoil the broth".

A file's stability is also a good indicator for the defect count. One possible explanation of this observation is a problem domain that is not well understood with often changing requirements.

Best results are obtained by the combined analysis of the variables stability and fluctuation. Accordingly, unstable and fluctuating files are about three times more fault-prone than stable and non-fluctuating files. In addition, fluctuating unstable files make up a very little part of all files and are significantly more fault-prone than all other files.

The hypotheses concerning the file's age could be accepted only partly. Generally, the file's age is an indicator for its defects. There are slight but significant differences in the mean defect counts of **newborn**, **young** and **old** files. But, in contrast to the original hypotheses, the most fault-prone files are **old** files, followed by **newborn** and **young** files. Combined analyses of defect variance show that unstable or fluctuating files are more fault-prone than non-fluctuating files independently of a file's age. Thus, stability and fluctuation are better indicators for defects than the file's age.

Finally, there is no statistical evidence in the analysed context for the hypothesis concerning the relationship between the number of co-changed files and defects.

To sum up, best indicators are the file's stability and fluctuation. In combination, these indicators show that fluctuating files that have been changed frequently are clearly the files with the highest defect count. Concurrently, these files make up a very little part of the system's files.

In order to search for explanations for the observations obtained by analysing historical characteristics of files, structural characteristics can be considered, too. For instance, size and complexity metrics can be analysed in order to explore whether structural characteristics, e.g. the file's size measured by the LOC metric or its complexity are possible explanations for the observation that fluctuating and unstable files are more fault-prone than other files. Is the file's size a possible explanation for its fluctuation? Is the file's complexity a possible explanation for its instability? Is bad structure in terms of e.g. bad smells a possible explanation for its fluctuation? Thus, detailed analyses of defect variance can be performed in order to answer these questions and to get more precise results by combining historical and structural characteristics.

10.6 Discussion

In this section, advantages and disadvantages of analyses on different granularity levels, for instance on file vs. on package level are discussed. In addition, the conclusions drawn by applying the approach in practice are presented.

In all analyses, a simple categorisation has been chosen. The more detailed a categorisation is the more precise are the results. But increasing the analysis granularity means on the other hand that the effort to evaluate the results increases, too. Therefore, a trade-off between a coarse grained (= easy to apply and analyse in practice) and fine grained analysis (= precise results but costly to analyse) has to be performed.

Exploring the history of software projects requires the cleaning up, processing, transformation, analysis, and interpretation of large amounts of data. Thus, measurements that synthesise the evolution of a software entity have to be defined (Girba, Lanza, and Ducasse 2005). For analysing the relationship between a file's defect count and its age, a classification of the data into three groups (**`newborn`**, **`young`** and **`old`**) has been chosen. A more detailed classification would lead to more precise results. However, a simple categorisation has been chosen for the following two main reasons:

a) **Applicability in practice.** The main advantage of the empirical approach presented in Chapter **6** is its applicability in practice. The definition of simple categories supports this approach because the more detailed a categorisation is, the more time-consuming is it to analyse and interpret in practice.

b) **Metaphorisation:** Having less but meaningful categories simplifies the communication and interpretation of the results. It is more difficult to find meaningful names for a higher number (e.g. for 10) of categories.

A trade-off between a coarse grained categorisation (= easy to apply and analyse in practice) and fine grained categorisation (= precise results but costly to analyse) has to be performed. In addition, an existing categorisation can be refined. This can be necessary when for instance current results differ significantly from results obtained in past analyses. Furthermore, the testers' experience may play an important role when deciding to perform detailed analyses. In cases that the results do not reflect the testers' expectations/hypotheses, a detailed categorisation would help to get more precise results.

All analyses presented in this chapter have been performed at file level. The main reason for not performing analyses on a higher level, e.g. on package level, is that a package consists of several very heterogeneous files with respect to their age, number of authors performing HTs, etc. Consequently, an aggregation (by computing the sum, maximum, average, or median) is difficult and looses too much information. An aggregation is for instance best suited for the lines of code metric (LOC). The total LOC of a package has a "meaning" and can be computed by summing up the LOC-metrics of the files/classes contained in it.

The empirical approach presented in Chapter **6** proved of value in practice. The concept has been easily understood by testers. Above all, the DVAs allow a quick overview of the data and refuted the concern of the testers that the approach is not easy to be understood. Testers reported that they assumed that parts of the old code cause problems. But now, they have the evidence for this based on the data. The results underline how important it is to have a justification for subjective impressions. But they also show that there are other indicators for a file's defect count not considered by testers yet and that it is worthwhile to combine experience and facts in order to determine indicators for defects.

Another lesson learned is that the data collection procedure is time-consuming. Reconstructing information from past is much more complex and inconvenient than when relevant information is collected and connected at creation time. Another recommendation with a great benefit that is easy to realise is the connection of the VCS and the DTS. The restriction, that code can be checked in only with a valid requirement or defect ID, allows a more efficient analysis, since it is easy to relate defects to corresponding files.

10.7 Threats to validity

Similar to the open source context, a threat to validity is the problem of collective check-ins (Section **7.7.5**). A collective check-in refers to HTs where a set of files is checked in after a developer has removed two or more defects. In this case, an email containing more than a single defect ID but several referenced

files is sent. Thus, collective check-ins are a threat to validity and can lead to imprecision in the defect count.

Another threat to validity is that a developer corrects a defect, checks in the files but does not specify the corresponding defect ID within the email generator. Since the developers do not use the DTS to set the status of a defect, sending an email is (beside direct communication) a comfortable way to notify testers that particular defects have been corrected and that retesting activities can be started. Thus, developers are motivated to insert the (correct) defect ID and to send the notification email as soon as the defect has been corrected. This fact diminishes the threat to validity.

10.8 Related work

In this section, related work is presented. There are several other studies that focus on predicting the defect count of a software entity by combining product metrics and historical metrics: (Graves et al. 2000), (Arisholm and Briand 2006), (Khoshgoftaar, Seliya, and Sundaresh 2006), (Ostrand, Weyuker, and Bell 2004), (Bell 2005), (Schröter et al. 2006), (Ohlsson et al. 1999), (Pighin and Marzona 2003), (Zimmermann, Premraj, and Zeller 2007). In contrast to the study presented in this chapter, the aim of the published studies is defect prediction. However, the main goal of this study is to analyse the extent to which historical characteristics are good indicators for the software's defect count without selecting the best prediction model.

In (Graves et al. 2000), (Khoshgoftaar et al. 1998), (Ostrand, Weyuker, and Bell 2005), (Bell 2005), (Ohlsson et al. 1999), and (Pighin and Marzona 2003) **age** is used as an independent variable but the definitions used in these studies differ from the classification presented in this chapter. For instance, in (Graves et al. 2000) and (Gyimothy, Ferenc, and Siket 2005) only two file categories are defined: “new” and “pre-existing in a previous release”. In (Fenton and Pfleeger 1998), the age of a file is measured by the number of previous releases in which that file appeared, whereas in (Fischer, Pinzger, and Gall 2003) the age is measured in months. All these studies confirm the hypothesis stated in this chapter that age is an indicator for a file’s defect count. But the results differ partly from those obtained in this study. Independent of the measures used for a software entity’s age, the studies report that the younger a file is, the higher is its defect count. In contrast, the results of this study show that newborn and old files are the most fault-prone ones. One possible cause for such different results is the fact that the design or architecture does not support local changes. Each new functionality induces changes that affect old code and thus lead to defects.

Except the study reported in (Schröter et al. 2006), all other studies (Arisholm and Briand 2006), (Graves et al. 2000), (Khoshgoftaar et al. 1998), (Ohlsson et al. 1999), (Ostrand, Weyuker, and Bell 2005), (Bell 2005), (Weyuker, Ostrand, and Bell 2007), (Schröter et al. 2006) support the finding of this chapter with respect to the relationship between the number of changes performed to a software en-

tity and its defects. In (Schröter et al. 2006), only pre-release defects correlate with the number of changes performed to software entities. The authors define pre-release defects to be all defects found six months before release.

The studies presented in (Bell 2005), (Weyuker, Ostrand, and Bell 2007), (Schröter et al. 2006), and in (Graves et al. 2000) analyse the relationship between the number of authors performing changes to files and the software's defects. The study reported in (Weyuker, Ostrand, and Bell 2007) confirms the results presented in this chapter. In (Schröter et al. 2006), only pre-release defects correlate with the number of authors performing changes. The results reported in (Bell 2005) and in (Graves et al. 2000) differ from the results of this study. It can be concluded that the suitability of the metric "number of authors" as indicator for defects in code highly depends on the analysed context so that it has to be analysed in each context whether it is applicable or not. Possible influencing factors could be communication characteristics, the team size or the process model used.

The relationship between the number of co-changed files and defects is not reported in any study. Most studies analysing this relationship are more fine-grained, i.e. they analyse the extent to which the number of changed lines of code impacts on the defect count, for example in (Nagappan and Ball 2005), (Layman, Kudrjavets, and Nagappan 2008) and in (Purushothaman 2005). The results of these studies are inconsistent. In (Nagappan and Ball 2005) and (Layman, Kudrjavets, and Nagappan 2008), code churn metrics are reported to be good indicators for defects whereas the results reported in (Purushothaman 2005) show that there is a low probability (< 4%) that a change concerning a single line has defects.

Other related research considers structural characteristics of software, e.g. its size or complexity and explores their relationship with defects in code. An overview on this kind of related work is given in Chapter 9.6.

10.9 Chapter summary

In this chapter, the relationship between a file's historical characteristics and its defect count has been investigated (Illes-Seifert and Paech, 2010), (Illes-Seifert and Paech, 2008a/b). The results show that the software's history is a good indicator for its quality expressed in terms of the number of defects.

Particularly, good indicators for defects are the file's fluctuation and its stability. **Fluctuation** categorises files with respect to the number of distinct authors that performed changes to it; fluctuating files have been changed by a number of authors above average whereas non-fluctuating files have been changed by a number of authors below average. The analyses show that fluctuating files are more fault-prone than non-fluctuating files. Possible explanations are the lack of responsibility for a piece of code or its bad structure.

Stability categorises files with respect to their change frequency; unstable files have been changed frequently (above average), stable ones below average. The analyses show that unstable files are more fault-prone than stable ones. This

observation indicates that particular parts of the application are not well understood and often need rework. Consequently, these files are fault-prone.

The empirical results do not support all hypotheses concerning the relationship between a file's age and its defect count. In fact, a file's age is an indicator for its defect count. There are slight but significant differences in the mean defect counts of **`newborn`**, **`young`** and **`old`** files. But in contrast to an intuitive expectation, **`old`** files proved to be the most fault-prone files.

Detailed analyses can be performed in order to get more precise results and to restrict the set of fault-prone files. By analysing different indicators *in combination*, more detailed results can be derived. Such a detailed analysis should be performed in order to specify the results obtained by a simple analysis. In this study, the relationship between a file's stability, its fluctuation and the defect count has been analysed. The results show that **`unstable fluctuating`** files are the most fault-prone files. Consequently, the file's stability and its fluctuation are the best indicators for defects in the analysed context. In addition, unstable and fluctuating files make up a very small part of the system's files so that the set of fault-prone files could be constrained.

Knowing which particular historical characteristics are indicators for a file's quality (e.g. expressed by its defect count) is useful for different roles in the development process. Testers can focus their testing activities on files they expect to be faulty, for instance **`unstable and fluctuating`** files. Quality engineers can monitor development activities and initiate reviews, for example for often changed files in order to prevent a high defect count. Additionally, **`old`** files that have been often changed by a number of authors above average cause high defect counts and can therefore be indicators for bad design. Thus, maintainers can identify candidates for refactoring.

In order to search for explanations for the observations obtained by analysing historical characteristics, structural characteristics of files can be also considered. For example, the code's size or its complexity can be analysed. The following questions are of interest in this context: Are unstable and fluctuating files large files? Have unstable and fluctuating files a high complexity metric? Etc.

All analyses presented in this chapter have been performed by applying the empirical approach presented in Chapter **6** that gives guidance in finding indicators for (poor) software quality. Since the analyses' results are not encrypted within complex formulae, the approach is easy to understand and to apply. In addition, visual representations for the analyses have been used. Thus, a standardised intuitive interpretation of the results is possible. All results obtained by visual means are statistically validated. Consequently, more reliable decisions can be made because the probability of accidental effects is minimised.

The study helped testers to justify their presumptions by facts. In addition, based on the results of the study several improvements of the development process could be proposed. Finally, the study shows how complex it is to collect and reconstruct information from the past and motivates a goal-oriented meas-

urement. Having a goal in mind, all data that have to be collected can be provided at creation time. In addition, the infrastructure can be extended appropriately in order to allow a (semi-)automatic collection of the relevant data.

CHAPTER 11 Synopsis

In this chapter, a review of the main contributions of this thesis is given. In addition, the contributions are related to the main goals of the thesis. Finally, an overview of future research directions is given.

11.1 Summary and conclusions

During the lifecycle of a software, large amounts of data are recorded within a variety of tools, for instance in defect tracking or versioning control systems. This information documents the whole lifecycle of the software. But in order to be able to draw conclusions from the data and to support the decision making process, the data have to be analysed thoroughly and purposefully. This is often not the case in practice and applies also to software testing. Since testing resources are limited, testers have to decide which parts of the software to test, (the *test foci*) and which not.

But how to decide what to test? Which parts will have defects? In practice, the available resources are usually uniformly distributed among all parts of the software with the risk that parts which really contain defects are not tested enough, whereas mature parts that contain no defects are tested too intensively. In the case that test foci are defined in practice, this decision is based on testers' experience rather than on facts. Although experience is important in testing, testers report the lack of a systematic approach when deciding on test foci (Illes-Seifert and Paech 2008).

In literature, there are two basic kinds of approaches that support the decision on test foci. On the one hand, text books that propose heuristics for defects, for instance parts of the software developed by a distributed team, new components, new technology, etc. The main drawback of these heuristics is the lack of empirical validation. In addition, testers have to select those aspects that they think to be applicable in their context. On the other hand, another piece of research work focuses on the development of more and more complex algorithms for defect prediction like neuronal nets or decision trees. The main drawback of these approaches is that they are not applicable in practice. This is the case for the following main reasons:

a. The approaches propose complex algorithms that are not easy to use in practice because they are difficult to understand.

b. The complex formulae hinder that the nature of the detected relationships is understood.

c. There is no empirical validated consensus over the superiority of one model over another.

This thesis has two main goals. First, it aims to contribute towards a systematic approach for the selection of the test foci that is applicable in practice. Second, it aims to increase the empirical body of knowledge in the area of empirically validated indicators for defects in code.

The main results of this thesis that contribute to achieve these goals are summarised in Figure 11.1 and are detailed in the rest of this section.

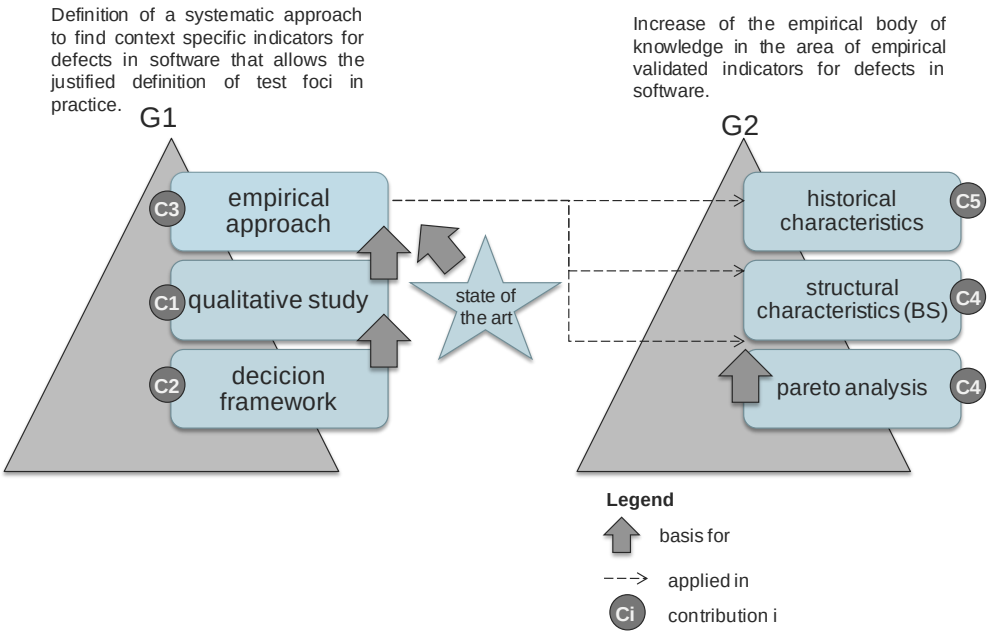


Figure 11.1 - Results of the thesis

The state of practice concerning testing processes in industry has been analysed in a qualitative study (Contribution 1)[46]. The study has been performed based on a decision framework that structures the testing process by a decision hierarchy (Contribution 2)[47]. The results of the qualitative analysis as well as the evaluation of the state of the art motivate the empirical approach for the test focus selection (Contribution 3)[48]. Mature sciences have a solid empirical body of knowledge in common. The state of the art in empirical software engineering is still immature. Thus, extensive experimentation on indicators for defects in software has been performed (Contribution 4)[49] and (Contribution 5). On the one hand, the empirical studies contribute to the enrichment of the empirical body of knowledge in the area of empirically validated indicators for defects in software. On the other hand, they serve to validate the empirical approach. Particularly, the studies explore the empirical evidence for the Pareto-Principle as well as the usefulness of structural and historical characteristics of software as indicators for defects. The empirical studies that make up Contribution 4 have been performed in the context of open source development. The empirical study that makes up Contribution 5 has been performed in an industrial setting.

Contribution 1 – Qualitative analysis of the testing process. The main *goal* of the qualitative analysis is to identify strengths and weaknesses of testing processes in practice in order to develop solutions that address the main problems

[46] (Illes-Seifert and Paech 2008)

[47] (Borner, Illes, and Paech 2007), (Borner, Illes-Seifert, and Paech 2007)

[48] (Illes-Seifert and Paech 2010)

[49](Illes-Seifert and Paech 2008a), (Illes-Seifert and Paech 2008b), (Illes-Seifert and Paech 2009), (Illes-Seifert and Paech 2010)

identified by testers. The analysis is performed as an interview study with experienced testers. The *benefit* of a qualitative study is that it gives a detailed picture of complex characteristics and issues in software testing. The *results* of this study show the state of practice with respect to testing processes. Particularly, testers emphasise that testing activities require system specific experience. In addition, one of the main challenges for testers is the definition of the test foci. From the researcher's point of view, the results are important because they highlight issues encountered in practice that should be considered in research. Particularly, it is important to consider the "man in the loop" when developing new methods, tools, and processes. In addition, the study shows how important it is to involve practitioners when developing new approaches. The results of this analysis served as the basis for this thesis as it addresses the main issues mentioned by testers.

Contribution 2 – Decision based framework for the characterisation of test processes. The main *goal* of developing the decision based framework for the characterisation of test processes is to structure the testing process from the point of view of the decisions to be made during it. The main *benefit* of a decision based view of processes in general, and of the testing process in particular is that the awareness of decisions to be made increases their quality. It forces the decision-makers, in this case the testers, to search for alternatives and to trade off between them. The *result* is a decision hierarchy that comprises all decisions made during testing and reflects dependencies between them. The hierarchy is useful for researchers and testers. From the practitioner's point of view, the hierarchy is useful because it highlights decisions that are often made implicitly and that therefore are of poor quality. In addition, practitioners get a deeper understanding of the complex decision making process during testing. Thus, the hierarchy can be used as an introducing guideline to the complex area of testing processes. From the researcher's point of view, the decision hierarchy is useful, too. First, it enriches the body of knowledge on the subject of decision-making in the area of testing and builds the foundation for further research in the area of rationale management. Rationale management research aims at making design and development decisions explicit to all stakeholders. As shown in this thesis, the decision hierarchy can be used by researchers as an evaluation framework in many contexts.

Contribution 3 – Empirical approach for justified definition of the test foci. The main *goal* of the empirical approach is to provide a systematic procedure for the justified selection of the test foci that is applicable in practice. The *resulting* approach describes how to identify indicators for defects in software. It proposes a combination of statistical procedures and visual representations in order to analyse the program's structure and its history and to search for empirically validated indicators for defects in software. For first exploratory analyses that aim to explore the data, simple analyses of defect variance are proposed. The visualisation occurs in terms of a DVA (defect variance analysis) diagram. Then, detailed analyses are performed in order to get more precise results. By combined analyses of defect variance, the relationship between more

indicators and defects in software is analysed. Again, an adjusted DVA is used to visualise the results. The approach showed the following *benefits* from the researchers' as well as from the practitioners' point of view*:*

a) It is easy to understand and to use (due to visual representations);

b) It is based on facts (statistical significance of the results is required);

c) It is experience based, i.e. the approach involves testers in the selection and validation of indicators for defects in software.

Contribution 4 – Extensive experimentation. The empirical approach is validated in the context of seven large open source programs. The main *goals* of the empirical analyses are the validation of the empirical approach on the one hand and the enrichment of the empirical body of knowledge in the area of empirical validated indicators for defects in software on the other hand. The results show that the approach is general enough to be applied in order to determine structural and historical indicators for defects. It is also specific enough to highlight indicators for defects in software for each analysed open source program. In the following, the main contributions that enlarge the empirical body of knowledge are presented.

4.1 PARETO-Analysis

The main *goal* of the PARETO-Analysis is to analyse whether a small part of the software contributes to most of the defects. The *results* show that a small number of files accounts for the majority of the defects. This result is confirmed by a high number of other empirical studies on this topic. But, there is no evidence that a small part of the system's code size accounts for the majority of the defects. This result is also supported by other researchers. Apart from one study, the analysis of the Pareto principle across several releases has not been focused by researchers so far. Consequently, there is some empirical evidence that the Pareto principle holds for all releases of software. Table 11.1 summarises the hypotheses considered in this thesis, as well as the results along with a comparison to related work. The first column indicates the ID of the corresponding hypothesis, the second column its description. The third column shows the results of the empirical analysis presented in this thesis. The last two columns indicate the results obtained by other researchers as well as the amount of empirical research that has been conducted on the corresponding topic.

To sum up, defects concentrate on a small part of the files but not on a small part of the code. From the *practitioner's point of view* it can be concluded that testers and maintainers need additional indicators to prioritise testing and maintenance activities. In addition, testers can use the results of the Pareto analysis in order to select parts of the code for which they require an intensification of the unit testing coverage criteria. From the *researchers' point of view* it is important to consider that algorithms that determine the most fault-prone files, like those presented in (Kim et al. 2008), are only useful when considering the

amount of code covered by these files. Finally, further empirical studies that address the hypotheses P2 and P4 have to be conducted in order to gain more empirical evidence for the validity of the Pareto principle across several releases.

H ID	Hypothesis Description	Result	Evidence from related work	
		☑ ☒ ☐	Result ☑ ☒ ☐	Number of studies ⇧ ⇩ ⇔
P1	A small number of files accounts for the majority of the defects.	☑	☑	⇧
P2	If P1 applies to one release, then it applies to all releases of a software project.	☑	☑	⇩
P3	A small part of the system's code size accounts for the majority of the defects.	☒	☒	⇔
P4	If P3 applies to one release, then it applies to all releases of a software project.	☒	☒	⇩

Table 11.1 - Pareto analysis summary

Legend

☑ Hypothesis is confirmed.
☒ Hypothesis is rejected.
☐ Hypothesis is partly confirmed.
⇧ A high amount of study exists to the corresponding topic (>10).
⇩There are very few studies to the corresponding topic (0-3).
⇔There are some studies to the corresponding topic (4-9).

4.3 BAD SMELL-Analysis

The main *goal* of the BAD SMELL-Analysis is to explore whether entities for which particular bad smells apply are more fault-prone than entities for which bad smells do not apply. The *results* show that there are some bad smells that are good indicators for defects, whereas the God Class (GC) bad smell is the best indicator for a class' defects. On average, files containing classes for which the GC bad smell applies are 6 times more fault-prone than files that do not contain classes for which the GC bad smell applies. Apart from the study presented in (Shatnawi and Li 2006), there are no empirical results on this topic. The study presented in (Shatnawi and Li 2006) considers only a part of the bad smells analysed in this thesis. For this part, the results are similar to the results obtained in this thesis.

From the *practitioners' point of view* it is important to know which bad smells are good indicators for defects for several purposes. First, testers can use these indicators to define the test foci and maintainers can prioritise refactoring activities not only based on factors like understandability, changeability etc., but also based on analyses on fault-proneness, i.e. if parts of the software for which a bad smell applies are more fault-prone than parts for which a bad smell does not apply, maintainers can use this information to prioritises maintenance ac-

tivities. Knowing which bad smells are indicators for defects in code is also useful for developers. The integration of "smell detectors" in their programming environment enables early warning on possible defects.

From the *researchers' point of view* it is important to replicate empirical studies focusing on the relationship between bad smells and defects in code. Apart from one study, this research area has been neglected in research.

Table 11.2 summarises the hypotheses formulated for the BAD SMELL ANALYSIS along with the results obtained as well as a comparison to the study presented in (Shatnawi and Li 2006).

H ID	Hypothesis Description	Result	Results obtained by (Shatnawi and Li 2006)
A-FE Feature Envy	A file with at least one method for which the FE bad smell applies is more fault-prone than a file that has no methods with this bad smell.	☐	
A-GM God Method	A file with at least one method for which the GM bad smell applies is more fault-prone than a file that has no methods with this bad smell.	☑	☑
A-GC God Class	A file containing at least one class for which the GC bad smell applies is more fault-prone than a file that has no class for which the GC bad smell applies.	☑	☑
A-SS Shotgun Surgery	A file containing at least one class for which the SS bad smell applies is more fault-prone than a file that has no class for which the SS bad smell applies.	☑	☑
A-RB Refused Bequest	A file containing at least one class for which the RB bad smell applies is more fault-prone than a file that has no class for which the RB bad smell applies.	☐	☒
A-MC Misplaced Class	A file containing at least one class for which the MC bad smell applies is more fault-prone than a file that has no class for which the MC bad smell applies.	☒	-
A-DC Data Class	A file containing at least one class for which the DC bad smell applies is more fault-prone than a file that has no class for which the DC bad smell applies.	☒	☒
A-LoSta Lack of State	A file containing at least one class for which the LoSta bad smell applies is more fault-prone than a file that has no class for which the LoSta bad smell applies.	☑	-
A-ISP ISP Violation	A file containing at least one class for which the ISP bad smell applies is more fault-prone than a file that has no class for which the ISP bad smell applies.	☑	-
A-LoStr Lack of Strategy	A file containing at least one class for which the LoStr bad smell applies is more fault-prone than a file that has no class for which the LoStr bad smell applies.	☐	-
A-LoV Lack of Visitor	A file containing at least one class for which the LoV bad smell applies is more fault-prone than a file that has no class for which the LoV bad smell applies.	☒	-
A-LoB Lack of Bridge	A file containing at least one class for which the LoB bad smell applies is more fault-prone than a file that has no class for which the LoB bad smell applies.	☒	-
A-GP God Package	A package for which the GP bad smell applies is more fault-prone than packages for which the GP bad smell does not apply.	☑	-
A-WSI Wide System Interface	A package for which the WSI bad smell applies is more fault-prone than packages for which the WSI bad smell does not apply.	☒	-

Table 11.2 - Bad smell analysis summary

Legend

☑ Hypothesis is confirmed.
☒ Hypothesis is rejected.
☐ Hypothesis is partly confirmed.
- Hypothesis not analysed in literature so far.

Contribution 5

The empirical approach presented in Chapter 6 has been applied in an industrial context. The main *goals* of this study are:

a) to increase of the empirical body of knowledge and

b) to validate the approach in practice.

In the following, the results of the HISTORY-Analysis are summarised along with the lessons learned from the application of the empirical approach.

4.2 HISTORY-Analysis

The main *goal* of the HISTORY-Analysis is to investigate whether historical characteristics indicate defects in software. The *results* show that there are some historical characteristics that are good indicators for a file's defect count and consequently, these characteristics are good indicators for the selection of the test foci. Particularly, the file's fluctuation and the file's stability proved to be good indicators in the analysed industrial context. By combining these indicators, more precise results could be obtained and the set of fault-prone files can be restricted. Accordingly, fluctuating unstable files proved to be significantly more fault-prone than the other files in the analysed context.

The file's age can be used as indicator for defects, but the file's fluctuation and its stability proved to be better indicators in the analysed context. The results of the study also show that there is little evidence from the data that the number of co-changed files is a good indicator for a file's defects.

The hypothesis concerning the file's fluctuation is confirmed by results in literature only partly. In contrast, the FC metric is confirmed by a high number of studies as a good indicator for defects in software. For the variable "age", there are too few empirical studies to be able to derive evidence in favour or against one indicator. In the case of the number of co-changed files, there is no empirical study that addressed the evaluation of this particular characteristic.

From the *practitioner's point of view* it can be concluded that there exist historical characteristics that are indicators for defects in software. Since results are often inconclusive, it is important to apply the empirical approach *in context* in order to determine which indicators apply in the own organisation or project. The best indicator that is also supported by a high number of other studies is the frequency of change. This is a good starting point for prioritizing testing activities.

From the *researchers' point of view* it is important to replicate empirical studies in order to get a deeper understanding of factors that influence the software's defect count and to build a reliable empirical body of knowledge.

Table 11.3 summarises the hypothesis formulated for the HISTORY ANALYSIS along with the results obtained as well as a comparison to similar studies reported by other researchers.

H ID	Hypothesis Description	Result	Evidence from related work	
		☑ ☒ ☐	Result ☑ ☒ ☐	Number of studies ⇧ ⇩ ⇔
H-CS-1	Fluctuating files have on average more defects than non-fluctuating files.	☑	☐	⇔
H-CS-2	Stable files have on average more defects than unstable files.	☑	☑	⇧
H-CS-3	Files with a number of co-changed files above average are more fault-prone than files with a number of co-changed files below average.	☒	-	⇩
H-CS-4	A file's age is an indicator for its defect count.	☑	☐	⇔
H-CS-4.1	Newborn and young files are the most fault-prone files.	☒	☐	⇔
H-CS-4.2	Old files have the lowest defect count.	☒	☑	⇔

Table 11.3 - History analysis summary

Legend

☑ Hypothesis is confirmed.
☒ Hypothesis is rejected.
☐ Hypothesis is partly confirmed.
- Hypothesis not analysed in literature so far.
⇧ A high amount of study exists to the corresponding topic (>10).
⇩There are very few studies to the corresponding topic (0-3).
⇔There are some studies to the corresponding topic (4-9).

The study performed in an industrial context also aims to validate the empirical approach. From the *practitioners' point of view*, this study shows several *benefits*. First, the study helped testers to justify their presumptions by facts. This approach helps them to prioritise testing activities and to select the test foci for testing new functionality but also to select the test foci for regression testing. In addition, based on the results of the study, several improvements of the development process could be proposed above all concerning a better tool support for data collection and a purposeful selection of code reviewing activities. One improvement particularly concerns the adjustment of the VCS so that it is mandatory to indicate the defect ID (or the requirement ID) when code is checked in. With this adjustment, a better analysis is possible, since it is easy to relate defects to corresponding software entities.

From the *practitioners' and researchers' point of view*, the study shows how time-consuming it is to collect and reconstruct information from the past and motivates a goal-oriented measurement. Having a goal in mind, the needed infrastructure for a (semi-)automatic data collection can be provided. Thus, the collection of the data at creation time is facilitated and avoids the time-consuming re-construction of lost information.

11.2 Future research directions

The first goal of this thesis is the definition of an empirical approach to find context specific indicators that allow the justified selection of the test foci. The approach presented in Chapter 6 is an important step towards this goal.

Possible further improvements include the following aspects:

- The development of concepts for tool support containing data extractors, data analysis components, as well as a visualisation dashboard that allow testers to quickly obtain information relevant for making testing decisions based on data contained in several systems, for instance in defect tracking systems, versioning control systems, etc. A high degree of automation of the data collection and analysis process is prerequisite for this approach to be used in practice.
- In addition, tool support should be developed that gives immediate feedback to developers about parts of the software that could contain defects. This enables fast feedback to developers and prevents defects.
- The approach presented in this thesis addresses the justified selection of the test foci. In future research, this approach can be generalised and evaluated for any kind of quality characteristics like maintainability, testability, etc.
- In addition, in this research work, the main focus is on the analysis of historical and structural characteristics of the software's code. Further research should consider other characteristics. For instance, structural characteristics of other artefacts like the requirements specification can be analysed. In this context, questions like "Does a complex requirement lead to more fault-prone software components than a simple requirement?" should be considered. Furthermore, additional historical characteristics of the whole software development process, for instance the history of a requirement, should be addressed. In this context, questions like "Does an often changed requirement lead to more fault-prone software components than a stable requirement?" should be analysed.
- One issue mentioned by testers that is not addressed in this thesis concerns the poor quality of the requirements specification. Research in the area of requirements engineering should consider the testers as stakeholders of the requirements specification and integrate them into the process of requirements specification and validation.

The second goal of this thesis is to enrich the empirical body of knowledge in the area of empirical validated indicators for defects in software. The extensive empirical studies comprising several releases of seven open source programs as well as a commercial system are a step towards this goal. Nevertheless, in order to be able to have empirical evidence, experimentation in this area should be intensified. For instance, apart from one empirical study, there is no empirical work on the relationship between bad smells and defects in literature.

The approach presented in this thesis proved of value when applied in practice. It shows strong indications for its feasibility and efficiency. Nevertheless, research work to be done in future must concentrate on putting the approach into practice within different organisations and project contexts (e.g. organisations that differ in size, application domain, development processes, etc.). This will make it possible to improve the approach based on the effects that its usage shows within several project environments.

In future, the awareness of the importance of making empirically justified decisions (e.g. for test foci definition but also in general, when deciding between alternative methods and tools) will increase. Simultaneously, more and more heterogeneous tools will be developed that produce an immense amount of data. Thus, it will become more and more important for researchers and practitioners to be able to analyse the data produced during the lifecycle of the software purposefully and efficiently. This thesis proposes a generic approach that addresses some of the issues that arise when large amounts of data have to be collected and analysed in order to make justified decisions on the test foci.

APPENDIX

A 1 Validation of the decision hierarchy

The results of the case studies performed to validate the decision hierarchy presented in Chapter 4 are summarised below.

A 1.1 Refinement

The presented hierarchy is generic, i.e. it is independent from the testing level (e.g. system testing level or unit testing level). But the hierarchy can also be refined in order to identify the specific issues and decisions by instantiating the generic decision hierarchy. To illustrate the refinement, the decision hierarchy has been applied to the system testing process (STP)

Figure A. 1.1 shows the refined decisions. It illustrates all decision levels (left column) as well as the corresponding decisions of the generic testing process (middle column) and the specific decisions of the system testing process (right column). Specific decisions in the right column refine corresponding decisions of the generic testing process at the same decision level. This is illustrated in Figure A. 1.1 by using two labels within one "decision box". The upper label of a box describes the decision of the generic testing process. The lower label specifies the corresponding specific decision of the STP.

Within the STP, decisions concerning the test basis and test focus are refined. These are functional and quality requirements within the STP in order to decide on critical parts to be tested. On test approach level, decisions on model coverage and the degree of automation refine the generic decisions.

At test design level, the kind of external systems and the automation tools to be used in the test execution phase are decided. In addition, decisions on the optimal test case order minimizing the setup-overhead for the test cases play an important role.

At test realisation level, the STP refines the decisions on concrete test data and concrete test cases. In addition, decisions concerning GUI steps are important in order to define the concrete test cases. Moreover, GUI data are used to select concrete test data. In parallel, the GUI layout, i.e. how the GUI data are arranged on the screen, influences the concrete test cases.

At the last two decision levels there are no specific decisions within the STP.

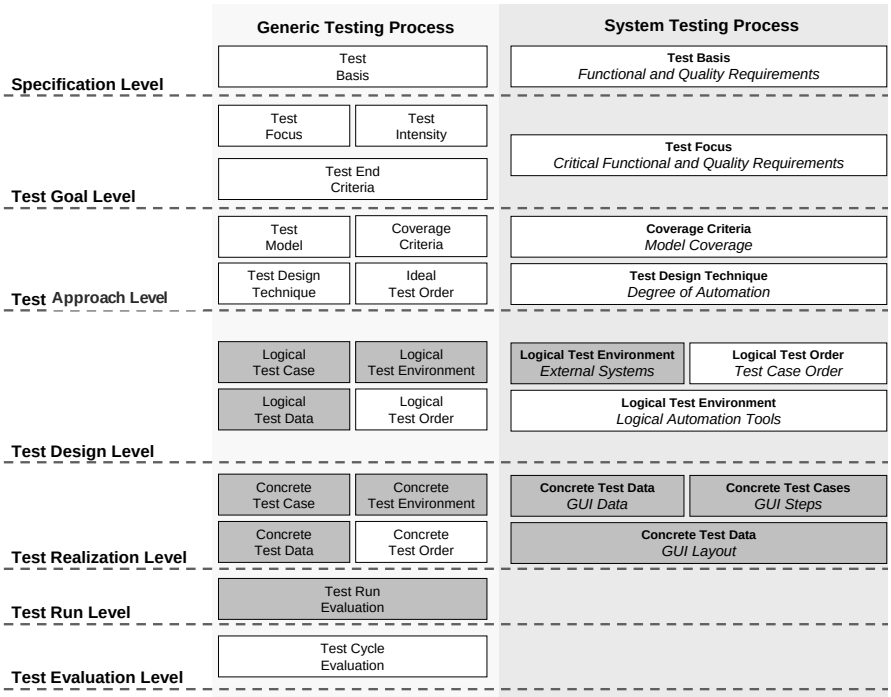


Figure A. 1.1 - Decision refinement for the system testing process
(Borner, Illes-Seifert, and Paech 2007a/b)

A 1.2 Test process analysis for process improvement

Based on the decision hierarchy, the testing process of an organisation has been analysed in order to find its strengths and weaknesses. The organisation provides system solutions in the area of real-time operations. Testers in this organisation are organised in an independent testing group. The ratio of testers to developers is 1:4. The test process analysis was based on document reviews as well as on interviews. All interviewees are experienced testers, with up to ten years of experience.

All decisions at specification level are made by the requirements engineering team, whereas the rest of the decisions are made by the testing team. Furthermore, there are decisions made implicitly (e.g. all decisions at test goal and test strategy level) and decisions made explicitly (e.g. all decisions at test design level). Implicit decisions are not documented, whereas explicit decisions are (partially) documented within test artefacts. All decisions on test goal and test strategy level are made implicitly. The testing team does not perform a risk analysis in order to make sound decisions on test foci or test intensities. Thus, the end of testing activities is not determined by criteria defined in advance, but by current test results and the "feeling" of the testing team regarding the maturity and quality of the product. The test team uses two "standard" test design techniques (domain testing and boundary value analysis). Other techniques are not considered and evaluated with respect to their efficiency in the project's context. Thus, decisions on the test model, the design technique as well as on

coverage criteria are made implicitly, without a thorough analysis of alternatives.

Logical test cases and test data are explicitly defined on the basis of requirements and documented within a test management tool. Decisions concerning concrete test cases and test data are made explicitly and are mostly documented during test execution within test protocols. The decision on the concrete test order is made explicitly, but only documented in case of a failed test run. A matrix of concrete test environments is also managed by the testing team. Decisions on logical test environments as well as on the logical test order are made implicitly and are not documented.

The evaluation of a test run is made explicitly for each executed test case. If a failure occurs, a process concerning the life cycle of a defect is passed through, from its classification, localisation and correction until its retest. At the end of a test cycle, the test team evaluates the results. This decision is made explicitly, but only summarises the test results. Since the definition of test end criteria is not performed, the evaluation of the test cycle occurs without a reference to defined criteria.

Implications: The decision based analysis highlights the following main strengths and weaknesses of the testing process. Missing involvement of the testing team into decisions at specification level leads to input which is not well suited to be used in the testing process. Thus, complex user scenarios are not part of the documentation provided by requirements engineers. However, these scenarios would be very precious for system testing as they lead to realistic test cases.

Another weakness concerns the unstructured decision process on test goal as well as on test approach level. Thus, a thorough evaluation against goals is not possible. Improvement efforts should concentrate on methodologies that help testers to define objective and measurable goals in advance. A strength of the testing process is the thorough documentation of decisions concerning test cases and test data supporting the repeatability of test runs for instance within regression testing.

A 1.3 Evaluation framework for testing approaches in the literature

The decision hierarchy can also be used as a framework for the comparison of different testing approaches. It permits the classification of approaches depending on whether they provide (automated) support for a specific decision or not. Table A.1.1 exemplifies how approaches for use case based testing can be compared with one another on the basis of the decision hierarchy, where this example considers only three of the seven decision levels. A complete overview of all approaches is presented in (Illes and Paech 2006). Comparing the approaches on the basis of the decision hierarchy allows the analysis of their similarities and differences. As illustrated in Table A.1.1, some decisions (e.g. the decision

concerning the test model) are supported by all approaches, whereas other decisions (e.g. the decision concerning quality requirements) are partially supported by only a subset of the approaches.

	Decisions		Approaches								
Decision level	Generic tesing process	System testing process	Path analysis [1]	Extended UCs [2]	TOTEM [3]	Structural Testing with UCs [4]	ASM Based Testing [5]	Requirements by Contracts [6]	Testing with UCs [7]	SCENT [8]	Simulation and Test Models [9]
Specification level	Test basis	*Functional requirements*	X	(X)	(X)	(X)	X	(X)	X	X	X
		Quality requirements							(X)	(X)	(X)
Test goal level	Test focus	*Critical functional requirements*	X	(X)		(X)		(X)			
		Critical quality requirements								(X)	
	Test intensity	Test intensity	(X)								
	Test end criteria	Test end criteria									
Test strategy level	Test model	Test model	X	X	X	X	X	X	X	X	X
	Coverage criteria	*Model coverage*	X	X	X	X	X	X	X	X	X
	Test design technique	*Degree of automation*			X		X	X		(X)	X
	Ideal test order	Ideal test order			(X)						

Table A.1.1- Applying the decision hierarchy to compare testing approaches

X = Approach supports decision, (X) = Approach partially supports decision
(Illes and Paech 2006)

[1] → (Ahlowalia 2002), [2] → (Binder 1999), [3] → (Briand and Labiche 2002)
[4] → (Carniello, Jino, and Lordello 2004), [5] → (Grieskamp et al. 2001),
[6] → (Nebut et al. 2003), [7] → (Rupp and Queins 2003), [8] → (Ryser and Glinz 2003)
[9] → (Whittle, Chakraborty, and Krueger 2005)

A 1.4 Evaluation framework for testing tools

The decision hierarchy served as the basis for the design of a questionnaire used within a survey evaluating 13 commercial and open source test management tools (Illes et al. 2006). The evaluation is primarily based on the information provided by tool vendors who completed the questionnaire. The goal was to analyse to what extent a decision is supported by a test management tool. Based on the decision hierarchy, questions addressing the functional characteristics of the testing tools can easily be derived. For instance, if a test management tool integrates requirements management functionality, it would provide support for decisions on specification level by facilitating the identification of functional and quality requirements.

A 2 Pareto distribution of defects in code

Table A.2.1 shows the frequency distribution of defects in code. For each analysed release of an OSP, the percentage of code and the percentage of the most fault-prone files that contain approximately 80% of the defects are indicated.

OS-Project	Release	% of code	% of files	at 80% of defects
1. Ant	ant 1.5.3	22,72	8,59	80,0
	ant 1.6	29,25	14,06	79,9
	ant 1.7	36,57	11,09	79,9
2. ApacheFOP	ApacheFOP 0.93	31,16	11,93	79,8
	ApacheFOP 0.94	17,93	8,87	80,0
3. CDK	CDK 2005	29,66	17,18	80,1
	CDK 2006	42,33	17,18	80,0
	CDK 2	17,24	7,03	79,9
4. Freenet	freenet 0.5.0	31,73	14,66	79,9
	freenet 0.5.1	18,71	8,3	80,05
	freenet 0.7	64,47	36,64	80
5. Jmol	Jmol 9	15,62	10,06	78,5
	Jmol 10	44,56	15,38	79,9
	Jmol 11.2	54,44	22,89	80,0
6.OS Cache	oscache 2.0.1.	43,14	13,98	80
	oscache 2.1.1.	30,25	8,16	80,6
	oscache 2.4	39,77	13,39	81,1
7. TVBrowser	tbrowser 0.9	34,09	14,0	80,4
	tbrowser 1.0	16,28	8,65	82,0
	tbrowser 2.6	22,72	8,59	80,0

Table A.2.1 - Pareto distribution of defects in code

A 3 Bad smell detection strategies

In the following, a summary of bad smell detection strategies as presented in (Marinescu 2002)is given.

Method level bad smells

FeatureEnvy := ((AID, HigherThan(4)) and (AID, TopValues(10%)) and (ALD, LowerThan(3)) and (NIC, LowerThan(3))

AID = Access of Import-Data

ALS = Access of Local Data

NIC = Number of Import Classes

GodMethod := (LOC, TopValues(20%)) but not in (LOC, LowerThan(70)) and

((NOP, HigherThan(4) or (NOLV, HigherThan(4))) and (MNOB, HigherThan(4))

LOC = Lines Of Code

NOP = Number Of Parameters

NOLV = Number Of Local Variables

MNOB = Maximum Number Of Branches

Class level bad smells

DataClasses := ((WOC, BottomValues(33%)) and (WOC, LowerThan(0.33))) and ((NOPA, HigherThan(5)) or (NOAM, HigherThan(5)))

WOC = Weight of a Class

NOPA = Number Of Public Attributes

NOAM = Number Of Accessor Methods

GodClasses := ((ATFD, TopValues(20%)) and (ATFD, HigherThan(4)))

and ((WMC, HigherThan(20)) or (TCC, LowerThan(0.33))

ATFD = Access To Foreign Data

WMC = Weighted Method Count

TCC = Tight Class Cohesion

ShotgunSurgery := ((CM, TopValues(20%)) and (CM, HigherThan(10))) and (CC, HigherThan(5))

CM = Changing Methods

WCM = Weighted Changing Methods

CC = Changing Classes

RefusedBequest := ((AIUR, BottomValues(25%)) but not in (DIT, LowerThan(1))) and (AIUR, LowerThan(0.33)))

IUR = Inheritance Usage Ratio

AIUR = Average Inheritance Usage Ratio

DIT = Depth of Inheritance Tree

MisplacedClass := (CL, LowerThan(0.33) and ((NOED, TopValues(25%)) and (NOED, HigherThan(6))) and (DD, LowerThan(3))

NOED = Number Of External Dependencies

CL = Class Locality

DD = Dependency Dispersion

Lack-of-Strategy bad smells

LackOfBridge := LackOfBridge-deep or LackOfBridge-shallow

LackOfBridge-deep := (NOD, HigherThan(8)) and ((HIT, HigherThan(1)) and (LR,HigherThan(0.66))) and (NPubM, HigherThan(3))

LackOfBridge-shallow := (NOD, HigherThan(6)) and ((CR, HigherThan(0.75)) and (HIT, HigherThan(0))) and(NPubM, HigherThan(3))

NOD = Number Of Descendants

HIT = Height of Inheritance Tree

LR = Leaves Ratio

CR = Child Ratio

NPubM = Number Of Public Methods

LackOfState := (AMW, HigherThan(4)) and (NOA, HigherThan(3)) and ((WMC, HigherThan(10)) or (NPubM, HigherThan(3)))

AMV = Average Method Weight

NOA = Number Of Attributes

NPubM = Number Of Public Methods

WMC = Weighted Method Count

LackOfStrategy := LackOfStrategy-OneClass or LackOfStrategy-ClassHierarchy

LackOfStrategy-OneClass := (WMC, HigherThan(20) and (WMC, TopValues(25%))) and (NOM, HigherThan(20)) or (TCC, LowerThan(33%))

LackOfStrategy-ClassHierarchy := (ANOM, HigherThan(1.0)) and (NPubM, HigherThan(3))

WMC = Weighted Method Count

TCC = Tight Class Cohesion

NPubM = Number Of Public Methods

NOM = Number Of Methods

ANOM = Average Number of Overridden Methods

LackOfVisitor := (AOR, HigherThan(0.5) and (NOD, HigherThan(3)) and (NPubM, HigherThan(5))

OR = Override Ratio

AOR = Average Override Ratio

NOD = Number Of Descendants

NPubM = Number Of Public Methods

ISPViolation := ((CIW, TopValues(20%) butnotin (CIW, LowerThan(10))) and (AUF, LowerThan(0.5)) and (COC, HigherThan(3))

CIW = Class Interface Width

COC = Clients Of Class

AUF = Average Use of Interface

Package level bad smells

GodPackage := ((PS, HigherThan(20)) and (PS, TopValues(25%)) and

(NOCC, HigherThan(20)) and (NOCP, HigherThan(3))

PS = Package Size

NOCC = Number Of Client Classes

NOCP = Number Of Client Packages

PC = Package Cohesion

WideSubystemInterface := (PIS, HigherThan(10)) and (PUR, HigherThan(0.75))

PIS = Package Interface Size

PUR = Package Usage Ratio

A 4 Mann-Whitney test for the bad smell FE and GM

In this chapter, the results of the Mann-Whitney non parametric test for the bad smell Feature Envy and God Method are presented.

OSP	Group	%	Mean Rank	Rank Sum	Mann-Whitney U	Z	Significance FE
ANT 1.5.3	¬FE-Group	79,31	336,80	193660,50	28060,50	-6,71	1,96102E-11
	FE-Group	20,69	463,43	69514,50			
ANT 1.6.0	¬FE-Group	77,74	434,66	305131,50	58378,50	-4,89	9,96118E-07
	FE-Group	22,26	512,56	103024,50			
ANT 1.7.0	¬FE-Group	75,65	517,47	434161,00	81781,00	-10,23	1,38197E-24
	FE-Group	24,35	671,61	181334,00			
Apache FOP 0.9.3	¬FE-Group	80,00	417,65	285675,00	51405,00	-3,72	0,000196005
	FE-Group	20,00	469,39	80265,00			
Apache FOP 0.9.4	¬FE-Group	100,00	451,50	407253,00			n/a
	FE-Group	0,00	0,00	0,00			
CDK 2005	¬FE-Group	100,00	483,50	467061,00			n/a
	FE-Group	0,00	0,00	0,00			
CDK 2006	¬FE-Group	80,05	615,46	629619,50	105843,50	-5,57	2,58526E-08
	FE-Group	19,95	735,93	187661,50			
CDK 1.0.1	¬FE-Group	78,13	518,60	420581,00	91315,00	-0,32	0,747565056
	FE-Group	21,87	522,73	118660,00			
Freenet 0.5.0	¬FE-Group	99,86	358,31	256189,50	219,50	-0,77	0,443653503
	FE-Group	0,14	496,50	496,50			
Freenet 0.5.1	¬FE-Group	100,00	994,50	1977066,00			n/a
	FE-Group	0,00	0,00	0,00			
Freenet 0.7	¬FE-Group	71,34	214,10	70867,00	15921,00	-4,80	1,55528E-06
	FE-Group	28,66	278,29	37013,00			
Jmol 9	¬FE-Group	98,82	85,14	14218,00	144,00	-0,56	0,573769181
	FE-Group	1,18	73,50	147,00			
Jmol 10	¬FE-Group	79,12	86,40	12442,00	2002,00	-3,08	0,002086516
	FE-Group	20,88	110,82	4211,00			
Jmol 11	¬FE-Group	78,01	155,09	40169,00	6499,00	-4,39	1,13715E-05
	FE-Group	21,99	206,97	15109,00			
OSCache 2.0.1	¬FE-Group	100,00	47,00	4371,00			n/a
	FE-Group	0,00	0,00	0,00			
OSCache 2.1.1	¬FE-Group	100,00	49,50	4851,00			n/a
	FE-Group	0,00	0,00	0,00			
OSCache 2.4.1	¬FE-Group	76,99	51,78	4505,00	677,00	-4,49	7,12752E-06
	FE-Group	23,01	74,46	1936,00			
TVBrowser 0.9.1	¬FE-Group	100,00	25,50	1275,00			n/a
	FE-Group	0,00	0,00	0,00			
TVBrowser 1.0	¬FE-Group	99,46	93,06	17123,50	80,50	-0,38	0,706913249
	FE-Group	0,54	81,50	81,50			
TVBrowser 2.6	¬FE-Group	67,23	394,97	219603,00	64757,00	-5,61	2,0274E-08
	FE-Group	32,77	453,04	122775,00			

Table A.4.1 - Man-Whitney test for the FE bad smell

OSP	Group	%	Mean Rank	Rank Sum	Mann-Whitney U	Z	Significance GM
ANT 1.5.3	¬GM-Group	95,31	350,93	242489,50	3403,50	-7,12	1,08368E-12
	GM-Group	4,69	608,40	20685,50			
ANT 1.6.0	¬GM-Group	95,35	441,71	380311,00	9220,00	-7,04	1,99226E-12
	GM-Group	4,65	662,98	27845,00			
ANT 1.7.0	¬GM-Group	96,03	542,50	577761,50	10116,50	-9,52	1,80071E-21
	GM-Group	3,97	857,58	37733,50			
Apache FOP 0.9.3	¬GM-Group	85,61	411,44	301172,00	32894,00	-7,27	3,54931E-13
	GM-Group	14,39	526,57	64768,00			
Apache FOP 0.9.4	¬GM-Group	92,68	441,93	369454,00	19588,00	-5,53	3,14826E-08
	GM-Group	7,32	572,71	37799,00			
CDK 2005	¬GM-Group	93,37	471,03	424867,50	17614,50	-6,16	7,27723E-10
	GM-Group	6,63	659,27	42193,50			
CDK 2006	¬GM-Group	92,49	622,55	735857,50	36704,50	-6,88	6,11821E-12
	GM-Group	7,51	848,16	81423,50			
CDK 1.0.1	¬GM-Group	88,63	501,88	461725,50	38065,50	-9,27	1,9499E-20
	GM-Group	11,37	656,91	77515,50			
Freenet 0.5.0	¬GM-Group	95,25	351,45	239691,50	6788,50	-4,68	2,81116E-06
	GM-Group	4,75	499,84	16994,50			
Freenet 0.5.1	¬GM-Group	97,89	994,50	1935297,00	40866,00	0,00	1
	GM-Group	2,11	994,50	41769,00			
Freenet 0.7	¬GM-Group	92,46	226,18	97032,50	4797,50	-3,66	0,000252089
	GM-Group	7,54	309,93	10847,50			
Jmol 9	¬GM-Group	90,53	85,08	13018,00	1211,00	-0,12	0,906512121
	GM-Group	9,47	84,19	1347,00			
Jmol 10	¬GM-Group	93,96	88,99	15216,50	510,50	-3,08	0,002103888
	GM-Group	6,04	130,59	1436,50			
Jmol 11.2.14	¬GM-Group	84,34	158,82	44470,50	5130,50	-3,64	0,000273721
	GM-Group	15,66	207,84	10807,50			
OSCache 2.0.1	¬GM-Group	95,70	45,76	4072,50	67,50	-2,82	0,004866787
	GM-Group	4,30	74,63	298,50			
OSCache 2.1.1	¬GM-Group	95,92	48,21	4532,00	67,00	-3,57	0,000352962
	GM-Group	4,08	79,75	319,00			
OSCache 2.4.1	¬GM-Group	94,69	55,89	5980,50	202,50	-2,20	0,027824904
	GM-Group	5,31	76,75	460,50			
TVBrowser 0.9.1	¬GM-Group	98,00	25,18	1234,00	9,00	-1,36	0,173990666
	GM-Group	2,00	41,00	41,00			
TVBrowser 1.0	¬GM-Group	94,59	91,51	16015,00	615,00	-2,76	0,005842258
	GM-Group	5,41	119,00	1190,00			
TVBrowser 2.6	¬GM-Group	92,02	407,17	309854,00	19913,00	-4,77	1,79742E-06
	GM-Group	7,98	492,79	32524,00			

Table A.4.2 - Man-Whitney test for the GM bad smell

A 5 Mann-Whitney test for class level bad smells

In this chapter, the results of the Mann-Whitney non parametric test for all class level bad smells analysed in Chapter 9.

OSP	Group	%	Mean Rank	Rank Sum	Mann-Whitney U	Z	Significance MC
ANT 1.5.3	¬MC-Group	78,34	337,56	191733,00	30137,00	-6,33	2,46412E-10
	MC-Group	21,66	455,04	71442,00			
ANT 1.6.0	¬MC-Group	78,85	428,86	305346,00	51518,00	-6,75	1,51805E-11
	MC-Group	21,15	538,27	102810,00			
ANT 1.7.0	¬MC-Group	80,88	541,47	485700,00	82947,00	-4,31	1,66514E-05
	MC-Group	19,12	612,24	129795,00			
Apache FOP 0.9.3	¬MC-Group	85,96	422,38	310446,00	39966,00	-2,51	0,01223956
	MC-Group	14,04	462,45	55494,00			
Apache FOP 0.9.4	¬MC-Group	100,00	451,50	407253,00			n/a
	MC-Group	0,00	0,00	0,00			
CDK 2005	¬MC-Group	93,48	482,70	435876,00	27720,00	-0,40	0,689427221
	MC-Group	6,52	495,00	31185,00			
CDK 2006	¬MC-Group	95,38	639,37	779388,00	35798,00	-0,07	0,94413601
	MC-Group	4,62	642,25	37893,00			
CDK 1.0.1	¬MC-Group	73,89	520,86	399500,50	102884,50	-0,43	0,666381821
	MC-Group	26,11	515,65	139740,50			
Freenet 0.5.0	¬MC-Group	86,73	348,69	216536,50	23405,50	-3,72	0,000196979
	MC-Group	13,27	422,63	40149,50			
Freenet 0.5.1	¬MC-Group	78,67	1126,77	2060861,00	387326,00	-8,68	3,91868E-18
	MC-Group	21,33	1296,60	643114,00			
Freenet 0.7	¬MC-Group	69,18	222,52	71430,00	19749,00	-2,47	0,013369792
	MC-Group	30,82	254,90	36450,00			
Jmol 9	¬MC-Group	51,48	87,13	7580,00	3382,00	-0,98	0,327582807
	MC-Group	48,52	82,74	6785,00			
Jmol 10	¬MC-Group	74,73	88,80	12076,50	2760,50	-1,44	0,149548289
	MC-Group	25,27	99,49	4576,50			
Jmol 11.2.14	¬MC-Group	61,14	160,05	32491,00	11785,00	-1,65	0,098578308
	MC-Group	38,86	176,64	22787,00			
OSCache 2.0.1	¬MC-Group	100,00	47,00	4371,00			n/a
	MC-Group	0,00	0,00	0,00			
OSCache 2.1.1	¬MC-Group	100,00	49,50	4851,00			n/a
	MC-Group	0,00	0,00	0,00			
OSCache 2.4.1	¬MC-Group	59,29	52,46	3514,50	1236,50	-2,58	0,009884618
	MC-Group	40,71	63,62	2926,50			
TVBrowser 0.9.1	¬MC-Group	54,00	25,00	675,00	297,00	-0,33	0,73943116
	MC-Group	46,00	26,09	600,00			
TVBrowser 1.0	¬MC-Group	67,57	88,78	11098,00	3223,00	-2,70	0,006957197
	MC-Group	32,43	101,78	6107,00			
TVBrowser 2.6	¬MC-Group	71,58	419,56	248378,50	66269,50	-1,82	0,069446026
	MC-Group	28,42	400,00	93999,50			

Table A.5.1- Man-Whitney test for the MC bad smell

OSP	Group	%	Mean Rank	Rank Sum	Mann-Whitney U	Z	Significance DC
ANT 1.5.3	¬DC-Group	91,45	355,72	235842,50	15726,50	-3,11	0,001848522
	DC-Group	8,55	440,85	27332,50			
ANT 1.6.0	¬DC-Group	91,58	443,69	366935,50	24557,50	-4,14	3,52959E-05
	DC-Group	8,42	542,38	41220,50			
ANT 1.7.0	¬DC-Group	90,35	543,16	544248,50	41745,50	-5,60	2,0827E-08
	DC-Group	9,65	665,86	71246,50			
Apache FOP 0.9.3	¬DC-Group	92,87	421,07	334328,50	18713,50	-4,50	6,77854E-06
	DC-Group	7,13	518,22	31611,50			
Apache FOP 0.9.4	¬DC-Group	100,00	451,50	407253,00			n/a
	DC-Group	0,00	0,00	0,00			
CDK 2005	¬DC-Group	95,76	482,49	446299,50	18024,50	-0,63	0,526283398
	DC-Group	4,24	506,38	20761,50			
CDK 2006	¬DC-Group	96,17	636,95	782812,50	26977,50	-1,48	0,139832848
	DC-Group	3,83	703,44	34468,50			
CDK 1.0.1	¬DC-Group	95,76	520,22	517098,50	21152,50	-0,64	0,519495362
	DC-Group	4,24	503,24	22142,50			
Freenet 0.5.0	¬DC-Group	100,00	358,50	256686,00			n/a
	DC-Group	0,00	0,00	0,00			
Freenet 0.5.1	¬DC-Group	100,00	994,50	1977066,00			n/a
	DC-Group	0,00	0,00	0,00			
Freenet 0.7	¬DC-Group	100,00	232,50	107880,00			n/a
	DC-Group	0,00	0,00	0,00			
Jmol 9	¬DC-Group	99,41	85,07	14291,50	72,50	-0,40	0,69168571
	DC-Group	0,59	73,50	73,50			
Jmol 10	¬DC-Group	100,00	91,50	16653,00			n/a
	DC-Group	0,00	0,00	0,00			
Jmol 11.2.14	¬DC-Group	99,10	166,50	54777,50	492,50	-0,01	0,994811985
	DC-Group	0,90	166,83	500,50			
OSCache 2.0.1	¬DC-Group	100,00	47,00	4371,00			n/a
	DC-Group	0,00	0,00	0,00			
OSCache 2.1.1	¬DC-Group	100,00	49,50	4851,00			n/a
	DC-Group	0,00	0,00	0,00			
OSCache 2.4.1	¬DC-Group	100,00	57,00	6441,00			n/a
	DC-Group	0,00	0,00	0,00			
TVBrowser 0.9.1	¬DC-Group	92,00	24,40	1122,50	41,50	-2,29	0,022269872
	DC-Group	8,00	38,13	152,50			
TVBrowser 1.0	¬DC-Group	92,43	91,87	15709,00	1003,00	-1,76	0,078661187
	DC-Group	7,57	106,86	1496,00			
TVBrowser 2.6	¬DC-Group	91,41	409,52	309597,00	23451,00	-3,01	0,002625053
	DC-Group	8,59	461,70	32781,00			

Table A.5.2 - Man-Whitney test for the DC bad smell

OSP	Group	%	Mean Rank	Rank Sum	Mann-Whitney U	Z	Significance GC
ANT 1.5.3	¬GC-Group	96,69	354,94	248809,50	2758,50	-5,70	1,19312E-08
	GC-Group	3,31	598,56	14365,50			
ANT 1.6.0	¬GC-Group	96,23	442,82	384813,00	6798,00	-7,00	2,4749E-12
	GC-Group	3,77	686,56	23343,00			
ANT 1.7.0	¬GC-Group	95,85	540,10	574129,50	8613,50	-11,08	1,5655E-28
	GC-Group	4,15	899,25	41365,50			
Apache FOP 0.9.3	¬GC-Group	79,53	408,88	278036,50	46496,50	-6,78	1,16888E-11
	GC-Group	20,47	502,31	87903,50			
Apache FOP 0.9.4	¬GC-Group	96,23	445,76	386919,00	9773,00	-4,71	2,44894E-06
	GC-Group	3,77	598,06	20334,00			
CDK 2005	¬GC-Group	99,07	480,44	459777,00	1374,00	-4,16	3,2222E-05
	GC-Group	0,93	809,33	7284,00			
CDK 2006	¬GC-Group	99,22	636,46	807036,00	2490,00	-3,95	7,68961E-05
	GC-Group	0,78	1024,50	10245,00			
CDK 1.0.1	¬GC-Group	99,13	517,99	533009,50	3074,50	-3,04	0,002333734
	GC-Group	0,87	692,39	6231,50			
Freenet 0.5.0	¬GC-Group	98,18	353,36	248413,50	957,50	-5,61	2,04452E-08
	GC-Group	1,82	636,35	8272,50			
Freenet 0.5.1	¬GC-Group	98,92	1153,30	2652590,00	6440,00	-11,61	3,70482E-31
	GC-Group	1,08	2055,40	51385,00			
Freenet 0.7	¬GC-Group	97,84	228,97	103952,00	667,00	-3,94	8,24136E-05
	GC-Group	2,16	392,80	3928,00			
Jmol 9	¬GC-Group	93,49	84,25	13311,50	750,50	-1,27	0,203913068
	GC-Group	6,51	95,77	1053,50			
Jmol 10	¬GC-Group	89,01	86,95	14086,50	883,50	-4,01	5,98934E-05
	GC-Group	10,99	128,33	2566,50			
Jmol 11.2.14	¬GC-Group	87,35	153,04	44381,00	2186,00	-7,23	4,9676E-13
	GC-Group	12,65	259,45	10897,00			
OSCache 2.0.1	¬GC-Group	100,00	47,00	4371,00			n/a
	GC-Group	0,00	0,00	0,00			
OSCache 2.1.1	¬GC-Group	100,00	49,50	4851,00			n/a
	GC-Group	0,00	0,00	0,00			
OSCache 2.4.1	¬GC-Group	100,00	57,00	6441,00			n/a
	GC-Group	0,00	0,00	0,00			
TVBrowser 0.9.1	¬GC-Group	98,00	25,18	1234,00	9,00	-1,36	0,173990666
	GC-Group	2,00	41,00	41,00			
TVBrowser 1.0	¬GC-Group	98,38	91,70	16688,50	35,50	-4,51	6,54801E-06
	GC-Group	1,62	172,17	516,50			
TVBrowser 2.6	¬GC-Group	95,53	406,08	320801,00	8356,00	-7,53	4,92369E-14
	GC-Group	4,47	583,16	21577,00			

Table A.5.3 - Man-Whitney test for the GC bad smell

OSP	Group	%	Mean Rank	Rank Sum	Mann-Whitney U	Z	Significance SS
ANT 1.5.3	¬SS-Group	91,17	347,72	229843,00	11052,00	-6,42	1,34022E-10
	SS-Group	8,83	520,81	33332,00			
ANT 1.6.0	¬SS-Group	90,70	441,53	361616,00	25826,00	-4,93	8,05131E-07
	SS-Group	9,30	554,05	46540,00			
ANT 1.7.0	¬SS-Group	89,72	536,39	533708,50	38198,50	-8,51	1,79118E-17
	SS-Group	10,28	717,43	81786,50			
Apache FOP 0.9.3	¬SS-Group	81,64	419,26	292644,50	48693,50	-3,32	0,000913117
	SS-Group	18,36	466,85	73295,50			
Apache FOP 0.9.4	¬SS-Group	81,93	450,68	333053,00	59623,00	-0,28	0,776848772
	SS-Group	18,07	455,21	74200,00			
CDK 2005	¬SS-Group	90,58	468,10	409589,00	26339,00	-6,28	3,34421E-10
	SS-Group	9,42	631,56	57472,00			
CDK 2006	¬SS-Group	89,91	622,62	715392,00	54717,00	-5,83	5,69886E-09
	SS-Group	10,09	789,84	101889,00			
CDK 1.0.1	¬SS-Group	89,79	514,29	479315,50	44537,50	-2,91	0,003612168
	SS-Group	10,21	565,33	59925,50			
Freenet 0.5.0	¬SS-Group	83,52	347,30	207685,50	28584,50	-3,74	0,0001823
	SS-Group	16,48	415,26	49000,50			
Freenet 0.5.1	¬SS-Group	90,54	1138,01	2395513,50	178948,50	-9,64	5,1706E-22
	SS-Group	9,46	1402,10	308461,50			
Freenet 0.7	¬SS-Group	83,84	219,28	85301,50	9446,50	-4,98	6,31991E-07
	SS-Group	16,16	301,05	22578,50			
Jmol 9	¬SS-Group	84,62	82,47	11793,00	1497,00	-2,65	0,007964769
	SS-Group	15,38	98,92	2572,00			
Jmol 10	¬SS-Group	81,87	88,41	13173,50	1998,50	-2,03	0,041880513
	SS-Group	18,13	105,44	3479,50			
Jmol 11.2.14	¬SS-Group	83,73	155,50	43228,50	4447,50	-5,10	3,40882E-07
	SS-Group	16,27	223,14	12049,50			
OSCache 2.0.1	¬SS-Group	92,47	44,97	3867,00	126,00	-3,43	0,000605374
	SS-Group	7,53	72,00	504,00			
OSCache 2.1.1	¬SS-Group	91,84	48,47	4362,00	267,00	-1,98	0,047198629
	SS-Group	8,16	61,13	489,00			
OSCache 2.4.1	¬SS-Group	92,92	54,96	5771,00	206,00	-3,47	0,000514812
	SS-Group	7,08	83,75	670,00			
TVBrowser 0.9.1	¬SS-Group	86,00	25,34	1089,50	143,50	-0,25	0,804352474
	SS-Group	14,00	26,50	185,50			
TVBrowser 1.0	¬SS-Group	83,78	92,33	14310,50	2220,50	-0,68	0,496717108
	SS-Group	16,22	96,48	2894,50			
TVBrowser 2.6	¬SS-Group	85,97	402,21	285970,00	32854,00	-6,01	1,88012E-09
	SS-Group	14,03	486,28	56408,00			

Table A.5.4 - Man-Whitney test for the SS bad smell

OSP	Group	%	Mean Rank	Rank Sum	Mann-Whitney U	Z	Significance RB
ANT 1.5.3	¬RB-Group	84,28	347,89	212559,00	25593,00	-4,58	4,7379E-06
	RB-Group	15,72	444,00	50616,00			
ANT 1.6.0	¬RB-Group	47,95	416,80	180474,50	86513,50	-5,10	3,38103E-07
	RB-Group	52,05	484,43	227681,50			
ANT 1.7.0	¬RB-Group	84,13	541,22	504955,00	69244,00	-4,91	9,09553E-07
	RB-Group	15,87	628,07	110540,00			
Apache FOP 0.9.3	¬RB-Group	88,65	423,63	321115,00	33454,00	-2,20	0,028076218
	RB-Group	11,35	462,11	44825,00			
Apache FOP 0.9.4	¬RB-Group	88,80	459,17	367795,50	34306,50	-3,51	0,00044931
	RB-Group	11,20	390,67	39457,50			
CDK 2005	¬RB-Group	94,62	480,32	439012,50	20857,50	-1,75	0,079428763
	RB-Group	5,38	539,39	28048,50			
CDK 2006	¬RB-Group	93,90	646,47	775764,00	38436,00	-3,16	0,001569408
	RB-Group	6,10	532,27	41517,00			
CDK 1.0.1	¬RB-Group	94,12	521,44	509443,50	27906,50	-1,46	0,144534903
	RB-Group	5,88	488,48	29797,50			
Freenet 0.5.0	¬RB-Group	92,18	358,16	236382,50	18252,50	-0,18	0,860571558
	RB-Group	7,82	362,56	20303,50			
Freenet 0.5.1	¬RB-Group	95,44	1151,83	2555909,50	92819,50	-6,38	1,80284E-10
	RB-Group	4,56	1396,84	148065,50			
Freenet 0.7	¬RB-Group	94,40	228,81	100217,50	4076,50	-2,51	0,012125261
	RB-Group	5,60	294,71	7662,50			
Jmol 9	¬RB-Group	66,27	83,98	9406,00	3078,00	-0,64	0,523652759
	RB-Group	33,73	87,00	4959,00			
Jmol 10	¬RB-Group	80,77	91,56	13459,50	2563,50	-0,04	0,968956256
	RB-Group	19,23	91,24	3193,50			
Jmol 11.2.14	¬RB-Group	60,24	154,69	30938,00	10838,00	-2,97	0,002981695
	RB-Group	39,76	184,39	24340,00			
OSCache 2.0.1	¬RB-Group	89,25	46,63	3870,00	384,00	-0,52	0,604919695
	RB-Group	10,75	50,10	501,00			
OSCache 2.1.1	¬RB-Group	88,78	49,79	4332,00	453,00	-0,47	0,636942644
	RB-Group	11,22	47,18	519,00			
OSCache 2.4.1	¬RB-Group	90,27	54,66	5575,00	322,00	-3,36	0,000790758
	RB-Group	9,73	78,73	866,00			
TVBrowser 0.9.1	¬RB-Group	84,00	25,76	1082,00	157,00	-0,37	0,712545593
	RB-Group	16,00	24,13	193,00			
TVBrowser 1.0	¬RB-Group	74,59	92,29	12736,00	3145,00	-0,54	0,589412711
	RB-Group	25,41	95,09	4469,00			
TVBrowser 2.6	¬RB-Group	74,12	409,87	251252,00	63061,00	-1,44	0,150569987
	RB-Group	25,88	425,82	91126,00			

Table A.5.5 - Man-Whitney test for the RB bad smell

OSP	Group	%	Mean Rank	Rank Sum	Mann-Whitney U	Z	Significance MC
ANT 1.5.3	¬MC-Group	78,34	337,56	191733,00	30137,00	-6,33	2,46412E-10
	MC-Group	21,66	455,04	71442,00			
ANT 1.6.0	¬MC-Group	78,85	428,86	305346,00	51518,00	-6,75	1,51805E-11
	MC-Group	21,15	538,27	102810,00			
ANT 1.7.0	¬MC-Group	80,88	541,47	485700,00	82947,00	-4,31	1,66514E-05
	MC-Group	19,12	612,24	129795,00			
Apache FOP 0.9.3	¬MC-Group	85,96	422,38	310446,00	39966,00	-2,51	0,01223956
	MC-Group	14,04	462,45	55494,00			
Apache FOP 0.9.4	¬MC-Group	100,00	451,50	407253,00			n/a
	MC-Group	0,00	0,00	0,00			
CDK 2005	¬MC-Group	93,48	482,70	435876,00	27720,00	-0,40	0,689427221
	MC-Group	6,52	495,00	31185,00			
CDK 2006	¬MC-Group	95,38	639,37	779388,00	35798,00	-0,07	0,94413601
	MC-Group	4,62	642,25	37893,00			
CDK 1.0.1	¬MC-Group	73,89	520,86	399500,50	102884,50	-0,43	0,666381821
	MC-Group	26,11	515,65	139740,50			
Freenet 0.5.0	¬MC-Group	86,73	348,69	216536,50	23405,50	-3,72	0,000196979
	MC-Group	13,27	422,63	40149,50			
Freenet 0.5.1	¬MC-Group	78,67	1126,77	2060861,00	387326,00	-8,68	3,91868E-18
	MC-Group	21,33	1296,60	643114,00			
Freenet 0.7	¬MC-Group	69,18	222,52	71430,00	19749,00	-2,47	0,013369792
	MC-Group	30,82	254,90	36450,00			
Jmol 9	¬MC-Group	51,48	87,13	7580,00	3382,00	-0,98	0,327582807
	MC-Group	48,52	82,74	6785,00			
Jmol 10	¬MC-Group	74,73	88,80	12076,50	2760,50	-1,44	0,149548289
	MC-Group	25,27	99,49	4576,50			
Jmol 11.2.14	¬MC-Group	61,14	160,05	32491,00	11785,00	-1,65	0,098578308
	MC-Group	38,86	176,64	22787,00			
OSCache 2.0.1	¬MC-Group	100,00	47,00	4371,00			n/a
	MC-Group	0,00	0,00	0,00			
OSCache 2.1.1	¬MC-Group	100,00	49,50	4851,00			n/a
	MC-Group	0,00	0,00	0,00			
OSCache 2.4.1	¬MC-Group	59,29	52,46	3514,50	1236,50	-2,58	0,009884618
	MC-Group	40,71	63,62	2926,50			
TVBrowser 0.9.1	¬MC-Group	54,00	25,00	675,00	297,00	-0,33	0,73943116
	MC-Group	46,00	26,09	600,00			
TVBrowser 1.0	¬MC-Group	67,57	88,78	11098,00	3223,00	-2,70	0,006957197
	MC-Group	32,43	101,78	6107,00			
TVBrowser 2.6	¬MC-Group	71,58	419,56	248378,50	66269,50	-1,82	0,069446026
	MC-Group	28,42	400,00	93999,50			

Table A.5.6 - Man-Whitney test for the MC bad smell

A 6 Mann-Whitney test for "Lack-Of" bad smells

In this chapter, the results of the Mann-Whitney non parametric test for all Lack-of bad smells analysed in Chapter 9 are presented.

OSP	Group	%	Mean Rank	Rank Sum	Mann-Whitney U	Z	Significance LoBr
ANT 1.5.3	¬LoB-Group	82,90	353,64	212535,00	31634,00	-2,70	0,007009656
	LoB-Group	17,10	408,39	50640,00			
ANT 1.6.0	¬LoB-Group	82,95	444,03	332582,00	51707,00	-2,65	0,007999114
	LoB-Group	17,05	490,74	75574,00			
ANT 1.7.0	¬LoB-Group	81,51	538,16	486500,50	77440,50	-5,47	4,4963E-08
	LoB-Group	18,49	629,24	128994,50			
Apache FOP 0.9.3	¬LoB-Group	86,90	426,97	317239,00	40843,00	-0,48	0,633175054
	LoB-Group	13,10	434,83	48701,00			
Apache FOP 0.9.4	¬LoB-Group	96,90	453,41	396281,50	10565,50	-1,73	0,082771316
	LoB-Group	3,10	391,84	10971,50			
CDK 2005	¬LoB-Group	93,79	482,16	436838,00	25967,00	-0,68	0,493675233
	LoB-Group	6,21	503,72	30223,00			
CDK 2006	¬LoB-Group	89,67	632,20	724495,50	67264,50	-2,49	0,012806005
	LoB-Group	10,33	702,92	92785,50			
CDK 1.0.1	¬LoB-Group	88,82	520,94	480306,50	52148,50	-0,76	0,444735414
	LoB-Group	11,18	508,06	58934,50			
Freenet 0.5.0	¬LoB-Group	73,88	361,47	191215,00	47893,00	-0,74	0,459167035
	LoB-Group	26,12	350,11	65471,00			
Freenet 0.5.1	¬LoB-Group	87,48	1140,99	2320764,00	251169,00	-7,26	3,8049E-13
	LoB-Group	12,52	1316,88	383211,00			
Freenet 0.7	¬LoB-Group	78,66	239,72	87499,00	15431,00	-2,30	0,021711707
	LoB-Group	21,34	205,87	20381,00			
Jmol 9	¬LoB-Group	88,76	85,32	12797,50	1377,50	-0,40	0,690857706
	LoB-Group	11,24	82,50	1567,50			
Jmol 10	¬LoB-Group	91,76	90,84	15170,00	1142,00	-0,68	0,493481584
	LoB-Group	8,24	98,87	1483,00			
Jmol 11.2.14	¬LoB-Group	83,13	162,66	44893,50	6667,50	-1,74	0,081412504
	LoB-Group	16,87	185,44	10384,50			
OSCache 2.0.1	¬LoB-Group	100,00	47,00	4371,00			n/a
	LoB-Group	0,00	0,00	0,00			
OSCache 2.1.1	¬LoB-Group	100,00	49,50	4851,00			n/a
	LoB-Group	0,00	0,00	0,00			
OSCache 2.4.1	¬LoB-Group	100,00	57,00	6441,00			n/a
	LoB-Group	0,00	0,00	0,00			
TVBrowser 0.9.1	¬LoB-Group	78,00	26,90	1049,00	160,00	-1,62	0,106199271
	LoB-Group	22,00	20,55	226,00			
TVBrowser 1.0	¬LoB-Group	86,49	94,21	15073,00	1807,00	-1,35	0,175922183
	LoB-Group	13,51	85,28	2132,00			
TVBrowser 2.6	¬LoB-Group	87,18	417,36	300919,50	35787,50	-1,81	0,070987346
	LoB-Group	12,82	391,12	41458,50			

Table A.6.1 - Man-Whitney test for the LoB bad smell

OSP	Group	%	Mean Rank	Rank Sum	Mann-Whitney U	Z	Significance LoState
ANT 1.5.3	¬LoState-Group	81,66	324,01	191811,00	16283,00	-10,76	5,30722E-27
	LoState-Group	18,34	536,57	71364,00			
ANT 1.6.0	¬LoState-Group	80,40	420,60	305355,50	41454,50	-9,60	7,90731E-22
	LoState-Group	19,60	580,79	102800,50			
ANT 1.7.0	¬LoState-Group	82,33	512,82	468203,50	50962,50	-14,09	4,63039E-45
	LoState-Group	17,67	751,49	147291,50			
Apache FOP 0.9.3	¬LoState-Group	76,26	404,31	263612,00	50734,00	-7,64	2,17397E-14
	LoState-Group	23,74	504,08	102328,00			
Apache FOP 0.9.4	¬LoState-Group	87,58	443,44	350315,50	37870,50	-3,48	0,000503695
	LoState-Group	12,42	508,37	56937,50			
CDK 2005	¬LoState-Group	90,27	471,83	411438,50	30810,50	-4,68	2,93905E-06
	LoState-Group	9,73	591,73	55622,50			
CDK 2006	¬LoState-Group	91,08	617,55	718822,50	40792,50	-8,11	4,9418E-16
	LoState-Group	8,92	863,67	98458,50			
CDK 1.0.1	¬LoState-Group	85,93	504,01	449576,00	51298,00	-7,21	5,64307E-13
	LoState-Group	14,07	614,14	89665,00			
Freenet 0.5.0	¬LoState-Group	90,50	345,24	223713,50	13437,50	-6,08	1,22304E-09
	LoState-Group	9,50	484,89	32972,50			
Freenet 0.5.1	¬LoState-Group	94,92	1133,57	2501783,50	65255,50	-15,88	8,37092E-57
	LoState-Group	5,08	1713,49	202191,50			
Freenet 0.7	¬LoState-Group	83,84	213,61	83094,00	7239,00	-7,12	1,07889E-12
	LoState-Group	16,16	330,48	24786,00			
Jmol 9	¬LoState-Group	86,39	83,38	12173,00	1442,00	-1,83	0,067545265
	LoState-Group	13,61	95,30	2192,00			
Jmol 10	¬LoState-Group	91,76	88,64	14803,00	775,00	-2,96	0,003085137
	LoState-Group	8,24	123,33	1850,00			
Jmol 11.2.14	¬LoState-Group	79,22	152,53	40116,50	5400,50	-5,57	2,55E-08
	LoState-Group	20,78	219,73	15161,50			
OSCache 2.0.1	¬LoState-Group	90,32	44,05	3700,50	130,50	-4,33	1,5061E-05
	LoState-Group	9,68	74,50	670,50			
OSCache 2.1.1	¬LoState-Group	90,82	48,02	4273,50	268,50	-2,67	0,007573499
	LoState-Group	9,18	64,17	577,50			
OSCache 2.4.1	¬LoState-Group	89,38	54,40	5494,50	343,50	-3,55	0,000390387
	LoState-Group	10,62	78,88	946,50			
TVBrowser 0.9.1	¬LoState-Group	94,00	24,15	1135,00	7,00	-3,28	0,001026086
	LoState-Group	6,00	46,67	140,00			
TVBrowser 1.0	¬LoState-Group	89,73	91,04	15112,50	1251,50	-2,57	0,010153947
	LoState-Group	10,27	110,13	2092,50			
TVBrowser 2.6	¬LoState-Group	85,37	397,96	280962,00	31391,00	-7,97	1,56321E-15
	LoState-Group	14,63	507,57	61416,00			

Table A.6.2 - Man-Whitney test for the LoSta bad smell

OSP	Group	%	Mean Rank	Rank Sum	Mann-Whitney U	Z	Significance LoStr
ANT 1.5.3	¬LoStra-Group	96,69	355,03	248875,00	2824,00	-5,63	1,75404E-08
	LoSt-Group	3,31	595,83	14300,00			
ANT 1.6.0	¬LoStra-Group	91,81	445,76	369533,00	25498,00	-3,15	0,001607902
	LoSt-Group	8,19	521,93	38623,00			
ANT 1.7.0	¬LoStra-Group	95,94	543,67	578467,50	11887,50	-8,52	1,56309E-17
	LoSt-Group	4,06	822,83	37027,50			
Apache FOP 0.9.3	¬LoStra-Group	87,60	425,39	318615,00	37740,00	-1,25	0,211313776
	LoSt-Group	12,40	446,46	47325,00			
Apache FOP 0.9.4	¬LoStra-Group	94,46	451,68	384833,50	21144,50	-0,12	0,90258474
	LoSt-Group	5,54	448,39	22419,50			
CDK 2005	¬LoStra-Group	95,76	479,20	443263,50	14988,50	-2,68	0,007258707
	LoSt-Group	4,24	580,43	23797,50			
CDK 2006	¬LoStra-Group	93,11	628,66	748110,00	39465,00	-4,61	4,06234E-06
	LoSt-Group	6,89	786,03	69171,00			
CDK 1.0.1	¬LoStra-Group	94,12	517,78	505873,00	28120,00	-1,29	0,195507053
	LoSt-Group	5,88	547,02	33368,00			
Freenet 0.5.0	¬LoStra-Group	95,25	352,77	240590,50	7687,50	-3,81	0,000140176
	LoSt-Group	4,75	473,40	16095,50			
Freenet 0.5.1	¬LoStra-Group	97,81	1151,25	2617951,00	31276,00	-9,79	1,28231E-22
	LoSt-Group	2,19	1686,75	86024,00			
Freenet 0.7	¬LoStra-Group	97,41	230,17	104039,00	1661,00	-2,36	0,018189266
	LoSt-Group	2,59	320,08	3841,00			
Jmol 9	¬LoStra-Group	98,82	84,63	14132,50	104,50	-1,53	0,126374195
	LoSt-Group	1,18	116,25	232,50			
Jmol 10	¬LoStra-Group	98,90	91,31	16435,50	145,50	-0,56	0,572769532
	LoSt-Group	1,10	108,75	217,50			
Jmol 11.2.14	¬LoStra-Group	96,39	162,82	52103,50	743,50	-3,88	0,000105159
	LoSt-Group	3,61	264,54	3174,50			
OSCache 2.0.1	¬LoStra-Group	94,62	46,35	4078,50	162,50	-1,32	0,187525517
	LoSt-Group	5,38	58,50	292,50			
OSCache 2.1.1	¬LoStra-Group	93,88	48,34	4447,00	169,00	-2,61	0,009116246
	LoSt-Group	6,12	67,33	404,00			
OSCache 2.4.1	¬LoStra-Group	94,69	54,83	5866,50	88,50	-4,32	1,58912E-05
	LoSt-Group	5,31	95,75	574,50			
TVBrowser 0.9.1	¬LoStra-Group	98,00	25,64	1256,50	17,50	-0,61	0,539238966
	LoSt-Group	2,00	18,50	18,50			
TVBrowser 1.0	¬LoStra-Group	96,76	92,86	16621,50	511,50	-0,35	0,73002059
	LoSt-Group	3,24	97,25	583,50			
TVBrowser 2.6	¬LoStra-Group	95,53	413,35	326548,50	14103,50	-0,62	0,53809918
	LoSt-Group	4,47	427,82	15829,50			

Table A.6.3 - Man-Whitney test for the LoStra bad smell

OSP	Group	%	Mean Rank	Rank Sum	Mann-Whitney U	Z	Significance LoV
ANT 1.5.3	¬LoV-Group	89,52	360,48	233954,00	23029,00	-0,96	0,336207725
	LoV-Group	10,48	384,49	29221,00			
ANT 1.6.0	¬LoV-Group	93,13	449,46	377992,50	23931,50	-1,41	0,157191645
	LoV-Group	6,87	486,51	30163,50			
ANT 1.7.0	¬LoV-Group	88,73	548,41	539638,00	55018,00	-2,86	0,004240705
	LoV-Group	11,27	606,86	75857,00			
Apache FOP 0.9.3	¬LoV-Group	87,84	426,92	320620,00	38244,00	-0,52	0,60283747
	LoV-Group	12,16	435,77	45320,00			
Apache FOP 0.9.4	¬LoV-Group	99,45	451,59	405077,50	2160,50	-0,20	0,842325756
	LoV-Group	0,55	435,10	2175,50			
CDK 2005	¬LoV-Group	90,17	484,52	422018,50	40482,50	-0,41	0,683965814
	LoV-Group	9,83	474,13	45042,50			
CDK 2006	¬LoV-Group	98,67	638,04	804564,50	8873,50	-1,46	0,145045553
	LoV-Group	1,33	748,03	12716,50			
CDK 1.0.1	¬LoV-Group	86,99	521,71	471104,00	58957,00	-1,08	0,281918694
	LoV-Group	13,01	504,72	68137,00			
Freenet 0.5.0	¬LoV-Group	99,30	358,50	254891,00	1775,00	-0,01	0,995034284
	LoV-Group	0,70	359,00	1795,00			
Freenet 0.5.1	¬LoV-Group	90,58	1146,72	2414994,50	196323,50	-6,30	2,99703E-10
	LoV-Group	9,42	1319,55	288980,50			
Freenet 0.7	¬LoV-Group	80,82	242,16	90810,50	13064,50	-3,28	0,001030493
	LoV-Group	19,18	191,79	17069,50			
Jmol 9	¬LoV-Group	91,72	85,49	13250,50	1009,50	-0,72	0,468806256
	LoV-Group	8,28	79,61	1114,50			
Jmol 10	¬LoV-Group	93,96	90,72	15512,50	806,50	-0,96	0,337908448
	LoV-Group	6,04	103,68	1140,50			
Jmol 11.2.14	¬LoV-Group	87,05	164,81	47631,00	5726,00	-0,89	0,371677922
	LoV-Group	12,95	177,84	7647,00			
OSCache 2.0.1	¬LoV-Group	97,85	47,24	4299,00	69,00	-0,78	0,433015427
	LoV-Group	2,15	36,00	72,00			
OSCache 2.1.1	¬LoV-Group	97,96	49,65	4766,00	82,00	-0,58	0,562916342
	LoV-Group	2,04	42,50	85,00			
OSCache 2.4.1	¬LoV-Group	98,23	57,20	6349,00	89,00	-0,69	0,48737302
	LoV-Group	1,77	46,00	92,00			
TVBrowser 0.9.1	¬LoV-Group	78,00	26,90	1049,00	160,00	-1,62	0,106199271
	LoV-Group	22,00	20,55	226,00			
TVBrowser 1.0	¬LoV-Group	88,65	93,90	15399,00	1575,00	-1,11	0,266594461
	LoV-Group	11,35	86,00	1806,00			
TVBrowser 2.6	¬LoV-Group	89,60	417,54	309398,00	29239,00	-2,14	0,032425481
	LoV-Group	10,40	383,49	32980,00			

Table A.6.4 - Man-Whitney test for the LoV bad smell

OSP	Group	%	Mean Rank	Rank Sum	Mann-Whitney U	Z	Significance ISP
ANT 1.5.3	¬ISP-Group	95,31	351,40	242819,00	3733,00	-6,84	8,01505E-12
	ISP-Group	4,69	598,71	20356,00			
ANT 1.6.0	¬ISP-Group	94,57	442,28	377704,00	12619,00	-6,13	8,85914E-10
	ISP-Group	5,43	621,47	30452,00			
ANT 1.7.0	¬ISP-Group	94,68	542,20	569308,50	17533,50	-8,36	6,49928E-17
	ISP-Group	5,32	782,82	46186,50			
Apache FOP 0.9.3	¬ISP-Group	87,84	419,88	315331,50	32955,50	-3,93	8,63924E-05
	ISP-Group	12,16	486,62	50608,50			
Apache FOP 0.9.4	¬ISP-Group	92,79	446,49	373712,50	23009,50	-2,92	0,00349524
	ISP-Group	7,21	516,01	33540,50			
CDK 2005	¬ISP-Group	96,79	478,97	447835,50	10255,50	-3,27	0,001059527
	ISP-Group	3,21	620,18	19225,50			
CDK 2006	¬ISP-Group	96,48	634,08	781826,00	21065,00	-3,28	0,001044369
	ISP-Group	3,52	787,89	35455,00			
CDK 1.0.1	¬ISP-Group	95,86	515,79	513206,50	17696,50	-3,36	0,000768016
	ISP-Group	4,14	605,45	26034,50			
Freenet 0.5.0	¬ISP-Group	95,39	352,49	240748,00	7162,00	-4,06	4,88597E-05
	ISP-Group	4,61	482,97	15938,00			
Freenet 0.5.1	¬ISP-Group	97,98	1151,70	2623577,00	27796,00	-9,81	9,75865E-23
	ISP-Group	2,02	1710,60	80398,00			
Freenet 0.7	¬ISP-Group	92,03	225,32	96213,50	4835,50	-4,03	5,47751E-05
	ISP-Group	7,97	315,31	11666,50			
Jmol 9	¬ISP-Group	92,31	83,29	12993,50	747,50	-2,65	0,008167248
	ISP-Group	7,69	105,50	1371,50			
Jmol 10	¬ISP-Group	96,15	90,28	15799,00	399,00	-1,89	0,058489508
	ISP-Group	3,85	122,00	854,00			
Jmol 11.2.14	¬ISP-Group	90,06	158,44	47375,00	2525,00	-4,95	7,30415E-07
	ISP-Group	9,94	239,48	7903,00			
OSCache 2.0.1	¬ISP-Group	94,62	45,39	3994,50	78,50	-3,24	0,001181681
	ISP-Group	5,38	75,30	376,50			
OSCache 2.1.1	¬ISP-Group	93,88	48,84	4493,00	215,00	-1,49	0,137117307
	ISP-Group	6,12	59,67	358,00			
OSCache 2.4.1	¬ISP-Group	93,81	55,49	5882,00	211,00	-2,76	0,005731867
	ISP-Group	6,19	79,86	559,00			
TVBrowser 0.9.1	¬ISP-Group	90,00	24,36	1096,00	61,00	-2,11	0,035035847
	ISP-Group	10,00	35,80	179,00			
TVBrowser 1.0	¬ISP-Group	92,43	92,47	15812,00	1106,00	-0,82	0,409447134
	ISP-Group	7,57	99,50	1393,00			
TVBrowser 2.6	¬ISP-Group	92,99	406,83	312854,50	16789,50	-5,37	7,84715E-08
	ISP-Group	7,01	509,03	29523,50			

Table A.6.5 - Man-Whitney test for the ISP bad smell

A 7 Visual analyses for class level bad smells

Figure A. 7.1 - Figure A. 7.3 show the DVAs for all OSPs with respect to class level bad smells.

Data Glass

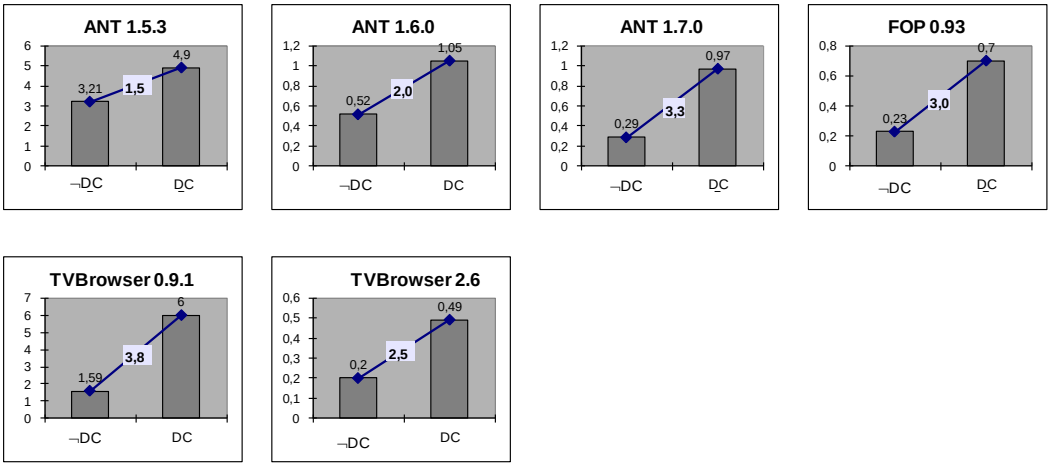


Figure A. 7.1 - Visual mean defect count analysis for the GC bad smell

God Glass

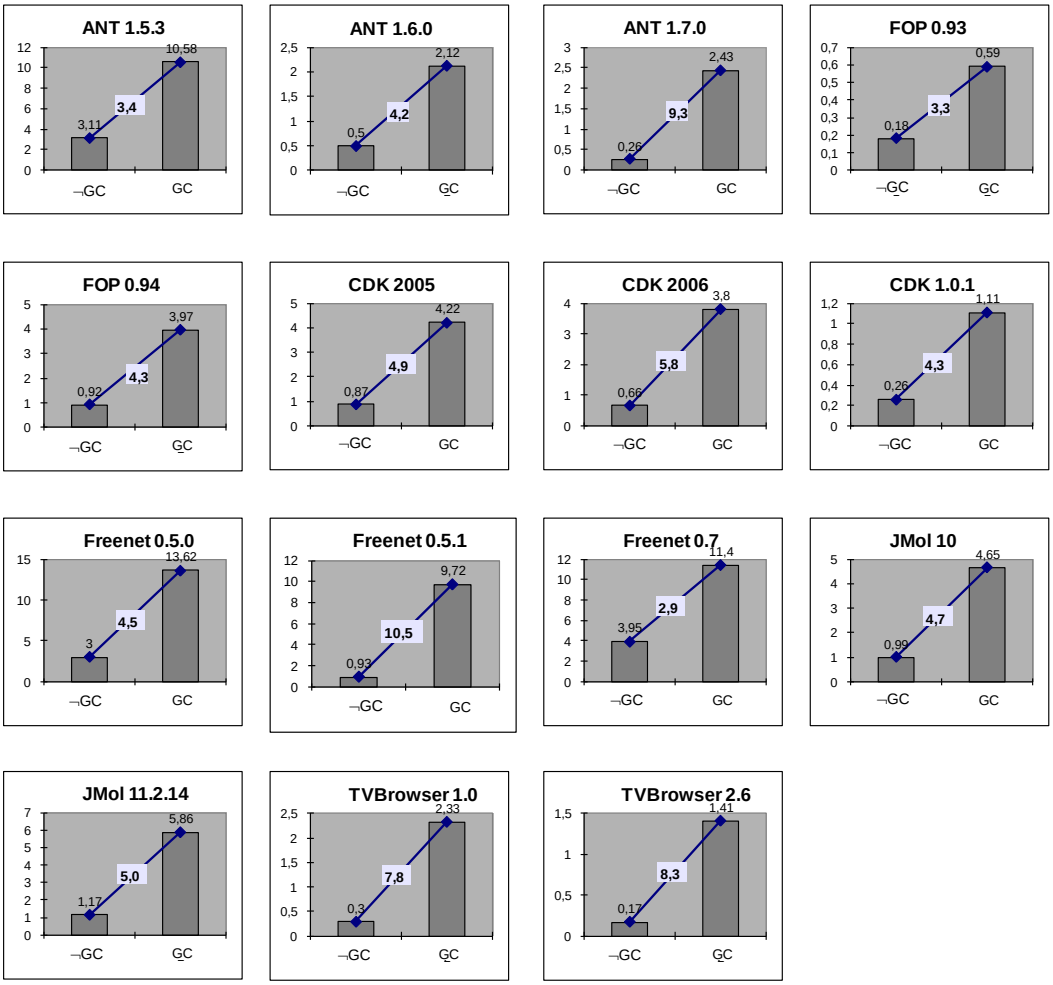


Figure A. 7.2 - Visual mean defect count analysis for the GC bad smell

Shotgun Surgery

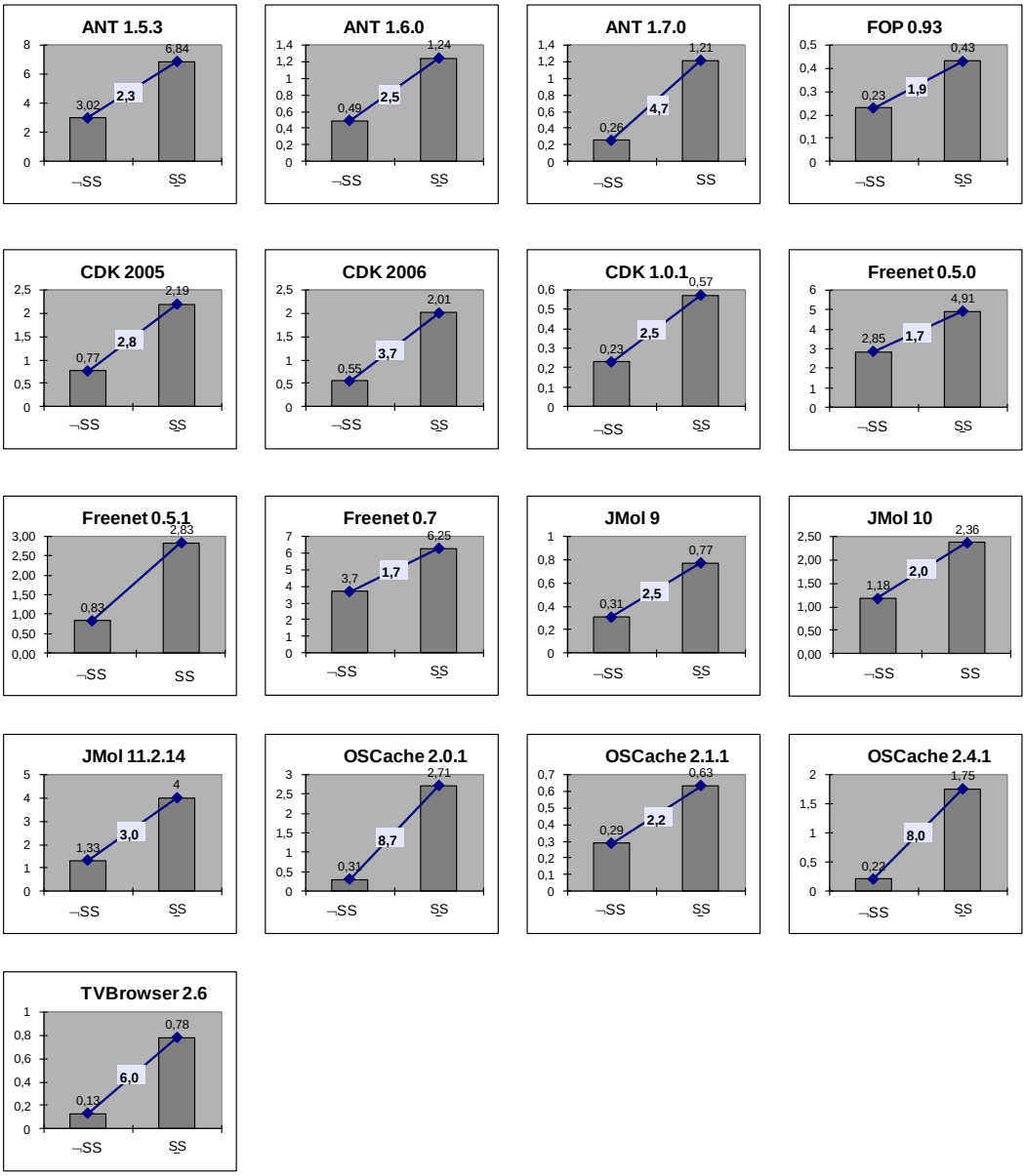


Figure A. 7.3 - Visual mean defect count analysis for the SS bad smell

A 8 Visual analyses for class level "Lack-Of" bad smells

Figure A. 8.1 - Figure A. 8.2 show the DVAs for all OSPs with respect to "Lack-Of" bad smells.

Lack of State

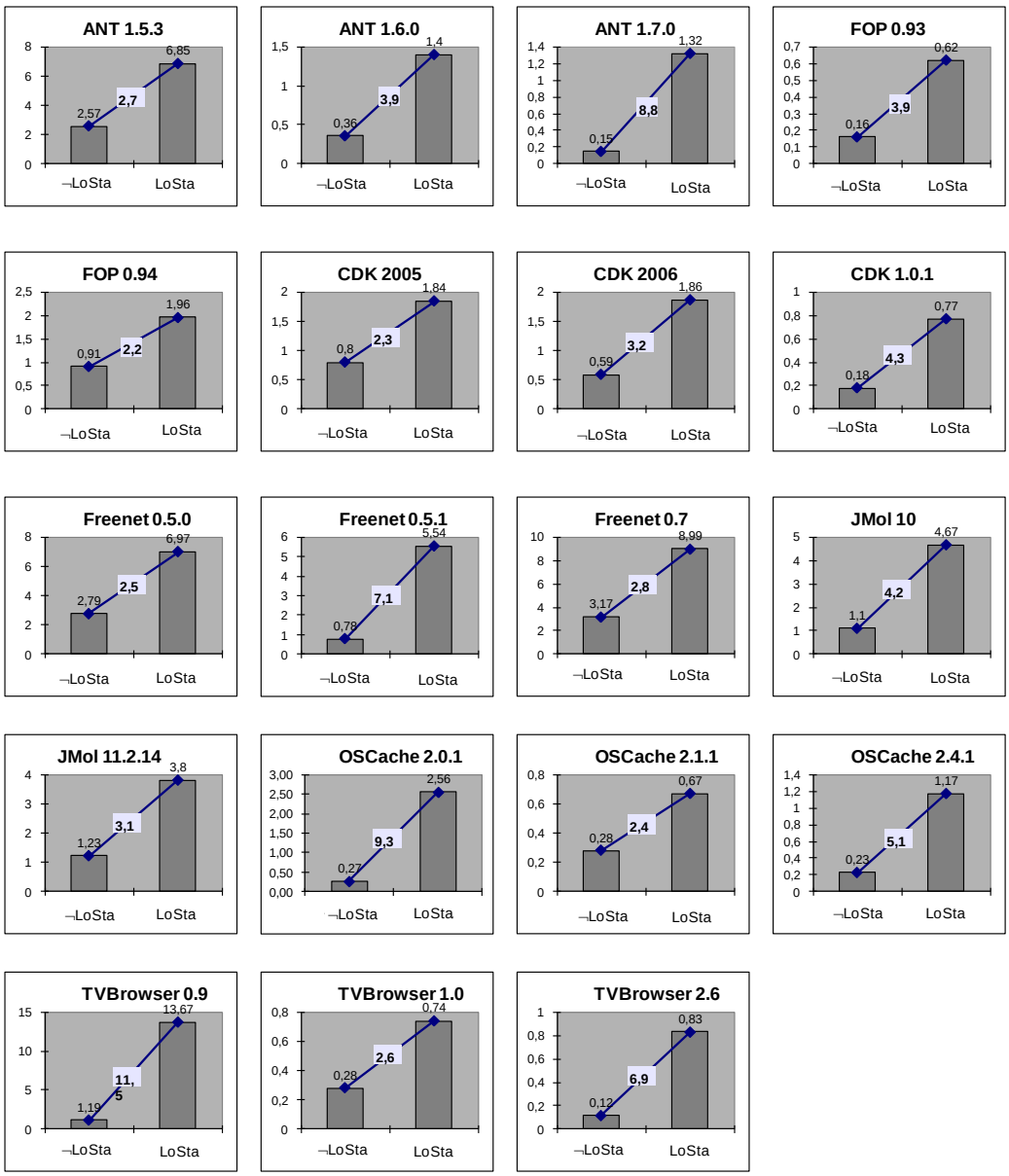


Figure A. 8.1 - Visual analysis for the LoSta bad smell

ISP

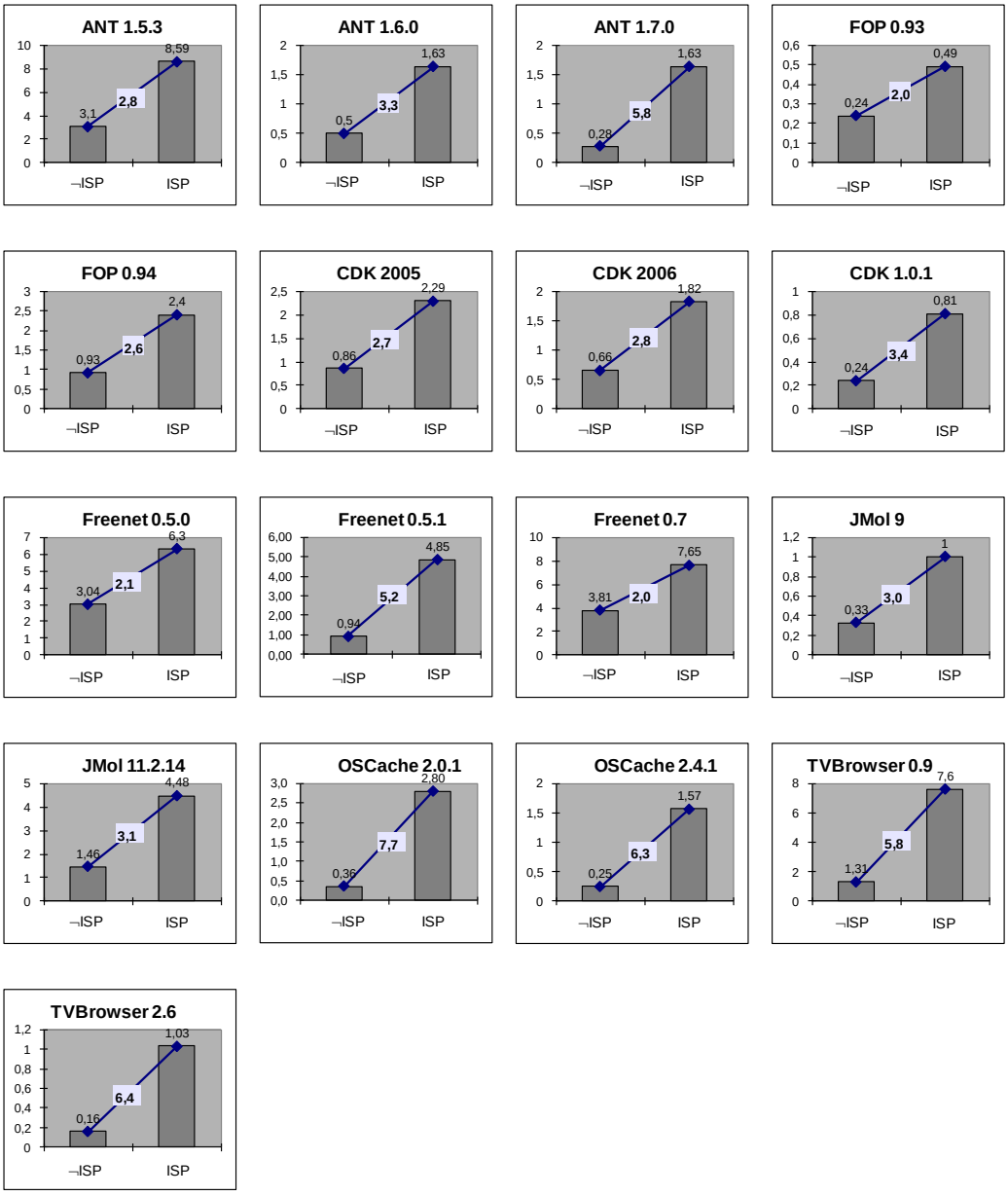


Figure A. 8.2 - Visual analysis for the ISP bad smell

A 9 Mann-Whitney test for package level bad smells

In this chapter, the results of the Mann-Whitney non parametric test for all package level bad smells analysed in Chapter 9 are presented.

OSP	Group	%	Mean Rank	Rank Sum	Mann-Whitney U	Z	Significance GP
ANT 1.5.3	¬GP-Group	59,00	30,12	1777,00	7,00	-3,13	0,001731313
	GP-Group	4,00	59,75	239,00			
ANT 1.6.0	¬GP-Group	70,00	35,67	2497,00	12,00	-3,18	0,001451503
	GP-Group	4,00	69,50	278,00			
ANT 1.7.0	¬GP-Group	75,00	38,21	2866,00	16,00	-4,05	5,02402E-05
	GP-Group	6,00	75,83	455,00			
Apache FOP 0.9.3	¬GP-Group	66,00	34,47	2275,00	64,00	-4,03	5,51502E-05
	GP-Group	9,00	63,89	575,00			
Apache FOP 0.9.4	¬GP-Group	69,00	37,68	2600,00	185,00	-2,59	0,009481878
	GP-Group	10,00	56,00	560,00			
CDK 2005	¬GP-Group	133,00	67,32	8953,50	42,50	-3,57	0,000356346
	GP-Group	5,00	127,50	637,50			
CDK 2006	¬GP-Group	156,00	79,83	12453,00	207,00	-2,96	0,003076585
	GP-Group	7,00	130,43	913,00			
CDK 1.0.1	¬GP-Group	97,00	49,80	4831,00	78,00	-2,31	0,02114546
	GP-Group	4,00	80,00	320,00			
Freenet 0.5.0	¬GP-Group	46,00	25,40	1168,50	87,50	-3,08	0,002077127
	GP-Group	10,00	42,75	427,50			
Freenet 0.5.1	¬GP-Group	82,00	46,68	3828,00	425,00	-2,11	0,034890145
	GP-Group	15,00	61,67	925,00			
Freenet 0.7	¬GP-Group	21,00	11,81	248,00	17,00	-2,31	0,020864902
	GP-Group	5,00	20,60	103,00			
Jmol 9	¬GP-Group	8,00	4,50	36,00	0,00	-1,84	0,065663192
	GP-Group	1,00	9,00	9,00			
Jmol 10	¬GP-Group	9,00	5,00	45,00			n/a
	GP-Group	0,00	0,00	0,00			
Jmol 11	¬GP-Group	32,00	16,69	534,00	6,00	-2,94	0,003283805
	GP-Group	4,00	33,00	132,00			
OSCache 2.0.1	¬GP-Group	12,00	6,50	78,00			n/a
	GP-Group	0,00	0,00	0,00			
OSCache 2.1.1	¬GP-Group	11,00	6,09	67,00	1,00	-1,40	0,162392238
	GP-Group	1,00	11,00	11,00			
OSCache 2.4.1	¬GP-Group	12,00	6,58	79,00	1,00	-1,41	0,158574494
	GP-Group	1,00	12,00	12,00			
TVBrowser 0.9.1	¬GP-Group	12,00	6,50	78,00			n/a
	GP-Group	0,00	0,00	0,00			
TVBrowser 1.0	¬GP-Group	30,00	16,13	484,00	19,00	-1,08	0,280123268
	GP-Group	2,00	22,00	44,00			
TVBrowser 2.6	¬GP-Group	127,00	65,50	8318,00	190,00	-3,38	0,000724054
	GP-Group	8,00	107,75	862,00			

Table A.9.1 - Man-Whitney test for the GP bad smell

OSP	Group	%	Mean Rank	Rank Sum	Mann-Whitney U	Z	Significance WSI
ANT 1.5.3	¬WSI-Group	63,00	32,00	2016,00			n/a
	WSI-Group	0,00	0,00	0,00			
ANT 1.6.0	¬WSI-Group	74,00	37,50	2775,00			n/a
	WSI-Group	0,00	0,00	0,00			
ANT 1.7.0	¬WSI-Group	80,00	40,77	3261,50	21,50	-0,85	0,394665759
	WSI-Group	1,00	59,50	59,50			
Apache FOP 0.9.3	¬WSI-Group	71,00	37,01	2628,00	72,00	-1,75	0,079750829
	WSI-Group	4,00	55,50	222,00			
Apache FOP 0.9.4	¬WSI-Group	74,00	39,12	2895,00	120,00	-1,44	0,150098719
	WSI-Group	5,00	53,00	265,00			
CDK 2005	¬WSI-Group	134,00	68,24	9143,50	98,50	-2,32	0,020099979
	WSI-Group	4,00	111,88	447,50			
CDK 2006	¬WSI-Group	150,00	80,10	12014,50	689,50	-1,87	0,06211495
	WSI-Group	13,00	103,96	1351,50			
CDK 1.0.1	¬WSI-Group	97,00	50,15	4864,50	111,50	-1,64	0,101088337
	WSI-Group	4,00	71,63	286,50			
Freenet 0.5.0	¬WSI-Group	51,00	27,31	1393,00	67,00	-1,76	0,079135866
	WSI-Group	5,00	40,60	203,00			
Freenet 0.5.1	¬WSI-Group	82,00	48,96	4014,50	611,50	-0,04	0,969000712
	WSI-Group	15,00	49,23	738,50			
Freenet 0.7	¬WSI-Group	26,00	13,50	351,00			n/a
	WSI-Group	0,00	0,00	0,00			
Jmol 9	¬WSI-Group	9,00	5,00	45,00			n/a
	WSI-Group	0,00	0,00	0,00			
Jmol 10	¬WSI-Group	9,00	5,00	45,00			n/a
	WSI-Group	0,00	0,00	0,00			
Jmol 11.2.14	¬WSI-Group	35,00	18,41	644,50	14,50	-0,29	0,771207518
	WSI-Group	1,00	21,50	21,50			
OSCache 2.0.1	¬WSI-Group	11,00	6,32	69,50	3,50	-0,58	0,5595837
	WSI-Group	1,00	8,50	8,50			
OSCache 2.1.1	¬WSI-Group	11,00	6,77	74,50	2,50	-0,93	0,351656707
	WSI-Group	1,00	3,50	3,50			
OSCache 2.4.1	¬WSI-Group	13,00	7,00	91,00			n/a
	WSI-Group	0,00	0,00	0,00			
TVBrowser 0.9.1	¬WSI-Group	12,00	6,50	78,00			n/a
	WSI-Group	0,00	0,00	0,00			
TVBrowser 1.0	¬WSI-Group	30,00	16,35	490,50	25,50	-0,44	0,658608207
	WSI-Group	2,00	18,75	37,50			
TVBrowser 2.6	¬WSI-Group	133,00	67,73	9008,50	97,50	-0,74	0,460817171
	WSI-Group	2,00	85,75	171,50			

Table A.9.2 - Man-Whitney test for the WSI bad smell

A 10 Visual analyses for the GP bad smell

Figure A. 10.1 shows the DVAs for all OSPs with respect to the GP bad smells.

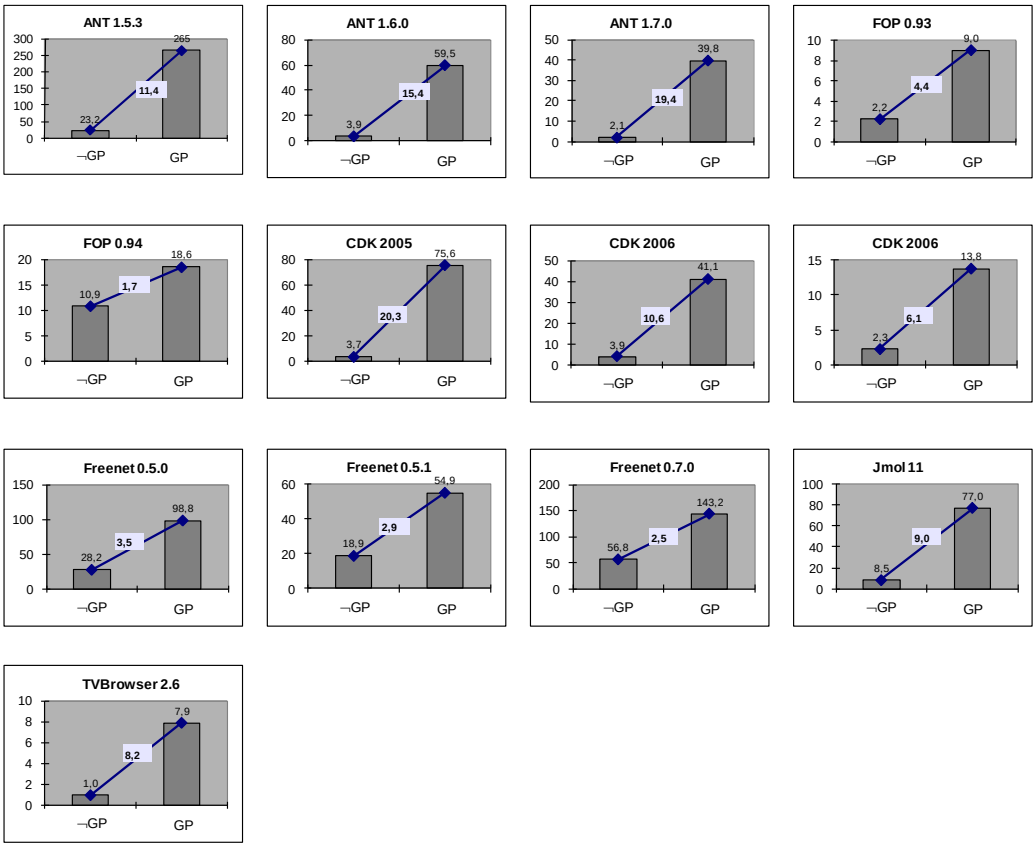


Figure A. 10.1 - Visual analysis for the GP smell

A 11 Statistical tests for DA

In this chapter, the results of the Mann-Whitney non-parametric test for DA and and the results of the Kruskal-Wallis test for DA (for three groups) are presented. The Mann-Whitney test analyses differences between files with lower than average (la) and higher than average (ha) DA metrics. The Kruskal-Wallis test analyses differences between *more than two* populations. In this case, the population (files in commercial software) has been divided into three groups:

- Group 1 – Files that have been modified by one author.
- Group 2 – Files that have been modified by two authors.
- Group 3 – Files that have been modified by three or more authors.

The null hypothesis is in both cases that the defect count is the same in all analysed groups; the alternative hypothesis is that it is not.

Distinct authors	Mean Rank	Rank Sum	Mann-Whitney-U	Z	Signifi-cance
la (non-fluctuating)	243,3777056	112440,5	5487,5	-5,07729282	0,01
ha (fluctuating)	359,7840909	15830,5			

Table A.11.1 - Mann-Whitney test for DA

Distinct authors	Mean Rank	Chi-Square	Df	Significance
1	243,37770563	27.128265129	2	1,2858E-6
2	324,4			
3	378,0862069			

Table A.11.2 - Kruskal-Wallis test for DA, three groups

Based on the results of both tests, the null-hypothesis has to be rejected, i.e. there is a significant difference within the analysed groups.

A 12 Statistical tests for FC

Table A.12.1 shows the results of the Mann-Whitney non-parametric test for the FC metric.

Distinct authors	Mean Rank	Rank Sum	Mann-Whitney-U	Z	Significance
stable	219,7210366	72068,5	18112,5	-7,09793206	1,2664E-12
unstable	315,7443820	56202,5			

Table A.12.1 - Mann-Whitney test for FC

A 13 Statistical tests for CF

Table A.13.1 and Table A.13.2 show the results of the Mann-Whitney non-parametric test for CF-SUM and CF-MAX.

CF-SUM	Mean Rank	Rank Sum	Mann-Whitney-U	Z	Significance
la	236,85761589404	71531	3477	-3,05078248	0,209884194
ha	277,014778325123	6889			

Table A.13.1 - Mann-Whitney non-parametric test for CF-SUM

CF-MAX	Mean Rank	Rank Sum	Mann-Whitney-U	Z	Significance
la	242,5021277	56988	29258	-2	0,12912925
ha	262,137037	70777			

Table A.13.2 Mann-Whitney non-parametric test for CF-MAX

A 14 Statistical tests for age

Table A.14.1 shows the results of the Kruskal-Wallis test for the variable age.

age	Mean Rank	Chi-Square	df	Significance
F-N	252,77272727	6,39162204893984	2	0,05
F-Y	229,54661017			
F-O	275,8			

Table A.14.1 - Statistical test for age

A 15 Non-parametric tests for combined analyses

Age X Stability

Table A.15.1 shows the results of the Kruskal-Wallis test for the Age x Stability combined analysis.

Categories	Mean Rank	Chi-Square	df	Significance
N-unst	218,1036585	53,9611337284692	5	2,1347E-10
Y-unst	202,4230769			
O-unst	244,9178082			
N-stab	316,6573034			
Y-stab	320,962963			
O-stab	312,1612903			

Table A.15.1 - Kruskal-Wallis test for Age X Stability

Age X Fluctuation

Table A.15.2 shows the results of the Kruskal-Wallis test for the age x fluctuation combined analysis.

Categories	Mean Rank	Chi-Square	df	Significance
N-nF	243,1709957	31,1190498476427	5	8,8737E-06
Y-nF	221,9045455			
O-nF	263,2933884			
N-F	353,5909091			
Y-F	334,625			
O-F	383,8928571			

Table A.15.2 - Kruskal-Wallis test for Age X Fluctuation

Stability X Fluctuation

Table A.15.3 shows the results of the Kruskal-Wallis test for stability x fluctuation.

Categories	Mean Rank	Chi-Square	df	Significance
stab-nF	216,7852564103	66,1249753506352	3	2,882E-14
stab-F	276,96875			
unstab-nF	298,69			
unstab-F	407,1071428571			

Table A.15.3 - Kruskal-Wallis test for Stability X Fluctuation

LIST OF FIGURES

Figure 1.1 - Data, Information, Knowledge 22
Figure 1.2 - Generic approach 24
Figure 2.1 - Measures, metrics, attributes and entities....38
Figure 2.2 - Types of data and measurement scales 40
Figure 2.3 - Measurement scales and operations....41
Figure 2.4 - Histograms for different distributions 46
Figure 3.1 - Risk = damage x probability of failure 50
Figure 4.1 - Decision levels and corresponding decisions....61
Figure 5.1 - Test process characteristics 71
Figure 5.2 - Documentation needs during testing 71
Figure 5.3 - Communication characteristics 73
Figure 6.1 - Empirical approach....81
Figure 6.2 - Quality and quality indicators. An example 83
Figure 6.3 - Simple DVA: Mean defect count vs. file age....86
Figure 6.4 - Combined DVA for Ant: Mean defect count vs. file age and stability 89
Figure 6.5 - Testers' experience is needed when 90
Figure 7.1 - Defect count and characteristics of a file....96
Figure 7.2 - Computing the defect count for files in OSPs 99
Figure 7.3 - Defect correction density in HTs 103
Figure 7.4 - Defect detection per algorithm level....104
Figure 7.5 - Percentage of HTs for different defect counts per HT....106
Figure 8.1 - Pareto distribution of defects in files of the OSCache project 112
Figure 8.2 - Pareto distribution of defects....113
Figure 8.3 - Pareto distribution of defects in code 115
Figure 8.4 - Pareto distribution of defects in code across releases 116
Figure 9.1 - DVA: Visual mean defect count analysis for the FE bad smell 131
Figure 9.2 - Method level bad smell analysis 132
Figure 9.3 - DVA:Visual mean defect count analysis for the GM bad smell 133

Figure 9.4 - Class level bad smell analysis....134
Figure 9.5- Lack-of Patterns....136
Figure 9.6 - Package level bad smell analysis....137
Figure 10.1 - Communication of defects to the project team....148
Figure 10.2 - DVA for DA....149
Figure 10.3 - DVA for DA (three groups)....150
Figure 10.4 - Mean defect count for stable vs. unstable files....151
Figure 10.5 - DVA for CF-SUM and CF-MAX....152
Figure 10.6 - Simple DVA for ANT: Mean defect count vs. file age....153
Figure 10.7 - Combined DVA: Mean defect count vs.....154
Figure 10.8 - Combined DVA: Mean defect count vs. file age x fluctuation....155
Figure 10.9 - Combined DVA: Mean defect count vs. file age X stability....156
Figure 11.1 - Results of the thesis....166

APPENDIX

Figure A. 1.1 - Decision refinement for the system testing process 177
Figure A. 7.2 - Visual mean defect count analysis for the GC bad smell 199
Figure A. 7.3 - Visual mean defect count analysis for the SS bad smell 200
Figure A. 8.1 - Visual analysis for the LoSta bad smell 201
Figure A. 8.2 - Visual analysis for the ISP bad smell 202
Figure A. 10.1 - Visual analysis for the GP smell 205

LIST OF TABLES

Table 2.1 - Empirical software engineering research 36
Table 2.5 - Parametric and non-parametric tests for different designs 44
Table 3.2 - Fundamental ideas of the GQM approach 52
Table 3.3 - Models and indicators for defects in software 57
Table 5.1 - Participants' characteristics 69
Table 6.1 - Category definition matrix for 88
Table 7.1 - Subject programs 98
Table 7.2 - Algorithm performance 101
Table 7.3 - Algorithm performance 102
Table 7.4 - Algorithm performance 103
Table 7.5 - Average, maximum and minimum defect count per HT 105
Table 7.6 - Methods for classification of HTs 109
Table 8.1 - Pareto distribution of defects in files across releases 114
Table 8.2 – Pareto principle, related work 120
Table 9.1 - Hypotheses for method level bad smells 127
Table 9.2 - Hypotheses for class level bad smells 128
Table 9.3 - Hypotheses for lack-of-pattern bad smells 129
Table 9.4 - Hypotheses for package level bad smells 130
Table 9.5 - Which bad smells are the best indicators for defects in code? 139
Table 9.6 – Bad smells and defects - (Shatnawi and Li 2006) vs. results of this thesis 141
Table 10.1 - Characteristics of the analysed CS 146
Table 10.2 - Independent variables 147
Table 10.3 - Descriptive statistics for DA 149
Table 10.4- Category definition matrix for stability x fluctuation 154
Table 10.5 - Category definition matrix for age x fluctuation 155
Table 10.6- Category definition matrix for Age X Stability 156
Table 11.1 - Pareto analysis summary 169
Table 11.2 - Bad smell analysis summary 171
Table 11.3 - History analysis summary 173

Table A.1.1 - Applying the decision hierarchy to compare testing approaches 179
Table A.2.1 - Pareto distribution of defects in code 180
Table A.4.1 - Man-Whitney test for the FE bad smell.....185
Table A.4.2 - Man-Whitney test for the GM bad smell 186
Table A.5.1- Man-Whitney test for the MC bad smell.....187
Table A.5.2 - Man-Whitney test for the DC bad smell 188
Table A.5.3 - Man-Whitney test for the GC bad smell 189
Table A.5.4 - Man-Whitney test for the SS bad smell 190
Table A.5.5 - Man-Whitney test for the RB bad smell 191
Table A.5.6 - Man-Whitney test for the MC bad smell 192
Table A.6.1 - Man-Whitney test for the LoB bad smell 193
Table A.6.2 - Man-Whitney test for the LoSta bad smell 194
Table A.6.3 - Man-Whitney test for the LoStra bad smell 195
Table A.6.4 - Man-Whitney test for the LoV bad smell 196
Table A.6.5 - Man-Whitney test for the ISP bad smell 197
Table A.9.1 - Man-Whitney test for the GP bad smell 203
Table A.9.2 - Man-Whitney test for the WSI bad smell 204
Table A.11.1 - Mann-Whitney test for DA 206
Table A.11.2 - Kruskal-Wallis test for DA, three groups 206
Table A.12.1 - Mann-Whitney test for FC 207
Table A.11.2 - Kruskal-Wallis test for DA, three groups 206
Table A.12.1 - Mann-Whitney test for FC 207
Table A.13.1 - Mann-Whitney non-parametric test for CF-SUM 207
Table A.13.2 - Mann-Whitney non-parametric test for CF-MAX 207
Table A.14.1 - Statistical test for age 207
Table A.15.1 - Kruskal-Wallis test for Age X Stability 208
Table A.15.2 - Kruskal-Wallis test for Age X Fluctuation 208
Table A.15.3 - Kruskal-Wallis test for Stability X Fluctuation 209

List of Abbreviations

BS	Bad Smell
CF	Co-changed files
CS	Commercial Software
DA	Distinct Authors
DC	Data Class
DTS	Defect Tracking System
DVA	Defect Variance Analyses Diagram
FC	Frequency of Change
FE	Feature Envy
F-N	Newborn File
F-O	Old File
F-Y	Young File
GC	God Class
GM	God Method
GP	God Package
HT	History Touch
ISP	ISP Violation
LM	Last Minute Fix
LoB	Lack of Bridge
LOC	Lines of Code
LoSta	Lack of State
LoStr	Lack of Strategy
LoV	Lack of Visitor
MC	Misplaced Class
Mod	Moderation
OSP	Open Source Program
PostR	Post-release
PreR	Pre-release
RB	Refused Bequest
SS	Shotgun Surgery
VCS	Versioning Control System
WSI	Wide Subsystem Interface

PUBLICATIONS

This thesis builds on the following papers:

(Illes-Seifert and Paech 2010): Illes-Seifert, T. and B. Paech. 2010. "Exploring the relationship of a file's history and its fault-proneness: An empirical method and its application to open source programs." *Journal of Information & Software Technology*, May, pp. 539-558.

(Illes-Seifert and Paech 2009): Illes-Seifert, T. and B. Paech 2009. "The Vital Few and Trivial Many: An Empirical Analysis of the Pareto Distribution of Defects." Pp. 151-162 in *Software Engineering 2009: Fachtagung des GI-Fachbereichs Softwaretechnik*, edited by P. Liggesmeyer, G. Engels, J. Münch, J. Dörr, and N. Riegel. Kaiserslautern: Lecture Notes in Informatics (LNI 143).

(Illes-Seifert and Paech 2008a): Illes-Seifert, T. and B. Paech 2008. "Exploring the Relationship of a File's History and Its Fault-Proneness: An Empirical Study." Pp. 13-22 in *Proceedings of the Testing: Academic & industrial Conference - Practice and Research Techniques, TAIC PART 2008*. IEEE Computer Society, Washington, DC.

(Illes-Seifert and Paech 2008b): Illes-Seifert, T. and B. Paech 2008. "Exploring the relationship of history characteristics and defect count: an empirical study." Pp. 11-15 in *Proceedings of the 2008 Workshop on Defects in Large Software Systems, DEFECTS '08*. ACM, New York, NY.

(Illes-Seifert and Paech 2008c): Illes-Seifert, T. and B. Paech. 2008. "On the Role of Communication, Documentation and Experience during System Testing - An Interview Study." *Proceedings of the Multikonferenz Wirtschaftsinformatik MKWI, Workshop Prozessinnovation mit Unternehmenssoftware - PRIMIUM*.

(Borner, Illes, and Paech 2007a): Borner, L., T. Illes, and B. Paech 2007. "Entscheidungen im Testprozess." Pp. 247-248 in *Software Engineering 2007 (SE 2007), Fachtagung des GI-Fachbereichs Softwaretechnik*, edited by W.-G. Bleek, J. Raasch, and H. Züllighoven. Hamburg: Lecture Notes in Informatics (LNI).

(Borner, Illes-Seifert, and Paech 2007b): Borner, L., T. Illes-Seifert, and B. Paech. 2007. "The Testing Process - A Decision Based Approach." *Proceedings of the international Conference on Software Engineering Advances (ICSEA)*, August 25 - 31, pp. 41-51. IEEE Computer Society, Washington, DC.

(Illes and Paech 2006): Illes, T. and B. Paech 2006. "An Analysis of Use Case Based Testing Approaches Based on a Defect Taxonomy." Pp. 211-222 in *Software Engineering Techniques: Design for Quality, IFIP International Federation for Information Processing*, edited by K. Sacha. Springer Verlag.

List of the publications that are not part of the thesis (listed in chronological order)

(Illes, Herrmann, Paech, and Rückert, 2005): Illes, T., A. Herrmann, B. Paech, and J. Rückert. 2005. "Criteria for Software Testing Tool Evaluation. A Task Oriented View." *Proceedings of the 3rd World Congress for Software Quality*, September, pp. 213-222.

(Illes and Paech, 2005): Illes, T. and B. Paech. 2005. "What is a Good Test Specification?" *Proceedings of the 2nd European Symposium about Verification and Validation of Software Systems (VVSS 2005)*, November 24, pp. 15-24.

(Illes et al. 2006): Illes, T., H. Pohlmann, T. Roßner, A. Schlatter, and M. Winter. 2006. *Software-Testmanagement Planung, Design, Durchführung und Auswertung von Tests - Methodenbericht und Analyse unterstützender Werkzeuge*. Hannover: Heise Zeitschriften Verlag.

(Herrmann et al. 2006): Herrmann, A., B. Paech, S. Kirn, D. Kossmann, G. Müller, C. Binnig, M. Gilliot, T. Illes, L. Lowis, and D. Weiß. 2006. "Durchgängige Qualität von Unternehmenssoftware." *Industrie Management*, pp. 59-61.

(Illes and Paech, 2006): Illes, T. and B. Paech. 2006. "From "V" to "U" or: How Can We Bridge the V-Gap Between Requirements and Test?" *Software & Systems Quality Conferences*, May 10.

(Illes-Seifert et al. 2007): Illes-Seifert, T., A. Herrmann, M. Geisser, and T. Hildenbrand. 2007. "The Challenges of Distributed Software Engineering and Requirements Engineering: Results of an Online Survey." *Proceedings of the 1st International Global Requirements Engineering Workshop (GREW07)*, August 27, pp. 55-65.

(Illes, Borner, and Paech, 2007): Illes, T., L. Borner, and B. Paech 2007. "Testfallgenerierung aus semi-formalen Use Cases." Pp. 387-392 in *Informatik 2007 Informatik trifft Logistik, Beiträge der 37. Jahrestagung der Gesellschaft für Informatik e.V.*, edited by R. Koschke, O. Herzog, K-H. Rödiger, and M. Ronthaler. Bremen: Lecture Notes in Informatics (LNI), Gesellschaft für Informatik (GI).

(Weiß et al. 2007a): Weiß, D., J. Kaack, S. Kirn, M. Gilliot, L. Lowis, G. Müller, A. Herrmann, C. Binnig, T. Illes, B. Paech, and D. Kossmann. 2007. "Die SIKOSA-Methodik." *Wirtschaftsinformatik*, pp. 188-198.

(Weiß et al. 2007b): Weiß, D, J. Kaack, S. Kirn, M. Gilliot, L. Lowis, G. Müller, A. Herrmann, C. Binnig, T. Illes, B. Paech, and D. Kossmann. 2007. "Die SIKOSA-Methodik - Unterstützung der industriellen Softwareproduktion durch methodisch integrierte Softwareentwicklungsprozesse." *Wirtschaftsinformatik*, pp. 188-198.

(Weiß et al. 2007c): Weiß, D., S. Kirn, B. Paech, G. Müller, D. Kossmann, A. Herrmann, M. Gilliot, L. Lowis, C. Binnig, and T. Illes 2007. "Ein Beitrag zur Berücksichtigung von Compliance in der Softwareentwicklung." in *Aktuelle Trends in der Softwareforschung. Tagungsband zum doIT Softwareforschungstag*, edited by K. Haasis, A. Heinzl, and D. Klumpp. Mannheim.

BIBLIOGRAPHY

Adams, E.N. 1984. "Optimizing Preventive Service of Software Products." *IBM Journal of Research and Development*, pp. 2-14.

Ahlowalia, N. 2002. "Testing from Use Cases Using Path Analysis Technique." *Proceedings of the International Conference On Software Testing Analysis & Review, Orlando, Florida*.

Albright, S. C., W. L. Winston, and C. Zappe. 1999. *Data Analysis and Decision Making with Microsoft Excel*. USA: Duxbury Pr.

Amland, S. 2000. "Risk-based testing: risk analysis fundamentals and metrics for software testing including a financial application case study." *J. Syst. Softw.*, pp. 287-295.

Andersson, C. and P. Runeson. 2007. "A Replicated Quantitative Analysis of Fault Distributions in Complex Software Systems." *IEEE Trans. Softw. Eng.*, pp. 273-286.

Arisholm, E. and L.C. Briand. 2006. "Predicting fault-prone components in a Java legacy system." *Proceedings of the 2006 ACM/IEEE international Symposium on Empirical Software Engineering (Rio de Janeiro, Brazil, September 21 - 22, 2006)*, pp. 8-17.

Aurum, A., C. Wohlin, and A. Porter. 2006. "Aligning Software Project Decisions: a Case Study." *International Journal of Software Engineering and Knowledge Management*, pp. 795 – 718.

Bach, J. 1999. "Heuristic Risk-Based Testing." *Software Testing and Quality Engineering*.

Bach, J. 2003. "Troubleshooting Risk-Based Testing." *Software Testing and Quality Engineering*.

Basili, V. R., L.C. Briand, and W.L. Melo. 1996. "A validation of object-oriented design metrics as quality indicators." *IEEE Transactions on Software Engineering*, pp. 751-761.

Basili, V., G. Caldiera, and D. Rombach 1994. "The Goal Question Metric Appraoch." in *Encyclopedia of Software Engineering*. Wiley.

Basili, V. R. and F. Lanubile. 1999. "Building Knowledge through Families of Experiments." *IEEE Transactions of Software Engineering*, pp. 456-473.

Basili, V.R. and D.M. Weiss. 1984. "A Methodology for Collecting Valid Software Engineering Data." *IEEE Transactions on Software Engineering*, November, pp. 728-738.

Beizer, B. 1990. *Software Testing Techniques*. 2nd ed. New York: Van Nostrand Reinhold.

Bell, R. M. 2005. "Predicting the Location and Number of Faults in Large Software Systems." *IEEE Trans. Softw. Eng.*, April, pp. 340-355.

Bellinger, G, D Castro, and A Mills. 2010. "Systems Thinking." Data, Information, Knowledge and Wisdom. Retrieved May 07, 2010 (http://www.systems-thinking.org/dikw/dikw.htm).

Bell, R.M., T.J. Ostrand, and E.J. Weyuker. 2006. "Looking for bugs in all the right places." *Proceedings of the 2006 international Symposium on Software Testing and Analysis (Portland, Maine, USA, July 17 - 20, 2006), ISSTA '06*, pp. 61-72.

Binder, R. V. 1999. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co.

Binkley, A. B. and S. R. Schach. 1998. "Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures." *Proceedings of the 20th international conference on Software engineering, ICSE'98*, pp. 452–455.

Boehm, B. W., J. R. Brown, H. Kaspar, M. Lipow, G. McLeod, and M. Merritt. 1978. "Characteristics of Software Quality." *TRW series on software, Vol 1., North Holland*.

Borner, L., T. Illes, and B. Paech 2007. "Entscheidungen im Testprozess." Pp. 247-248 in *Software Engineering 2007 (SE 2007), Fachtagung des GI-Fachbereichs Softwaretechnik*, edited by W.-G. Bleek, J. Raasch, and H. Züllighoven. Hamburg: Lecture Notes in Informatics (LNI).

Borner, L., T. Illes-Seifert, and B. Paech. 2007. "The Testing Process - A Decision Based Approach." *Proceedings of the international Conference on Software Engineering Advances (ICSEA)*, August 25 - 31, pp. 41-51.

Briand, L. C., J. W. Daly, and J. Wüst. 1998. "A Unified Framework for Cohesion Measurement in Object-OrientedSystems." *Empirical Softw. Engg.*, Jul., pp. 65-117.

Briand, L. C., J. W. Daly, and J. K. Wüst. 1999. "A Unified Framework for Coupling Measurement in Object-Oriented Systems." *IEEE Trans. Softw. Eng.*, Jan., pp. 91-121.

Briand, L. C., P. Devanbu, and W. Melo. 1997. "An investigation into coupling measures." *Proceedings of the 19th International Conference on Software Engineering, ICSE'97*, pp. 412–421.

Briand, L. and Y. Labiche. 2002. "A UML-based Approach to System Testing." Carleton University.

Briand, L. C., Y. Labiche, and Y. Wang. 2003. “An Investigation of Graph-Based Class Integration Test Order Strategies.” *IEEE Trans. Softw. Eng.*, Jul., pp. 594-607.

Briand, L. C., J. Wüst, J. W. Daly, and D. V. Porter. 2000. “Exploring the relationship between design measures and software quality in object-oriented systems.” *J. Syst. Softw.*, May, pp. 245-273.

Burnstein, I., T. Suwannasart, and C. R. Carlson. 1996. “Developing a Testing Maturity Model for Software Test Process Evaluation and Improvement.” *Proceedings of the IEEE International Test Conference on Test and Design Validity*.

Card, S., J. Mackinlay, and B. Shneiderman. 1999. *Readings in Information Visualization - Using Vision to Think*. USA: Morgan Kaufmann.

Carniello, A., M. Jino, and M. Lordello. 2004. “Structural Testing with Use Cases.” *Proceedings of the WER04 - Workshop em Engenharia de Requisitos*.

Cartwright, M. and M. Shepperd. 2000. “An Empirical Investigation of an Object-Oriented Software System.” *IEEE Trans. Softw. Eng.*, Aug., pp. 786-796.

Chidamber, S. R. and C. F. Kemerer. 1994. “A Metrics Suite for Object Oriented Design.” *IEEE Trans. Softw. Eng.*, Jun., pp. 476-493.

Chikofsky, E. J. and J. H. Cross II. 1990. “Reverse Engineering and Design Recovery: A Taxonomy.” *IEEE Software*, Jan., pp. 13-17.

Čubranić, D. and G. C. Murphy. 2003. “Hipikat: recommending pertinent software development artifacts.” *Proceedings of the 25th international Conference on Software Engineering (Portland, Oregon, May 03 - 10, 2003)*.

Cui, Y., L. Li, and S. Yao. 2009. “A New Strategy for Pairwise Test Case Generation.” *Proceedings of the Third International Symposium on Intelligent Information Technology Application*, pp. 303-306.

Dahlstedt, A. 2005. “Guidelines Regarding Requirements Engineering Practices in order to Facilitate System Testing.” *Proceeding of the 11th International Workshop on Requirements Engineering: Foundation for Software Quality*.

Davenport, T and L Prusak. 1998. *Working Knowledge*. Boston, MA: Harvard Business Scholl Press.

Denaro, G., S. Morasca, and M. Pezzè. 2002. “Deriving models of software fault-proneness.” *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering Ischia, Italy*, pp. 361 - 368.

Denaro, G. and M. Pezzè. 2002. “An empirical evaluation of fault-proneness models.” *roceedings of the International Conference on Software Engineering (ICSE 2002), Orlando, Florida, USA*, pp. 241-251.

Do, H. and G. Rothermel. 2006. “An empirical study of regression testing techniques incorporating context and lifetime factors and improved cost-benefit models.” *Proceedings of the 14th ACM SIGSOFT international Symposium on Foundations of Software Engineering (Portland, Oregon, USA, November 05 - 11, 2006). SIGSOFT '06/FSE-14*, pp. 141-151.

Dowson, M. 1997. “The Ariane 5 software failure.” *SIGSOFT Softw. Eng. Notes, ACM Press*, March, pp. 84.

Dromey, R. G. 1996. “Concerning the Chimera [software quality].” *IEEE Software*, pp. 33-43.

Dutoit, A. H., R. McCall, R. Mistrik, and B. Paech. 2006. *Rationale Management in Software Engineering*. Berlin Heidelberg: Springer-Verlag.

Eisenhardt, K.M. 1989. “Building Theories from Case Study Research.” *Academy of Management Review*, pp. 532-550.

Elbaum, S., G. Rothermel, S. Kanduri, and A. Malishevsky. 2004. “Selecting a Cost-Effective Test Case Prioritization Technique.” *Software Quality Control*, Sept., pp. 185-210.

Emam, K. E., S. Benlarbi, N. Goel, and S. N. Rai. 2001. “Comparing case-based reasoning classifiers for predicting high risk software components.” *J. Syst. Softw.*, Jan., pp. 301-320.

Emam, K. E., W. Melo, and J. C. Machado. 2001. “The prediction of faulty classes using object-oriented design metrics.” *J. Syst. Softw.*, Feb., pp. 63-75.

Emerson, D.J. and J. Strenio 2000. “Boxplots and Batch Comparison.” in *Understanding Robust and Exploratory Data Analysis*. New York, Toronto: John Wiley&Sons, Wiley Classic Library Edition.

Endres, A. 1975. “An analysis of errors and their causes in system programs.” *SIGPLAN Not.*, pp. 327-336.

Fenton, N. E. and N. Ohlsson. 2000. “Quantitative Analysis of Faults and Failures in a Complex Software System.” *IEEE Trans. Softw. Eng.*, Dec., pp. 797-814.

Fenton, N. E. and S. L. Pfleeger. 1998. *Software Metrics: A Rigorous and Practical Approach*. 2nd ed. Boston, MA, USA: Course Technology.

Few, S. 2009. *Now You see it, Simpple Visualization Techniques for Quantitative Analysis*. Oakland, CA: Analytics Press.

Fischer, M., M. Pinzger, and H. Gall. 2003. “Populating a Release History Database from Version Control and Bug Tracking Systems.” *Proceedings of the international Conference on Software Maintenance (September 22 - 26, 2003). ICSM.*

Fowler, M., K. Beck, J. Brant, W. Opdyke, and D. Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. NY, USA: Addison-Wesley.

Girba, T., M. Lanza, and S. Ducasse. 2005. "Characterizing the Evolution of Class." *Software Maintenance and Reengineering*, pp. 2- 11.

Gras, J.-J., R. Gupta, and E. Perez-Minana. 2006. "Generating a Test Strategy with Bayesian Networks and Common Sense." *TAIC-PART '06: Proceedings of the Testing: Academic & Industrial Conference on Practice And Research Techniques*, pp. 29--40.

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

Grieskamp, W., M. Lepper, W. Schulte, and N Tillmann. 2001. "Testable Use Cases in the Abstract State Machine Language." *Proceedings of the Second Asia-Pacific Conference on Quality Software*.

Guo, L., Y. Ma, B. Cukic, and H. Singh. 2004. "Robust Prediction of Fault-Proneness by Random Forests." *Proceedings of the 15th international Symposium on Software Reliability Engineering (November 02 - 05, 2004). ISSRE*, pp. 417-428.

Gyimothy, T., R. Ferenc, and I. Siket. 2005. "Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction." *IEEE Trans. Softw. Eng.*, pp. 897-910.

Halstead, M. H. 1977. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc.

Hatton, L. 1997. "Reexamining the Fault Density-Component Size Connection." *IEEE Softw.*, Mar., pp. 89-97.

Hatton, L. 2008. "The role of empiricism in improving the reliability of future software." Keynote Talk at TAIC PART 2008. Retrieved May 10, 2010 (http://www.leshatton.org/Documents/TAIC2008-29-08-2008.pdf).

Holschuh, T., M Pauser, K. Herzig, T. Zimmermann, R. Premraj, and A. Zeller. 2009. "Predicting defects in SAP Java code: An experience report." *Proceedings of the 31th International Conference on Software Engineering*, May, pp. 172-181.

IEEE Std, 610.12. 1990. *IEEE Standard Glossary of Software Engineering Terminology*. New York, USA: IEEE Standards Association.

IEEE Std., 829. 1998. *IEEE standard for software test documentation*. USA: Software Engineering Technical Committee of the IEEE Computer Society.

IEEE Std., 829-1998. 1998. *IEEE standard for software test documentation*. USA: Software Engineering Technical Committee of the IEEE Computer Society.

Illes, T. and B. Paech 2006. “An Analysis of Use Case Based Testing Approaches Based on a Defect Taxonomy.” Pp. 211-222 in *Software Engineering Techniques: Design for Quality, IFIP International Federation for Information Processing*. Boston: Springer.

Illes, T., H. Pohlmann, T. Roßner, A. Schlatter, and M. Winter. 2006. *Software-Testmanagement Planung, Design, Durchführung und Auswertung von Tests - Methodenbericht und Analyse unterstützender Werkzeuge*. Hannover: Heise Zeitschriften Verlag.

Illes-Seifert, T. and B. Paech. 2008. “On the Role of Communication, Documentation and Experience during System Testing - An Interview Study.” *Proceedings of the Multikonferenz Wirtschaftsinformatik MKWI, Workshop Prozessinnovation mit Unternehmenssoftware - PRIMIUM*.

Illes-Seifert, T. and B. Paech 2009. “The Vital Few and Trivial Many: An Empirical Analysis of the Pareto Distribution of Defects.” Pp. 151-162 in *Software Engineering 2009: Fachtagung des GI-Fachbereichs Softwaretechnik*, edited by P. Liggesmeyer, G. Engels, J. Münch, J. Dörr, and N. Riegel. Kaiserslautern: Lecture Notes in Informatics (LNI 143).

Illes-Seifert, T. and B. Paech. 2010. “Exploring the relationship of a file's history and its fault-proneness: An empirical method and its application to open source programs.” *Information & Software Technology*, May, pp. 539-558.

ISO/IEC. 2001. *Software engineering - Product quality - Part 1*. Geneva, Switzerland.

ISO/IEC Standard, 9126. 2001. *ISO/IEC Standard 9126: Software Engineering -- Product Quality*. Part 1: International Organization for Standarization.

ISTQB. 2007. “Standard glossary of terms used in Software Testing Version 2.0.” *Produced by the ‘Glossary Working Party’ International Software Testing Qualifications Board*, Dec..

Jeffrey, D. and N. Gupta. 2008. “Experiments with test case prioritization using relevant slices.” *J. Syst. Softw.*, Feb., pp. 196-221.

Jéron, T., J.-M. Jézéquel, Y. Le Traon, and P. Morel. 1999. “Efficient Strategies for Integration and Regression Testing of OO Systems.” *Proceedings of the 10th International Symposium on Software Reliability Engineering*, pp. 260.

Jiang, Y., B. Cukic, and Y. Ma. 2008. “Techniques for evaluating fault prediction models.” *Empirical Softw. Eng*, Oct., pp. 561-595.

Jiang, Y., B. Cukic, and Y Ma. 2008. “Techniques for evaluating fault prediction models.” *Empirical Softw. Eng.*, pp. 561-595.

Jones, C. 2008. *Applied Software Measurement, Global Analysis of Productivity and Quality*. 3rd ed. USA: McGraw-Hill.

Juran, J.M. and Jr.F.M Gryna. 1988. *Quality Control Handbook*. 4th ed. USA: McGraw Hill.

Juristo, N. and A. Moreno. 2001. *Basics of Software Engineering Experimentation*. Boston MA., USA,: Kluwer Academic Publishers.

Juristo, N., A. M. Moreno, and S. Vegas 2003. "Limitations of empirical testing technique knowledge." Pp. 1-38 in *Software Engineering And Knowledge Engineering, Lecture Notes on Empirical Software Engineering*. River Edge, NJ: World Scientific Publishing Co.

Juristo, N., A. M. Moreno, and S. Vegas. 2004. "Reviewing 25 Years of Testing Technique Experiments." *Empirical Softw. Eng.*, Mar., pp. 7-44.

Kaâniche, M. and K. Kanoun. 1996. "Reliability of a commercial telecommunications system." *Proceedings of the the Seventh international Symposium on Software Reliability Engineering (ISSRE '96)*, October 30 - November 02.

Kaner, C., J. Bach, and B. Pettichord. 2002. *Lessons Learned in Software Testing, A Context Driven Approach*. New York, Canada: John Wiley & Sons, Inc.

Khoshgoftaar, T.M., E.B. Allen, R. Halstead, G. P. Trio, and R. M. Flass. 1998. "Using Process History to Predict Software Quality." *IEEE Computer*, pp. 66-72.

Khoshgoftaar, T.M., E.B. Allen, W.D. Jones, and J.P. Hudepohl. 2000. "Classification-tree models of software-quality over multiple releases." *IEEE Transactions on Reliability*, pp. 4-11.

Khoshgoftaar, T. M., N. Seliya, and N. Sundaresh. 2006. "An empirical study of predicting software faults with case-based reasoning." *Software Quality Control*, Jun., pp. 85-111.

Kim, S., T. Zimmermann, E. J. Whitehead, and A. Zeller. 2008. "Predicting faults from cached history." *Proceedings of the 1st Conference on India Software Engineering Conference ISEC*.

Koomen, T. and M. Pol. 1999. *Test Process Improvement, A step-by-step guide to structured testing*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Koru, A. G., K. E. Emam, D. Zhang, H. Liu, and D. Mathew. 2008. "Theory of relative defect proneness." *Empirical Softw. Eng.*, Oct., pp. 473-498.

Koru, A. G. and J. Tian. 2003. "An empirical comparison and characterization of high defect and high complexity modules." *J. Syst. Softw.*, pp. 153-163.

Koru, A. G., D. Zhang, and H. Liu. 2007. "Modeling the Effect of Size on Defect Proneness for Open-Source Software." *Proceedings of the Third international*

Workshop on Predictor Models in Software Engineering (May 20 - 26, 2007). International Conference on Software Engineering, pp. 10.

Kvale, S. 1996. *InterViews: An Introduction to Qualitative Research Interviewing*. London: Sage.

Layman, L., G. Kudrjavets, and N. Nagappan. 2008. "Iterative identification of fault-prone binaries using in-process metrics." *Proceedings of the Second ACM-IEEE international Symposium on Empirical Software Engineering and Measurement (Kaiserslautern, Germany, October 09 - 10, 2008), ESEM '08*, pp. 206-212.

Lessmann, S., B. Baesens, C. Mues, and S Pietsch. 2008. "Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings." *IEEE Trans. Softw. Eng.*, Jul., pp. 485-496.

Ludewig, J. and H. Lichter. 2007. *Software Engineering, Grundlagen, Menschen, Prozesse, Techniken*. Heidelberg: dpunkt.verlag.

Lukowicz, P., E. Heinz, L. Prechelt, and W. Tichy. 1994. "Experimental evaluation in computer science: A quantitative study." *Technical Report 17/94*.

Marinescu, R. 2002. *Measurement and Quality in Object-Oriented Design*. Politehnica" University of Timisoara: PhD thesis.

Martin, R.C. 1996. "Interface Segregation Principle." objectmentor. C++ Report. Retrieved May 10, 2010 (http://www.objectmentor.com/resources/articles/isp.pdf).

Martin, R.C. 2000. "Design Principles and Patterns." objectmentor. Object Menotor. Retrieved May 10, 2010 (http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf).

McCabe, T.J. 1976. "A complexity measure." *IEEE Transactions on Software Engineering*.

McCall, J. A., P. K. Richards, and G. F. Walters. 1977. "Factors in Software Quality." *Nat'l Tech.Information*, Vol. 1, 2 and 3.

Mende, T. and R. Koschke. 2009. "Revisiting the evaluation of defect prediction models." *Proceedings of the 5th international Conference on Predictor Models in Software Engineering (Vancouver, British Columbia, Canada, May 18 - 19, 2009). PROMISE '09*, pp. 1-10.

Mens, T. and T. Tourwe. 2004. "A survey of software." *Software Engineering, IEEE Transactions*, pp. 126-139.

Menzies, T., A. Dekhtyar, J. Distefano, and J. Greenwald. 2007. "Problems with Precision: A Response to "Comments on 'Data Mining Static Code Attributes to Learn Defect Predictors'"." *IEEE Trans. Softw. Eng.*, pp. 637-640.

Menzies, T., R. Lutz, and I. C. Mikulski. 2003. "Better Analysis of Defect Data at NASA." *International Conference on Software Engineering and Knowledge Engineering, SEKE'03*, pp. 607-611.

Menzies, T., B. Turhan, A. Bener, G. Gay, B. Cukic, and Y. Jiang. 2008. "Implications of ceiling effects in defect predictors." *Proceedings of the 4th international Workshop on Predictor Models in Software Engineering (Leipzig, Germany, May 12 - 13, 2008), PROMISE '08*, pp. 47-54.

Mockus, A., P. Zhang, and P. Li. 2005. "Predictors of customer perceived software quality." *Proceedings of the 27th International Conference on Software Engineering, ICSE'05*, pp. 225–233.

Montgomery, D.C. 1997. *Design and Analysis of Experiments*. 4th ed. New York: John Wiley&Sons.

Morasca, S. 2001. "Software Measurement." Pp. 239 - 276 in *Handbook of Software Engineering and Knowledge Engineering - Volume 1: Fundamentals*. London, UK: World Scientific Publishing Co. Pte. Ltd.

Mosley, D. J. and B. A. Posey. 2002. *Just Enough Software Test Automation*. Upper Saddle River, NJ, USA: Prentice Hall.

Munson, J. C. and T. M. Khoshgoftaar. 1992. "The Detection of Fault-Prone Programs." *IEEE Trans. Softw. Eng.*, pp. 423-433.

Myers, G. J. 1979. *The Art of Software Testing*. New York, NY, USA: John Wiley & Sons, Inc.

Myrtveit, I. and E. Stensrud. 1999. "A Controlled Experiment to Assess the Benefits of Estimating with Analogy and Regression Models." *IEEE Trans. Softw. Eng.*, Jul, pp. 510-525.

Myrtveit, I., E. Stensrud, and M. Shepperd. 2005. "Reliability and Validity in Comparative Studies of Software Prediction Models." *IEEE Trans. Softw. Eng.*, May, pp. 380-391.

Nagappan, N. and T. Ball. 2005. "Use of relative code churn measures to predict system defect density." *Proceedings of the 27th international Conference on Software Engineering (St. Louis, MO, USA, May 15 - 21, 2005), ICSE '05*, pp. 284-292.

Nagappan, N., T. Ball, and A. Zeller. 2006. "Mining metrics to predict component failures." *Proceedings of the International Conference on Software Engineering (ICSE 2006), Shanghai, China*, pp. 452 - 461.

Nebut, C., F. Fleurey, Y. Le Traon, and J.-M. Jézéquel. 2003. "Requirements by contracts allow automated system testing." *Proceedings of the 14th. IEEE International Symposium on Software Reliability Engineering (ISSRE'03)*.

Ohlsson, N. and H. Alberg. 1996. “Predicting Fault-Prone Software Modules in Telephone Switches.” *IEEE Trans. Softw. Eng.*, Dec., pp. 886-894.

Ohlsson, M., A. von Mayrhauser, B. McGuire, and C. Wohlin. 1999. “Code decay analysis of legacy software through successive releases.” *Proc. IEEE Aerospace Conf.*.

Ostrand, T. J. and E. J. Weyuker. 2002. “The distribution of faults in a large industrial software system.” *SIGSOFT Softw. Eng. Notes*, Jul., pp. 55-64.

Ostrand, T. J., E. J. Weyuker, and R. M. Bell. 2004. “Where the bugs are.” *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pp. 86--96.

Ostrand, T. J., E. J. Weyuker, and R. M. Bell. 2005. “Predicting the Location and Number of Faults in Large Software Systems.” *IEEE Trans. Softw. Eng.*, pp. 340--355.

Paech, B. and K. Kohler 2003. “Task-driven Requirements in object-oriented development.” in *Perspectives on Requirements Engineering*. Boston: Kluwer Academic Publishers.

Perry, D. E., A. A. Porter, and Jr. L. G. Votta. 2006. “Empirical Studies - Looking to Future.” Retrieved May 20, 2010 (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.3.894).

Pfleeger, S. L. 2005. “Soup or Art? The Role of Evidential Force in Empirical Software Engineering.” *IEEE Softw.*, Jan., pp. 66-73.

Pighin, M. and A. Marzona. 2003. “An Empirical Analysis of Fault Persistence Through Software Releases.” *International Symposium on Empirical Software Engineering*, pp. 206.

Pighin, M. and A. Marzona. 2003. “An Empirical Analysis of Fault Persistence Through Software Releases.” *Proceedings of the 2003 International Symposium on Empirical Software Engineering*, pp. 206.

Pinkster, I., B. van de Burgt, D. Janssen, and E. van Veenendaal. 2004. *Successful Test Management – An Integral Approach*. Berlin: Springer.

Porter, A. A. and R. W. Selby. 1990. “Evaluating techniques for generating metric-based classification trees.” *J. Syst. Softw.*, Jul., pp. 209-218.

Purushothaman, R. 2005. “Toward Understanding the Rhetoric of Small Source Code Changes.” *IEEE Trans. Softw. Eng.*, pp. 511-526.

Reh, J.F. 2005. “Pareto's Principle - The 80-20 Rule, How the 80/20 rule can help you be more effective.” about.com Management. http://management.about.com/cs/generalmanagement/a/Pareto081202_2.htm.

Retrieved May 07, 2010 (http://management.about.com/cs/generalmanagement/a/Pareto081202_2.htm).

Riel, A.J. 1996. *Object-Oriented Design Heuristics*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Rupp, C. and S. Queins. 2003. "Vom Use-Case zum Test-Case." *OBJEKTspektrum*.

Ryser, J. and M. Glinz. 2003. "SCENT: A Method Employing Scenarios to Systematically Derive Test Cases for System Test." University of Zürich.

Salkind, N J. 2007. *Encyclopedia of Measurement and Statistics 3-Volume Set*. California, USA: SAGE Publications.

Schäfer, H. 2004. "Risk based testing, How to choose what to test more and less." www.cs.tut.fi. Retrieved May 10, 2010 (http://www.cs.tut.fi/tapahtumat/testaus04/schaefer.pdf).

Schröter, A., T. Zimmermann, R. Premraj, and A. Zeller. 2006. "If Your Bug Database Could Talk." *Proceedings of the 5th International Symposium on Empirical Software Engineering*, pp. 18-20.

Seaman, C.B. 1999. "Qualitative Methods in Empirical Studies of Software Engineering." *IEEE Transactions on Software Engineering*, July/August, pp. 557-572.

Shatnawi, R. and W. Li. 2006. "An Investigation of Bad Smells in Object-Oriented Design." *ITNG '06: Proceedings of the Third International Conference on Information Technology: New Generations*, pp. 161--165.

Shepperd, M. and G. Kadoda. 2001. "Comparing Software Prediction Techniques Using Simulation." *IEEE Trans. Softw. Eng.*, pp. 1014-1022.

Siegel, S. and N.J. Castellan. 1988. *Non-parametric statistics for the Behavioral Sciences*. 2nd ed. New York: McGraw-Hill.

Sjoberg, D. I., T. Dyba, and M. Jorgensen. 2007. "The Future of Empirical Methods in Software Engineering Research." *2007 Future of Software Engineering (May 23 - 25, 2007). International Conference on Software Engineering*, pp. 358-378.

Śliwerski, J., T. Zimmermann, and A. Zeller. 2005. "When do changes induce fixes?" *Proceedings of the 2005 international Workshop on Mining Software Repositories*, May.

Sliwersky, J., T. Zimmermann, and A. Zeller. 2005. "When do changes induce fixes? On Fridays." *Proceedings of the International Workshop on Mining Software Repositories (MSR 2005)*.

Spillner, A. and T. Linz. 2010. *Software Testing Foundations - A Study Guide for the Certified Tester Exam - Foundation Level - ISTQB compliant*. Heidelberg: dpunkt.verlag.

Srikanth, H. 2006. *Value-Driven System Level Test Case Prioritization*. North Carolina State University: Doctoral Thesis. UMI Order Number: AAI3223211.

Srivastva, P. R., K. Kumar, and G. Raghurama. 2008. "Test case prioritization based on requirements and risk factors." *SIGSOFT Softw. Eng. Notes*, pp. 1-5.

State Justice Institute. 1999. "A Judge's Deskbook on the Basic Philosophies and Methods of Science." Retrieved May 10, 2010 (http://www.judicialstudies.unr.edu/JudgesDeskbookFullDoc.pdf).

Subramanyam, R. and M. S. Krishnan. 2003. "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects." *IEEE Transactions*, pp. 297–310.

Szabo, R. M. and T. M. Khoshgoftaar. 1995. "An assessment of software quality in a C++ environment." *Proceedings of the Sixth International Symposium on Software Reliability Engineering, Toulouse, France*, pp. 240- 249.

Thwin, M. M. and T. Quah. 2005. "Application of neural networks for software quality prediction using object-oriented metrics." *J. Syst. Softw.*, May, pp. 147-156.

Tichy, W. F. 1998. "Should Computer Scientists Experiment More?" *IEEE Computer*, May., pp. 32-40.

Tjortjis, C. and P Layzell. 2001. "Expert Maintainers' Strategies and Needs when Understanding Software: A Case Study Approach." *Proceedings of the Eighth Asia-Pacific on Software Engineering Conference (APSEC)*.

Tmmi Foundation; eds. van Veenendaal, E. 2009. *Test Maturity Model Integration, TMMi*. Ireland: Tmmi Foundation.

van der Aalst, L. 2006. "Risk Based Test Strategy Judged." *7th International Conference on Software Testing, ICSTEST, Düsseldorf, Germany*, May.

Wainer, J., C. G. Novoa Barsottini, D. Lacerda, and L. R. Magalhães de Marco. 2009. "mpirical evaluation in Computer Science research published by ACM." *Inf. Softw. Technol.*, Jun., pp. 1081-1085.

Weyuker, E.J. and T.J. Ostrand. 2008. "Comparing methods to identify defect reports in a change management database." *Proceedings of the 2008 Workshop on Defects in Large Software Systems (Seattle, Washington, July 20, 2008)*, pp. 27-31.

Weyuker, E. J. and T. J. Ostrand. 2008. "Comparing methods to identify defect reports in a change management database." *Proceedings of the 2008 Workshop on Defects in Large Software Systems, ACM*, July 20.

Weyuker, E. J., T. J. Ostrand, and R.M. Bell. 2007. “Using Developer Information as a Factor for Fault Prediction.” *Proceedings of the Third international Workshop on Predictor Models in Software Engineering (May 20 - 26, 2007), International Conference on Software Engineering*.

Weyuker, E. J., T. J. Ostrand, and R. M. Bell. 2008. “Do too many cooks spoil the broth? Using the number of developers to enhance defect prediction models.” *Empirical Softw. Engg.*, Oct., pp. 539-559.

Whittle, J., J. Chakraborty, and I. Krueger. 2005. “Generating Simulation and Test Models from Scenarios.” *Proceedings of the 3rd World Congress for Software Quality, München*.

Wohlin, C., P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. 2000. *Experimentation in Software Engineering: an Introduction*. Norwell, MA, USA: Kluwer Academic Publishers.

Wolf, T. and A. H. Dutoit. 2004. “Sysiphus: Combining system modeling with collaboration and rationale.” wwwbruegge.in.tum.de. Retrieved May 10, 2010 (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.87.8769).

Wu, S., Q. Wang, and Y. Yang. 2008. “Quantitative analysis of faults and failures with multiple releases of softpm.” *Proceedings of the Second ACM-IEEE international Symposium on Empirical Software Engineering and Measurement (Kaiserslautern, Germany, October 09 - 10, 2008). ESEM ’08*, pp. 198-205.

Yin, R.K. 2003. *Case Study Research, Design and Methods*. USA: SAGE Publications.

Zelkowitz, M.V. and D Wallace. 1998. “Experimental models for validating technology.” *IEEE Computer*, pp. 23-31.

Zhang, M., T. Hall, N. Baddoo, and P. Wernick. 2008. “Do bad smells indicate "trouble" in code?” *Proceedings of the 2008 Workshop on Defects in Large Software Systems (Seattle, Washington, July 20 - 20, 2008)*, pp. 43-44.

Zhang, H. and X. Zhang. 2007. “Comments on "Data Mining Static Code Attributes to Learn Defect Predictors".” *IEEE Trans. Softw. Eng.*, Sept..

Zimmermann, T., N. Nagappan, H. Gall, E. Giger, and B. Murphy. 2009. “Cross-project defect prediction: a large scale experiment on data vs. domain vs. process.” *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (Amsterdam, The Netherlands, August 24 - 28, 2009). ESEC/FSE ’09*, pp. 91-100.

Zimmermann, T., N. Nagappan, and A. Zeller 2008. “Predicting Bugs from History.” in *Software Evolution*. Berlin: Springer.

Zimmermann, T., R. Premraj, and A. Zeller. 2007. "Predicting Defects for Eclipse." *Proceedings of the Third international Workshop on Predictor Models in Software Engineering*, May.