Cost-Optimisation
of
Analytical Software Quality Assurance

Stefan Wagner

Institut für Informatik
der Technischen Universität München

Cost-Optimisation of Analytical Software Quality Assurance

Stefan Wagner

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr. Helmut Veith
Prüfer der Dissertation:		
	1.	Univ.-Prof. Dr. Dr. h.c. Manfred Broy
	2.	Univ.-Prof. Dr. Peter Liggesmeyer, Technische Universität Kaiserslautern

Die Dissertation wurde am 01.08.2006 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 14.02.2007 angenommen.

Abstract

Analytical software quality assurance (SQA) constitutes a significant part of the total development costs of a software system. Most estimates say that about 50% of the costs can be attributed to defect-detection and removal. Hence, this is a promising area for cost-optimisation. Various defect-detection techniques can be used to improve the quality of software. Those techniques differ in many respects and it is consolidated knowledge that the best result in terms of defect-detection can be obtained by combining several diverse techniques.

The main question in that context is then how to use those different techniques in a cost-optimal way. In detail, this boils down to the questions (1) which techniques to use, (2) in which order, (3) with how much effort, and (4) with which distribution over the components. The major problem stated by several authors is that the costs and benefits of SQA and the interrelationships of the influencing factors are still not totally understood.

The main contribution of this dissertation is an analytical and stochastic model of the economics of analytical SQA, based on expected values. The model is more general than existing analytical models, as it is able to handle different types of techniques, i.e., static and dynamic ones. Nevertheless, the proposed model is more concrete than comparable economic models of QA because it includes the technical factors that influence defect detection. The model can be used (1) to analyse different techniques theoretically and (2) to optimise the SQA in a company using historical project data. Relevant data for the model is collected from the literature to give average values for different techniques and defects. This allows an easier application of the model in practice. Based on the empirical data the model is subject to sensitivity analysis resulting in a quantitative order of importance of the input factors. This allows to determine which factors are most beneficial to be analysed in more detail.

Finally, an approach to identify defect-prone components of a system based on measures of detailed design models allows to concentrate the SQA on these components. It thereby improves the efficiency of defect-detection.

The approach for cost-optimisation of SQA is calibrated using several case studies. A complete validation was not feasible because the model needs calibration over several projects in a company to yield reasonable predictions. The largest case study is an evaluation of model-based testing in an industrial project in which the network controller of an automotive network was tested. Further studies were done in various domains and evaluating different techniques in different phases. All these case studies provided valuable feedback on the model and also contributed to the body of knowledge on the efficiency of the techniques.

Preface

I would like to thank all the people who helped in making this project a success. First I want to express my gratitude to my supervisor Manfred Broy who has always been very supportive although my topics are not very close to his main research interests. Only the close connections to industry and the multitude of opportunities that he offered me made this dissertation possible. I also would like to thank my co-supervisor Peter Liggesmeyer who gave valuable feedback.

In addition, my thanks also goes to all the people of the research group of Prof. Broy which I had the pleasure to work in over the last three and a half years. The atmosphere has always been friendly and supportive. Special thanks goes to Sebastian Winter, Frank Marschall, Silke Müller, Florian Deißenböck and Céline Laurent. Many discussions – not only about informatics – have been very valuable to me and I hope that this will continue.

A lot of my research was influenced by the work of and discussions with Alexander Pretschner and Wolfgang Prenninger. Especially the MOST case study was mainly their effort. Thank you! I am also grateful to Jan Romberg who has been a great colleague. It was a pleasure to share the office with him. Further thanks go to Ulrike Hammerschall, Jan Jürjens, Leonid Kof, Christian Pfaller, Markus Pister, Markus Pizka, Herbert Reiter, and Tilman Seifert for valuable discussions and joined work.

Also the work of the secretary and the IT administration at the chair made working there a pleasure. Thank you! I also like to thank all the students that I supervised. It has been an interesting experience that I nearly always enjoyed. I am especially grateful to Claudia Koller who did excellent work on the evaluation of bug finding tools.

I am am very grateful to Sandro Morasca, Bev Littlewood, and Bernd Freimut who gave valuable feedback on various stages of my work although they have not been directly involved. Thanks goes also to Helmut Fischer and Peter Trischberger who allowed me to do interesting case studies with industrial data.

My friends and family have always supported what I do and I cannot thank them enough for that. Especially my mother and late father made me the way I am. I am sure he would be very proud of me. I am especially grateful to Josef Stadlmaier who improved my English in some chapters of this dissertation.

Last but not least, I would like to thank Patricia de Crignis for the wonderful time we had together, the experiences we shared, and the encouragement she often gave to me.

Contents

Abstract iii

Preface v

1 Introduction 1
1.1 Background and Motivation . . . 1
1.2 Problem . . . 2
1.3 Contribution . . . 2
1.4 Related Work . . . 3
1.5 Contents . . . 3

2 Software Quality Assurance 5
2.1 Software Quality . . . 5
2.1.1 Views and Standards . . . 5
2.1.2 Reliability . . . 7
2.2 Defects . . . 8
2.2.1 Definitions . . . 8
2.2.2 Defect Types . . . 9
2.3 Techniques . . . 11
2.3.1 General . . . 11
2.3.2 Test . . . 12
2.3.3 Inspection . . . 14
2.3.4 Static Analysis Tools . . . 15
2.4 Quality Models . . . 15
2.4.1 Classification . . . 16
2.4.2 Software Reliability Models . . . 18
2.4.3 Software Quality Costs . . . 20

3 State of the Art 23
3.1 Defect Introduction and Removal . . . 23
3.2 Software Quality Economics . . . 24
3.3 Analytical Models of Quality Assurance . . . 25
3.3.1 Inspection . . . 26
3.3.2 Test . . . 26
3.3.3 General . . . 27
3.4 Literature Reviews . . . 28
3.5 Defect-Proneness . . . 28

4 Quality Economics 31
4.1 Basic Factors 31
4.1.1 Software Quality Costs Revisited 31
4.1.2 Influencing Factors 32
4.2 An Analytical Model 35
4.2.1 Basics 35
4.2.2 Model Components 37
4.2.3 Forms of the Difficulty Functions 41
4.2.4 Example 43
4.2.5 Empirical Knowledge 44
4.2.6 Sensitivity Analysis 58
4.2.7 Discussion 64
4.3 Practical Model 65
4.3.1 Basics 65
4.3.2 Components 66
4.3.3 Empirical Knowledge 67
4.3.4 Sensitivity Analysis 69
4.3.5 Cost-Optimisation 69
4.4 Application 70
4.4.1 General 70
4.4.2 Embedding in the V-Modell XT 72
4.5 Example 75
4.5.1 Basic Setting 75
4.5.2 Manual Planing 76
4.5.3 Optimised Planing 76
4.6 Summary 78

5 Defect-Proneness of Components 81
5.1 General 81
5.2 Metrics 82
5.2.1 Development Process 82
5.2.2 Measures of the Static Structure 82
5.2.3 Measure of Behaviour 84
5.2.4 Metrics Suite 86
5.2.5 Properties 86
5.3 Fault Proneness 87
5.4 Failure Proneness 88
5.5 Summary 89

6 Case Studies 91
6.1 Automatic Collision Notification 91
6.1.1 Description 91
6.1.2 Device Design 91
6.1.3 Results 93
6.1.4 Summary 94
6.2 MOST NetworkMaster 94
6.2.1 Approach 95

6.2.2 Model and System 98
6.2.3 Tests 100
6.2.4 Observations and Interpretations 103
6.2.5 Summary 111
6.3 Mobile Services Backend Systems 112
6.3.1 Projects 113
6.3.2 Approach 114
6.3.3 Analysis 116
6.3.4 Discussion 121
6.3.5 Summary 123
6.4 Telecommunication Systems 123
6.4.1 Geometric Progression of Failure Probabilities 123
6.4.2 Approach 124
6.4.3 Results 125
6.4.4 Summary 129

7 Summary and Outlook 131
7.1 Summary 131
7.2 Outlook 132
7.2.1 Practical and Theoretical Improvements 132
7.2.2 An Integrated Approach 133

1 Introduction

The bitterness of poor quality remains long after low pricing is forgotten!
Leon M. Cautillo

1.1 Background and Motivation

The quality of software is an important factor in the success of any software system today. Business value for the vendor as well as for the customer strongly depends on quality. Quality decides whether the system is used at all and whether it provides the expected benefits for the customer. For the vendor, quality of software stands for the needed effort for fault removal and adding new functionality.

For example in [44] it is stated that the success of the software industry in Germany mainly depends on how well it can manage the quality of its software products. However, quality is a difficult concept in itself. How do you decide if something is *of quality*? How do you measure and compare? Often quality has a flavour of subjectivity as works of art do. Still, software quality can be boiled down to costs and benefits in the economical sense because usually software is written for some business reasons.

Software quality assurance (SQA) comprises methods and techniques to assure that a software has a certain (desired) quality. There are two fundamental approaches: (1) constructive and (2) analytical techniques. Constructive techniques improve the process of software development to *prevent* defects and other problems from being introduced. In contrast to that, analytical techniques appraise the software with the aim to *detect* defects and other problems. These two types have completely different characteristics. Defect-detection deals with existing defects whereas defects should not come into existence using defect prevention. Hence, the analysis of these types of techniques should be done separately.

The costs for analytical SQA are significant. Many estimates say that analytical SQA constitutes about 50% of the total development costs. This figure is attributed to Myers [144]. Jones [94] still assigns 30–40% of the development costs to quality assurance and defect removal. In a study from 2002, the National Institute of Standards and Technology of the United States [179] even 80% of the development costs are assigned to the detection and removal of defects. There is a huge opportunity for cost savings in this area and that is why we focus on analytical techniques, also called defect-detection techniques, in the following.

A further point of view is the distribution of defects over the components of a software. It is often observed that this distribution follows a Pareto principle with 20% of the components being responsible for 80% of the defects [20, 55]. This suggests that SQA should not be uniformly distributed over the components but be concentrated on the components that contain the most defects.

1.2 Problem

To save costs for SQA, we need to optimise its usage in the development process with the aim to reduce costs and increase benefits. There are many ways to address this problem. In particular two approaches are possible: (1) improve existing techniques or (2) use the existing techniques in a cost-optimal way. We concentrate on the second approach because this offers project leaders and quality managers the possibility to plan the quality assurance with respect to economic considerations. This approach boils down to four main questions:

1. Which techniques should be used?
2. In what sequence?
3. With how much effort?
4. With which distribution over the components?

The answers to these questions involve an analysis and comparison of the different available techniques. It needs to be determined whether they are applicable to a specific kind of software and how their defect detection capabilities compare. Ntafos discusses in [149] that "cost is clearly a central factor in any realistic comparison but it is hard to measure, data are not easy to obtain, and little has been done to deal with it." Costs are the only possibility to compare all the influencing factors in SQA because they are the unit all those factors can be reduced to. Because of that Rai et al. identify in [173] mathematical models of the economics of software quality assurance as an important research area. "A better understanding of the costs and benefits of SQA and improvements to existing quantitative models should be useful to decision-makers."

The distribution of effort over the components is also very hard to answer, as we cannot know beforehand which components contain the most defects. However, there are various approaches that try to predict the *fault-proneness* of components or classes based on several metrics. Hence, an approach that helps to distribute the optimal effort calculated using the cost model over the components would be helpful.

1.3 Contribution

The dissertation mainly focuses on one of the quality attributes: *reliability*. There are several reasons for that. The main reason is that it simplifies the model building to concentrate on one aspect rather than trying to solve "everything", i.e., all aspects of quality, in one model. Reliability is one of the major factors that affects users and customers and is therefore of major importance. Nevertheless, this does not mean that other quality aspects are totally ignored as a complete separation is not possible. For example, the effort for corrective maintenance also depends on the number of defects. Moreover, maintenance in general is more important in the long run as it constitutes most of the total product costs [30]. So, it is important to be able to extend the approach to explicitly incorporate other quality factors. However, note that the proposed approach currently does not deal with adaptive and perfective maintenance, i.e., enhancements of the system apart from defect removal.

The main contribution of this thesis is an analytical model of the economics of analytical SQA. It consists of (1) a theoretical model that contains many detailed factors and can be used for theoretical analyses, and (2) a simplified practical model that is aimed at usage in software development projects. This practical model has been derived from the theoretical one, has a reduced set of input factors and offers means for optimisation. The main change is that we consider defect types instead of single faults because measurement for the latter is only possible in controlled experiments which are not feasible in real project environments. Furthermore, the methodological usage of this model is explained and it is exemplarily embedded in the V-Modell XT.

Furthermore, we propose an approach to identify defect-prone components early in the development process based on detailed UML models. This approach adapts existing and proven metrics to a specific set of model types and thereby forms a metrics suite that can yield an order of fault-proneness of components. A combination with operational profiles can even identify the failure-prone components, i.e., the ones that are most likely to fail in the field.

Finally, the thesis contains several case studies that (partly) validate the proposed approaches. A full validation of the model in practice is out of scope of this dissertation because the model needs calibration over several projects. However, the studies contribute to the body of knowledge on the effectiveness, efficiency, and costs of defect-detection techniques.

1.4 Related Work

The available related work can generally be classified into three categories: (1) theoretical models of the effectiveness and efficiency of either test techniques or inspections [63, 112, 136, 219], (2) economics-oriented, abstract models for quality assurance in general [21, 66, 193], and (3) approaches to identify defect-prone components [36, 119]. Models of the first type are able to incorporate interesting technical details but are typically restricted to a specific type of techniques and often economical considerations are not taken into account. The second type of models typically stems from more management-oriented researchers that consider economic constraints and are able to analyse different types of defect-detection but often deal with the technical details in a very abstract way. The third category of related work contains approaches that use code metrics to identify fault-prone components. One problem with code metrics is that they are only available late in the development process, i.e., when the code has already been created. Model metrics can be measured earlier but have not been used so far for the identification of fault-prone components. Furthermore, the combination with operational profiles is unique in our approach.

1.5 Contents

The remainder of this thesis is organised as follows: In Chap. 2 we review the basics that are important for the following chapters. In particular, we describe software quality and different views and standards for it. We give definitions for the often confused terms for defects and give a classification and high-level description of the available

techniques for SQA. Finally, quality models are introduced and two examples are described in more detail.

Chap. 3 explains the state of the art in the areas that constitute our main contributions. We describe the available work on analytical models for defect-detection techniques, defect introduction and removal models, and models for software quality economics. The available literature surveys and the state of the art in the identification of fault-prone components are also explained.

The main contribution is presented in Chap. 4. First, we describe the basic factors that are important for the economics of quality assurance and combine them in an analytical model. Based on the empirical knowledge on those factors we perform a sensitivity analysis to investigate the importance of the factors. Then we derive a simpler, more practical model, that is again analysed and it is shown how this model can be used to optimise the usage of SQA in a company.

The approach to identify defect-prone components using detailed design models is proposed in Chap. 5. We describe the chosen and adjusted metrics for UML models and show how the metrics suite can be used to predict fault-prone and – in combination with operational profiles – failure-prone components, i.e., components which are likely to fail during operation.

Chap. 6 contains the accompanying case studies that were used to (partly) validate the proposed approaches and in particular analyse specific factors used to calibrate the analytical model. We describe case studies of an automatic collision notification system in automobiles, of a network controller for an automotive infotainment network, of backend systems for mobile services, and of telecommunication systems.

We close with final conclusions and an outlook on further research in Chap. 7.

Previously Published Material

The material covered in this thesis is based, in part, on our contributions in [100, 168, 201–212, 212–214].

2 Software Quality Assurance

This chapter introduces the basic terms and approaches that we consider as the basis for the remainder of this dissertation. We start in Sec. 2.1 by giving and discussing a definition of software quality. Sec. 2.2 has definitions for the different terms and notions related to defects. In Sec. 2.3 we describe and classify common techniques for software quality assurance. A classification with two important examples of quality models is given in Sec. 2.4. All this is considered as general knowledge in the field of software quality assurance although there are still disputable parts. In the following Chap. 3 we will discuss the state of the art that is of specific relevance for the contributions of this dissertation.

2.1 Software Quality

Because of the multitude of views on software quality, we first discuss these views and definitions. The quality attribute *reliability* is also defined and discussed in more detail because the contributions of this dissertation focus on dealing in particular with this attribute.

2.1.1 Views and Standards

Quality is a difficult concept in itself. The roots of describing and defining quality can be found in philosophy, especially in the view of Plato on beauty [160]. The question is how to define and judge if something is "better" than something else. This is often expressed in saying: "I know quality when I see it". An interesting discussion on those philosophical aspects can be found in in a book by Pirsig [159].

However, for a practical use of the notion of quality in engineering, this is not sufficient. We need measurable and comparable quantities. Garvin gives in [67] a comparison of different approaches to define quality. He calls the philosophical approach described above *transcendent*. Apart from that he identifies four further approaches:

- Product-based approach
- User-based approach
- Manufacturing approach
- Value-based approach

All these approaches have their own definitions of quality. In the *product-based* approach, quality describes differences in the quantity of some desired attributes. Hence, in contrast to the transcendent approach, this is precisely measurable. This assumes that we exactly know and are able to describe what is desired. For example, the quality of tires can be measured with the time they can be used. However, this approach is

difficult for software because metrics for software are still not able to give satisfying quantifications. It is not clear how desired attributes of software, such as maintainability, could be measured. Note that the easy to measure and often used quantities *number of found defects* or *defects per kLOC* are not a *desired* attribute. Such metrics are closer to the manufacturing view that is discussed later where conformance to specification is an issue. In the product-based approach only "ideal" attributes are considered.

Closer to the transcendent view is the *user-based* approach. It assumes that the product that satisfies the needs of the user best has the highest quality. This is related to the product-based approach as the desired attributes must be defined by users. The difference is that the emphasis is not on metrics but on the subjective impression of the users. However, Garvin argues that user satisfaction and quality are not necessarily the same. The reason is that users can be satisfied with a product they do not consider of high quality. For example, a product can be not as well produced as high quality products but is cheap enough to leave the customer satisfied.

The *manufacturing* approach takes a more internal view and defines quality as conformance with specified requirements. This definition, however, includes all the problems we have with developing exhaustive and useful specifications. It assumes that it is always possible to define the requirements of a product completely and hence a deviation of the specification can be easily recognised. The metric *defects per kLOC* from above is then useful as we measure the deviation from the specification by the defect density of the code.

The last approach is the *value-based* approach. It assigns costs to conformance and nonconformance and hence can calculate the value of a product. It assumes that we are able to assign a value to all involved factors. This actually blends two concepts as Garvin points out: "Quality, which is a measure of excellence, is being equated with value, which is a measure of worth." Nevertheless, we can relate these two concepts. We discuss this issue of quality costs in Sec. 2.4.3.

In the software community, the manufacturing approach as well as the user-based approach are accepted and acceptable. The reason is that in most cases a defect is easy to define as a deviation from the specification but this is not always clear. Hence, also deviation from customer or user expectations must be taken into account. The IEEE defines in [88] *quality* as follows:

Definition 1 (Quality) *(1) The degree to which a system, component, or process meets specified requirements. (2) The degree to which a system, component, or process meets customer or user needs or expectations.*

This also reflects the difficulty of specifying complete requirements especially for software which often has quickly changing requirements. Especially when considering the comparison of different quality assurance approaches defining quality only by conformance to requirements is problematic. For example, requirements reviews detect defects in requirements but cannot be handled with this limited definition of quality. Hence, a hybrid definition of software quality works best for our overall goal of analysing and optimising SQA.

The software quality can be expressed by *quality attributes* which are – following [88] – "a feature or characteristic that affects an item's quality". *Quality factor* can be used synonymously. There is no generally accepted set and description of quality attributes but a well-known set is defined in the standard ISO 9126:

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
- Portability

The ordering of quality attributes in a tree structure is proposed in [22] which allows a iterative refinement with the aim to get measurable quantities at the lowest level. For example, maintainability can be split into understandability and testability. This approach is criticised and improved for maintainability in a multi-dimensional model in [30]. Another important classification is defined by Laprie [116] that concentrates more on the related view of *dependability* which is essential the same as quality in our discussions above. However, the basic entities in all these classifications are similar to a large extent and the differences lie mainly in the attribution to different categories.

2.1.2 Reliability

The quality attribute *reliability* is arguably, besides usability, the most important attribute for the user of a software. Moreover, most quality assurance techniques mainly aim at improving the reliability. Hence, this work has its main focus on this attribute. As a foundation for the later chapters, we give a definition of software reliability and discuss the specific issues of the reliability of software. A common definition also adopted by the IEEE [88] is the following:

Definition 2 (Reliability) *Reliability is the probability of failure-free functioning of a system or component under stated conditions and environment for a specified period of time.*

This definition implies that reliability is based on a notion of time and that we look at reliability always at specific conditions and in a specific and unchanged environment. This is important because the reliability of a software can be extremely different in different environments. The same observation holds for hardware reliability. In extremely wet environments hardware will wear off earlier than in normal room conditions.

However, there are important differences between software reliability and hardware reliability. The reason is that software is intangible. Hence, software does not wear out but the main sources for failures are "design" faults. Those design faults can successively be removed and therefore, the reliability of software grows generally over time whereas hardware reliability decreases because of the wear-out. Design faults obviously exist in hardware as well but are typically not considered as a main source of failure.

For software, also no faults are introduced by the manufacturing process as there is no classic manufacturing. In general, it is seen as the process of transforming raw materials into a final product. This process is for software nonexistent apart from copying

the compiled code. Manufacturing faults are hence not an important issue for software because they stem from physical properties of the raw materials that introduce random differences in the product. Note that the compilation of source code in particular is not part of the manufacturing process. It only creates the first instance of the product but does not manufacture all the products.

The definition of reliability already implies the randomness of the failure process. Hence, mainly stochastic models are used to estimate and predict the reliability of software. We will describe those models in more detail in Sec. 2.4.2.

2.2 Defects

This section first gives definitions of defect and related terms and then introduces the classification of defects in defect types.

2.2.1 Definitions

There are various definitions of the terms related to faults and failures. Although there are public standards that define these terms, e.g., [88], there are still no commonly accepted definitions. Hence, we present our understanding of the most important terms in the following.

Definition 3 (Failure) *A failure is an incorrect output of a software visible to the user.*

An example would be a computed output of 12 when the correct output is 10. The notion of a *failure* is the most intuitive term and it is easy to define to a certain extent. During the run of a software something goes wrong and the user of the software notices that. Hence, output is anything that can be visible to the user, e.g., command-line, GUI, or actuators in an embedded system. Incorrect output can range from a wrong colour on the user interface over a wrong calculation to a total system crash. No output where output is expected is also considered to be incorrect.

The problem with defining a failure comes in when looking in more detail on the term *incorrect*. It is difficult to define this formally. Most of the time it is sufficient to define that a result is incorrect if it does not conform to the requirements as in the manufacturing approach to quality from Sec. 2.1. However, sometimes the user did not specify suitably or this part of the software is underspecified. For our purposes, we again take the hybrid approach of requirements conformance and user expectation. For most cases it is sufficient to understand failures as deviations from the requirements specification. However, for some cases – requirements defects – it is necessary to include the user expectation.

Definition 4 (Fault) *A fault is the cause of a potential failure inside the code or other artefacts.*

Having the definition of a failure, the definition of a fault seems to be simple. However, the concept of a fault is actually very difficult. It is often not possible to give a concrete location of the source of a failure. When the fault is an omission it is something non-existing and it is not always clear where the fix should be introduced.

Another problem constitute interface faults, i.e., the interfaces of two components do not interact correctly. It is possible to change either of the components to prevent the failure in the future. Frankl et al. discuss this in [58]. They state that the term *fault* “is not precise, and is difficult to make precise”. They define a fault using the notion of *failure region*. It consists of failure points – input values that cause a failure – that all cause no failure after a particular defect removal. This allows a formal and correct definition of a fault but does not allow a relation to programmer mistakes and any classification is very difficult. Therefore, we will use our own more ambiguous definition given above. Note that we consider wrong or missing parts in requirements or design specifications also as faults, i.e., this term is not restricted to code.

Definition 5 (Defect) *Defects are the superset of faults and failures.*

The notion of a *defect* is often very helpful if it is not important whether we are currently considering a fault or failure. This is because there is always some kind of relationship between the two and at a certain abstraction layer it is useful to have a common term for both.

Definition 6 (Mistake) *A mistake is a human action that produces a fault.*

The prime example is an incorrect action on the part of a developer including also omissions. This is also sometimes called *error*. While discussing the notion of *fault*, we already saw that it might be interesting to have the relation to the actions of the developers – the mistakes they make. This is important for classification purposes as well as for helping to prevent those kinds of faults in the future. Hence, we have extended the 2-layer model of faults and failures to a 3-layer model where mistakes cause faults and faults cause failures.

Definition 7 (Error) *An error is that part of the system state that may cause a subsequent failure.*

This definition of error is taken from [5]. It extends the 3-layer model with mistakes, faults, and failures to a 4-layered approach with errors between the faults and failures. Fig. 2.1 gives an overview of the terms and the different layers. Intuitively, when running the program, a fault in the code is executed. Then the software does something not expected by the programmer and reaches some erroneous state. However, it has not yet produced a failure. At this stage so-called *fault-tolerance* mechanisms can take counter-measures to avoid the failure. Also the error might not lead to a failure because it has no consequence on the user-visible behaviour, or it is masked by another error. In other cases, however, the erroneous state becomes visible to the user and hence results in a failure. More formal definitions of these terms can be found in [28]. However, these informal, intuitive definitions are sufficient for the following.

2.2.2 Defect Types

There is already a variety of research on the differences of defects and their nature. We can roughly divide them in three categories: (1) defect taxonomies, (2) root cause analysis, and (3) defect classification. Defect taxonomies are categorisations of faults,

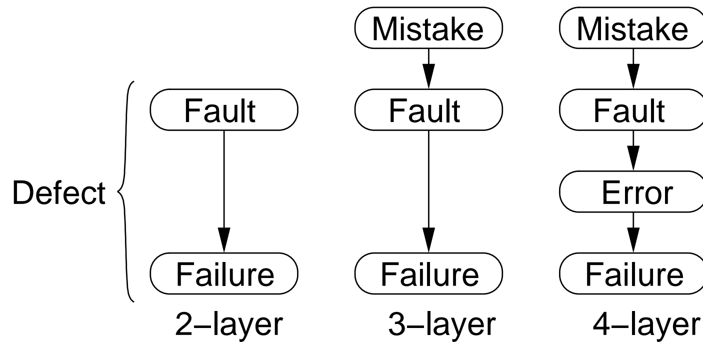


Figure 2.1: Overview of the terms related to defects

mostly in code, that are based on the details of the implementation solution, e.g., wrong type declaration, wrong variable scope, or wrong interrupt handling. A well-known example is the taxonomy of Beizer [9] although it goes beyond a simple taxonomy. An even more detailed approach is root cause analysis where not only the faults themselves are analysed but also their cause, i.e., the mistakes made by the development team. The goal is to identify these root causes and eliminate them to prevent faults in the future. Experience with this approach at IBM is documented in [128]. In general, root cause analysis is perceived as rather elaborate and the cost/benefit relation is not clear. Therefore, defect classifications aim at reducing the costs but sustain the benefits at the same time. The categorisation uses more coarse-grained *defect types* that typically have multiple dimensions.

An IEEE standard [87] defines several dimensions of defects that should be collected. This starts from the process activity and phase the defect was detected, over the suspected cause, to the so-called type that is similar to a taxonomy. Interestingly, also the source in terms of the document or artefact is proposed as a dimension of the classification. However, applications of this standard classification are not frequently reported.

The mainly used defect classification approaches have been proposed by companies: IBM and HP. The IBM approach is called *Orthogonal Defect Classification (ODC)* [40]. A defect is classified across the dimensions

- defect type,
- source,
- impact,
- trigger,
- phase found, and
- severity.

The defect type is here one of eight possibilities that allow an easy and quick classification of defects and are sufficient for analysing trends in the defect detection. Triggers are the defect-detection techniques that detect the defects and hence it is possible to

establish a relationship between defect types and triggers. Kan [101] criticises that the association between defect type and project phases is still an open question and that the distribution of defect types depends also on the processes and maturity of the company.

Similar to ODC is the HP approach called *Defect Origins, Types, and Modes* [70]. The name already gives the three dimensions a defect is classified in. The origin is the source of the defect – as in the IEEE standard –, the types are also a coarse-grained categorisation of what is wrong, and the mode can be one of *missing*, *unclear*, or *wrong*. Again the type of artefact – the origin – is documented as opposed to the activity. In contrast to ODC we can analyse the relationships between defects and document types but the defect-detection techniques – the triggers in ODC – are not directly documented.

However, it has been found in different case studies [49, 64, 212] that general defect type classifications are difficult to use in practice and need to be refined or adapted to the specific domain and project environment. In [62, 64] an approach is proposed for defining such an adaption of the defect type classification.

2.3 Techniques

To be able to investigate how we can cost-optimise the usage of analytical quality assurance, we have to know what (analytical) quality assurance is and what methods and techniques are available. We introduce the terms in general and describe the main techniques test, inspection, and automated static analysis in more detail.

2.3.1 General

Quality assurance (QA) is defined by the IEEE in [88] as "(1) A planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements. (2) A set of activities designed to evaluate the process by which products are developed or manufactured." Hence, quality assurance can be focused on the product or the process. This dissertation concentrates on QA in the first sense. It can be further classified in *constructive* and *analytical* quality assurance.

Constructive Quality Assurance. Constructive QA is the improvement of software quality by improving the process of software development. Hence, we do not want to detect and remove defects but prevent the introduction of defects in the first place. The emphasis is often on the activities that take place before the actual code development such as domain analysis, architecture design, or detailed design. However, also specific coding techniques or standards belong to constructive quality assurance. Hence, in some sense all software engineering activities apart from defect detection and removal can be seen as constructive QA. There are models available that allow to measure the quality of the process of a company to some extent. Hence, they are QA in the second sense of the above definition. The main examples of those models are the CMMI [42] and SPICE [90]. Unfortunately, a high ranking in one of the models

does not necessarily imply good quality or low defect rates. The models can only give guidelines for process improvement.

Analytical Quality Assurance. In contrast to constructive QA that aims to prevent defects, analytical QA detects existing defects. For this, techniques such as reviews and tests are in use. Although constructive QA can prevent many defects, humans always will make mistakes and hence we have to detect and remove the corresponding faults using defect-detection techniques. There are two main categories of analytical QA: (1) static and (2) dynamic analysis. Static analysis contains all techniques that do not need to run the software but operate on the code and/or other documents and detect defects in them. On the contrary, dynamic analysis runs the compiled software with the aim to observe failures. In that case the failures need to be traced back to its corresponding faults what needs more effort than with static techniques that detect the fault directly. However, it is not always clear if the fault would have resulted in a failure and there is a strong mixture with maintenance-related improvements. Analytical QA is also called *verification and validation* which is defined by the IEEE [88] as follows:

Definition 8 (Verification and Validation (V&V)) *Verification and validation is the process of determining whether the requirements for a system or component are complete and correct, the products of each development phase fulfil the requirements or conditions imposed by the previous phase, and the final system or component complies with specified requirements.*

Hence, the three important areas in analytical QA are (1) the analysis of the requirements, (2) the analysis of products between phases, and (3) the analysis of the final system. We need the requirements to be able to judge if the final system conforms to them (see also the definition of failure in Sec. 2.2). Nevertheless, it is also important to analyse and assure the quality during the whole process.

2.3.2 Test

There are various possibilities to classify different test techniques. We base our classification on standard books about testing [9, 144] and the classification in [98]. One can identify at least two dimensions to structure the techniques: (1) The granularity of the test object and (2) the test case derivation technique. Fig. 2.2 shows these two dimensions and contains some concrete examples and how they can be placed according to these dimensions.

The types of test case derivation can be divided on the top level into (1) functional and (2) structural test techniques. The first only uses the specification to design tests, whereas the latter relies on the source code and the specification. In functional testing generally techniques such as equivalence partitioning and boundary value analysis are used. Structural testing is often divided into *control-flow* and *data-flow* techniques. For the control-flow coverage metrics such as statement coverage or condition coverage are in use. The data-flow metrics measure the number and types of uses of variables.

On the granularity dimension we often see the phases *unit*, *module* or *component test*, *integration test*, and *system test*. In unit tests only basic components of the system are tested using stubs to simulate the environment. During integration tests the

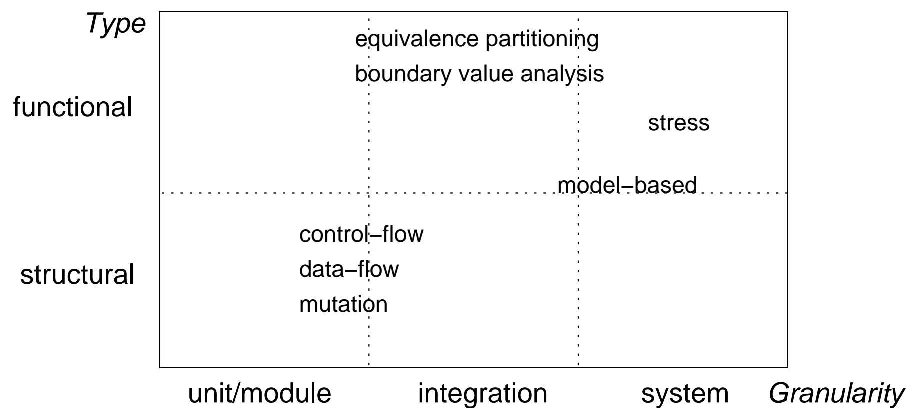


Figure 2.2: The two basic dimensions of test techniques

components are combined and their interaction is analysed. Finally, in system testing the whole system is tested, often with some similarity to the later operational profile. This also corresponds to the development phases. Hence, the granularity dimension can also be seen as *phase* dimension.

Finally, there are some special types of testing either with a special purpose or with the aim to simplify or automate certain parts of the test process. Model-based testing, for example, uses explicit behaviour models as basis for test case derivation, possibly with an automatic generation. Stress tests check the behaviour of the system under heavy load conditions. Random tests are generated without the specific guidance of a test case specification but use a stochastic approach to cover the input range. The opposite to random tests can be called *directed* tests as the test cases are directed at revealing failures. Various other types of testing can be found in the literature [9, 15, 144].

Model-Based Testing

A special approach to testing that appeals research as well as practice is *model-based testing*. We give a short introduction to this technique because we will later in this dissertation analyse it in more detail in a case study (Sec. 6.2). Some people argue that all kinds of testing are model-based [15]. Even if we do not have explicit test models, the tester has a "mental" model of the system and its environment. However, model-based testing more often denotes tests based on an explicit behaviour model of the software. Still, there is a large range of model-based testing and test tools [31].

From these explicit – and preferably executable – behaviour models tests are derived. This can be achieved with various techniques. In particular, there are automatic and manual approaches. If model-based testing is to be successful, the model must be easier to comprehend and validate than the code implementation itself. To achieve this, besides graphical notations, the model must be more abstract than the system. In general, for abstraction we can use two different ways [165]: (1) by means of encapsulation: macro-expansion mechanisms introduce (automatically) additional information, or (2) by deliberately omitting details and losing information such as timing

behaviour. If the model was not more abstract than the system, especially in the second sense, then there would be no difference in the effort of validating the model and the effort of validating the system itself.

For the automatic derivation process, test case specifications are used to guide the generator. They can either be structural or functional. Structural test case specifications use coverage metrics on the model similar to what traditional structural tests use on code [68]. Hence, a coverage of states or transitions is required. Structural test case specifications identify "interesting" cases in the large state space of the system [166].

In general, it can be noted that there are not many evaluations of the effectiveness and efficiency of model-based testing. In particular, the costs and benefits have only been evaluated to a small extent [167].

2.3.3 Inspection

We use the term *inspection* in a broad sense including all reading techniques for artefacts in a development process with the aim to find defects and with a defined process. The pioneer in the field of software inspections was Michael Fagan. An overview of his work can be found in [52]. He proposed the first approach of a "formal" inspection process in [51]. Since then many variations on the process and the reading techniques have been proposed. We follow mainly [115] in the following and use his taxonomy of software inspections. There are four dimensions in this taxonomy.

- Technical dimension: methodological variations
- Economic dimension: economic effects on the project and vice versa
- Organisational dimension: effect on the organisation and vice versa
- Tool dimension: support with tools

The dimension we mainly look at in the following is the technical dimension. An overview is depicted in Fig. 2.3. The process of an inspection typically consists of the phases *planning*, *overview*, *detection*, *collection*, *correction*, and *follow up*. In the planning phase, a particular inspection is organised when artefacts pass the entry criteria to the inspection. We need to select the participants, assign them to roles and schedule the meetings. The overview phase – sometimes called *kickoff meeting* – is not compulsory and the literature differs in this respect. This phase is intended to explain the artefact and its relationships to the inspectors. This can be helpful for very complex artefacts or for early documents such as requirements descriptions.

The next phase *defect detection* is the main part of the inspection. This is often divided into *preparation* and a *meeting*. How the details of this phase look like differs in the literature. Some authors suggest that already the preparation part should be used for defect detection while others propose to only try to understand the artefact and do the actual defect detection in the meeting part.

We use the term *inspection* here in a broad sense for all kinds of document reading with the aim of defect-detection. We can then identify differences mainly in the technical dimension, e.g., in the process of the inspections, for example whether explicit preparation is required. Other differences lie in the used reading techniques, e.g. checklists, in the required roles, or in the products that are inspected.

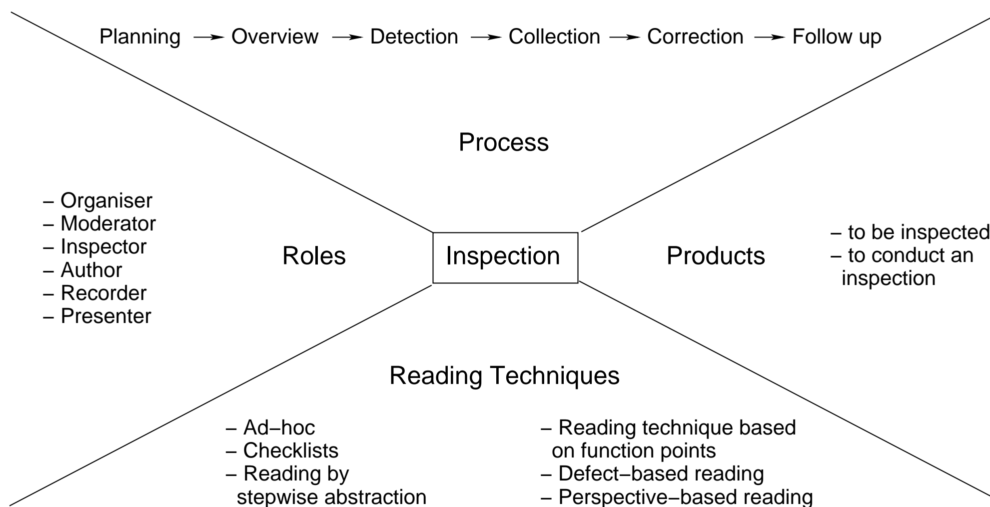


Figure 2.3: The technical dimension of software inspections (source: [115])

A prominent example is the formal or Fagan inspection that has a well-defined process with a separate preparation and meeting and defined roles. Another often used technique is the walkthrough. In this technique the moderator guides through the code but no preparation is required.

2.3.4 Static Analysis Tools

Static analysis tools are a class of programs that aim to find defects in code by static analysis similarly to a compiler, e.g. [6, 57, 77]. The results of using such a tool are, however, not always real defects but can be seen as a warning that a piece of code is critical in some way. Hence, the analysis with respect to true and false positives is essential in the usage of static analysis tools. There are various techniques to identify such critical code pieces. The most common one is to define typical bug patterns that are derived from experience and published common pitfalls in a certain programming language. Furthermore, coding guidelines and standards can be checked to allow for a better readability. Also, more sophisticated analysis techniques based on the dataflow and controlflow are used. Finally, additional annotations in the code are introduced by some tools [57] to allow an extended static checking and a combination with model checking.

2.4 Quality Models

In this section we give an overview of models used to describe and evaluate quality for software. First, we classify the different types of available models and then describe two specific types of quality models – software reliability models and software quality economics – in more detail.

2.4.1 Classification

In Sec. 2.1 we discussed the meanings of quality and gave a definition for software quality. During software development it is important to analyse the quality to improve the product and the process and to support the project management, e.g. the release management. Hence, we need quantitative measures of software quality and models to interpret the results.

Definition 9 (Quality Model) *A* quality model *or* quality-evaluation model *is an abstraction of the relationships of attributes of the artefacts, process, and people and one or more quality attributes of the product.*

Purpose. Following [200], those models have either the goal to (1) assess the current quality, (2) accurately predict the future quality, or (3) identify problem areas in the artefacts or process. We use a similar approach to categorise the models. We start with the two types of quality assurance: constructive and analytical. Hence, we also have constructive and analytical quality models. *Constructive quality models* are used to explain the relationship between constructive actions during development and one or more quality attributes. For example, adding watchdogs to the architecture increases the availability of the system. *Analytical quality models* can be further broken down to *assessing* and *predictive* quality models. The assessing models corresponds to the goals 1 and 3 from above. They have the aim to estimate the current value of one or more quality attributes and thereby might identify problem areas. The predictive quality models make predictions of the future development of quality attributes based on current and historical information. Hence, we can classify quality models in three types:

- Constructive quality models
- Assessing quality models
- Predictive quality models

Quality View. A main issue are the different views on software quality discussed in Sec. 2.1. Different quality models support different views better or worse. For example, usability models are inherently user-based quality models. The main aim is to build a system which can be easily used. Economic models of software quality such as the one proposed in Chap. 4 support obviously more of a value-based view on quality. Examples can be found for all of the discussed views.

Specificness. Tian [200] adds to this the further dimension of abstraction or specificness. This is important as for different purposes different levels of detail are necessary. We first distinguish between two types of models with a large difference in terms of abstraction. *Generalised quality models* describe product quality in terms of industrial average or general trends. These do not require product- or process-specific data to be applicable but obviously can give only coarse-grained results. Well-known models of this type can be found in the work of Jones [94]. The second type are *product-specific quality models* that use actual data from a specific product, project, or company. An overview is depicted in Fig. 2.4.

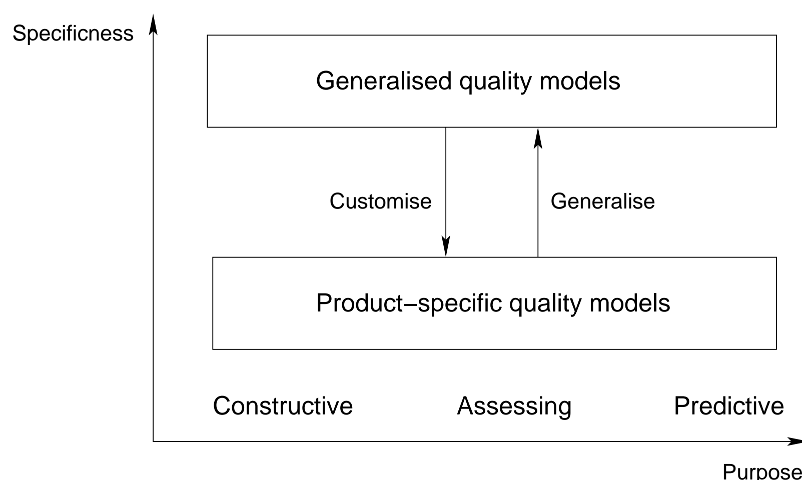


Figure 2.4: Classification of quality models (adapted from [200])

Generalised Models. Generalised Models are quality models that are not specific to a certain product or project. The most general quality models are overall models that give a single estimate of the overall quality in industry. An example is to reduce quality to the number of defects per line of code in the defect-density model and give an overall average value of this metric. An overall model can be structured into a segmented model that has different estimates for different industrial segments. For example, we can give typical failure rates for software products in different domains. Finally, we can introduce the additional dimension time into the analysis, i.e., the progression over time of those quality metrics is considered. This allows to analyse trends in the development of quality and larger environmental influences. Those models are called dynamic models.

Product-Specific Models. Firstly, we have history-based models that use historical data, typically from older releases or similar projects, to predict quality for the current project. It is also possible to customise generalised models for a specific project. An example is the distribution of found defects over the development phases. Observation-based models combine current observations of the software and the process activities to estimate the quality of the software. An example are software reliability growth models that relate usage time and time between failures to evaluate the quality aspect reliability. Thirdly, measurement-based models do not rely on observations, i.e., the execution of the software but measure some static aspects. Such models define relationships between early measurements and product quality to be able to use this information in quality assurance and improvement. Code complexity metrics can be used to identify fault-prone modules, for example. This information allows to concentrate the quality assurance on these modules.

Further Specificness. Finally, quality models can have further dimensions of specificness. As discussed above, they can be general or product-specific. However, there are also other types of specificness that are important. Models are often *technique-specific*, i.e., analyse the effects of a specific development or defect-

detection technique. For example, there are several models that are dedicated to analysing inspections. Quality models can also be *phase-specific*, e.g., most reliability models concentrate only on system test and field use. Finally, many models are *attribute-specific* by concentrating on one quality attribute in particular. There are several examples for that such as maintainability or reliability models.

Measurement. Each of the quality models has different data requirements that have to be fulfilled to be able to use the model. The resulting effort to collect the data can differ significantly. A simple defect-density model only requires to measure the number of defects and the size of the software. On the other extreme, detailed software quality economics (cf. Sec. 2.4.3) often need data about the cost of field failures and the cost of fault removal. We can group the types of measurements into three groups [200]:

- Product measurement means to evaluate different attributes and characteristics of the software and related artefacts.
- Activity measurement is concerned with the effort, time, and resources needed for the activities in the development.
- Environmental measurement mainly measures characteristics of the process and the related people. It is often neglected but can give precious insights.

The two models we propose in this dissertation can be categorised then as product-specific models, the analytical model of quality economics is a history-based, predictive model whereas the metrics suite for defect-proneness is a measurement-based, predictive model.

2.4.2 Software Reliability Models

We focus on the quality attribute reliability in this dissertation. Reliability theory and models for estimating and predicting reliability have been developed for several decades. We explain some basics and summarise the merits and limitations in the following. The general idea behind most reliability models is that we want to predict the future failure behaviour – and thereby the reliability – of a software based on actual failure data. In the definition from Sec. 2.1.2 reliability is already defined as probability. That means that we use data from failures as sample data in a stochastic model to estimate the model parameters. There are other kinds of reliability models that use different mechanisms but they are not as broadly used.

The failure data that comprises the sample data can be one of two types: (1) time between failure (TBF) data or (2) grouped data. The first type contains each failure and the time that has passed since the last failure. The second type has the length of a test or operation interval and the number of failures that occurred in that interval. The latter type is not as accurate as the first but the data is easier to collect in real world projects.

Having used the sample data to estimate the model parameters we can use the model with these parameters to predict future behaviour. That means the model can calculate useful quantities of the software at a point in time in the future. Apart from reliability

the mostly used quantities are the expected number of failures up to a certain time t (often denoted by $\mu(t)$) and its derivative, the failure intensity (denoted by $\lambda(t)$). The latter can intuitively be seen as the average number of failures that occur in a time interval at that time t. Based on these quantities we can also predict, for example, for how long we have to continue testing to reach a certain failure intensity objective.

An excellent introduction to the topic is a book by Musa [140]. A wide variety of partly practical relevance is described in a handbook [123]. Finally, the most detailed description of these models and the theory behind them can be found in [141]

Execution Time and Calendar Time

Experience indicates that the best measure of time is the actual CPU execution time [141]. The reliability of software as well as hardware that is not executed does not change. Only when it is run there is a possibility of failure and only then there can be a change in reliability. As discussed in Sec. 2.1.2, the main cause of failure for hardware is considered the wear-out. Therefore, it is possible to relate the execution time to calendar time in some cases. For software this seems to be more difficult and hence execution time should be used.

However, CPU time may not be available, and it is possible to reformulate the measurements and reliability models in terms of other exposure metrics: clock time, in-service time (usually a sum of clock times due to many software applications running simultaneously on various single- or multiple-CPU systems), logical time (such as number of executed test cases, database queries, or telephone calls), or structural coverage (such as achieved statement or branch coverage). 100 months of in-service time may be associated with 50 months (clock time) of two systems or 1 month (clock time) of 100 systems. All of these approximations are also referred to as *usage time*.

In any case, some combination of statistical sampling with estimates of units sold is much better than using calendar time because of the so-called loading, or ramping, effect [96]. If this effect is not accounted for, most models assume a constant usage over time which is not reasonable under many practical circumstances. The difficulty is to handle differing amounts of installations and testing effort during the life cycle of the software.

Parameter Estimation

Parameter estimation is the most important part of applying a software reliability model to a real development project. The model itself is hopefully a faithful abstraction of the failure process but only the proper estimation of the model parameters can fit the model to the current problem. There are several ways to accomplish that but a consensus seems to be reached that a Maximum Likelihood approach on the observed failure data fits best to most models. Another possibility is to use other software metrics or failure data from old projects as basis for the estimation but this is only advisable when no failure data is available, i.e., before system testing.

Merits and Limitations

The usage of stochastic reliability models for software is a proven practice and has been under research for decades now. Those models have a sound mathematical basis

and are able to yield useful metrics that help in determining the current reliability level and in deciding the release problem.

However, the models are often difficult to use because (1) some understanding of the underlying statistics and (2) laborious, detailed metrics documentation and collection is necessary. Moreover, the available tool support is still not satisfying. The main deficiency is, however, that the models are only applicable during system test and field usage and even during system test they depend on usage-based testing, i.e., they assume that the tests follow the operational profile that mirrors the future usage in the field. The approach to use other metrics or failure data from old project is very fragile. Especially to use other software metrics have not lead to a satisfying predictive validity. This narrows the merit of those models.

2.4.3 Software Quality Costs

A unifying view on the different aspects of software quality is difficult. However, we can often express the various factors in monetary units, i.e. the costs and benefits (cf. Sec. 2.1). This section describes the area of quality economics, sometimes called quality costs. Quality cost models describe the different types of costs and their relationships.

Types of Costs. The costs of quality are an area that is under research in various domains. We understand them as the costs that are associated with preventing, finding, and correcting defective work. These costs are divided into *conformance* and *nonconformance* costs, also called *control costs* and *failure of control costs*. We can further break down the costs into the distinction between prevention, appraisal, and failure costs which gives the model the name *PAF* model [97]. The basic model was derived from the manufacturing industry but has been used repeatedly for software quality as well [106, 108, 193]. A graphical overview is given in Fig. 2.5.

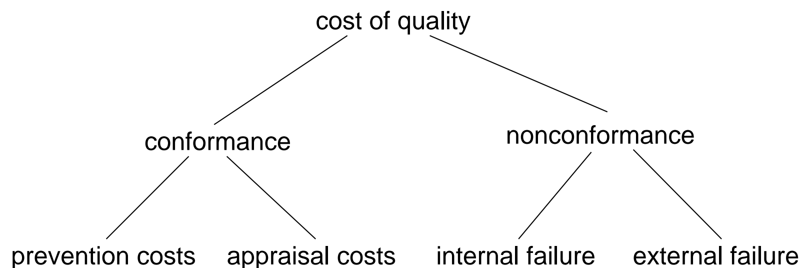


Figure 2.5: Cost types

Conformance Costs. The conformance costs comprise all costs that need to be spent to build the software in a way that it conforms to its quality requirements. This can be further broken down to *prevention* and *appraisal* costs. Prevention costs are for example developer training, tool costs, or quality audits, i.e. costs for means to prevent the injection of faults. The appraisal costs are caused by using various types of tests and reviews.

Nonconformance Costs. The *nonconformance* costs come into play when the software does not conform to the quality requirements. These costs are divided into *internal failure* costs and *external failure* costs. The former contains costs caused by failures that occurred during development, the latter describes costs that result from failures at the client.

Summary. The PAF model for the costs of quality is a widely accepted basis for software quality economics. It supports primarily the manufacturing-approach to quality, i.e., focuses on conformance and non-conformance to the specification or requirements. The problem with this model is that it stays rather abstract and it needs to be refined to be used in practice.

3 State of the Art

We describe the state of the art in the areas that we address in the following chapters. For each of the areas the most important results are summarised and its strength and weaknesses are identified. We first analyse the work on models for defect introduction and removal. Then the research on software quality economics is summarised. Following that, we describe more technical, analytical models of the effectiveness and efficiency of defect-detection techniques. Finally, we present the work on empirical surveys of quality assurance techniques and proposals related to the fault-proneness of components.

3.1 Defect Introduction and Removal

Boehm [19] already introduced a simple defect introduction/removal model in which in different phases different defect classes are introduced such as requirements, design, code, and documentation. Different QA techniques are used to eliminate those defects with different effectiveness based on costs. He also cites and summarises some studies but these details do not directly influence the cost estimation in the COCOMO model.

Jones [94] looks at *defect creation* and *defect discovery* with the emphasis on different activities in the development process. For example, during requirements analysis, design or coding defects are introduced and discovered during coding and testing. Hence, there is no concept of propagation in the sense that a defect in the requirements causes a defect in the code but this defect is assumed to be the same.

The most detailed and comprehensive model of defect introduction and removal was developed by Chulani and Boehm [41, 47]. It is part of COQUALMO which is an extension of COCOMO. It is similar to the model of Jones and fails to distinguish defect classes.

Kan [101] uses a model of *defect injection and removal* that is similar to the model of Jones. The focus is on a process step which is an activity in the development process. There are existing defects before the step, defects are injected during development and by incorrect repairs after defect detection. The undetected and newly introduced defects constitute the defects after the step. By using this generic notion of process step the differences between development and defect detection are not clear in this model. Furthermore, it does also not consider defects in different types of artefacts and the propagation of those.

Summary. The basic view of the available work is that there are activities in the development process that introduce defects in different phases and correspondingly activities that remove those defects. The defects after the application of a defect-detection technique are the defects that existed before without the ones removed. In summary, current defect introduction and removal models do not consider defects in different

artefacts and the relationships between them. Only the defect classes of Boehm [19] are an exception. However, this is important because different defect-detection techniques appraise different types of artefacts.

3.2 Software Quality Economics

Models for quality economics stem mainly from the manufacturing field but have been adapted to software as well. A comparison of quality cost models is done in [86]. Generally, those models are rather abstract and aim at structuring the different types of costs that are related to quality. Most of them are too abstract to make them operational directly.

Mandeville describes in [126] software quality costs, basically an adaption of the PAF model (cf. Sec. 2.4.3), a general methodology for cost collection, and how specific data from these costs can be used in communication with management.

In [106] the model of software quality costs is set into relation to the Capability Maturity Model (CMM) [155]. The emphasis is hence on the prevention costs and how the improvement in terms of CMM levels helps in preventing failures.

Humphrey presents in [83] his understanding of software quality economics. The defined cost metrics do not represent monetary values but only fractions of the total development time. Furthermore, the effort for testing is classified as failure cost instead of appraisal cost.

Raffo et al. [172] define a model considering the defect potential. In essence this is very similar to the PAF model of quality costs (cf. Sec. 2.4.3). The only difference can be seen in the differentiation between the cost of isolating a defect and the cost of fixing a defect. Furthermore, they give a simple equation that combines theses high-level factors but omit guidelines how to use it in practice.

Ntafos describes some considerations on the cost of software failure in [148]. The difficulties of collecting appropriate data are shown but the model itself is described only on a very abstract level. Guidelines for applying a quality cost model in a business environment in general are given in [103] but mainly from the accounting point of view.

Galin extends in [65, 66] the software quality costs model – the PAF model – with managerial aspects but the extensions are not relevant in the context of analytical SQA.

A view on the economics of software quality assurance with the focus on different types of errors is described in [3]. It distinguishes between design errors, logic errors, and syntax errors and shows that the design errors are responsible for most costs. However, no analytical model is given that would allow to use that information in planning quality assurance.

In [109, 193] a metric called *return on software quality (ROSQ)* is defined. It is intended to financially justify investments in quality improvement. It aims mainly on measuring the effects of process improvements, i. e. constructive quality assurance. The specifics of analytical quality assurance are not accounted for. Furthermore, the calculations are mainly based on the average defect content in the software and further characteristics of the defects are not considered.

Boehm et al. present in [21] the iDAVE model that is based on COCOMO II and COQUALMO. This model allows a thorough analysis of the ROI of dependability.

The main critic is again the granularity. For example, only an average cost saving per defect is considered.

Building on iDAVE, Huang and Boehm propose a value-based approach for determining how much quality assurance is enough in [80]. This model does not resolve the granularity problem as the defect levels used (and borrowed from COQUALMO) are only rough estimates and hence more thorough analyses are not possible. However, it contains an interesting component that deals with time to market costs that is currently missing from most models.

A similar model to COQUALMO in terms of the description of the defect introduction and removal process is described by Jalote and Vishal [91]. They consider different fault injection rates in different phases and optimise the effort distribution of quality assurance based on the effort needed for fault removal and the effectiveness of the used techniques. However, all faults are treated equally in terms of effort and effectiveness although there can be huge differences. Moreover, effort is not the only relevant cost factor.

Holzmann [76] looks in particular at the economics of formal verification. He tries to show that those verification techniques make economic sense because they are able to find the defects with low failure probability but catastrophic consequences. Intuitively, this seems to be correct although there are not enough studies about the nature of defects detected by formal verification to support this claim.

Jones [94] classifies the costs of software defect removal activities in (1) preparation costs, (2) execution costs, and (3) repair costs. However, it is not clear whether preparation costs are fixed and execution costs are variable with respect to the spent effort. He also identifies field failure costs such as field service, maintenance, warranty repairs, and in some cases liability damages and litigation expenses. These failure costs are not further elaborated.

Summary. The models in this category suffer mainly from the problem that they are too abstract. They model the costs of quality based on the well-known PAF model and extend it or build on it. However, most work is still not directly applicable and ignore technical influences. COQUALMO, iDAVE, and ROSQ are notably different in this respect. They allow to calculate concrete values for the cost of quality or the return on investment. However, they still do not make use of the detailed, technical factors that were identified in analytical models of quality assurance techniques (cf. Sec. 3.3). The input factors are mainly coarse-grained and hence the results are also coarse-grained. Moreover, most of the models of quality costs concentrate on process improvement and therefore constructive QA.

3.3 Analytical Models of Quality Assurance

Analytical models of analytical quality assurance contain a wide variety of models. We include all models that in some sense evaluate analytical quality assurance or specific techniques in this category. The range goes from simple defect counts up to cost-based equations. Often the terms *effectiveness* and *efficiency* are used in this context. Effectiveness describes defect detection using defect counts whereas efficiency also takes the effort for the defect detection into account. A summary of some basic efficiency

models and metrics can be found in [101, 201] which we will not explain in the following but concentrate on specific proposals that include costs in some way.

3.3.1 Inspection

Gilb and Graham [69] describe a simple efficiency model for inspections by dividing the number of detected defects divided by the cost consumed by the inspection. Hence, efficiency is measured in defects per unit of effort. This is a classical efficiency metric. The main advantage of this is that the data is typically directly available because it is part of standard inspection result forms. However, neither the influence of the nature of found defects nor of other techniques is included. For example, the effect on the reliability of the software is unclear [201, 202].

Kusumoto et al. describe in [111, 112] a metric for cost effectiveness mainly aimed at software reviews. They introduce the concept of virtual software test costs that denote the testing cost that would have been needed if no reviews were done. This implies that we always want a certain level of quality and it is not clear whether the field costs are included. However, this virtual test costs allow a better comparison of different applications of reviews.

An extension of the Kusumoto model is developed by Sabaliauskaite et al. in [182, 183]. They detail the metrics so that false positives and their effects on the costs can be considered. This extension is also validated and the effect of false positives is unique to this approach. Nevertheless, the Sabaliauskaite model inherited the problems of the Kusumoto model discussed above.

Freimut, Briand, and Vollei [63] also use the Kusumoto model as a basis. They identified the problem that this model assumes that one defect in a phase assumes that it results in exactly one in the next whereas defects can propagate in several defects in later phases, e.g., a design defect results in several defects in the code. Furthermore, they present an approach how to use expert opinion for the factors of the model where no measured data is available.

The economics of the inspection process are also investigated by Biffl et al. [10, 12]. This model operates on defect classes that are not precisely defined but are related to the severity and impact on later phases. The basic equation for the benefits of an inspection includes the number of defects, the average benefit per defect – the savings in later phases –, and the effectiveness per defect class. Effort is not directly part of the model but indirectly by the number of inspectors and the inspection duration. By this, this approach is very specific for inspections but allows to determine the optimal team size for an inspection.

3.3.2 Test

There are also several analytical models of software testing with different emphases. They are often used to compare the defect detection capabilities of different techniques and they do not always include cost considerations. Weyuker and Jeng [219] proposed an analytical model of partition testing to evaluate different strategies. It is based on the input domain and the distinction between *correct* and *failure-causing* inputs. Different strategies have different probabilities to select these differing inputs. They deliberately ignored operational profiles and assumed all input values of a program to be equally

likely to be selected in a test case. Although the model is quite simple it set the framework for further models. Chen and Yu [38] extend this model by analysing best and worst cases. Boland, Singh, and Cukic [23] built a similar model for comparing partition and random testing with different mathematical techniques – majorization and Schur functions – that allow them to refine the previous models. Morasca and Serra-Capizzano [136] consider hierarchies of the failure rates of different sub-domains of the input to compare different testing techniques.

The emphasis of the model proposed by Frankl and Weyuker [59,60] is more on test case selection criteria. Such criteria can be a data-flow criteria or structural coverage measure. They show that it is possible to define reasonable relations between different criteria based on their fault-detecting capability. Frankl et al. [58] moved the emphasis from detecting a failure to delivered reliability. In particular, they compared debug testing to operational testing, i.e., testing with the aim to quickly reveal failures to testing that simulates operational behaviour. They answer this using an expected value model that models the effect of testing as transforming failure-causing inputs into correct inputs. Reliability is then the probability of selecting a remaining failure-causing input during operation.

Pham describes in [156] various flavours of a software cost model for the purpose of deciding when to stop testing. It is a representative of models that are based on reliability growth models. Other examples include [73, 79, 124, 151]. The main problem with such models is that they are only able to analyse system testing and no other defect-detection techniques. In addition, the differences of different test techniques cannot be considered.

3.3.3 General

Cengarle [37] discusses some basic properties of test and inspection and their combination. It is stated that the combination of different techniques yields the best results but it is not based on a mathematical model. Littlewood et al. [120] propose an analytical, stochastic model of the diversity of defect-detection techniques and they can prove this statement based on this model. The basis are *difficulty* functions that intuitively describe the difficulty of a specific defect-detection technique to detect a specific defect. The model gives important insights but is hard to use in practice hence the authors suggest to group the defects in defect classes for simplification. Furthermore, a notion of time or effort for the technique application is missing in the model and it is assumed that a technique can be applied once or twice and so on.

Collofello and Woodfield [43] propose a general model for the *cost effectiveness* of an *error-detection process* as the costs saved by the process divided by the cost consumed by the process. The error-detection process denotes the process of defect-detection and fault removal and the operation of the software is the final process. It uses simple average values for the effectiveness of techniques and the cost, i.e. effort, for each error detection process that is used. The only differentiation between defects is done be classifying them into design and code defects. Costs apart from the labour costs for using the techniques and defect removal such as tool costs or additional support costs are not considered.

Summary. Most of the available analytical models are technique-specific, i.e., concentrate on inspections or testing. This allows deeps insights in those techniques but does not help in deciding how to optimally use different techniques. Some models already use different types of defects and defects in different types of artefacts to allow a more detailed analysis. In general, the most of the available models fail in incorporating costs apart from personnel effort and do not consider that different techniques detect different defects. The only approach that explicitly models the latter concept is from Littlewood et al. However, they do not elaborate their model to make it applicable in practice.

3.4 Literature Reviews

Juristo et al. summarise in [98] the main experiments regarding testing techniques of the last 25 years. Their main focus is to classify the techniques and experiments and compare the techniques but not to collect and compare actual figures.

Laitenberger published an extensive survey on inspection technologies in [115]. He presents a taxonomy of inspections and inspection techniques and structures the available work according to it. He also included data on effectiveness and effort but without relating it to a model or conducting further analyses.

Briand et al. [29] use several sources from the literature for inspection efficiency were used to build efficiency benchmarks. The intent is to analyse and document the current practice of inspections so that companies are able to compare their own practices with the average. For this they analysed several studies for effectiveness and effort, mainly of inspections but also testing and related it based on the inspection models described in Sec. 3.3.1.

Summary. There have been few approaches to summarise the knowledge on software quality assurance techniques. The available ones mainly concentrate on specific techniques and only Briand et al. related the empirical results to an analytical model which is aiming mainly on inspections.

3.5 Defect-Proneness

Cartwright and Sheppard [36] analysed a large industrial system that was developed with the Shlaer-Mellor method. The found correlation of some simple metrics of the Shlaer-Mellor method and the defect densities of the corresponding classes. For example, the use of inheritance had a strong influence. This shows that it is possible to predict the defect-proneness using models. However, this work does not consider the more widespread description techniques of UML and prediction is only done for faults. Hence, no analysis of the probability of failure of classes is possible.

There have been few other approaches that consider reliability metrics on the model level: In [215] an approach is proposed that includes a reliability model that is based only on the static software architecture. A complexity metric that is in principle applicable to models as well as to code is discussed in [35], but it also only involves static structure as well. In [18] the cyclomatic complexity is suggested for most aspects of a design metric but not further elaborated.

In et al. describe in [89] an automatic metrics counter for UML. They classify their metrics into various categories including fault proneness. The metrics in this category are WMC, NOC, and DIT. The latter two are the same as in our approach. The calculation of WMC is given as the sum of the complexities of the methods but no further explanation is given how this complexity should be calculated from the model. State machines are not analysed.

A white paper by Douglass [48] contains numerous proposals of model metrics for all types of UML models. Therefore this work has several metrics that are not relevant for the fault-proneness. Moreover, detailed explanations of the metrics is not available for all of them. The above mentioned DIT metric is similar to the *Class Inheritance Depth (CID)*, and NOC is comparable to *Number of Children (NC)*. The *Class Coupling (CC)* describes the relations between classes but it does not consider the interfaces but the associations. Finally, there is a complexity metric for state machines called *Douglass Cyclomatic Complexity (DCC)* that is based on the metric from McCabe. It handles nesting and and-states in a way that describes the understandability of the diagram. differently. Also triggers and guards are ignored. Douglass considers the aspect of the complexity in terms of comprehensibility whereas we want to capture the inherent complexity of the behaviour of the component. He gives rough guidelines for values that indicate "good" models but does not relate the metrics to fault-proneness.

Other approaches have been used for dependability analysis based on UML models, although these do not consider complexity metrics: In [25] an approach to automatic dependability analysis using UML is explained where automatic transformations are defined for the generation of models to capture systems dependability attributes such as reliability. The transformation concentrates on structural UML views and aims to capture only the information relevant for dependability. Critical parts can be selected to avoid explosion of the state space. A method is presented in [24] in which design tools based on UML are augmented with validation and analysis techniques that provide useful information in the early phases of system design. Automatic transformations are defined for the generation of models to capture system behavioural properties, dependability and performance. There is a method for quantitative dependability analysis of systems modelled using UML statechart diagrams in [84]. The UML models are transformed to stochastic reward nets, which allow performance-related measures using available tools, while dependability analysis requires explicit modelling of erroneous states and faulty behaviour.

Nagappan et al. report on a study of mining metrics to predict component failures in [145]. They concentrate on metrics other than model metrics. Nevertheless, the results are interesting as they show that there are always metrics that are good predictors of failures but the set of metrics is not uniform across projects. Hence, the metrics in a metrics suite for prediction is probably domain- or even project-specific.

Finally, there are approaches that aid in selecting the most suitable unit test techniques based on software metrics, mainly code metrics. Liggesmeyer proposes in [118, 119] such metrics and an approach for selection. However, this is only possible late in the process when code already is available. For an earlier planning of the QA process design and model metrics are more suitable.

Summary. Most of the available metrics suite concentrate on code metrics which are available only late in the development process when the code has already been developed. The approaches that exist for models either have slightly different aims, e.g., analysing dependability attributes or readability, or concentrate on the static structure. However, the complexity of the dynamic behaviour is also an important factor. Furthermore, all approaches analyse the fault-proneness. Yet, in many cases failure-proneness is more interesting because there might faults that never appear in the field. In summary, there are no metrics suite for UML models available that aim to identify fault- and failure-prone components and take the dynamic behaviour into account.

4 Quality Economics

> Quality is never an accident; it is always the result of high intention, sincere effort, intelligent direction and skillful execution; it represents the wise choice of many alternatives. *William A. Foster*

This chapter presents the main contribution of this dissertation: an analytical model of the quality economics of defect-detection techniques. For this, we revisit the general model of software quality costs and identify the main influencing factors of the economics. They are combined in (1) a theoretical and (2) a practical, analytical model. For these models, we analyse and consolidate the available empirical knowledge. Using this data, we perform sensitivity analysis to identify the most important factors and finally, give a detailed methodical approach to using the model in practice.

4.1 Basic Factors

We first need to identify and discuss the basic factors that influence the quality economics of defect-detection techniques. We revisit the PAF model of quality costs (cf. Sec. 2.4.3) and refine it. Then we derive the factors that will be used in the analytical model.

4.1.1 Software Quality Costs Revisited

In Sec. 2.4.3, we described the basics of quality costs and software quality costs in particular. In this section we revisit the PAF (Prevention, Appraisal, Failure) model of quality costs and adapt it to our specific needs as a first step to an analytical model of the economics of defect-detection techniques.

We discussed in Chap. 1 that we concentrate on analytical SQA because constructive QA has significantly different characteristics and influencing factors. Hence, a model that is able to handle all types of quality assurance would be extremely complex and difficult to use. Therefore, we can eliminate the *prevention costs* from the software quality costs as we do not look at preventing potential defects but at detecting existing ones. Nevertheless, one could argue that when considering requirements or design reviews there is prevention of defects in code. However, we consider all kinds of defects in all kinds of documents. Those defects sometimes propagate to the next phases and documents but sometimes they do not. The review process is then more an appraisal of the documents with the aim to detect and remove defects than an activity to prevent defects.

Having reduced the PAF model essentially to an AF (Appraisal, Failure) model, the remaining parts are now refined to be able to identify the relevant cost factors from a reliability point of view. Note that there are more types that could be included. For example, costs for adaptive or perfective maintenance are affected by techniques such

as inspections that can improve readability and changeability. However, they are out of scope of this dissertation because we concentrate on defect-detection and removal. The complete refined model is shown in Fig. 4.1. The appraisal costs are detailed to the *setup* and *execution* costs. The former constituting all initial costs for buying test tools, configuring the test environment, and so on. The latter means all the costs that are connected to actual test executions or review meetings, mainly personnel costs.

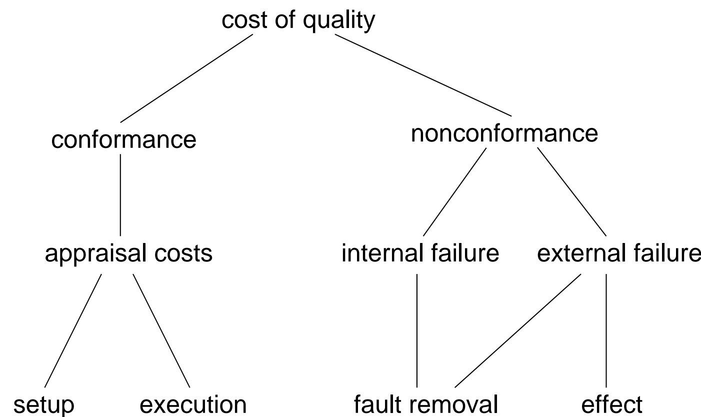


Figure 4.1: The refined cost types

On the nonconformance side, we have *fault removal* costs that can be attributed to the internal failure costs as well as the external failure costs. This is because if we found a fault and want to remove it, it would always result in costs no matter whether caused in an internal or external failure. Actually, there does not have to be a failure at all. Considering code inspections, faults are found and removed that have never caused a failure during testing. For example, the removal costs can be quite different regarding different techniques. When a test identifies a failure, there needs to be considerable effort spent to find the corresponding fault. During an inspection, faults are found directly. Fault removal costs also contain the costs for necessary re-testing and re-inspections.

External failures also cause *effect* costs. Those are all further costs associated with the failure apart from the removal costs. For example, compensation costs could be part of the effect costs, if the failure caused some kind of damage at the customer site. We might also include other costs such as loss of sales because of bad reputation in the effect costs.

4.1.2 Influencing Factors

In the following, we describe all major influencing factors that we identified based on the state of the art (cf. Chap. 3) and the refined software quality cost model from Sec. 4.1.1. We structure the factors starting with the simple, directly from the quality cost model derivable, factors to the more complicated, technical factors of the defect-detection techniques.

Cost Factors

First, we discuss the four cost factors that we get from the refinement of the PAF model. We also add a fifth, secondary factor *labour costs* that influences all of the four others.

Setup Costs. The costs to set up the application of a specific technique are one main part of the costs of analytical QA. The importance of this factor is directly derived from the quality cost model and it can be significant considering the effort needed to build a suitable test environment. The personnel effort is also dependent on the labour costs.

Execution Costs. In addition to the fixed setup costs, we have variable execution costs. This factor is also directly derived from the quality cost model. The execution costs are, as discussed above, mainly personnel costs and hence dependent on another factor, the labour costs.

Fault Removal Costs. When a defect is detected, the second step is to remove that defect. This has also different costs mainly consisting of personnel costs. Hence, we have an influence from the labour costs. Many other factors have an effect on this: inspections detect faults whereas tests only detect failures and leave the fault isolation as additional step. Hence, this has an influence on how laborious the removal is. Furthermore, it is dependent on the type of document and phase in which the defect is detected. A found defect during requirements analysis involves only to change the requirements document. Detecting the same defect during system test might require to change several documents, including the code and the design, and to re-inspect and re-test the software. When the defect is revealed in the field, we may have more costs for support staff. It is common software engineering knowledge that defect removal is the more expensive the later the defect is revealed. For example, in [20] it is stated that finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase.

Effect Costs. This is the most difficult to determine factor in the quality economics of defect-detection techniques. It contains all further costs that a defect in the field has apart from the removal costs. This may include compensation costs in case the user of the software had any damage because of the software defect. Other factors such as lost sales because of bad reputation belong to this factor as well but are not further considered in this dissertation. For example, in [190] issues of reputation of companies are discussed that help to measure those costs.

Labour Costs. The labour costs are a *secondary* factor because they do not influence the quality costs directly but have an effect on several of the other cost factors. Therefore, it is worthwhile to include it in the list and analyse it later on. Most of the costs incurring in analytical QA can be attributed to personnel effort. Hence, we can often measure the time effort spent for different activities and multiply them with the labour costs. Those need not only to contain the salary of the staff but all additional

costs caused such as sick days or training. This is often called a *loaded* labour rate. We assume this to be available as this is a standard economics approach.

Technical Factors

The following four factors are more technical in nature as they characterise the defects and techniques used for defect-detection.

Difficulty of Defect Detection. A more technical but nevertheless very important factor is the difficulty that a specific technique has in detecting specific defects. We have already noted that it is consolidated knowledge that a combination of different techniques can yield the best results [37, 120]. The reason for this lies in the fact that different, i.e., diverse, techniques detect different defects. Intuitively, this means that each technique has types or classes of defects that it is better suited to detect than others. Hence, we can say that for a specific technique it can be more or less *difficult* to detect a certain defect. Littlewood et al. have proposed a model containing this notion of difficulty in [120] that we will use in the following.

Failure Probability. Finally, it is not only important to detect defects but to detect the ones that would be most likely to cause a failure in the field. In [20] it is shown that about 20% of the faults cause about 80% of the downtime of software systems. We express this with the failure probability of a fault.

Document Types. A seldom considered factor is the document type the defect is contained in. We analyse all types of analytical SQA and therefore also requirements and design reviews that detect defects that could be called *requirements defects* and *design defects*. Those kind of defects cannot be detected with most of the available test techniques or code inspections.

Defect Propagation. Various models consider a concept of defect propagation over documents, phases, and technique usages. The model of pipes and drains is very popular in this context (cf. Sec. 3.1). Defects can be carried on from on document to the next and from one technique to the next. The most obvious case is when a specific technique fails to detect an existing defect in a document. More complicated is the constellation when one defect in a document causes other defects in other documents. For example, a design defect might cause several defects in code documents.

Marketing Factors

Finally, there are also two important factors that are not directly related to the mere act of developing the software. Most software is developed for some business reasons and hence needs to be sold on a market. The following two factors stem from this market. We will not elaborate on these factors in our economics model because we focus on the technical factors and leave their incorporation as future work.

Time to Market. So far we implicitly assumed that the time when we finish the quality assurance does not matter and has no effects on the costs as long as we can fix enough defects in-house for a positive balance of costs and revenues. However, in the markets, especially the software market, the time to market can have significant effects. For some products an early market introduction with more residual defects can be beneficial as the customers might prefer the first product on the market although the quality is not optimal. However, this aspect is not further considered in this dissertation as this is an aspect that depends more on economic and marketing decisions. Huang and Boehm [80] have used such a factor in their model.

Marketing-Driven Quality Requirements. Similar to the time to market, this influence is out of scope of this dissertation. Nevertheless, we want to discuss the factor. We assumed so far that the quality requirements are solely based on the cost/benefit relation expressed by the factors above. However, for reasons lying in the marketing of the product, it might be beneficial to have differing (higher or lower) requirements on specific quality attributes. For example, in some domains there are standards or even legal regulations that require specific safety or availability levels. Even so an economic analysis might suggest that testing is enough, those standards might introduce additional constraints.

4.2 An Analytical Model

We propose a general, analytical model of defect-detection techniques in the following. It is general with respect to the various types of techniques it is able to analyse. We principally analyse different types of testing which essentially detect failures and static analysis techniques that reveal faults in the code or other documents. It can be seen as a refinement and extension of the model by Collofello and Woodfield [43] that uses fewer input factors. We first describe the model and its assumptions in general, and then give equations for each component of the model for a single technique and for the combination of several techniques.

4.2.1 Basics

In this section, we concentrate on an ideal model of quality economics as we do not consider the practical use of the model but want to mirror the actual relationships as faithfully as possible. The model is stochastic meaning that it is based on expected values as basis for decision making. This approach is already common in other engineering fields [17] to compare different alternatives. Furthermore, a model that incorporates all important input factors for these differing techniques needs to use the universal unit of money, i.e., units such as euro or dollar. This is the only possibility to combine different cost parts such as personnel effort and tool costs. Hence, our model is cost-based.

Components

We divide the model into three main components which all are dependent on the spent effort t as a global parameter:

- Direct costs $d(t)$
- Future costs $f(t)$
- Revenues / saved costs $r(t)$

The direct costs are characterised by containing only costs that can be directly measured during the application of the technique. The future costs and revenues are both concerned with the (potential) costs in the field but can be distinguished because the future costs contain the costs that are really incurred whereas the revenues are comprised of saved costs. As discussed above, we will consider the expected values of those components – denoted by E in the following equations.

Assumptions

The main assumptions in the model are:

- Found faults are perfectly removed.
- The amount or duration of a technique can be freely varied.

The first assumption is often used in software reliability modelling to simplify the stochastic models. It states that each fault detected is instantly removed without introducing new faults. Although this is often not true in real defect removal, it is largely independent of the used defect-detection technique and the newly introduced faults can be handled like initial faults which introduces only a small blurring as long as the probability of introducing new faults is not too high.

The second assumption is needed because we have a notion of time effort in the model to express for how long and with how many people a technique is used. This notion of time can be freely varied although for real defect-detection techniques this might not always make sense, especially when considering inspections or static analysis tools where a certain basic effort or none at all has to be spent. Still, even for those techniques, the effort can be varied by changing the speed of reading, for example.

Difficulty

We adapt the general notion of the difficulty of an application of technique A to find a specific fault i from [120] denoted by $\theta_A(i)$ as a basic quantity for our model. In essence, it is the probability that A does not detect i. In the original definition this is independent of time or effort but describes a "single application". We extend this using the length of the technique application t_A. With length we do not mean calendar time but effort measured in staff-days, for example, that was spent for this technique application. Hence, we can define the refined difficulty function as follows:

Definition 10 (Difficulty Function) *The difficulty function $\theta_A(i, t_A)$ yields the probability that the technique A is not able to detect the fault i when applied with effort t_A.*

In the following equations we are often interested in the case when a fault is detected at least once by a technique which can be expressed as $1 - \theta_A(i, t_A)$. We also assume

that in the difficulty functions the concept of defect classes is handled. A defect class is a group of defects based on the document type it is contained in. Hence, we have for each defect also its document class c, e.g., requirements defects or code defects. This has an effect considering that some techniques cannot be applied to all types of documents, e.g., functional testing cannot reveal a defect in a design document directly. It may however detect its successor in code.

Defect Propagation

A further aspect to consider is that the defects occurring during development are not independent of each other. There are various dependencies between defects. However, most importantly there is dependency in terms of propagation. Defects from earlier phases in the development process propagate to later phases and over process steps. We do not consider the phases to be the important factor here but the document types. In every development process there are different types of documents, or artifacts, that are created. Usually, those are requirements documents, design documents, code, and test specifications. Then one defect in one of these documents can lead to none, one, or more defects in later derived documents. A schematic overview is given in Fig. 4.2.

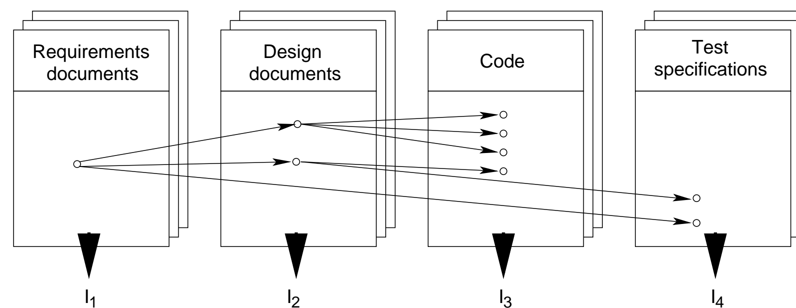


Figure 4.2: How defects propagate over documents

We see that a requirements defect can lead to several defects in design documents as well as test specifications. The design defects can again propagate to the code and to (glass-box) test specifications. For each document type c we have the set of defects I_c and hence the total set of defects I is $I = \bigcup I_c$. Furthermore, for each defect, we also look at its predecessor defects R_i. For the model this has the effect that a defect can only be found by a technique if neither the defect itself nor one of its predecessors was detected by an earlier used technique.

4.2.2 Model Components

We give an equation for each of the three components with respect to single defect-detection technique first and later for a combination of techniques. Keep in mind that the main basis of the model are expected values, i.e., we combine cost data with probabilities. For the sake of simplification we do not consider the defect propagation in these first equations but will introduce them later when describing the combination of more than one technique.

Direct Costs

The direct costs are those costs that can be directly measured from the application of a defect-detection technique. They are dependent on the length t of the application. Fig. 4.3 shows schematically the details of the direct costs for an application of technique A.

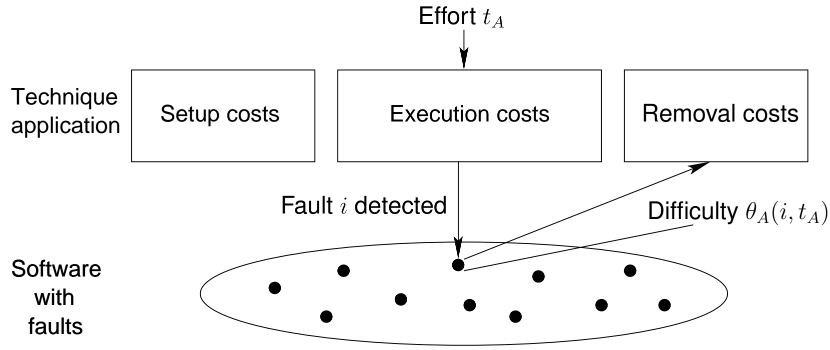


Figure 4.3: The components of the direct costs

It contains the two main cost blocks – setup costs and execution costs. The latter is dependent on the spent effort for A denoted by t_A. From the execution costs we can derive the difficulty of detecting the faults in the software which represents the probability that the fault is not detected. However, if a fault is detected it incurs costs for its removal. From this we can derive the following definition for the expected value of the direct costs $E[d_A(t_A)]$:

$$E[d_A(t_A)] = u_A + e_A(t_A) + \sum_i (1 - \theta_A(i, t_A))v_A(i), \tag{4.1}$$

where u_A are the setup costs, $e_A(t_A)$ the execution costs, and $v_A(i)$ the fault removal costs specific to that technique. Hence, we have for a technique its fixed setup costs, execution costs depending on the length of using the technique and removal costs for each fault in the software if the technique is able to find it.

Future Costs

In case some defects are not found, these will result in costs in the future denoted by $E[f_A(t_A)]$. We divide these costs into the two parts fault removal costs in the field $v_F(i)$ and failure effect costs $c_F(i)$. The latter contain all support and compensation costs as well as annoyed customers as far as it is possible to determine them (cf. Sec. 4.1.2).

$$E[f_A(t_A)] = \sum_i \pi_i \theta_A(i, t_A)(v_F(i) + c_F(i)), \tag{4.2}$$

where $\pi_i = P$(fault i is activated by randomly selected input and is detected and fixed) [120]. Hence, it describes the probability that the defect leads to a failure in the field.

Revenues

It is necessary to consider not only the costs of the defect-detection techniques but also their revenues. They are essentially saved future costs. With each fault that we find in-house we avoid higher costs in the future. Therefore, we have the same cost categories but look at the faults that we find instead of the ones we are not able to detect. We denote the revenues with $E[r_A(t_A)]$.

$$E[r_A(t_A)] = \sum_i \pi_i(1 - \theta_A(i, t_A))(v_F(i) + c_F(i)) \tag{4.3}$$

Because the revenues are saved future costs this equation looks similar to Eq. 4.2. The difference is only that we consider the faults that have been found and hence use the probability of the negated difficulty, i.e., $1 - \theta_A(i, t_A)$.

Combination

In practice, more than one technique is used to find defects. The intuition behind that is that they find (partly) different defects. Hence, we conjecture that different techniques are *diverse* in their defect-detection capabilities. This is theoretically shown in a model of diversity of techniques from Littlewood et al. [120]. Also empirical studies have shown that the effectiveness of several techniques differs significantly considering different defect types [8]. However, this has not been incorporated explicitly in most existing efficiency and economics models of SQA. As discussed above, we adapted the difficulty functions from Littlewood et al. [120] and thereby are able to express this diversity. The reason is that those functions are defined per fault and technique. This allows to express these differences.

To model the combination, we define that X is the ordered set of the applied defect-detection techniques. Hence, the sum over all technique applications X gives us the total direct costs. For each technique application we use Eq. 4.1 with the extension that not only the probability that the technique finds the fault is taken into account. Also the probability that the techniques applied before have not detected it is necessary. Moreover, the defect propagation needs to be considered. Not only the defect itself but also its predecessors R_i have not been detected.

For the sake of readability we introduce the abbreviation $\Theta(x, i)$ for the probability that a fault and its predecessors have not been found by previous – before x – applications of defect-detection techniques.

$$\Theta(x, i) = \prod_{y<x} \Big[\theta_y(i, t_y) \prod_{j \in R_i} \theta_y(j, t_y)\Big], \tag{4.4}$$

hence, for each technique y that is applied before x we multiply the difficulty for the fault i and all its predecessors as described in the set R_i. The expected value of the combined direct costs d_X of a sequence of defect-detection technique applications X is then defined as follows:

$$E[d_X(t_X)] = \sum_{x \in X} \Big[u_x + e_x(t_x) + \sum_i \Big((1 - \theta_x(i, t_x))\Theta(x, i)\Big) v_x(i)\Big], \tag{4.5}$$

where t_X is the ordered set of efforts for the techniques in X. Note that by using $\Theta(x,i)$ the difference to Eq. 4.1 is rather small. We extended it by the sum over all technique applications and the probability that each fault and its predecessors have not been found by previous techniques expressed by $\Theta(x,i)$.

The equation for the expected value of the revenues r_X of several technique applications X uses again a sum over all technique applications. In this case we look at the faults that occur, that are detected by a technique and neither itself nor its predecessors have been detected by the earlier applied techniques.

$$E[r_X(t_X)] = \sum_{x \in X} \sum_{i} \left[\Big(\pi_i (1 - \theta_x(i, t_x)) \Theta(x,i) \Big) \big(v_F(i) + c_F(i) \big) \right] \tag{4.6}$$

The total future costs are simply the costs of each fault with the probability that it occurs and all techniques failed in detecting it and its predecessors. In this case, the abbreviation $\Theta(x,i)$ for accounting of the effects of previous technique applications cannot be directly used because the outermost sum is over all the faults and hence the probability that a previous technique detected the fault is not relevant. The abbreviation $\Theta'(x,i)$ that describes only the product of the difficulties of detecting the predecessors of i is hinted in the following equation for the expected value of the future cost f_X of several technique applications X.

$$E[f_X(t_X)] = \sum_{i} \left[\pi_i \prod_{x \in X} \theta_x(i, t_x) \underbrace{\prod_{y<x} \prod_{j \in R_i} \theta_y(j, t_y)}_{\Theta'(x,i)} (v_F(i) + c_F(i)) \right] \tag{4.7}$$

Model Output

Based on the three components of the model, we are able to calculate several different economical metrics of the quality assurance process. We give the metrics *total cost*, *profit*, and *return on investment* in the following.

Total Cost. The *total cost* describes the sum of all economic costs necessary for producing products. It is one possible metric that can be optimised. In our model, the total costs can be calculated straightforwardly by adding the direct costs and the future costs:

$$\text{total cost} = d_X + f_X \tag{4.8}$$

Profit. We describe the gain provided by the quality assurance with the term *profit*. Hence, it is the revenues less the total cost. In the terms of our model, this can be defined using the three components as follows:

$$\text{profit} = r_X - d_X - f_X \tag{4.9}$$

ROI. Another metric used in economic analyses is the *return on investment* (ROI) of the defect-detection techniques. The ROI – also called *rate of return* – is commonly defined as the gain divided by the used capital. Boehm et al. [21] use the equation $(\text{Benefits} - \text{Costs})/\text{Costs}$. To calculate the total ROI with our model we have to use Eqns. 4.5, 4.7, and 4.6.

$$\text{ROI} = \frac{r_X - d_X - f_X}{d_X + f_X} \tag{4.10}$$

All these metrics can be used for two purposes: (1) an up-front evaluation of the quality assurance plan as the *expected* total cost, profit, or ROI of performing it and (2) a single post-evaluation of the quality assurance of a project. In the second case we can substitute the initial estimates with actually measured values. However, not all of the factors can be directly measured, e.g., effect costs of defects removed in-house. Hence, also the post evaluation metric must be seen as an *estimated* metric.

4.2.3 Forms of the Difficulty Functions

The notion of *difficulty* of the defect detection is a very central one in the described model. As mentioned, this notion is based on an idea from [120]. However, the original difficulty functions had no concept of time or spent effort but only of a single or more applications. To be able to analyse and optimise the spent effort on each technique, we need to introduce that additional dimension in the difficulty functions, i.e., the functional form depending on the spent effort. This is similar to the informal curves shown by Boehm [19] describing the effectiveness of different defect-detection techniques depending on the spent costs. Actually, the equations given for the model above already contain that extended difficulty functions but they are not further elaborated. This gap is closed in the following.

Firstly, we do not have sufficient data to give an empirically founded basis for the forms of the difficulty functions. Nevertheless, we can formulate hypotheses to identify the most probable distributions for different defects. Despite this limitation, using these hypotheses we can analyse the influence of different functional forms on the model output. Secondly, keep in mind that a difficulty function is defined for a specific defect-detection technique detecting a specific defect. That means that each defect can have distinct distribution for each possible technique.

Exponential Function

One possible intuition for the relationship between difficulty and effort is that with more effort spent the difficulty decreases, i.e., the probability of detecting that defect increases. However, with increasing effort the rate of difficulty reduction slows down. The defect detection does get more and more complicated when the "obvious" cases all have been tried. This intuition can be matched by modelling the difficulty as an exponential function.

For this we use a function similar to the density function of an exponential distribution:

$$\theta(i,t) = \begin{cases} \lambda_i e^{-\lambda_i t} & \text{if } t > 0 \\ 1 & \text{otherwise} \end{cases}, \tag{4.11}$$

with λ_i being a parameter that is determined from empirical data from the technique and the defect. It is the inverse of the mean value of the empirically measured difficulty. Hence, it needs to be in the the range $0 \leq \lambda_i \leq 1$.

Constant Function

The constant function constitutes a special case of the forms of the difficulty functions. In this case the spent effort does not matter because the difficulty of detecting the defect is always the same. The intuitive explanation for this functional form is best explained using the example of a static analysis tool. These tools often use bug patterns specific for a language and thereby identify code sections that are critical. When searching for a specific bug pattern it is of no importance how much effort is spent but if the tool is not able to detect a specific pattern – or only in seldom cases – the probability of detection does not change. We can also use this distribution to model that a specific technique A cannot detect a specific defect i by specifying that $\theta_A(i,t) = 1$ for all t. This is useful in particular in the case that A cannot detect defects of a specific class, i.e., in a specific document type. For example, a design inspection cannot detect defects in code.

Linear Function

The linear difficulty function models the intuition that there is a steady decrease in difficulty when applying more effort. A review is an example that might exhibit such a behaviour. The more intensively the reviewer reads the document the higher the possibility that he detects that specific defect. The function can be formulated as follows:

$$\theta_A(\tau_i, t) = mt + 1, \tag{4.12}$$

where m is the (negative) slope of the straight line.

Sigmoid Function

For our purposes it is sufficient to see the sigmoid function as a variation of the exponential function. Its graph has an *S*-like shape and hence one local minimum and one local maximum. In this special case we actually use a complementary sigmoid function to get a turned *S*. We depict an example in Fig. 4.4.

In contrast to the exponential function, the sigmoid function models the intuition that in the beginning it is hard to detect a specific defect and the difficulty does decrease only slowly. However, when a certain amount of effort is spent, the rate increases and the chance of detecting the defect increases significantly until we reach a point of saturation – similar to the exponential function – where additional effort does not have a large impact. This distribution is also backed by the so-called S-curve of software testing [101]. That S-curve aims in a slightly different direction but also shows that early in testing only a limited number of failures are revealed, then the detection rate increases until a plateau of saturation is reached.

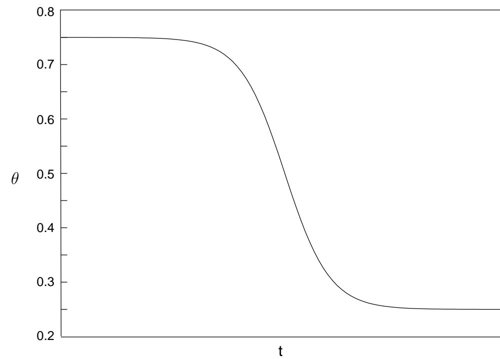


Figure 4.4: A sigmoid difficulty function

4.2.4 Example

We illustrate the model with a simple example. We oversimplify it here so that it is not too lengthy and too difficult to understand. However, for such a small system the use of the model can obviously be disputed. For the sake of the example we consider a software that only contains two faults 1 and 2 that are not related. We use only two different defect-detection techniques: an inspection A and a test suite B. Although this is not possible in practice, we assume that we are able to determine all of the input factors for the model w.r.t. this example system and quality assurance. The values can be found in Tab. 4.1. Furthermore, we have the setup costs $u_A = 100$ and $u_B = 500$. The loaded labour rate is assumed to be 100 and the execution costs for both techniques are simply the effort multiplied by the labour rate. We use different forms for the difficulty functions, i.e., an exponential function for $\theta_A(1)$, a constant function for $\theta_B(1)$, and linear functions for $\theta_A(2)$ and $\theta_B(2)$.

Table 4.1: The values of the input factors

Fault	v_F	f_F	π	θ_A	θ_B	v_A	v_B
1	5,000	1,000	0.8	$0.2e^{-0.2t_A}$	1	500	1000
2	2,000	20,000	0.2	$-0.02t_A + 1$	$-0.03t_B + 1$	400	900

We can directly compare the two defect-detection techniques A and B by looking at their factors w.r.t the two faults. Most notably, B needs significantly more costs to remove the faults than A. In this particular case, this probably stems from the fact that inspections allow easier debugging because they identify the fault directly whereas tests only detect failures. Furthermore, we see that B is not able to detect fault 1 because of its constant difficulty function. This implies that fault 1 is a defect that in principle cannot be detected by testing.

Based on these input factors, we can now choose an effort distribution over these two defect-detection techniques to plan the quality assurance. The model is then able to yield expected values for the different metrics given above. One possibility is to apply technique A with effort $t_A = 10$ person-hours and technique B with effort $t_B = 20$ person-hours in that order. We first use Eq. 4.4 to calculate the probabilities that faults 1 and 2 were found by the first used technique A. This information is later used for

technique B:

$$\Theta(B,1) = 0.2e^{-0.2\cdot 10} = 0.03$$

$$\Theta(B,2) = -0.02 \cdot 10 + 1 = 0.8$$

The calculations are rather simple here because we only apply two techniques and have no further relationships between faults.

Using these results, we can calculate the three components of the model. We want to show this for the direct costs $d_X(t_X)$ in more detail. The other two components are calculated accordingly. Using Eq. 4.5 we get the following:

$$\begin{aligned} E(d_X(t_X)) = \quad & [100 + 10 \cdot 100 && +(1 - 0.2e^{-0.2\cdot 10}) \cdot 500 \\ & && +(1 - (-0.02 \cdot 10 + 1)) \cdot 400] + \\ & [500 + 20 \cdot 100 && +((1 - 1) \cdot 0.03 \cdot 1,000 \\ & && +(1 - (-0.03 \cdot 20 + 1)) \cdot 0.8 \cdot 900 = 4,887 \end{aligned}$$

Hence, following the above quality assurance plan results in expected direct costs of 4,219. The other two components can be calculated similarly with the following results: $E(f_X(t_X)) = 15,544$ and $E(r_X(t_X)) = 6,592$. Having calculated the components it is straightforward to calculate different model outputs. The total cost is 20,431, i.e., the direct costs and the future costs added. Hence, this quality assurance plan costs over 20,000. This is the first factor that could be optimised to reduce the costs and thereby the invested capital.

However, we want to find out whether this QA plan is able to make a positive profit for the company. We use the equation for the profit and find it to be $-13,839$. We see that using this setting, we will have a loss of nearly 14,000. This is definitely not a desired result of quality assurance. Finally, to set the loss into relation to the invested capital, we also calculate the return on investment (ROI). For this we can simply divide the negative profit by the total cost which yields -0.68. This is definitely not acceptable. We need to change our QA plan either by changing the effort for the techniques or by using other or additional techniques. We will calculate examples for different effort distributions in a more detailed example in Sec. 4.5. This example here was only intended to give an idea what the model is capable of.

4.2.5 Empirical Knowledge

We review and summarise the empirical knowledge available for the quality economics of defect-detection techniques introducing the approach in general and then describing the relevant studies and results for each of the model factors for different types of techniques and defects. More details can be found in [205, 206].

Approach

This literature review aims at reviewing and summarising the existing empirical work that can be used to approximate the input parameters of the economics model proposed in Sec. 4.2. Literature review, also called meta-analysis, is a common technique in social sciences or medicine. Details on such a review can be found, for example, in [45]. For the meta-analysis we take all officially published sources into account, i.e., books,

journal articles, and papers in workshop and conference proceedings. In total we review 68 papers mainly following references from existing surveys and complementing those with newer publications. However, note that we only include studies with data relevant for the economics model. In particular, studies only with a comparison of techniques without detailed data for each were not taken into account.

We structure the available work in three parts for dynamic testing, review and inspection, and static analysis tools. We give a short characterisation for each category and describe briefly the available results for each relevant model input factor. We prefer to use and cite detailed results of single applications of techniques but also take mean values into account if necessary. We also summarise the combination of the results in terms of the lowest, highest, mean, and median value for each input factor and interesting other metrics in case there is enough data. These quantities can then be used in the model for various tasks, e.g., sensitivity analysis.

We deliberately refrain from assigning weights to the various values we combine although some of them are from single experiments while others represent average values. The reason is that we often lack knowledge on the sample size used and either we would estimate it or ignore the whole study result. An estimate of the sample size would introduce additional blurring into the data and omitting data considering the limited amount of data available is not advisable. Hence, we assume each data set of having equal weight.

Difficulty

The difficulty function θ is hard to determine because it is complex to analyse the difficulty of finding each potential fault with different defect-detection techniques. Hence, we need to use the available empirical studies to get reasonable estimates. Firstly, we can use the numerous results for the effectiveness of different test techniques. The effectiveness is the ratio of found defects to total defects and hence in some sense the counterpart to the difficulty function. In the paper of Littlewood et al. [120], where the idea of the difficulty function originated, *effectiveness* is actually defined this way. As a simple approximation we define the following for the difficulty:

$$\bar{\theta}_A = 1 - \text{effectiveness}_A \tag{4.13}$$

Using this equation we can determine the parameters of the different forms of the difficulty functions. For example, when using the linear function, we can use the average difficulty $\bar{\theta}$ and an average effort $\bar{t}$ to determine the slope m of the function. Then t can be varied to calculate the actual difficulty. The problem is that this is really a coarse-grained approximation that does not directly reflect the diversity of defect detection on different faults. Hence, we also need to analyse studies that use different defect types later in Sec. 4.3.3. Also the functional form cannot be determined in this way. The reason lies in the fact that most current studies do not analyse the effectiveness w.r.t. different amounts of effort.

Dynamic Testing

The first category of defect-detection techniques we look at is also the most important one in terms of practical usage. Dynamic testing is a technique that executes software

with the aim to find failures.

Setup Costs. The setup costs are mainly the staff-hours needed for understanding the specification in general and setting up the test environment. For this we can use data from [94]. There the typical setup effort is given in relation to the size of the software measured in function points (fp). Unit tests need 0.50 h/fp, function tests 0.75 h/fp, system test 1.00 h/fp, and field tests 0.50 h/fp. We have no data for average costs of tools and hardware but this can usually be found in accounting departments when using the economics model in practice.

Execution Costs. In the case of execution costs it is even easier than for setup costs as, apart from the labour costs, all other costs can be neglected. One could include costs such as energy consumption but they are extremely small compared to the costs for the testers. Hence, we can reduce this to the average labour costs. However, we also have average values per function point from [94]. There the average effort for unit tests is 0.25 h/fp, for function tests, system tests, and field tests 0.50 h/fp.

Effectiveness. There are nearly no studies that present direct results for the difficulty function of defect-detection techniques. Hence, we analyse the effectiveness and efficiency results first. Those are dependent on the test case derivation technique used. In the following, we summarise a series of studies that have been published regarding the effectiveness of testing in general and specific testing techniques.

- The experiment by Myers [143] resulted in an average percentage of defects found for functional testing of 36.0 and for structural of 38.0.
- Jones states in [94] that most forms of testing have an effectivity of less than 30%.
- He also states in [94] that a series of well-planned tests by a professionally staffed testing group can exceed 35% per stage and 80% in overall cumulative testing effectiveness.
- An experiment [132] showed that smoke tests for GUIs are able to detect more than 60% of the faults for most applications.
- In an experimental comparison of testing and inspection [113,114] the structural testing by teams had an effectiveness with a mean value of 0.17 and a std. dev. of 0.16.
- Hetzel reports in [75] on the average percentage of defects found for functional testing as 47.7 and for structural as 46.7.
- The study published in [8] compared testing over several successive usages to analyse the change in experience. In three phases of functional testing the mean effectiveness was with std. dev. in braces 0.64 (0.21), 0.47 (0.23), and 0.50 (0.15).

- Howden reports in [78] on an older experiment regarding different testing techniques. Path testing detected 18 of 28 faults and branch testing 6 of 28 faults. The combined use of different structural testing techniques revealed 25 of 28 faults.
- Weyuker reports in [218] on empirical results about flow-based testing. In particular they compared different metrics to measure the flow (executed paths and uses of variables. 71% of the known faults were exposed by at least all-du-paths, and 67% were exposed by all-c-uses, all-p-uses, all-uses, and all-du-paths.
- Paradkar describes an experiment for evaluating model-based testing using mutants in [154]. In two case studies the generated test suites were able to kill between 82% and 96% of the mutants.
- In [14] testing detects 7.2% of the defects.
- In [168] and Sec. 6.2 an evaluation of model-based testing is described. The effectiveness of eight test suites that can all be approximated as functional tests is given as 0.75, 0.88, 0.83, 0.33, 0.33, 0.46, 0.50, and 0.33.

A summary of the found effectiveness of functional and structural test techniques can be found in Tab. 4.2. We can observe that the mean and median values are all close which suggests that there are no strong outliers. However, the range in general is rather large, especially when considering all test techniques. When comparing functional and structural testing, there is no significant difference visible.

Table 4.2: Summary of the effectiveness of test techniques (in percentages)

Type	Lowest	Mean	Median	Highest
Functional	33	53.26	48.85	88
Structural	17	54.78	56.85	89
All	7.2	49.85	47	89

Difficulty. The approximation of the difficulty functions is given in Tab. 4.3. We used the results of the effectiveness summary above. Hence, the observations are accordingly.

Table 4.3: Approximation of the difficulty functions for testing

Type	Lowest	Mean	Median	Highest
Functional	12	46.74	51.15	67
Structural	11	45.22	43.15	83
All	11	50.15	53	92.8

Removal Costs. The removal costs are dependent on the second dimension of testing (cf. Sec. 2.3.2): the phase in which it is used. It is in general a very common observation in defect removal that it is significantly more expensive to fix defects in later phases than in earlier ones. Specific for testing is – in comparison with static techniques – that defect removal not only involves the act of changing the code but before that of localising the fault in the code. This is simply a result of the fact that testing always observes failures for which the causing fault is not necessarily obvious. We cite the results of several studies regarding those costs in the following.

- Shooman and Bolsky [191] analysed data from Bell Labs. They found the mean effort to identify the corresponding fault to a failure to be 3.05 hours. The minimum was 0.1 hours and the maximum 17 hours. The effort to correct those faults was then on average 1.98 hours with minimum 0.1 hours and maximum 35 hours. However, 53% of the corrections took between 0.1 and 0.25 hours.
- Jones [94] gives as industry averages during unit testing the effort to remove a defect to be 2.50 h/defect, during function testing to be 5.00 h/defect, and during system and field testing to be 10.00 h/defect.
- Collofello and Woodfield [43] report from a survey that asked for the effort needed to detect and correct a defect. The average result was 11.6 hours for testing.
- Franz and Shih [61] describe that the average effort per defect for unit testing at HP is 6 hours. During system testing the time to find and fix a defect is between 4 and 20 hours.
- Kelly et al. [104] state that it takes up to 17 hours to fix defects during testing.
- A study [179] found for the financial domain that a defect fix during coding and unit testing takes 4.9 hours, during integration 9.5 hours, and during beta-testing 12.1 hours.
- The same study [179] reports these measures for the transportation domain. The necessary hours to fix a defect during coding and unit testing are 2.4, during integration 4.1, and during beta-testing 6.2.
- Following [220] the effort to correct a requirements defect in a specific company in staff-days was (after 1991) 0.25 during unit test, 0.51 during integration test, 0.47 during functional test, 0.58 during system test.
- Rooijmans et al. [176] published data on the effort for the rework effort after testing in three projects. These were 4.0, 1.6, and 3.1 hours per defect, respectively.
- Möller [135] reports of removal costs during unit testing of $2,000$ DM and during system testing of $6,000$ DM.

Some statistics of the data above on the removal costs are summarised in Tab. 4.4. We assume a staff-day to consist of 6 staff-hours and combined the functional and

system test phases into the one phase “system test”. The removal costs (or efforts) of the three phases can be given with reasonable results. A combination of all values for a general average does not make sense as we get a huge range and a large difference between mean and median. This suggests a real difference in the removal costs over the different phases which is expected from standard software engineering literature, e.g. [19].

Table 4.4: Summary of the removal costs of test techniques (in staff-hours per defect)

Type	Lowest	Mean	Median	Highest
Unit	1.5	3.46	2.5	6
Integration	3.06	5.42	4.55	9.5
System	2.82	8.37	6.2	20
All	0.2	8	4.95	52

Review and Inspection

The second category of defect-detection techniques under consideration are reviews and inspections, i.e., reading documents with the aim to improve them.

Setup Costs. The first question is whether reviews and inspections do have setup costs. We considered setup costs to be fixed and independent of the time that the defect-detection technique is applied. In inspections we typically have a preparation and a meeting phase but both can be varied in length to detect more defects. Hence, they cannot be part of the setup costs. However, we have also an effort for the planning and the kick-off that is rather fixed. We consider those as the setup costs of inspections. One could also include costs for printing the documents but these costs can be neglected. Grady and van Slack describe in [71] the experience of Hewlett-Packard with inspections. They give an average time effort for the different inspection phases: for planning 2 staff-hours and for the kick-off 0.5 staff-hours.

Execution Costs. The execution costs are for inspections and reviews only the labour costs as long as there is no supporting software used. Hence, the execution costs are directly dependent on the factor t in our model. Nevertheless, there are average values for the execution costs of inspections.

- Grady and van Slack describe in [71] the experience of Hewlett-Packard with inspections. For the execution costs the typical time effort for the different inspection phases is as follows. The preparation phase has 2 staff-hours and the meeting 1.5 staff-hours. For cause and prevention analysis and follow-up usually take 0.5 staff-hours for each part.
- Jones has published average efforts in relation to the size in function points in [94]. Following this, a requirements review needs 0.25 h/fp, a design inspection 0.15 h/fp, and a code inspection 0.25 h/fp in the preparation phase. For the

meeting the values are for requirements reviews 1.00 h/fp, for design inspections 0.50 h/fp, and for code inspections 0.75 h/fp.

- In [198] usage-based reading (UBR) is compared to checklist-based reading (CBR). The mean preparation time was for UBR 53 minutes and for CBR 59 minutes.

- Porter et al. [162] conducted a long term experiment regarding the efficiency of inspections. They found that the median effort is about 22 person-hours per KNCSL.

- Jones gives in [94] typical rates for source code inspections as 150 LOC/h during preparation and 75 LOC/h during the meeting.

- Rösler describes in [178] his experiences with inspections. The effort for an inspection is on average one hour for 100 to 150 NLOC (non-commentary lines of code).

- In [1] the inspection of detailed design documents has a rate of 3.6 hours of individual preparation per thousand lines and 3.6 hours of meeting time per thousand lines. The results for code were 7.9 hours of preparation per thousand lines, 4.4 hours of meetings per thousand lines. Further results for detailed design documents were 5.76 h/KLOC for individual preparation, 4.54 h/KLOC for meetings. For code the results were 4.91 h/KLOC for preparation and 3.32 h/KLOC for meetings.

From these general tendencies, we can derive some LOC-based statistics for the execution costs of reviews. We assume for the sake of simplicity that all used varieties of the LOC metric are approximately equal. The results are summarised in Tab. 4.5. The mean and median values all are close. Only in code inspection meetings, there is a difference which can be explained by the small sample size. Note also that there is a significant difference between code and design inspections as the latter needs on average only half the execution costs. This might be explained by the fact that design documents are generally more abstract than code and hence easier to comprehend.

Table 4.5: Summary of the execution costs of inspection techniques (in staff-hours per KLOC)

Design	Lowest	Mean	Median	Highest
Preparation	3.6	4.68	4.68	5.76
Meeting	3.6	4.07	4.07	4.54
All	7.2	8.75	8.75	10.3
Code	**Lowest**	**Mean**	**Median**	**Highest**
Preparation	4.91	6.49	6.67	7.9
Meeting	3.32	7.02	4.4	13.33
All	6.67	13.2	11.15	22

Moreover, note that many authors give guidelines for the optimal inspection rate, i.e., how fast the inspectors read the documents. This seems to have an significant impact on the efficiency of the inspection.

- Rösler [178] argues for an optimal inspection rate of about 0.9 pages per hour.
- Gilb and Graham state in [69] than an optimal average rate is one page per hour.
- Krasner [108] gives the optimal bandwidth of the inspection rate as 1 ± 0.8 pages per hour where one page contains 300 words.

Hence, we can summarise this easily with saying that the optimal inspection rate lies about one page per hour. However, the effect of deviation from this optimum is not well understood. This, however, would increase the precision of economics models such as the one proposed above.

Effectiveness. Similar to the test techniques we start with analysing the effectiveness of inspections and reviews that is later used in the approximation of the difficulty.

- Jones [94] states that formal design and code inspections tend to be the most effective, and they alone can exceed 60%.
- Basili and Selby [8] compared three applications of code reading. The mean effectiveness was with std. dev. in braces 0.59 (0.28), 0.38 (0.28), and 0.57 (0.21).
- Defects in the space shuttle software are detected with inspections among other techniques [14]. Prebuild inspections are able to find 85.4% and other inspections further 7.3% of the total defects.
- Biffl et al. [13] describe experiments where the defect detection rate of inspections (share of defects found) has a mean of 45.2% with a standard dev. of 16.6%. In a second inspection cycle this is reduced to 36.5% with std.dev. 15.1%.
- Individual inspection effectiveness has a mean value of 0.52 with a std. dev. of 0.11. [113, 114]
- Biffl et al. analysed in [11] inspections and reinspections. They found that in the first inspection cycle 46% of all defects were found, whereas in the reinspection only 21% were detected.
- In [71] it is reported that typically 60 to 70 percent of the defects were found by inspections.
- In [8] three iterations of code reading were analysed. The mean effectiveness was with std. dev. in braces 0.47 (0.24), 0.39 (0.24), and 0.36 (0.20).
- Thelin et al. [199] report on several experiments on usage-based reading. The effectiveness was 0.29, 0.31, 0.34, and 0.32
- Thelin et al. [198] compare different reading techniques, in particular usage-based reading (UBR) and checklist-based reading (CBR). The effectiveness was 0.31 for UBR and 0.25 for CBR.

- Biffl and Halling [12] also looked at the cost benefits of CBR and scenario-based reading (SBR). The mean effectiveness (number of detected faults/number of all faults) in an experiment was the following with the roles user (SBR-U), designer (SBR-D), and tester (SBR-T). The results in percentages for different amounts of reading time can be found in [12].

We also summarise these results using the lowest, highest, mean, and median value in Tab. 4.6. We observe a quite stable mean value that is close to the median with about 30%. However, the range of values is huge. This suggests that an inspection is dependent on other factors to be effective.

Table 4.6: Summary of the effectiveness of inspection techniques (in percentage)

Lowest	Mean	Median	Highest
8.5	34.14	30	92.7

Difficulty. Using the simple approximation, we can derive statistics for the difficulty of inspections in reviews in Tab. 4.7. We also show our only results for the difficulty in dependence of the spent effort derived from [12] in Fig. 4.5. It shows the difficulty of several inspection techniques. We observe an exponential or linear form. The data are not enough to decide this completely.

Table 4.7: Derived difficulty of inspections using the approximation

Lowest	Mean	Median	Highest
7.3	65.86	70	91.5

Removal Costs. The removal costs of inspections only contain the fixing of the found faults because no additional localisation is required. As different document types can be inspected, we have to differentiate between them as well.

- During requirements reviews a removal effort of 1.00 hours per defect is needed. Design and code inspections have 1.50 h/defect. [94]
- In the book [171] the average effort to find and fix a major problem is given with 0.8 to 0.9 hours.
- In the report [130] the effort for rework for a defect is given as 2.5 staff-hours for in-house defects.
- Collofello and Woodfield [43] report from a survey that asked for the effort needed to detect and correct a defect. 7.5 hours were needed for a design defect and 6.3 hours for a code defect detected by inspections.

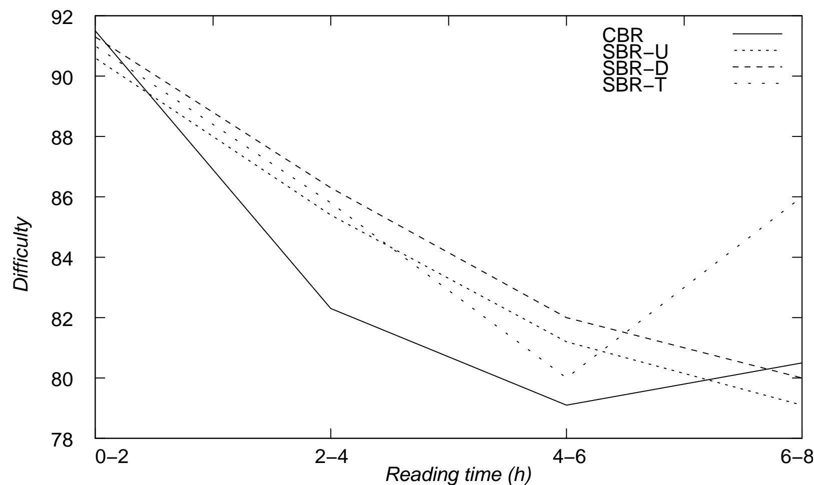


Figure 4.5: The difficulty of different inspection techniques in relation to the effort (reading time)

- Bourgeois published in [27] data on inspections. The average effort for inspections was 1.3 staff-hours per defect found and 2.7 staff-hours per defect found and fixed. In another project the average effort was 1.1 staff-hours per defect found and 1.4 to 1.8 staff-hours per defect found and fixed.

- The average effort per defect for code inspections was 1 hour (find and fix) at HP. [61]

- Kelly et al. [104] report that approximately 1.75 hours are needed to find and fix defects during design inspections, and approximately 1.46 hours during code inspections.

- The report in [179] states for the financial domain 1.2 hours to fix a defect during requirements analysis and 4.9 during coding. In the transportation domain the hours to fix a defect in the requirements phase are 2.0 and during coding 2.4.

- The effort to correct a requirements defect in staff-days was (after 1991) at Hughes Aircraft 0.05 during requirements analysis, 0.15 during preliminary design, 0.07 during detailed design and 0.17 during coding. [220]

- Möller [135] reports of removal costs during analysis, design, and coding of 500 DM. This value is not directly comparable to the other data based on person-effort. However, it can still give an intuition of the costs.

The summary of the removal costs can be found in Tab. 4.8. For the design reviews a strong difference between the mean and median can be observed. However, in this case this is not because of outliers in the data but because of the small sample size of only four data points. The large range when combining all the data for all types of inspections shows that there are probably significant differences between the inspection types.

Table 4.8: Summary of the removal costs of inspections (in staff-hours per defect)

Phase	Lowest	Mean	Median	Highest
Requirements	0.05	1.06	1.1	2
Design	0.07	2.31	0.83	6.3
Coding	0.17	2.71	1.95	6.3
All	0.05	1.91	1.2	7.5

Automated Static Analysis

The third and final category is tool-based analysis of software code to automatise the detection of certain types of defects.

Setup Costs. There are no studies with data about the setup and execution costs of using static analysis tools. In general, the setup costs are typically quite small consisting only of (possible) tool costs – although there are several freely available tools – and effort for the installation of the tools to have it ready for analysis.

Execution Costs. The execution costs are small in the first step because we only need to select the source files to be checked and run the automatic analysis. For tools that rely on additional annotations the execution costs are considerably higher. The second step, to distinguish between true and false positives, is more labour intensive than the first step. This requires possibly to read the code and analyse the interrelationships in the code which essentially constitutes a review of the code. Hence, the ratio of false positives is an important measure for the efficiency and execution costs of a tool.

- In [212] and Sec. 6.3 we found that the average ratio of false positives over three tools for Java was 66% ranging from 31% up to 96%.
- In [221] an evaluation of static analysis tools for C code regarding buffer overflows is described. The defects were injected and the fraction of buffer overflows found by each technique was measured. It is also noted that the rates of false positives or false alarms are unacceptably high.
- In [93] a static analysis tools for C code is discussed. The authors state that sophisticated analysis of, for example, pointers leads to far less false positives than simple syntactical checks.

Difficulty. The effectiveness of static analysis tools has only been investigated in a small number of studies and the results are mainly qualitative.

- In [181] among others PMD and FindBugs are compared based on their warnings which were not all checked for false positives. The findings are that although there is some overlap, the warnings generated by the tools are mostly distinct.

- Engler and Musuvathi discuss in [50] the comparison of their bug finding tool with model checking techniques. They argue that static analysis is able to check larger amounts of code and find more defects but model checking can check the implications of the code not just properties that are on the surface.

- We also analysed the effectiveness of three Java bug finding tools in [212] and Sec. 6.3. After eliminating the false positives, the tools were able to find 81% of the known defects over several projects. However, the defects had mainly a low severity where severity described the impact on the execution of the software. For the severest defects the effectiveness reduced to 22%, for the second severest defects even to 20%. For lower severities the effectiveness lies between 70% - 88%.

Field

In this section we look at the quantities that are independent from a specific defect-detection technique and can be associated to defects in the field. We are interested in removal costs in the field, failure severities as indicators of possible effect costs, and failure probabilities of faults.

Removal Costs. In this section we analyse only the removal costs of defects in the field as during development we consider the removal costs to be dependent on the used defect-detection technique.

- The ratio of the cost of finding and fixing a defect during design, test, and field use is: 1 to 13 to 92 [102] or 1 to 20 to 82 [175]

- The report [130] states that removal costs are 250 staff-hours per field-defect.

- The survey [43] resulted in the effort to detect and correct a defect in the field of 13.5 hours for a defect discovered.

- Following [179] the average effort to fix a defect in the financial domain after product release is 15.3 hours. In the transportation domain it is a bit lower with 13.1 hours per defect.

- Willis et al. [220] report that the rework per requirements defect in staff-days during maintenance was 0.65.

- In [180] an average effort to repair a defect after release to the customer is given as 4.5 staff-days.

- Bush gives $10,000 as average costs to fix a defect in the field in [33].

- Möller [135] gives $25,000$ DM as typical removal costs which constitutes a nearly exponential growth over the phases. This is also consistent with the value from Bush.

For the removal costs we have enough data to give reasonably some statistics. However, in the original results the mean and median are extremely different with 57.42

and 27.6, respectively. The mean is more than twice the median. This indicates that there are outliers in the data set that distort the mean value. Hence, we look at a box plot of the data that visualises the distribution with the aim to detect outliers. The box shows the lower and upper quartiles of the data with the median as a vertical line in the middle. The outer lines show the smallest and largest observation, respectively, with the outliers excluded. Those are denoted by "○" and "*" where the former denotes normal outliers and the latter extreme outliers.

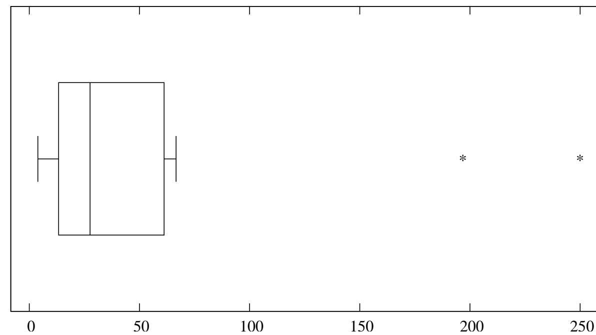


Figure 4.6: Box plot of the removal costs of field defects in staff-hours per defect

The box plot in Fig. 4.6 shows two extreme outliers, i.e., values that are more than three times of the interquartile range from the upper quartile away. We can eliminate those to get more reasonable results. With the reduced data set we get a mean value of 27.24 staff-hours per defect and a median of 27 staff-hours per defect. Hence, we have a more balanced data set with a mean value that can be further used. The results are summarised in Tab. 4.9. Fig. 4.7 shows the box plot of the data set without the outliers.

Table 4.9: Summary of the removal costs of field defects (in staff-hours per defect)

Lowest	Mean	Median	Highest
3.9	27.24	27	66.6

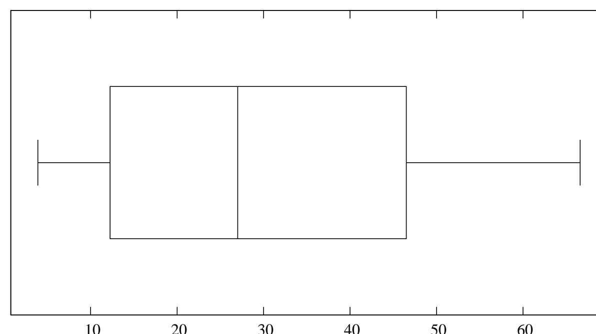


Figure 4.7: Box plot of the reduced data set of the removal costs of field defects in staff-hours per defect

Effect Costs. The effect costs are probably the most difficult ones to obtain. One reason is that these are highly domain-specific. Another is that companies often do not publish such data as it could influence their reputation negatively. There is also one more inherent problem. It is often just not possible to assign such costs to a single software fault. The highly complex configurations and the combination with hardware and possibly mechanics make such an assignment extremely difficult.

Yet, we cite two studies that published distribution of severity levels of defects. We consider the severity as the best available substitute of effect costs because more severe defects are probably more costly in that sense. However, this leaves us still with the need of a mapping of severity levels with typical effect costs for which there is no publicly available data.

Jones [94] states that the typical severity levels (1: System or program inoperable, 2: Major functions disabled or incorrect, 3: Minor functions disabled or incorrect, 4: Superficial error) have the following distribution:

1. 10%
2. 40%
3. 30%
4. 20%

Failure Probability. The failure probability of a fault is also one of the most difficult parts to determine in the economics model. Although there is the whole research field of software reliability engineering, there are only few studies that show representative distribution of such probabilities. The often cited paper by Adams [2] is one of the few exceptions. He mainly shows that the failure probabilities of the faults have an underlying geometric progression. This observation was also made in NASA studies reported in [146, 147]. This relationship can also be supported by data from Siemens when used in a reliability model [210] (cf. Sec. 6.4).

Labour Costs

The labour costs are difficult to capture correctly because they are constantly changing and are dependent on the inflation rate and many other factors. Hence, a general statement is not possible. Nevertheless, typical compensations for developers and testers can often be found in IT and software journals. A good source is also a book by Jones [95] that not only contains average salaries but also variations and overhead costs.

Discussion

Some of the summaries allow a comparison over different techniques. Most interestingly, the difficulty of finding defects is different between tests and inspections with inspections having more difficulties. Tests tend on average to a difficulty of 0.45 whereas inspections have about 0.65. The static analysis tools are hard to compare because of the limited data. They seem to be better in total but much worse considering severe defects.

The removal costs form a perfect series over the various techniques. As expected, the requirements reviews only need about 1 staff-hour of removal effort which rises over the other reviews to the unit tests with about 3.5 staff-hours. Over the testing phases we have again an increase to the system test with about 8 staff-hours. The field defects are then more than three times as expensive with 27 staff-hours. Hence, we can support the typical assumption that it gets more and more expensive to remove a defect over the development life-cycle.

We are aware that this survey can be criticised in many ways. One problem is clearly the combination of data from various sources without taking into account all the additional information. However, the aim of this survey is not to analyse specific techniques in detail and statistically test hypotheses but to determine some average values, some rules of thumb as approximations for the usage in an economics model. Furthermore, for many studies we do not have enough information for more sophisticated analyses.

Jones gives in [94] a rule of thumb: companies that have testing departments staffed by trained specialists will average about 10 to 15 percent higher in cumulative testing efficiency than companies which attempt testing by using their ordinary programming staff. Normal unit testing by programmers is seldom more than 25 percent efficient and most other forms of testing are usually less than 30 percent efficient when carried out by untrained generalists. A series of well-planned tests by a professionally staffed testing group can exceed 35 percent per stage, and 80 percent in overall cumulative testing efficiency. Hence, the staff experience can be seen as one of the influential factors on the variations in our results.

Nevertheless, we are able to provide data for most of the relevant factors for the model by synthesising the available empirical work. The main deficiencies are data for static analysis tools in principle and on effect costs of failures in the field. Furthermore, there is no data about the form of the difficulty functions, i.e., the difficulty or effectiveness in relation to the spent effort. Finally, in future empirical studies it would be helpful if the authors clearly add weights describing the size of the sample analysed to the results to simplify meta-analysis.

4.2.6 Sensitivity Analysis

Every newly proposed mathematical model should be subject to various analyses. Apart from the appropriateness of the model to the modelled reality and the validity of estimates and predictions, the dependence of the output on the input parameters is of interest. The quantification of this dependence is called *sensitivity analysis*. Local sensitivity analysis usually computes the derivative of the model response with respect to the model input parameters. More generally applicable is global sensitivity analysis that apportions the variation in the output variables to the variation of the input parameters. We base the following description of global sensitivity analysis mainly on [186].

Settings and Methods

Sensitivity analysis is the study of how the uncertainty in the output of a model can be apportioned to different sources of uncertainty in the model input. There are various

questions that can be answered by sensitivity analysis. As pointed out in [184] it is important to specify its purpose beforehand. In our context two settings are of most interest: (1) factors priorisation and (2) factors fixing.

Factors Priorisation (FP). The most important factor is the one that would lead to the greatest reduction in the variance of the output if fixed to its true value. Likewise the second most important factor can be defined and so on. The ideal use for the setting FP is for the prioritisation of research and this is one of the most common uses of sensitivity analysis in general. Under the hypothesis that all uncertain factors are susceptible to determination, at the same cost per factor, setting FP allows the identification of the factor that is most deserving of better experimental measurement in order to reduce the target output uncertainty the most. In our context that means that we can determine the factors that are most rewarding to measure most precisely.

Factors Fixing (FF). This setting is similar to factors priorisation but still has a slightly different flavour. Now, we do not aim to prioritise research in the factors but we want to simplify the model. For this we try to find the factors that can be fixed without reducing the output variance significantly. For our purposes this means that we can fix the input factor at any value in its uncertainty range without changing the outcome significantly.

FAST. There are various available methods for sensitivity analysis. The *Fourier amplitude sensitivity test (FAST)* is a commonly used approach that is based on Fourier developments of the output functions. It also allows a decomposition of the model output variance. In contrast to correlation or regression coefficients, it is not dependent on the goodness of fit of the regression model. The results give a quantification of the influences of the parameters, not only a qualitative ranking as for example the Morris method [137]. Furthermore, the FAST method was found computationally cheaper than comparable methods in [185].

With the latest developments of the FAST method, it is not only able to compute the first-order effects of each input parameter but also the higher-order and total effects. The first order effect is the influence of a single input parameter on the output variance, whereas the total effects also capture the interaction between input parameters. This is also important for the different settings as the first-order effects are used for the factors priorisation setting, the total-order effects for the factors fixing setting.

SimLab. We use the sensitivity analysis tool *SimLab* [192] for the analysis. Inside the tool we need to define all needed input parameters and their distributions of their uncertainty ranges. For this, different stochastic distributions are available. The tool then generates the samples needed for the analysis. This sample data is read from a file into the model – in our case a Java program – that is expected to write its output into a file with a specified format. This file is read again by SimLab and the first-order and total-order indexes are computed from the output.

Input Factors and Data

We describe the analysed scenario, factors and data needed for the sensitivity analysis in the following. The distributions are derived from the survey in Sec. 4.2.5. We base the analysis on an example software with 1,000 LOC and with 10–15 faults. The reason for the small number of faults is the increase in complexity of the analysis for higher numbers of faults. This small example can be a threat to the validity of the analysis but as most of the distribution of the input factors as analysed in Sec. 4.2.5 are dependent on the size of the software we are confident that this issue is not significant for the results.

Techniques. We have to base the sensitivity analysis on common or *average* distributions of the input factors. This also implies that we use a representative set of defect-detection techniques in the analysis. We choose seven commonly used defect-detection techniques and encode them with numbers: requirements inspection (0), design inspection (1), static analysis (2), code inspection (3), (structural) unit test (4), integration test (5), and (functional) system test (6). As indicated we assume that unit testing is a structural (glass-box) technique, system testing is functional (black-box), and integration testing is both. The usage of those seven techniques, however, does not imply that all of them are used in each sample as we allow the effort t to be null.

Additional Factors. To express the defect propagation concept of the model we added the additional factor ρ as the number of predecessors. The factor c represents the the defect class meaning the type of artefact the defect is contained in. This is important for the decision whether a certain technique is capable to find that specific defect at all. The factor ϕ encodes the form of difficulty function that is used for a specific fault and a specific technique. We include all the forms presented above in Sec. 4.2.3. The sequence s of techniques determines the order of execution of the techniques. We allow several different sequences including nonsense orders in which system testing is done first and requirements inspections as the last technique. Finally, the average labour costs per hour l is added because it is not explicitly included in the model equations from Sec. 4.2.2. Note, that we excluded the effect costs from the sensitivity analysis because we have not sufficient data to give any probability distribution.

Sigmoid Difficulty Function. For the purpose of sensitivity analysis we use a simple approximation of the sigmoid function because we believe that a very sophisticated function is in no relation to the limited amount of knowledge available on the real functional form. This approximation is a step function with two areas and corresponding two parameters γ and δ.

$$\theta(i,t) = \begin{cases} 0.9 & \text{if } t \leq \gamma \\ 0.1 & \text{if } \gamma < t \leq \delta \\ 0 & \text{otherwise} \end{cases} . \tag{4.14}$$

An example is shown in Fig. 4.8. The approximation we use is very rough simply because the data we have is not precise enough to justify a sophisticated function.

Hence, we assume we have some effort γ. When less than γ is spent it is very difficult to detect the defect and we assume a difficulty of 0.9. After γ the difficulty falls to 0.1 because the detection is now very probable. This approximation at least mirrors the intuition of the sigmoid function. Finally, we need also an effort δ that models the point where the difficulty reaches null. In the sigmoid function this point is in the infinity but for practical reasons, we make this an explicit point. This is because we want to use the collected empirical data and hence we need to be able to estimate the parameters from this data.

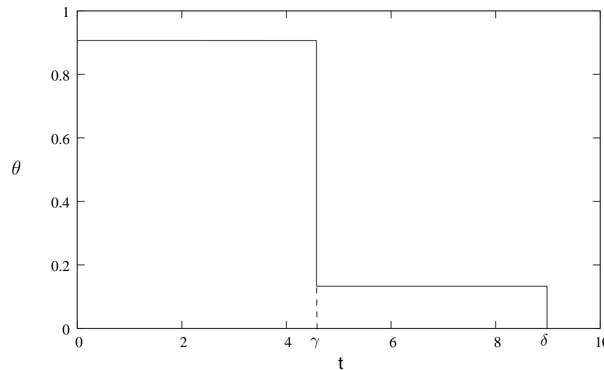


Figure 4.8: A difficulty step function

We cut off the function at double of the average effort with the intuition that most samples will lie inside this range. Hence, assuming that $\delta = 2\bar{t}$,

$$\gamma = 1.6\bar{\theta}\bar{t} - 0.4\bar{t}. \tag{4.15}$$

Data. We reuse the empirical knowledge that we compiled in Sec. 4.2.5. We model the input factors using triangular distributions with the minimum, maximum, and mean values synthesised from the analysed studies.

Results and Observations

This section summarises the results of applying the FAST method for sensitivity analysis on the data from the example above and discusses observations. The analysed output factor is the return-on-investment (ROI).

Abstract Grouping. We first take an abstract view on the input factors and group them without analysing the input factors for different techniques separately. The SimLab tool allows this grouping directly by combining the influence of the selected factors. Hence, we only have 11 input factors that are ordered with respect to their first and total order indexes in Tab. 4.10. The first order indexes are shown on the left, the total order indexes on the right.

The first order indexes are used for the factors priorisation setting. We see that the types of documents or artefacts the defects are contained in, denoted by a, are most rewarding to be investigated in more detail. One reason might be that we use a uniform

Table 4.10: The first and total order indexes of the abstract grouping

First order			Total order		
Document type	a	0.4698	Document type	a	0.8962
Exec. effort	t	0.1204	Remv. costs field	v_f	0.4473
Avg. difficulty	$\bar{\theta}$	0.0699	Avg. difficulty	$\bar{\theta}$	0.4255
Remv. costs field	v_f	0.0541	Setup costs	u	0.3916
Form of diff. function	ϕ	0.0365	Exec. effort	t	0.3859
Setup costs	u	0.0297	Form of diff. function	ϕ	0.2888
Remv. costs in-house	v	0.0264	Num. of predecessors	ρ	0.2711
Num. of predecessors	ρ	0.0256	Remv. costs in-house	v	0.2546
Failure prob. field	π	0.0158	Failure prob. field	π	0.2068
Tech. sequence	s	0.0083	Tech. sequence	s	0.1825
Labour costs	l	0.0010	Labour costs	l	0.1489

distribution because we do not have more information on the distribution of defects over document types. However, this seems to be an important information. The factor that ranks second highest is the spent effort t. In this analysis it is equal to $e(t)$ because we did not consider other cost factors during the execution. This approves the intuition that the effort has strong effects on the output and hence needs to be optimised. Also the average difficulty of finding a defect with a technique $\bar{\theta}$ and the costs of removing a defect in the field v_f are worth to be investigated further. Interestingly, the labour costs l, the sequence of technique application s, and the failure probability in the field π do not contribute strongly to the variance in the output. Hence, these factors should not be the focus in further research.

For the factors fixing setting, the ordering of the input factors is quite similar. Again the failure probability in the field π, the sequence of technique application s and the labour costs l can be fixed with changing the output variance the least. Note that this does not imply that the sequence and the labour costs are not important for the total outcome. Only for the variance and hence the validity of predictions they are not significant. For the labour costs, this can be explained by the fact that they appear in the costs as well as the revenues. Hence, they may change the ROI but not its variance. The factors that definitely cannot be fixed are again the document types, the removal costs in the field, and the average difficulty values. The setup costs rank higher with these indexes and hence should not be fixed.

Detailed Grouping. After the abstract grouping, we form smaller groups and differentiate between the factors with regard to different defect-detection techniques. The first and total order indexes are shown in Tab. 4.11 again with the first order indexes on the left and the total order indexes on the right.

The main observations from the abstract grouping for the factor priorisation setting are still valid. The type of artefact the defect is contained in (c) still ranks highest and has the most value in reducing the variance. However, in this detailed view, the failure probability in the field π ranks higher. This implies that this factor should not

Table 4.11: The first and total order indexes of the detailed grouping

First order			Total order		
Document type	a	0.2740	Document type	a	0.7750
Exec. effort 1	t_1	0.0601	Form of diff. function 4	ϕ_4	0.3634
Failure prob. field	π	0.0528	Exec. effort 1	t_1	0.3332
Form of diff. function 4	ϕ_4	0.0492	Failure prob. field	π	0.3200
Form of diff. function 1	ϕ_1	0.0391	Remv. costs field	v_f	0.2821
Remv. costs 3	v_3	0.0313	Remv. costs 4	v_3	0.2802
Form of diff. function 0	ϕ_0	0.0279	Form of diff. function 1	ϕ_1	0.2728
Num. of predecessors	ρ	0.0278	Num. of predecessors	ρ	0.2706
Form of diff. function 2	ϕ_2	0.0269	Remv. costs 1	v_1	0.2574
Remv. costs field	v_f	0.0252	Tech. sequence	s	0.2524
Form of diff. function 6	ϕ_6	0.0222	Avg. difficulty 5	$\bar{\theta}_5$	0.2493
Remv. costs 0	v_0	0.0219	Avg. difficulty 0	$\bar{\theta}_0$	0.2312
Form of diff. function 3	ϕ_3	0.0216	Avg. difficulty 3	$\bar{\theta}_3$	0.2300
Avg. difficulty 6	$\bar{\theta}_6$	0.0214	Form of diff. function 6	ϕ_6	0.2287
Remv. costs 5	v_5	0.0212	Form of diff. function 2	ϕ_2	0.2240
Avg. effort 0	$\bar{\theta}_0$	0.0209	Avg. effort 1	$\bar{\theta}_1$	0.2077
Tech. sequence	s	0.0208	Remv. costs 5	v_5	0.2039
Avg. difficulty 1	$\bar{\theta}_1$	0.0203	Form of diff. function 0	ϕ_0	0.1966
Remv. costs 1	v_1	0.0203	Setup costs 3	u_3	0.1913
Avg. difficulty 4	$\bar{\theta}_4$	0.0197	Remv. costs 0	v_0	0.1907
Form of diff. function 5	ϕ_5	0.0194	Avg. difficulty 6	$\bar{\theta}_6$	0.1894
Avg. difficulty 5	$\bar{\theta}_5$	0.0186	Form of diff. function	ϕ_5	0.1892
Exec. effort 2	t_2	0.0185	Form of diff. function 3	ϕ_3	0.1854
Avg. difficulty 3	$\bar{\theta}_3$	0.0181	Remv. costs 4	v_4	0.1807
Remv. costs 6	v_6	0.0142	Exec. effort	t_5	0.1719
Remv. costs 2	v_2	0.0139	Remv. costs 6	v_6	0.1709
Remv. costs 4	v_4	0.0120	Avg. difficulty 4	$\bar{\theta}_4$	0.1707
Avg. difficulty 2	$\bar{\theta}_2$	0.0109	Remv. costs 2	v_2	0.1633
Exec. effort 6	t_6	0.0089	Exec. effort 6	t_6	0.1619
Exec. effort 4	t_4	0.0058	Setup costs 5	u_5	0.1451
Exec. effort 3	t_3	0.0051	Exec. effort 4	t_4	0.1409
Exec. effort 5	t_5	0.0034	Setup costs 4	u_4	0.1404
Setup costs 5	u_5	0.0018	Exec. effort 2	t_2	0.1378
Setup costs 0	u_0	0.0013	Exec. effort 0	t_0	0.1268
Setup costs 4	u_4	0.0010	Labour costs	l	0.1222
Exec. effort 0	t_0	0.0009	Avg. difficulty 2	$\bar{\theta}_2$	0.1125
Setup costs 3	u_3	0.0007	Setup costs 6	u_6	0.1122
Setup costs 6	u_6	0.0007	Setup costs 0	u_0	0.1085
Setup costs 1	u_1	0.0005	Exec. effort 3	t_3	0.1053
Setup costs 2	u_2	0.0005	Setup costs 1	u_1	0.1034
Labour costs	l	0.0002	Setup costs 2	u_2	0.0996

be neglected. We also see that for some techniques the form of the difficulty function ϕ has a strong influence and that the setup costs u of most techniques rank low.

Similar observations can be made for the factors fixing setting and the total order indexes. The main observations are similar as in the abstract grouping. Again, the failure probability in the field π ranks higher. Hence, this factor cannot be fixed without changing the output variance significantly. A further observation is that some of the setup costs u can be set to fixed value what reduces the measurement effort.

Consequences

From the observations above we can conclude that the labour costs, the sequence of technique application and the removal costs of most techniques are not an important part of the model and the variation in effort does not have strong effects on the output, i.e., the ROI in our case. On the other hand, the type of artefact or document the defect is contained in, the difficulty of defect detection, and the removal costs in the field have the strongest influences. Interestingly, Sabaliauskaite supports in [182] the observation that it is important to consider the differences in artefacts for evaluating inspections.

This has several implications: (1) We need more empirical research on the distribution of defects over different document types and the removal costs of defects in the field to improve the model and confirm the importance of the factor, (2) we still need more empirical studies on the effectiveness of different techniques as this factor can largely reduce the output variance, (3) the labour costs do not have to be determined in detail and it does not seem to be relevant to reduce those costs, (4) further studies on the sequence and the removal costs are not necessary.

4.2.7 Discussion

The model so far is not suited for a practical application in a company as the quantities used are not easy to measure. Probably, we are unable to get values for the difficulty θ of each fault and defect-detection technique. Also the somehow fixed and distinct order of techniques is not completely realistic as some techniques may be used in parallel or only some parts of the software are analysed. However, in a more theoretical setting we can already use the model for important tasks including sensitivity analysis to identify important input factors.

Another application can be to analyse which techniques influence which parts of the model. For instance, the automatic derivation of test cases from explicit behaviour models (model-based testing) is a relatively new technique for defect detection. This technique can be analysed and compared with traditional, hand-crafted test techniques based on our model. Two of the factors are obviously affected by model-based testing: (1) the setup costs are considerably higher than in hand-crafted tests because not only the normal test environment has to be set up but also a formal (and preferably executable) model of the behaviour has to be developed. On the contrary, the execution cost per test case is then substantially smaller because the generation can be automated to some extent and the model can be used as an oracle to identify failures. Further influences on factors like the difficulty functions are not that obvious but need to be analysed. This example shows that the model can help to structure the comparison and analysis of defect-detection techniques.

Finally, we have to add that it might be interesting to relax the assumption of perfect debugging, i.e., that every found defect is removed without introducing new defects. In practice it is common that defect removal introduces new defects. Also sometimes there might be a decision that a defect is not removed as it would be too costly or the defect is particular minor and does not affect many customers. However, this would introduce further complexity to the model and in software reliability modelling (cf. 2.4.2) the experience has been that many models have a good predictive validity despite this assumption. Hence, we leave the influence of this factor for future work.

4.3 Practical Model

As we discussed above, the theoretical model can be used for analyses but is too detailed for a practical application. The main goal of this dissertation is, however, to optimise the usage of defect-detection techniques which requires applicability in practice. Hence, we need to simplify the model to reduce the needed quantities.

4.3.1 Basics

For the simplification of the model, we use the following additional assumptions:

- Faults can be categorised in useful defect types.
- Defect types have specific distributions regarding their detection difficulty, removal costs, and failure probability.
- The linear functional form of the difficulty approximates all other functional forms sufficiently.

We define τ_i to be the defect type of fault i. It is determined using the defect type distribution of older projects. In this way we do not have to look at individual faults but analyse and measure defect types for which the determination of the quantities is significantly easier. In the practical model we assume that the defects can be grouped in “useful” defect types. For reformulating the equation it is sufficient to consider the affiliation of a defect to a type but for using the model in practice we need to further elaborate on the nature of defect types and how to measure them. In Sec. 2.2.2 we described the current state in research and practice of defect types. For our economics model we consider the defect classification approaches from IBM [101] and HP [70] as most suitable because they are proven to be usable in real projects and have a categorisation that is coarse-grained enough to make sensible statements about each category.

We also remove the concept of defect propagation as it was shown not to have a high priority in the analyses above but it introduces significant complexity to the model. Hence, the practical model can be simplified notably. For the practical use of the model, we also need an estimate of the total number of defects in the artefacts. We can either use generalised quality models such as [94] or product-specific models such as COQUALMO [182]. To simplify further estimates other approaches can be used. For example, the defect removal effort for different defect types can be predicted using an association mining approach of Song et al. [194].

4.3.2 Components

Similar to Sec. 4.2.2 where we defined the basic equations of the ideal model, we formulate the equations for the practical model using the assumptions from above.

Single Economics

We start with the direct costs of a defect-detection technique. It is the counterpart of Eq. 4.1.

$$E[d_A(t_A)] = u_A + e_A(t_A) + \sum_i (1 - \theta_A(\tau_i, t_A)) v_A(\tau_i), \tag{4.16}$$

where u_A is the average setup cost for technique A, $e_A(t_A)$ is the average execution cost for A with length t, and $v_A(\tau_i)$ is the average removal cost in defect type τ_i. The main difference is that we consider defect types in the difficulty functions. The same applies to the revenues.

$$E[r_A(t_A)] = \sum_i \pi_{\tau_i} (1 - \theta_A(\tau_i, t_A))(v_F(\tau_i) + c_F(\tau_i)), \tag{4.17}$$

where $c_F(\tau_i)$ is the average effect costs of a fault of type τ_i. Finally, the future costs can be formulated accordingly.

$$E[f_A(t_A)] = \sum_i \pi_{\tau_i} \theta_A(\tau_i, t_A)(v_F(\tau_i) + c_F(\tau_i)). \tag{4.18}$$

With the additional assumptions, we can also formulate a unique form of the difficulty functions:

$$\theta_A(\tau_i, t_a) = m t_A + 1, \tag{4.19}$$

where m is the (negative) slope of the straight line. If a technique is not able to detect a certain type, we will set $m = 0$. Thus, the difficulty is always 1.

Combined Economics

Similarly, the extension to more than one technique can be done. The following equation for the expected value of the direct costs is the counterpart to Eq. 4.5:

$$E[d_X(t_X)] = \sum_{x \in X} \left[u_x + e_x(t_x) + \sum_i (1 - \theta_x(\tau_i, t_x)) \prod_{y<x} \Big(\theta_y(\tau_i, t_y) \Big) v_x(\tau_i) \right] \tag{4.20}$$

Also the expected value of the combined future costs f_X can be formulated in the practical model using defect types.

$$E[f_X(t_X)] = \sum_i \pi_{\tau_i} \prod_{x \in X} \Big(\theta_x(\tau_i, t_x) \Big) \Big(v_F(\tau_i) + c_F(\tau_i) \Big) \tag{4.21}$$

Finally, the expected value of the combined benefits b_X are defined accordingly.

$$E[b_X(t_X)] = \sum_{x \in X} \sum_i \pi_{\tau_i} (1 - \theta_x(\tau_i, t_x)) \prod_{y<x} \Big(\theta_y(\tau_i, t_y) \Big) \Big(v_F(\tau_i) + c_F(\tau_i) \Big) \tag{4.22}$$

4.3.3 Empirical Knowledge

For further analyses of the practical model, we also elicit the available empirical work on defect types in the following.

Dynamic Testing

To our knowledge, there is only one study that investigated the question of effectiveness of defect-detection techniques with respect to defect types. Basili and Selby analysed the effectiveness of functional and structural testing regarding different defect types in [8]. Tab. 4.12 shows the derived average difficulties $\bar{\theta}$ using the approximation based on the effectiveness (cf. Sec. 4.2.5).

Table 4.12: Difficulties of functional and structural testing for detecting different defect types in percentages

	Functional Testing	Structural Testing	Overall
Initial.	25.0	53.8	38.5
Control	33.3	51.2	47.2
Data	71.7	73.2	74.7
Computat.	35.8	41.2	75.4
Interface	69.3	75.4	66.9
Cosmetic	91.7	92.3	89.2

It is obvious that there are differences of the two techniques for some defect types, in particular initialisation and control defects. This study has been replicated and the findings were mainly confirmed [99].

Review and Inspection

Analogous to the test techniques, we only have one study about effectiveness and defect types [8]. The derived difficulty functions are given in Tab. 4.13. Also for inspections, large differences between the defect types are visible but the small number of studies does not guarantee generalisability.

Table 4.13: Difficulty of inspections to find different defect types in percentages

Initial.	35.4
Control	57.2
Data	79.3
Computat.	29.1
Interface	53.3
Cosmetic	83.3

Table 4.14: Average defect type distribution from inspections

Doc.	Std. use	Logic	Func.	Data	Syntax	Perf.	Others
44.27	21.05	7.65	6.36	5.00	4.66	2.59	8.42

O'Neill [152] also reports on the average distribution of defects types from a large study. The results can be found in Tab. 4.14 with the ratios given in percentages.

Automated Static Analysis

For static analysis tools there is no study comparable to the experiment of Basili and Selby [8]. Nevertheless, we summarise the available knowledge.

- In [78] are also some static analysis techniques evaluated. Interface consistency rules and anomaly analysis revealed 2 and 4 faults of 28, respectively.
- Bush et al. report in [34] on a static analyser for C and C++ code which is able to find several more dynamic programming errors. However, a comparison with tests was not done.
- Palsberg describes in [153] some bug finding tools that use type-based analysis. He shows that they are able to detect race conditions or memory leaks in programs.

Field

We only have data for the removal costs w.r.t. different defect types. In [117] it was found that interface defects consume about 25% of the effort and 75% can be attributed to implementation defects dominated by algorithm and functionality defects. The efforts in person days are on the average 4.6 for external, 6.2 for interface, 4.7 for implementation defects. Outliers are data design with 1.9 and inherited defects 32.8, unexpected interactions 11.1 and performance defects 9.3.

Defect Types

General distributions of defect types are probably difficult to obtain. However, we can determine the defect type distribution for certain application types. Yet, there is only little data published. Sullivan and Chillarege described the defect type distribution of the database systems DB2 and IMS in [195]. The distributions (in percentages) can be found in Tab. 4.15. Interestingly, the trend in this distributions was confirmed in [49] where several open source projects were analysed.

Lutz and Mikulski used for defects in NASA software a slightly different classification of defects in [122] but they also have algorithms and assignments as types with a lot of occurrences. The most often defect type, however, is *procedures* meaning missing procedures or wrong call of procedures. We can see that the defect types are strongly domain- and problem-specific and general conclusions are hard to make.

Table 4.15: Defect type distributions in database systems

System	Assignment Checking	Build	Data-Struct Algorithm	Function	Interface	Timing
DB2	48.19	3.6	19.82	12.16	2.25	13.96
IMS	56.22	2	23.38	1.99	9.95	6.47

4.3.4 Sensitivity Analysis

Similar to the analyses in Sec. 4.2.6 we determined the first and total order indexes of the practical model again with data from [205,206]. The results are shown in Tab. 4.16 with the first order indexes left and the total order indexes right. We have to remark that we only looked at defects in the code because we have no empirical data on defect types in other kinds of documents. Furthermore, we introduced the factor α that describes the defect type distribution. This is necessary because that distribution is not a part of the model but is assumed to be determined beforehand.

Table 4.16: The first and total order indexes from the practical model

First order			Total order		
Exec. effort	t	0.1196	Exec. effort	t	0.8855
Failure prob. field	π	0.1138	Remv. costs field	v_f	0.8670
Avg. difficulty	$\bar{\theta}$	0.1097	Tech. sequence	s	0.7881
Defect type distr.	α	0.0975	Avg. difficulty	$\bar{\theta}$	0.7857
Remv. costs field	v_f	0.0694	Labour costs	l	0.7772
Labour costs	l	0.0634	Defect type distr.	α	0.6676
Tech. sequence	s	0.0592	Failure prob. field	π	0.6200
Setup costs	u	0.0476	Setup costs	u	0.4902
Remv. costs in-house	v	0.0018	Remv. costs in-house	v	0.0958

We see that the effort for the techniques t ranks highest in both settings. The failure probability π again ranks high in the factors priorisation setting. Hence, this factor should be investigated in more detail which was not obvious from the analysis of the ideal model. However, similarly to the ideal model, the setup and removal costs of the techniques (u and v) do not contribute strongly to the output variance.

In the factors fixing setting, we see that the setup and removal costs can be fixed without changing the variance significantly. This implies that we can use coarse-grained values here. Also the failure probability π can be taken from literature values. More emphasis, however, should be put on the effort t, the removal costs in the field v_f, and the sequence of technique application s. The last one is surprising as for the ideal model this factor ranked rather low.

4.3.5 Cost-Optimisation

For the optimisation only two of the three cost-components of the model are important because the future costs and the saved costs (revenues) are dependent on each other.

There is a specific number of faults that have associated costs when they occur in the field. These costs are divided in the two parts that are associated with the revenues and the future costs, respectively. The total always stays the same, only the size of the parts varies depending on the defect-detection technique. Therefore, we use only the direct costs and the revenues for optimisation and consider the future costs to be dependent on the revenues. Because we use the revenues in the equation to be optimised we could also call it *profit*-optimisation.

Hence, the optimisation problem can be stated by: maximise $r_X - d_X$. By using Eq. 4.20 and Eq. 4.22 we get the following term to be maximised.

$$\sum_x \Big[-u_x - e_x(t_x) + \sum_i (1 - \theta_x(\tau_i, t_x)) \prod_{y<x} (\theta_y(\tau_j, t_y)) \big(\pi_{\tau_i} v_F(\tau_i) + \pi_{\tau_i} c_F(\tau_i) - v_x(\tau_i) \big) \Big] \tag{4.23}$$

The equation shows in a very concise way the important factors in the economics of defect-detection techniques. For each technique there is the fixed setup cost and the execution costs that depend on the effort. Then for each fault in the software (and over all defect types) we use the probability that the technique is able to find the fault and no other technique has found the fault before to calculate the expected values of the other costs. The revenues are the removal costs and effect costs in the field with respect to the failure probability because they only are relevant if the fault will lead to a failure. Finally, we have to subtract the removal costs for the fault of that technique which is typically much smaller than in the field.

For the optimisation purposes, we probably also have some restrictions, for example a maximum effort t_{max} with $\sum_x t_x \leq t_{max}$, either fixed length of 100 or none $t_A \in \{0, 100\}$, or some fixed orderings of techniques, that have to be taken into account. The latter is typically true for different forms of testing as system tests are always later in the development than unit tests.

Having defined the optimisation problem and the specific restrictions we can use standard algorithms for solving it. It is a hard problem because it involves multi-dimensional optimisation over a permutation space, i.e., not only the length of technique usage can be varied but also the sequence of the defect-detection techniques.

4.4 Application

In the sections above, we have already discussed several possible applications of the two models. We concentrate on the application of the practical model in company to analyse and optimise its analytical quality assurance in the following.

4.4.1 General

First we discuss general issues that are independent of the process model that is used in the development project.

Needed Quantities

Using the practical model, we identify only seven different types of quantities that are needed to use the model:

- Estimated number of faults: I
- Distribution of defect types
- Difficulty functions for each technique and type $\theta_x(\tau_i)$
- Average removal costs for type τ_i with technique x: $v_x(i)$
- Average removal costs for type τ_i in the field: $v_F(i)$
- Average effect costs for type τ_i in the field: $c_F(i)$
- Failure probability of fault of type τ_i: π_{τ_i}

For an early application of the model, average values from the literature review can be used as first estimates, especially for those that ranked low in the FF setting (cf. Sec. 4.2.6).

Expert Opinion

The practical application of the model depends in large parts on uncertain data that is elicited using expert opinion. We discussed several issues of this approach in [204]. The data is not really "measured" but estimated by people. Hence, the data is unreliable and subject to bias. Nevertheless, every data has imprecision as we cannot always be sure how accurately that data is recorded. Therefore, we have to deal with this unreliability and uncertainty using a structured approach to expert judgement. A possible approach is described in [133].

To this end we mainly use the approach developed by Freimut et al. [63] for their economics model of inspections. It is important that expert opinion is inherently uncertain. Hence, it is unrealistic to ask only for one estimate per parameter. Instead we use – similar to the synthesised empirical work – a triangular distribution with upper and lower bounds and a most likely value. There are also various methods to reduce the bias that is introduced in the data but we refer to [63] for details.

Defect Prediction

Apart from the analytical model proposed in this chapter, there are various other stochastic models that can be used to analyse the defect content of documents and code based on various metrics as also described in Sec. 2.4. For example, the COQUALMO model [41] contains a component that can be used for defect estimation.

During system testing software reliability models can be used to check the defect estimates. We describe such an approach in [213]. This is useful for recalculating the model in that late phase in which we have far more information about the system and its failure behaviour. Hence, the estimates on the defect content and the failure behaviour in the field can be improved.

Furthermore, there are also statistical methods used in the context of reviews. For example in [216], data from reviews were analysed and statistical distributions found that can be used to control the application of reviews. Also these technique-specific results can be used to improve the initial estimates that were input in the economics model.

4.4.2 Embedding in the V-Modell XT

Following the general discussion on the practical application of the economics model, we show an exemplary embedding of the usage of the model in an existing process model. For the example we choose the new version of the German standard model *V-Modell XT* [32, 174] but an embedding in other models such as RUP [110] could be done accordingly. For the embedding we need to extend and change some of the roles, products, and activities of the V-Modell. However, we first give a brief overview of the general process and then describe the extensions and changes in more detail.

Overview

To allow a concise overview of the usage of the model in the V-Modell, we built an abstract activity diagram shown in Fig. 4.9. The *quality manager* is responsible for cross-project quality standards, metrics, and methods. In particular, he defines and maintains the metrics catalogue that must contain all the necessary input factors summarised in Sec. 4.4.1. This catalogue is documented in the *quality management manual*. Based on that, the quality manager sets up and maintains an infrastructure that is able to store the corresponding metrics.

The main user of our model is the *project leader*. He performs the basic estimations for the project including the defect estimate needed for our model. Here, the metrics infrastructure can give guidance with data from similar projects. Based on the estimates, the project leader uses our model to calculate an optimised quality assurance and documents that in the *QA manual*. The *QA manager* further elaborates this QA manual. It is then used by the *evaluator* to prepare evaluation specifications. Note that we abstract here from the actual V-Modell because it differentiates different artefacts that can be evaluated. For some an explicit realisation of *evaluation procedures* is necessary. After evaluating the artefact, the evaluator writes an *evaluation report* that includes all measurements that can be later used in the model such as the number and type of the detected defects.

The *QA manager* collects those evaluation reports and prepares the *quality status report* which is later used by the project leader as part of the *project status report*. In that all necessary data of the project progress are cumulated. Based this report, a *project progress decision* is made possibly using our model to evaluate different scenarios and optimising the remaining quality assurance. This new QA plan is then part of the QA manual. When the project is finished, the project leader collects the *measurement data* relevant for our model and forwards it to the quality manager who stores it in the metrics infrastructure.

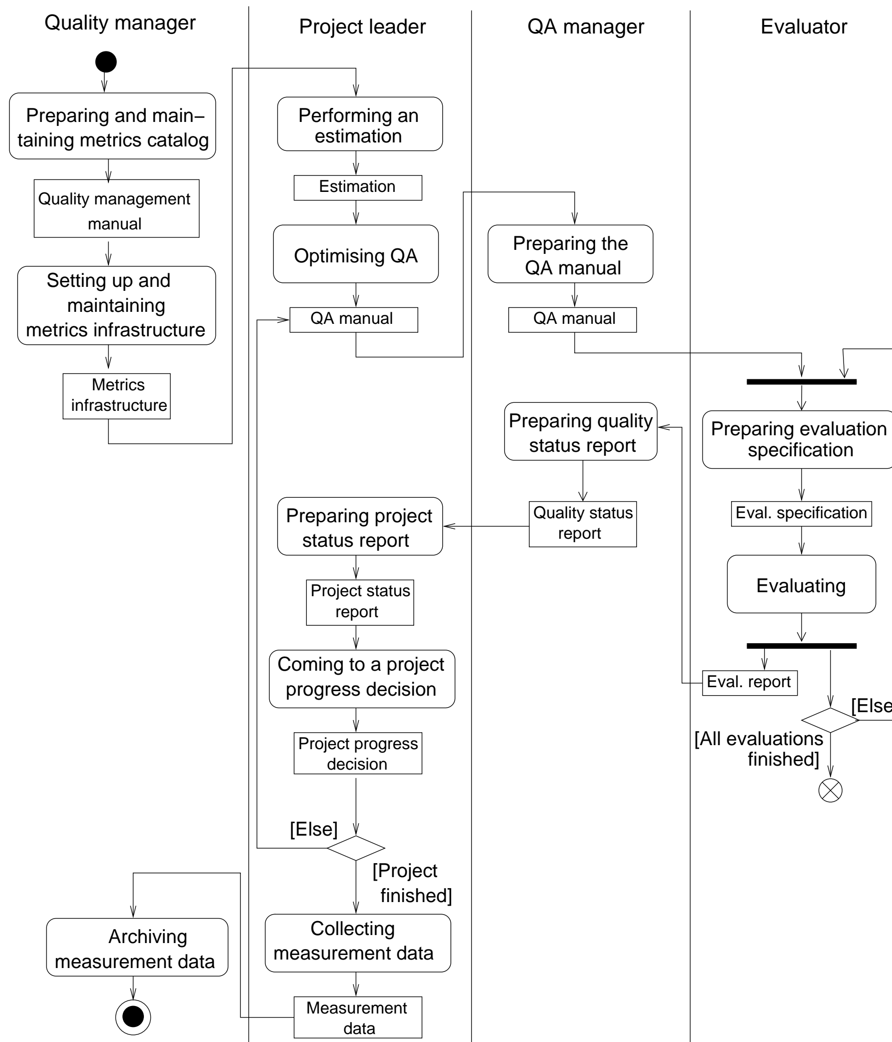


Figure 4.9: An activity diagram of the model application

Extensions and Changes

In the V-Modell XT there are four roles that are affected by the usage of the economics model:

- The *quality manager* is responsible for the quality assurance standards over all projects and for an efficient and effective quality management system. In particular, he develops a systematic quality management and creates and maintains the quality management manual. Most importantly in our context, he defines rules and approaches how projects plan and perform quality assurance techniques. Furthermore, he defines which QA techniques should be used in general and helps in choosing appropriate techniques for a specific project. The main change is that he is responsible for setting up and maintaining the metrics infrastructure.

- The *project leader* has the responsibility for the execution of the project. He

plans and controls the project progress. In particular, he makes the basic estimates for project planning and decides on future changes based on status reports. The main change for this role is that he uses the model for optimising the resource distribution for quality assurance and also collects the necessary measurement data for our model.

- The *QA manager* controls the quality in a project and thereby supervises all quality assurance. He is responsible for the quality status reports and also plans the QA work in collaboration with others. There is only the small change that in his quality status report the necessary measurements for the model must be contained.

- The *evaluator* – also called *inspector* although he not only uses inspections – creates evaluation specifications and using those evaluates the artefacts created in the project. Hence, he uses defect-detection techniques, e.g., reviews and tests, on those artifacts and reports the results. Also for the evaluator it is necessary that he documents the necessary measurements for the model factors.

Several activities and products need slight changes for the use of our model in the V-Modell. Mainly this is only more detail or the explicit measurement and collections of the data for the model input factors.

- The activity *preparing, introducing, and maintaining an organisation-specific process model* contains the sub-activity *preparing and maintaining metrics catalogue*. We assign this activity to the quality manager and explicitly require the incorporation of the factors from Sec. 4.4.1.

- The product *estimation* needs to have the sub-product *estimation of the defect content* that is used later in the model.

- The products *evaluation report* and *quality status report* must contain the necessary measurement data for the factors of Sec. 4.4.1.

- The activity *coming to a project progress decision* can use our economics model as basis for the decision. Different scenarios can be analysed and an optimal effort (or resource) distribution can be calculated.

- When *collecting measurement data* the project leader uses the quality status reports from the QA manager to extract the data that is to be stored for future project in the metrics infrastructure.

Finally, we also extend the V-Modell by several products and activities. They are mainly the activities of the quality manager that are not part of the V-Modell reference.

- The main new activity is *optimise QA*. This activity is performed by the project leader based on his estimates and data from similar projects. He calibrates the model and optimises it (w.r.t. cost or ROI) so that an optimal resource distribution is found. This is then documented in the QA manual.

- The *quality management manual* is mentioned in the V-Modell but it is not an official part of it. We find it useful in the context of our model because we can

organisation-widely define the metrics that need to be collected for the usage of the model.

- The *metrics infrastructure* is in essence similar to the project management infrastructure but not project-specific. In our context it needs to store the measured data for the relevant metrics of our model.
- The quality manager then is responsible for *setting up and maintaining the metrics infrastructure*. This means that a data repository needs to be available for the measurement data.
- Finally, the activity *archiving measurement data* describes that the quality manager stores the measurement data of all projects so that they are available for new projects.

In summary, we find that our model blends well with the V-Modell XT. The necessary activities to use the model fit on the existing roles and only three activities and two products need to be changed. Furthermore, we need two new activities and three new products to be able to embedded our model in the V-Modell. An embedding in other process models with a similar structuring should be possible with a comparable effort.

4.5 Example

This section shows in an example the usefulness of the model. In particular, we use the practical model with its optimisation procedure to improve a specific quality assurance plan of a project. The data we use does not stem from a real project although we based them on the empirical analyses from Sec. 4.2.5 and 4.3.3.

4.5.1 Basic Setting

For the example, we assume that we are responsible for the quality assurance of a software project with an expected size of 100 kLOC of C code. Based on historical data and experience we expect about 100 faults in that code – hence, a defect density of about 1 fault per kLOC. We use the average defect type distribution from the empirical knowledge and assume that we also have average data for the different attributes of these defect types such as their removal effort or the difficulty to detect them with different techniques. For all data that is not available, we use industry averages as given in Sec. 4.2.5 and 4.3.3, for example the distribution of defect types.

For the sake of simplicity, we consider only four different techniques for the quality assurance of the project:

- Code inspection
- Unit test
- Integration test
- System test

To simplify the example, we also do not look at different sequences of the technique application or several applications of the same technique at different points in time.

4.5.2 Manual Planing

First we want to manually try several different effort distributions on these settings to see how the model behaves. All the distributions are shown in Tab. 4.17. The effort is given in person-hours there. If we consider again the empirical averages from Sec. 4.2.5 we can come up with an effort distribution over the four different techniques which is called distribution 1. Distribution 2 reflects the setting in which the company decides to concentrate on testing and do no inspection. The effort is shifted to the testing techniques. With the distribution 3 we analyse a low effort on analytical SQA, i.e., there is not enough time allocated in the project for quality assurance. Finally, distribution 4 shows what happens when most of the quality assurance is done at the end during system testing. In contrast to distribution 2 nearly no unit and integration test is done.

Table 4.17: The manual effort distribution

Dist.	Inspection	Unit test	Integration test	System test	Profit	ROI
1	650	300	550	550	74,467	0.12
2	0	500	750	750	54,254	0.08
3	100	100	100	100	-111,410	-0.31
4	50	50	50	2000	-33,035	-0.04

Tab. 4.17 already contains the resulting return on investment of the different quality assurance plans. Distribution 1 exhibits a reasonable ROI of 0.12. Intuitively, if we follow this QA plan, we will have a 12% return on the invested capital. This is reasonable compared to current interest rates. Distribution 2 performs weaker with a ROI of 8%. Hence, we can conclude that in this setting it is beneficial to perform a code inspection.

Distribution 3 exhibits the worst ROI with -0.31, i.e., we lose nearly a third of the invested capital if we do not assign enough effort to the quality assurance. Finally, a large system test is also not beneficial with a negative ROI of 4%. However, considering the variance in the estimates this can still be seen as acceptable. We do not gain anything based on the quality but we also do not lose as much capital here. The loss is still over 30,000.

4.5.3 Optimised Planing

Having analysed several manual QA plans in the previous section, we present an cost-optimised (or profit-optimised) solution for the setting described above. The procedure for optimisation with the model is described in Sec. 4.3.5. There are various ways to optimise the result of the model using different algorithms and implementations. We choose a very simple optimisation here that does not guarantee a global optimum but at least allows us to show that the results from the manual plans can be improved.

Table 4.18: An optimal effort distribution

Inspection	Unit test	Integration test	System test	Profit	ROI
800	250	250	200	153,674	0.28

An optimised QA plan is shown in Tab. 4.18. We see that we are close to the distribution 1 from above which is the distribution derived from industry averages. However, note that the effort for the inspection is increased while the testing effort decreased. This is in accordance to the fact that defect removal is more expensive in later phases and hence early defect detection is beneficial. A graphical overview of all effort distributions is shown in Fig. 4.10.

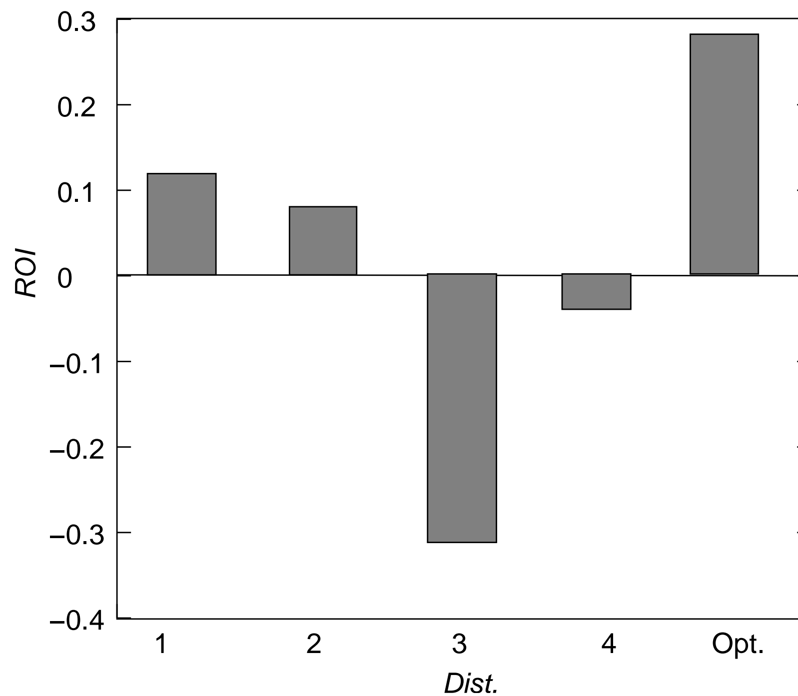


Figure 4.10: A graphical comparison of the ROIs of the effort distributions

While doing the optimisation, we noted again that the removal effort in the field, the difficulty, and the removal effort of the techniques have a strong influence on the ROI. Obviously, there are several thresholds where the use of a specific technique stops being beneficial. This is in strong correlation with the clustering of defects in defect types. We use a considerably coarse-grained structuring with only six defect types. It still needs to be investigated if they are enough to reflect the diversity of the different techniques in the sense of [120]. Only if this is the case, we can be sure that the simplification of using defect types instead of single defects does not destroy the predictions.

4.6 Summary

We proposed in this chapter an analytical, stochastic model for the quality economics or costs and benefits of analytical quality assurance. We defined two versions of that model: (1) an “ideal” model that incorporates many important factors and analyses single defects and (2) a practical model that uses defect types and aims at applications in the planing of real software projects. We reviewed the available empirical work on the input factors of the model and synthesised the data to make it usable in the models. Based on this data we performed sensitivity analysis on both models (1) to identify those factors that are most rewarding to be investigated in more detail and (2) to simply the model by finding the factors that can be fixed without changing the output significantly. Finally, we described issues of a practical application of the model and described an exemplary embedding of the model usage in the quality assurance process of the V-Modell XT.

The basic cost factors used in our model are similar to existing approaches. They are determined by the PAF model and the refinement is largely similar to the cost classification of Jones [94]. Many software quality cost models are also based on the PAF model but do not refine the cost factors [66,83,106,126]. Furthermore, our model is generally more detailed than those models because it includes also technical factors of the quality assurance process. Only iDAVE [21] and ROSQ [193] include similar technical influences but as very coarse-grained factors. Hence, our model allows to be more precise to this end and to make use of the available insights of analytical models of defect-detection techniques.

Most analytical models, however, concentrate on specific defect-detection techniques, such as inspections [12, 63, 112] or tests [58, 136, 156, 219]. The difficulty functions in our model are derived from the work of Littlewood et al. [120]. These functions allow to express defect detection capabilities and the diversity of different techniques. However, the incorporation of this factor into an analytical model and the combination with cost factors is unique to our model. The model of Collofello and Woodfield [43] is close to our model. However, we differentiate the factors in more detail. They do not consider different defect types apart from design and code defects and hence have also rather coarse-grained average values for the effectiveness and costs. Moreover, they only consider personnel effort for the costs which constitute a major part but other costs should not be neglected and are hence incorporated in our model.

We believe that our model is a reasonable combination of the high-level cost models and the detailed and technical analytical models. Thus we are able to make more precise and technically sound analyses and predictions of the economics of defect-detection techniques. To use expected values as the basis of our model is firstly a common approach in engineering economics [17]. Secondly, it allows the equations to be rather simple using sums and products. This improves the understanding of the equations and thereby of the relationships of the factors.

In addition, we proposed a new concept of defect propagation that differs from the existing defect introduction and removal models w.r.t. the consideration of different artefact types. This is necessary for our model to be able to incorporate defect-detection techniques that operate on requirements and design documents such as inspections.

We also contributed a large review of the empirical results that are available for defect-detection techniques. Comparable is only the study by Briand et al. [29] that is also based on a model but with partly different factors and a different aim. Our literature review allows to use (1) the found mean values as substitute for some of the factors and (2) the distributions as basis of a sensitivity analysis. This kind of sensitivity analysis is also unique to this class of models. Freimut et al. [63] also did a sensitivity analysis but only on the basis of the data of a case study. Our global sensitivity analysis identified the factors that are most beneficial to investigate in future studies and the results also help to increase the predictive validity of the model itself.

5 Defect-Proneness of Components

In the discussion on how to cost-optimise the general usage of defect-detection techniques and their corresponding effort distribution, we always assumed the software system to be analysed as a black-box. However, during the development more information is available on the structure and behaviour of the system. This information can be exploited to further optimise the SQA. In this case we propose to identify the fault- and failure-prone components of a system early in the development and concentrate the defect-detection on these components to reach a higher efficiency. For this, we define a metrics suite for software models based on UML 2.0. The results were also published in [100, 211]

5.1 General

We already argued that quality assurance constitutes a significant part of the total development costs for software. Especially formal verification is frequently perceived as rather costly. Therefore, there is a possibility for optimising costs by concentrating on the fault-prone components and thereby exploiting the existing resources as efficiently as possible. This is especially important when considering that studies showed that only a small number of components (or modules) of a system contain most of the defects [20]. Detailed design models offer the possibility to analyse the system early in the development life-cycle. One of the possibilities is to measure the complexity of the models to predict fault-proneness assuming that a high complexity leads to a high number of defects (as is done for example in [118]).

The complexity of software code has been studied to a large extent. With *complexity* we mean in this context mainly the complexity or difficulty to implement and test a software, to a lesser extent also the complexity to understand the model because this is related to the quality of the implementation in code. It is often stated that complexity is related to and a good indicator for the fault-proneness of software [105, 139, 163, 177]. There are two different approaches to the identification of fault-prone components. In the *estimative approach* models are used to predict the number of faults that are contained in each component. The *classification approach* categorises components into fault-prone classes, often simply low-fault and high-fault. We use the latter approach in the following because it is more suitable for the model metrics. The reason lies in the conceptual distance between the models and the defects in code. There are several steps involved in transforming the models into the final code. Hence, we believe that a quantitative statement cannot be reasonably made but a qualitative, coarse-grained classification is possible.

Although the traditional complexity metrics are not directly applicable to design models because of different means of structuring and abstractions, there are already a number of approaches that propose design metrics, e.g. [18, 35, 39, 215]. Most of

the metrics in [39] were found to be good estimators of fault-prone classes by Basili, Briand, and Melo [7] and are used in our approach as well. However, they concentrate mainly on the structure of the designs. Since the system structure is not sufficient as a source for the complexity of its components, which largely depends on their behaviour, we will also propose a metric for behavioural models.

5.2 Metrics

This section describes the possibilities to identify fault-prone components based on models built with UML 2.0 [150]. We introduce a design complexity metrics suite for a subset of model elements of the UML 2.0 and explain how to identify fault-prone components. This means we propose a kind of measurement-driven predictive model (cf. Sec. 2.4).

The basis of our metrics suite forms the suite from [39] for object-oriented code and the cyclomatic metric from [129]. In using a suite of metrics we follow [56, 131] stating that a single measure is usually inappropriate to measure complexity.

In [74] the correlation of metrics of design specifications and code metrics was analysed. One of the main results was that the code metrics such as the cyclomatic complexity are strongly dependent on the level of refinement of the specification, i.e. the metric has a lower value the more abstract the specification is. Models of software can be based on various different abstractions, such as functional or temporal abstractions [165]. Depending on the abstractions chosen for the model, various aspects may be omitted, which may have an effect on the metrics. Therefore, it is prudent to consider a suite of metrics rather than a single metric when measuring design complexity to assess fault-proneness of system components.

5.2.1 Development Process

The metric suite described below is generally applicable in all kinds of development processes. It does not need specific phases or sequences of phases to work. However, we need detailed design models of the software to which we apply the metrics. This is most rewarding in the early phases as the models then can serve various purposes.

We adjust metrics to parts of UML 2.0 based on the design approach taken in AUTOFOCUS [82], ROOM [188], or UML-RT [189], respectively. This means that we model the architecture of the software with structured classes (called actors in ROOM, capsules in UML-RT) that are connected by ports and connectors and which have associated state machines that describe their behaviour.

The metrics defined in this section are applicable to components as well as classes. However, we will concentrate on structured classes following the usage of classes in ROOM. The particular usage should nevertheless be determined by the actual development process.

5.2.2 Measures of the Static Structure

We start introducing the new measures with the ones that analyse the static structure of models. These are important because the interrelations and dependencies among model elements contribute significantly to their complexity.

Structured Classes

The concept of structured classes introduces composite structures that represent a composition of run-time instances collaborating over communication links. This allows UML classes to have an internal structure consisting of other classes that are bound by connectors. Furthermore, ports are used as a defined entry point to a class. A port can group various interfaces that are provided or required. A connection between two classes through ports can also be denoted by a connector. The parts of a class work together to achieve its behaviour. A state machine can also be defined to describe behaviour that is additional to the behaviour provided by the parts.

We start with three metrics, Number of Parts, Number of Required Interfaces, and Number of Provided Interfaces, which concern structural aspects of a system model. The metrics consider composite structure diagrams of single classes with their parts, interfaces, connectors, and possibly state machines. A corresponding example is given in Fig. 5.1.

Number of Parts (NOP). The number of parts of a structured class contributes obviously to its structural complexity. The more parts it has, the more coordination is necessary and the more dependencies there are, all of which may contribute to a fault. Therefore, we define *NOP* as the number of direct parts C_p of a class.

Number of Required Interfaces (NRI). This metric is (together with the NPI metric below) a substitute for the old *Coupling Between Objects (CBO)* that was criticised in [127] in that it does not represent the concept of coupling appropriately. It reduces ambiguity by giving a clear direction of the coupling. We use the required interfaces of a class to represent the usage of other classes. This is another increase of complexity which may as well lead to a fault, for example if the interfaces are not correctly defined. Therefore, we count the number of required interfaces I_r for this metric. Coupling metric as predictors of run-time failures were investigated in [16]. It shows that coupling metrics are suitable predictors of failures.

Number of Provided Interfaces (NPI). Very similar but not as important as NRI is the number of provided interfaces I_p. This is similarly a structural complexity measure that expresses the usage of a class by other entities in the system.

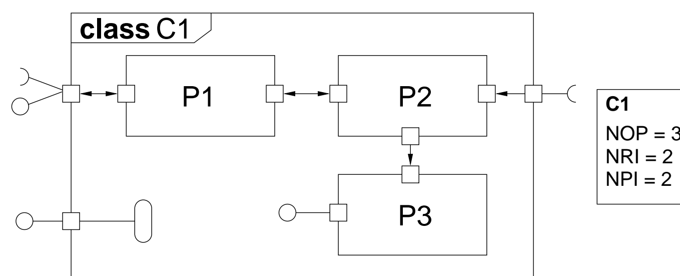


Figure 5.1: An example structured class with three parts and the corresponding metrics.

Example

The example in Figure 5.1 shows the composite structure diagram of a class with three ports, two required and two provided interfaces. It has three parts which have in turn ports, interfaces and connectors. However, these connecting elements are not counted in the metrics for the class itself because they are counted by the metrics for the parts, and these can later be summed up to consider the complexity of a class including its parts.

5.2.3 Measure of Behaviour

We proceed with a complexity metric for behavioural models because the behaviour determines the complexity of a component to a large extent.

State Machines

State machines with input and output – as defined in the UML 2.0 standard – can be used to model the behaviour of classes of a system. They describe the actions and state changes based on a partitioning of the state space of the class. Therefore, the associated state machine is also an indicator of the complexity of a class and hence its fault-proneness. State machines consist of states and transitions where states can be hierarchical. Transitions carry event triggers, guard conditions, and actions.

We use cyclomatic complexity [129] to measure the complexity of behavioural models represented as state machines because it fits most naturally to these models as well as to code. This makes the lifting of the concepts from code to model straightforward.

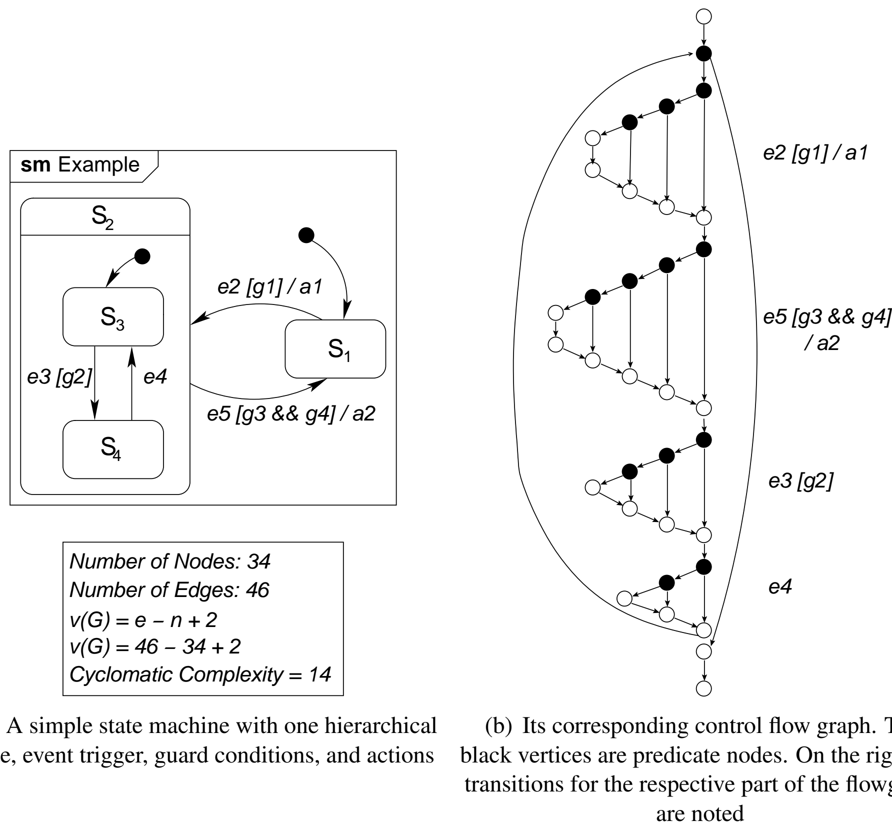
To find the cyclomatic complexity of a state machine we build a control flow graph similar to the one for a program in [129]. This is a digraph that represents the flow of control in a piece of software. For source code, a vertex is added for each statement in the program and arcs if there is a change in control, e.g. an if- or while-statement. This can be adjusted to state machines by considering the code implementation. The code transformation that we use as a basis for the metrics can be found in [188]. However, different implementation strategies could be used [158].

Example

An example of a state machine and its control flow graph is depicted in Fig. 5.2(a) and Fig. 5.2(b), respectively. At first we need an entry point as the first vertex. The second vertex starts the loop over the automata because we need to loop until the final state is reached or infinitely if there is no final state. The next vertices represent transitions, atomic expressions[1] of guard conditions, and event triggers of transitions. These vertices have two outgoing arcs each because of the two possibilities of the control flow, i.e. an evaluation to *true* or *false*. Such a branching flow is always joined in an additional vertex. The last vertex goes back to the loop vertex from the start and the loop vertex has an additional arc to one vertex at the end that represents the end of the loop. This vertex finally has an arc to the last vertex, the exit point.

[1] A guard condition can consist of several boolean expressions that are connected by conjunctions and disjunctions. An atomic expression is an expression only using other logical operators such as equivalence. For a more thorough definition see [129].

If we have such a graph we can calculate the cyclomatic complexity using the formula $v(G) = e - n + 2$, where v is the complexity, G the control graph, e the number of arcs, and n the number of vertices (nodes). There is also an alternative formula, $v(G) = p + 1$, which can also be used, where p is the number of binary *predicate nodes*. Predicate nodes are vertices where the flow of control branches.



(a) A simple state machine with one hierarchical state, event trigger, guard conditions, and actions

(b) Its corresponding control flow graph. The black vertices are predicate nodes. On the right the transitions for the respective part of the flowgraph are noted

Figure 5.2: An example of the cyclomatic complexity of a state machine

Hierarchical states in state machines are not incorporated in the metric. Therefore, the state machine must be transformed into an equivalent state machine with simple states. This appears to be preferable to handling hierarchy separately because we are not looking at understandability and we do not have to deal with hierarchy crossing transitions. Furthermore internal transitions are counted equally to normal transitions. Pseudo states are not counted themselves, but their triggers and guard conditions. Usage of the *InState* construct in guards is not considered.

Cyclomatic Complexity of State machine (CCS). Having explained the concepts based on the example flow graph above, the metric can be calculated directly from the state machine with a simplified complexity calculation. We count the atomic expressions and event triggers for each transition. Furthermore we need to add 1 for each transition because we have the implicit condition that the corresponding source

state is active. This results in the formula

$$CCS = |T| + |E| + |A_G| + 2. \tag{5.1}$$

where T is the multi-set of transitions, E is the multi-set of event triggers, and A_G is the multi-set of atomic expressions in the guard conditions. This formula yields exactly the same results as the longer version above but has the advantage that it is easier to calculate.

For this metric we have to consider two abstraction layers. First, we transform the state machine into its code representation[2] and second use the control flow graph of the code representation to measure structural complexity. The first "abstraction" is needed to establish the relationship to the corresponding code complexity because it is a good indicator of the fault-proneness of a program. The proposition is that the state machine reflects the major complexity attributes of the code that implements it. The second abstraction to the control flow graph was established in [129] and is needed for the determination of paths through the program which reflect the complexity of the behaviour.

5.2.4 Metrics Suite

In addition to the metrics which we defined above, we complete our metrics suite by adding two existing metrics from [39] that can be adjusted to be applicable to UML models. The metrics chosen are from the ones that were found to be good indicators of fault-prone classes in [7]. We omit *Response For a Class (RFC)* and *Coupling Between Objects (CBO)*[3] because they cannot be determined on the model level. The two adapted metrics are described in the following. The complete metrics suite can be found in Tab. 5.1. Note that all proposed metrics have an ordinal scale [54].

Depth of Inheritance Tree (DIT). This is the maximum depth of the inheritance graph T to a class c. This can be determined in any class diagram that includes inheritance.

Number of Children (NOC). This is the number of direct descendants C_d in the inheritance graph. This can again be counted in a class diagram.

5.2.5 Properties

Since there is a vast amount of so-called *complexity* metrics for software, several researchers developed more precise definitions for that term. We analyse whether our metrics are *structural complexity measures* by the definition in [131]. The definition says that for a set D of documents with a pre-order $\leq_D$ and the usual ordering $\leq_{\mathbb{R}}$ on the real numbers $\mathbb{R}$, a structural complexity measure is an order preserving function $m : (D, \leq_D) \longrightarrow (\mathbb{R}, \leq_{\mathbb{R}})$. This means that any structural complexity metric needs

[2] Note that this is done only for measuring purposes; our approach also applies if the actual implementation is not automatically generated from the UML model but manually implemented.

[3] RFC counts all methods of a class and all methods recursively called by the methods. CBO counts all references of a class to methods or fields of other classes.

Table 5.1: A summary of the metrics suite with its calculation

Name	Abbr.	Calculation
Depth of Inheritance Tree	DIT	$max(depth(T, c))$
Number of Children	NOC	$\|C_d\|$
Number of Parts	NOP	$\|C_p\|$
Number of Required Interfaces	NRI	$\|I_r\|$
Number of Provided Interfaces	NPI	$\|I_p\|$
Cyclomatic Complexity of State machine	CCS	$\|T\| + \|E\| + \|A_G\| + 2$

to be at least pre-ordered because this is necessary for comparing different documents. Each metric from the suite fulfils this definition with respect to a suitable pre-order on the relevant set of documents.

The document set D under consideration is depending on the metric: either a class diagram that shows inheritance and possibly interfaces, a composite structure diagram showing parts and possibly interfaces, or a state machine diagram. All the metrics use specific model elements in these diagrams as a measure. Therefore, there is a pre-order $\leq_D$ between the documents of each type based on the metrics: We define $d_1 \leq_D d_2$ for two diagrams d_1, d_2 in D if d_1 has fewer of the model elements specific to the metric under consideration than d_2. The mapping function m maps a diagram to its metric, which is the number of these elements. Hence m is order preserving and the metrics in the suite qualify as structural complexity measures.

Similarly, Weyuker [217] defined several properties that should be possessed by complexity metrics. These are partly more specific to complexity metrics based on the syntactic representation in a programming language. Hence, a transfer to model metrics is not obvious. Nevertheless, an interpretation for models shows that we are able to fulfil all nine properties. The only assumption we have to make is that concatenation of models may involve adding additional model elements to connect them.

5.3 Fault Proneness

As mentioned before, complexity metrics are good predictors for the reliability of components [105, 139]. Furthermore, the experiments in [7] show that most metrics from [39] are good estimators of fault-proneness. We adopted DIT and NOC from these metrics unchanged, therefore this relationship still holds. The cyclomatic complexity is also a good indicator for reliability [105] and this concept is used for CCS to be able to keep this relationship. The remaining three metrics were modelled similarly to existing metrics. NOP resembles NOC, NRI and NPI are similar to CBO. NOC and CBO are estimators for fault-proneness, therefore it is expected that the new metrics behave accordingly.

The metrics suite is used to determine the most fault-prone classes in a system. Different metrics are important for different components. Therefore, one cannot just take the sum over all metrics to find the most critical component. We propose to use the metrics so that we compute the metric values for each component and class and consider the ones that have the highest measures for each single metric. This way we can for example determine the components with complex behaviour or coupling.

We suggest to use *complexity levels* $L_C = \{high, low\}$. We assign each component such a complexity level by looking at the extreme values in the metrics results. Each component that exhibits a high value in at least one of the metrics is considered of having the complexity level *high*, all other components have the level *low*. It depends on the actual distribution of values to determine what is to be considered a high value. These complexity levels show the high-fault and low-fault components.

5.4 Failure Proneness

The following constitutes an extension to the analysis of fault proneness towards failure proneness. The fault-proneness of a component does not directly imply low reliability because a high number of faults does not mean that there is a high number of failures [201]. However, a direct reliability measurement is in general not possible on the model level. Nevertheless, we can get close by analysing the failure-proneness of a component, i.e. the probability that a fault leads to a failure that occurs during software execution.

It is not possible to express the probability of failures with exact figures based on the design models. We propose therefore to use more coarse-grained *failure levels*, e.g. $L_F = \{high, medium, low\}$, where L_F is the set of failure levels. This allows an abstract assessment of the failure probability. It is still not reliability as generally defined but the best estimate that we can get in early phases.

To determine the failure level of a component we use the complexity levels from above. Having assigned these complexity levels to the components, we know which components are highly fault-prone. The operational profile [140] is a description of the usage of the system, showing which functions are mostly used. We use this information to assign *usage levels* L_U to the components. This can be of various granularity. An example would be $L_U = \{high, medium, low\}$. When we know the usage of each component we can analyse the probability that the faults in the component lead to a failure.

The combination of complexity level and usage level leads us to the *failure level* L_F of the component. It expresses the probability that the component fails during software execution. We describe the mapping of the complexity level and usage level to the failure level with the function *fp*:

$$fp = L_C \times L_U \longrightarrow L_F, \text{where } L_F = L_U \cup \{low\} \tag{5.2}$$

What the function does is simply to map all components with a high complexity level to its usage level and all components with a low complexity level to *low*. However, this is only one possibility how *fp* can look like.

$$fp(x, y) = \begin{cases} y & \text{if } x = high \\ low & \text{otherwise} \end{cases} \tag{5.3}$$

This means that a component with high fault-proneness has a failure probability that depends on its usage and a component with low fault-proneness has generally a low failure probability.

Having these failure levels for each component we can use that information to guide the verification efforts in the project, e.g., assign the most amount of inspection and

testing on the components with a high failure level. Parts of critical systems such as an exception handler still need thorough testing although its failure level might be low. However, this is not part of this work.

5.5 Summary

In terms of abstraction, this chapter is on a more detailed level than the analytical model of Chap. 4. The aim is still to cost-optimally use analytical quality assurance but we do not distribute the effort between different techniques but we analyse how the effort is best distributed over the components of the system. This is done by identifying the most fault- and failure-prone components based on a metrics suite and detailed design models. In other words, the analytical model and the metrics suite are orthogonal: the model is used to concentrate the QS on defect-detection techniques and the metrics are used to concentrate on specific software components.

Based on the assumption that more complex components are more defect-prone we defined – partly based on prior empirical analyses – several metrics that combined are able to classify components using UML 2.0 models. These will be validated by two case studies in the following Chap. 6. The main difference to existing similar approaches is that we analyse models and hence the proposed approach can be applied early in the development process and hence the concentration on defect-prone components can be planed from early on. Furthermore, we also defined a metric for behaviour models – UML state machines – which is missing in similar metrics suite so far.

However, we have also to note that the relationship between complexity metrics and faults and failures is controversial discussed and this relationship is not always convincing in empirical studies. In particular, in a recent study at Microsoft [145] it was shown that there is always a set of metrics that is a good predictor for field failures – failure-proneness in some sense – but the members of the set differ from project to project. Nevertheless, for similar projects these sets tend to be similar too.

6 Case Studies

We describe four different case studies in the following chapter. They are used to analyse the predictive validity of the metrics suite from Chap. 5 and to derive metrics that can be used to calibrate the model from Chap. 4. In particular, we analyse the average difficulty of model-based testing and bug finding tools for which no empirical data has been available so far. Furthermore, we analyse the failure probability of faults in more detail. In addition, the case studies contribute to the empirical body of knowledge on defect detection.

6.1 Automatic Collision Notification

The first case study we use to validate our proposed defect-proneness analysis (cf. Chap. 5) is an automatic collision notification system as used in cars to provide automatic emergency calls. First, the system is described and designed using UML, then we analyse the model and present the results.

6.1.1 Description

The case study stems from the automotive domain, in particular from the Verisoft project[1] that aims at formally verifying all layers in a software system. Results from that are published in [26]. However, we take a different view here and try to analyse the defect-proneness of the system. The problem to be solved is that many accidents of automobiles only involve a single vehicle. Therefore it is possible that no or only a delayed emergency call is made. The chances for successful help for the casualties are significantly higher if an accurate call is made quickly. This has lead to the development of so called *Automatic Collision Notification (ACN)* systems, sometimes also called *mayday* systems. They automatically notify an emergency call response centre when a crash occurs. In addition, manual notification using the location data from a GPS device can be made. We use the public specification from the *Enterprise* program [196, 197] as a basis for the design model. Details of the implementation technology are not available. In this case study, we concentrate on the built-in device of the car and ignore the obviously necessary infrastructure such as the call centre.

6.1.2 Device Design

Following [196] we call the built-in device *MaydayDevice* and divide it into five components. The architecture is illustrated in Fig. 6.1 using a composite structure diagram of the device.

The device is a processing unit that is built into the vehicle and has the ability to communicate with an emergency call centre using a mobile telephone connection and

[1] `http://www.verisoft.de/`

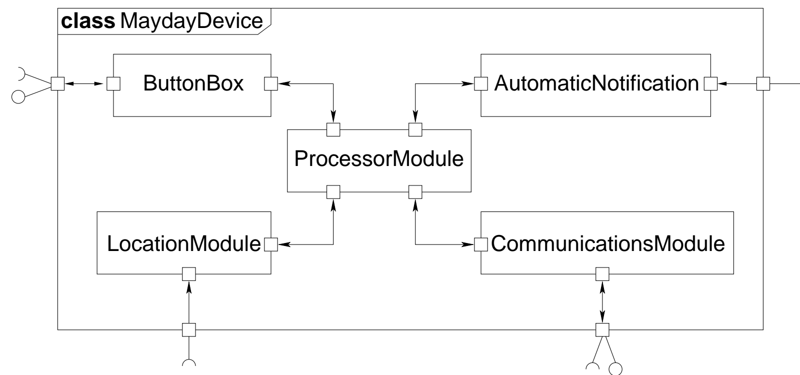


Figure 6.1: The composite structure diagram of the mayday device

retrieving position data using a GPS device. The components that constitute the mayday device are:

- *ProcessorModule*: This is the central component of the device. It controls the other components, retrieves data from them and stores it if necessary.
- *AutomaticNotification*: This component is responsible for notifying a serious crash to the processor module. It gets notified itself if an airbag is activated.
- *LocationModule*: The processor module requests the current position data from the location module that gathers the data from a GPS device.
- *CommunicationsModule*: The communications module is called from the processor module to send the location information to an emergency call centre. It uses a mobile communications device and is responsible for automatic retry if a connection fails.
- *ButtonBox*: This is finally the user interface that can be used to manually initiate an emergency call. It also controls a display that provides feedback to the user.

Each of the components of the mayday device has an associated state machine to describe its behaviour. We do not show all of the state machines but explain the two most interesting ones in more detail. This is, firstly, the state machine of the *ProcessorModule* called *Processor* in Fig. 6.2. It has three control states: *idle*, *retrieving*, and *calling*. The *idle* state is also the initial state. On request of an emergency call, either by *startCall* from the *ButtonBox* or *notify* from the *AutomaticNotification*, it changes to the *retrieving* state. This means that it waits for the GPS data. Having received this data, the state changes to *calling* because the *CommunicationsModule* is invoked to make the call. In case of success, it returns to the *idle* state and lights the green LED on the *ButtonBox*. Furthermore, the state machine can handle cancel requests and make a test call.

The second state machine is *Communications* in Fig. 6.3, the behaviour specification of *CommunicationsModule*. One of the main complicating factors here is the handling of four automatic retries until a failure is reported. The state machine starts in an *idle* state and changes to the *calling* state after the invocation of *makeCall*. The *offHook*

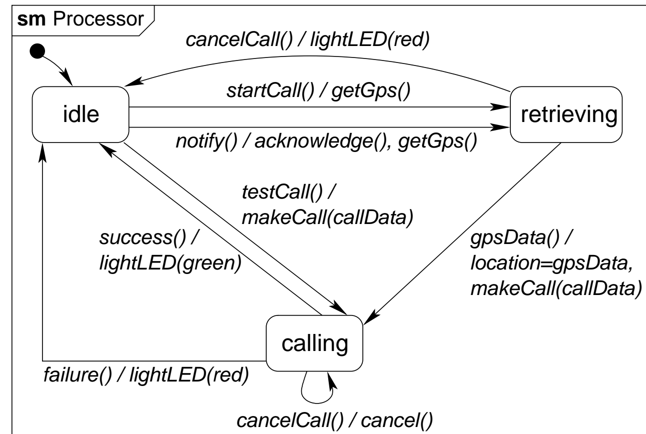


Figure 6.2: The state machine diagram of the *ProcessorModule*

signal is sent to the mobile communications device. Inside the *calling* state, we start in the state *opening line*. If the line is free, the *dialling* state is reached by dialling the emergency number. After the *connected* signal is received, the state is changed to *sending data* and the emergency data is sent. After all data is sent, the *finished* flag is set which leads to the *data sent* state after the *onHook* signal was sent to the mobile. After the mobile sends the *done* signal, the state machine reports *success* and returns to the idle state. In case of problems, the state is changed to *opening line* and the retries counter is incremented. After four retries the guard $[retries\ >= 5]$ evaluates to *true* and the call fails. It is also always possible to cancel the call which leads to a *failure* signal as well.

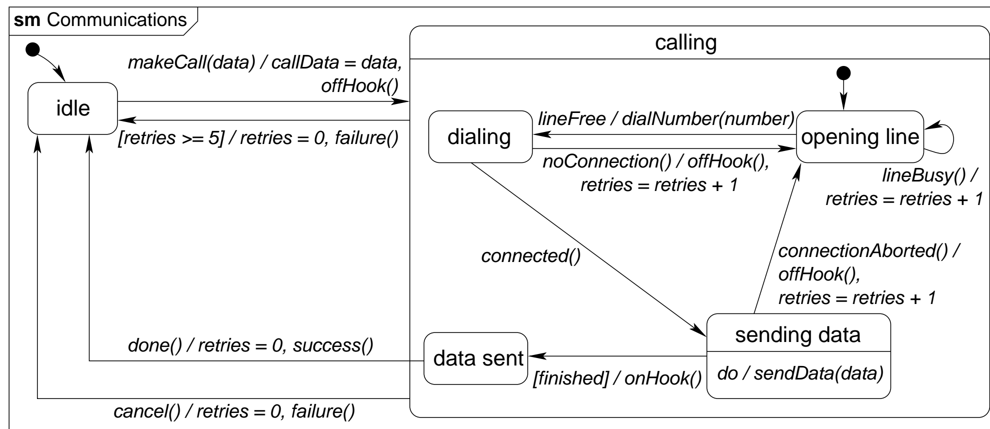


Figure 6.3: The state machine diagram of the *CommunicationsModule*

6.1.3 Results

The components of *MaydayDevice* are further analysed in the following. At first we use our metrics suite from Sec. 5.2 to gather data about the model. The results can be found in Tab. 6.1. It shows that we have no inheritance in the current abstraction

Table 6.1: The results of the metrics suite for the components of *MaydayDevice*.

Class	DIT	NOC	NOP	NRI	NPI	CCS
MaydayDevice	0	0	5	4	2	0
ProcessorModule	0	0	0	4	4	16
AutomaticNotification	0	0	0	2	1	4
LocationModule	0	0	0	1	2	4
CommunicationsModule	0	0	0	2	2	32
ButtonBox	0	0	0	2	2	8

level of our model and also that the considered classes have no parts apart from MaydayDevice itself. Therefore the metrics regarding these aspects are not helpful for this analysis.

More interesting are the metrics for the provided and required interfaces and their associated state machines. The class with the highest values for NRI and NPI is *ProcessorModule*. This shows that it has a high coupling and is therefore fault-prone. The same module has a high value for CCS but *CommunicationsModule* has a higher one and is also fault-prone.

In [197] there are detailed descriptions of acceptance and performance tests with the developed system. The system was tested by 14 volunteers. The usage of the system in the tests was mainly to provoke an emergency call by pressing the button on the button box.

The documentation in [197] shows that the main failures that occurred were failures in connecting to the call centre (even when cellular strength was good), no voice connect to the call centre, inability to clear the system after usage, and failures of the cancel function. These main failures can be attributed to the component *ProcessorModule* that is responsible for controlling the other components and *CommunicationsModule* that is responsible for the wireless communication. Therefore our analysis identified the correct components. The types of the corresponding faults of the failures are not available.

6.1.4 Summary

This case study is a first validation of the metrics suite for identifying defect-prone components. The metrics suite was able to identify the correct components in the ACN system. Although a single case study is not an empirical exhaustive validation it still is a proof-of-concept and gives first hints on the applicability. We will use the results from this case study in the summary of the the next case study in Sec. 6.2.5 to analyse the metrics and the correlation to the faults in more detail.

6.2 MOST NetworkMaster

The technique of model-based testing gets increasing attention in research and practice. Therefore, we need to be able to incorporate it in quality assurance plans that are based on our economics model from Chap. 4. However, to be able to consider this technique in the model we need more empirical information, especially on its av-

erage difficulty of defect detection. This case study evaluates this input parameter of the economics model for this specific technique as well as validates the approach for the identification of defect-prone components of Chap. 5. It has partly been published in [168, 211].

6.2.1 Approach

This case study aims at analysing effectiveness or difficulty (cf. Chap. 4) of model-based testing. We also use this case study as a validation for our approach to the identification of fault-proneness (cf. Chap. 5) and analyse some other interesting relationships concerning code and model coverage. For the case study, we built a model of a network controller for an automotive infotainment network – the MOST NetworkMaster – to assess an approach of (automated) model-based testing.

In particular, we address the following questions:

1. How do model-based tests compare to "traditional" test without an explicit model in terms of effectiveness (difficulty), coverage, and diversity?

2. How do manual tests – both with and without a model – compare to automatically generated tests?

3. How do random generated tests compare to "directed" tests based on test case specifications?

4. Can a suite of model metrics predict the most fault-prone components?

Furthermore, we analyse the relationships between coverage on the model and the implementation. Based on that we also investigate the correlation of coverage and defect detection. These relationships are not directly usable in the analytical model of Chap. 4 but could be a point of further refinement. Moreover, this information is valuable for the empirical body of knowledge of testing in general and model-based testing in particular.

First, we built an executable behaviour model of the network controller based on existing requirements documents. They consisted of the publicly available MOST specification [138] and a set of message sequence charts (MSCs) that specify several scenarios in more detail. The model building itself revealed a number of inconsistencies and omissions in the requirements documents which were updated later on.

The system under test (SUT) was a third-party software simulation of the NetworkMaster running on a PC that was connected to a MOST network. For the comparison we also had an existing test suite that was developed for the NetworkMaster by other test engineers without the use or knowledge of the model.

The test suites were applied to the implementation, failures were observed, analysed, and classified. The classification was done to count the corresponding faults based on *failure classes* (comparable to the *failure regions* of Sec. 2.2). The model itself was not included in the requirements documents. This explains why there are requirements faults at all: the updated specification MSCs did not capture all of the behaviour of the model and hence there were still requirements defects.

AutoFocus

The CASE tool AUTOFOCUS [81], intended for modelling reactive systems, was used to build the model for the case study. It allows to develop graphical specifications of embedded systems based on a simple, formally defined semantics. With the version 2.0 of the UML specification AUTOFOCUS is now similar to the UML. It is comparable to the subset of UML described in Sec. 5.2. AUTOFOCUS supports different views on the system model: structure, behaviour, interaction, and data type views. The core items of AUTOFOCUS specifications are components. A component is an independent computational unit that communicates with its environment via typed ports. Two or more components can be linked by connecting their ports with directed channels. Thereby, networks of components can be built and described by *system structure diagrams* (SSDs). Fig. 6.4 shows an example SSD.

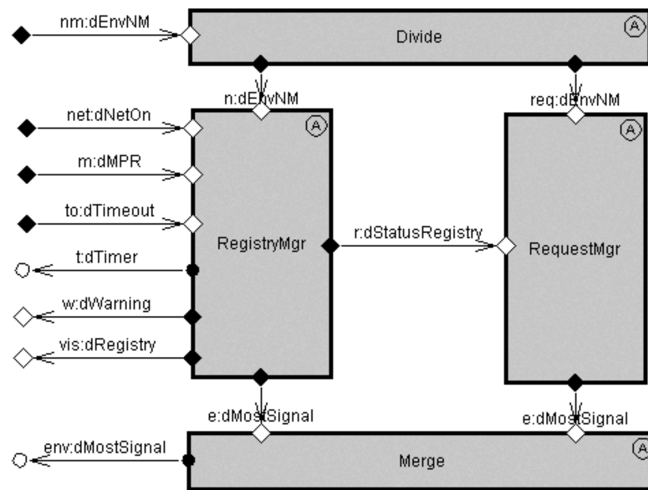


Figure 6.4: SSD of the MOST NetworkMaster

SSDs are hierarchical. Hence, each component can have a set of subcomponents. Atomic components are components which are not further refined. For these components a behaviour must be defined which is achieved by extended finite state machines called *state transition diagrams* (STDs). Fig. 6.5 depicts an example STD. The number at each transition denotes it priority.

An STD consists of a set of control states, transitions, and is associated with local variables. The local variables comprise the data state of the component. Each transition is defined by its source and destination control states, a guard with conditions on the input and the current data state, and an assignment for local variables and output ports. Transitions can fire if the condition on the current data state holds and the actual input matches the input conditions. After execution of the transition, the local variables are set accordingly, and the output ports are bound to the values computed in the output statements. These values are then copied to the input ports that are connected by channels. Guards and assignments are defined in a Gofer-like functional language that allows the definition of possibly recursive data types and functions.

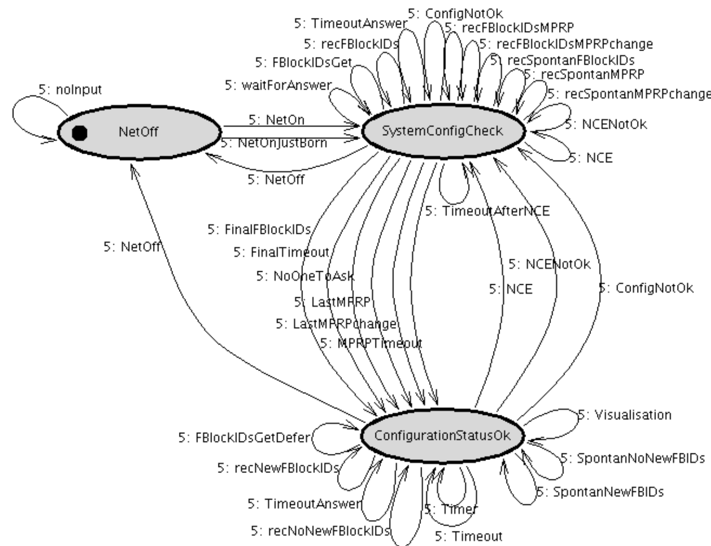


Figure 6.5: The behaviour of the *RegistryManager* described as STD

Components are timed by a common global clock. In particular, they all perform their computations simultaneously. Each clock cycle consists of two steps: (1) each component reads the values on its input ports and computes new values for local variables and output ports. (2) New values are copied to the output ports where they can be accessed immediately via the input ports of connected components. This simple time-synchronous execution and communication semantics of AUTOFOCUS and the use of a functional language as "action" language simplify the automatic test case generation [166].

The method of test case generation used in this case study is described in detail in [166, 169]. In short, the model is translated into a Constraint Logic Programming (CLP) language and test case specification are added. They guide the generation mechanism by identifying the "interesting" cases out of the infinite possibilities. By the execution of the CLP program all traces of the model are enumerated. In fact, the model is executed symbolically: rather than enumerating single traces – input, output, local data of all components – of the model, we work with sets of values in each step instead. States are not visited multiple times which is why in each step, the currently visited set of states is only taken into consideration if it is not a specialisation of a previously visited set of states.

MOST

MOST (Media Oriented Systems Transport) is an infotainment network tailored to the automotive domain. Its public specification [138] is maintained by the MOST cooperation that includes major automotive companies. MOST is a ring topology that supports synchronous and asynchronous communication at up to 24.8 Mbps. Various devices, such as a CD changer or a navigation system, can be connected so that they together provide infotainment services. These services are represented by so-called *function blocks* that are contained in MOST devices. Examples of a function block are

CDPlayer and the special function block *NetBlock*. This function block is available in every device and can be used to get information about the other function blocks. Each function block provides several functions that can be used by other function blocks. For instance, a *CDPlayer* can be asked to *start*, *stop*, etc. All function blocks and functions are addressed by standardised identifiers.

The network exhibits three central master function blocks, one of which is the NetworkMaster, the subject of our study. It is responsible for ensuring consistency of the various function blocks, for providing a lookup service, and for assigning logical addresses.

6.2.2 Model and System

We describe the model and the implementation, i.e., the SUT in the following.

Model of the NetworkMaster

Fig. 6.4 depicts the functional decomposition of the NetworkMaster into AUTOFOCUS components. It contains two components *Divide* and *Merge* that are only responsible for the correct distribution of messages. The *MonitoringMgr* checks the status of devices in the network but has no behaviour in the model, i.e. was functionally abstracted. The *RegistryMgr* is the main component. All devices need to register with it on startup and it manages this registry. Finally, the *RequestMgr* answers requests about the addresses of other devices.

The main complexity of the model lies in the component `RegistryMgr`. The STD that describes its behaviour is shown in Fig. 6.5. We do not provide detailed guards and actions on the transitions to keep the diagram comprehensible. We model the `RegistryMgr` using three control states:

- `NetOff` is the state when the system is switched off.
- The `SystemConfigCheck` state is used when the NetworkMaster checks the network and its slaves.
- The normal network operation is modelled by `ConfigurationStatusOk`.

Including the environment model, the model consists of 17 components with 100 channels and 138 ports, 12 STDs, 16 distinct control states, 16 local variables, and 104 transitions. 34 data types were defined by means of 80 constructors. The number of defined functions used in guards and assignments is 141. The model corresponds to about 12,300 lines of C code, without comments.

Five general abstraction principles were applied in the model. These were especially investigated by Prenninger and Pretschner [164, 165].

1. In terms of *functional abstraction*, we focused on the main functionality of the NetworkMaster, namely setting up and maintaining the registry, and providing the lookup service. We omitted node monitoring which checks from time to time whether or not all nodes in the ring are alive.

2. In terms of *data abstraction*, we reduced data complexity in the model, e.g., by narrowing the set of MOST signals to those which are relevant for the NetworkMaster behaviour, and by building equivalence classes on error codes which the NetworkMaster treats identically.

3. In terms of *communication abstraction*, we merged consecutive signals that concern the same transaction in actual hardware into one signal.

4. In terms of *temporal abstraction*, we abstracted from physical time. For instance, the timeout that indicates expiration of the time interval the NetworkMaster should wait for the response of a node is abstracted by introducing two symbolic events: one for starting the timer, and a nondeterministically occurring one for expiration of the timer. This nondeterministic event is raised outside of the NetworkMaster model which hence remains deterministic.

5. Finally, in terms of *structural abstraction*, the nodes in the environment of the NetworkMaster are not represented as AUTOFOCUS components, but instead by recursive data structures manipulated by one dedicated environment component. This enables us to parameterise the model in order to deal with a variable number of nodes in the network.

To show that the AUTOFOCUS design is simple to transfer to UML, the corresponding composite structure diagram to the SSD in Fig. 6.4 of the NetworkMaster is shown in Figure 6.6. Apart from some differences in notation, the main difference is that the ports in UML have no directions, i.e., there is no distinction between outgoing and ingoing ports. However, we kept the same number of ports in the UML model because we believe that this distinction is useful for a quick comprehension. The STDs of the AUTOFOCUS model are similarly transfered to UML state machines. Note that this does not meant that the semantics of the model stays the same. Nevertheless, for our metrics this is not important. Hence, such a translation is sufficient.

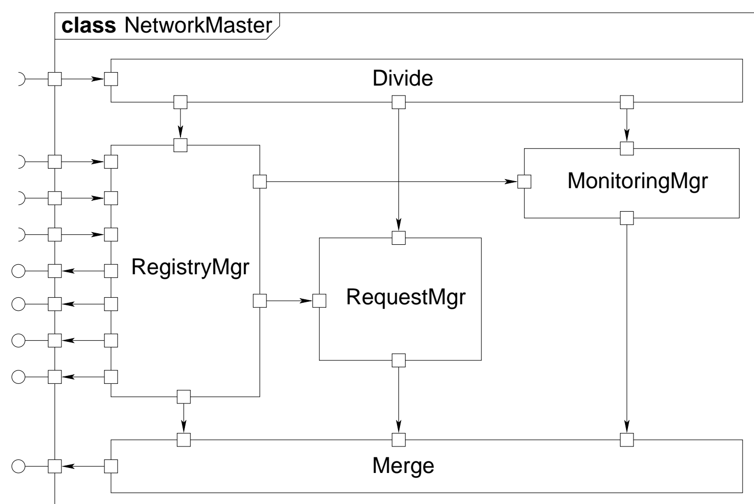


Figure 6.6: The composite structure diagram of the MOST network master.

Implementation

The SUT is a beta software simulation of the MOST NetworkMaster that is connected to an actual network. The network controller is intended to be built by different suppliers, not the automotive OEM who, nonetheless, needs a software-in-the-loop simulation for integration tasks with other devices. The NetworkMaster simulation was built by an external third-party. Roughly, the interface of the SUT is identical to that of hardware NetworkMasters.

In order to make the abstract test cases – model traces – applicable to the SUT, we wrote a compiler that translates them into 4CS[2] test programs. 4CS provides a test infrastructure for MOST networks. So-called *optolyzers* were used to stub actual nodes: these are freely programmable nodes in the MOST network and a Pascal-like language to formulate test cases. Via 4CS, we programmed them to behave like a corresponding test case. In this way, we can stimulate the SUT. In the 4CS programs, the SUT's output is compared to the intended output as given by the test case. At the end of each test case, the central registry of the SUT was downloaded and compared to the corresponding registry of the model which is also encoded in the test cases.

We omit details of the driver components responsible for input concretisation and output abstraction. For instance, in terms of data abstraction, one arbitrary representative of an equivalence class of error codes was chosen in order to instantiate signals. In terms of temporal abstraction, the expiration of a timer was instantiated by a wait statement containing the actual physical duration. Conversely, output of the model is converted into an executable verification statement. For example, if an output signal contains a list of items as parameter, a corresponding verification statement is created which checks if the actual list in the implementation's output is a permutation of the expected list in the model's output: the model is deliberately over-specified.

6.2.3 Tests

This section describes the general procedure of testing the NetworkMaster, different test suites, and observations.

Overview

Once the model had been built, we derived different test suites. The whole procedure is based on the technology and methodology of [166]. Further detail can be found in [164]. Except for hand-crafted test cases, these consist of abstract sequences of input and expected output. We turned them into executable test cases as described in [166]. Tests built without a model were manually lifted to the more abstract level of the model. Doing so allows us to apply all test cases to the implementation via the 4CS compiler, check for conformance with the model, and measure coverage at the level of the implementation. In addition, we applied the input part of each test case to the model and measured coverage at the level of the model. *Model coverage* is defined by means of coverage on Java (simulation) code that was generated from the model. *Implementation coverage*, on the other hand, was measured on the C code of the SUT. For the sake of comparability, we excluded those C functions that, as a

[2] `http://www.4cs.de/`

consequence of abstraction, do not have counterparts in the model. However, some of the abstracted behaviour is scattered over the C code, and we did not remove these parts for measurements.

Our coverage criterion is based on the control-flow of a program. *Condition/Decision (C/D) coverage* measures the number of different evaluations of each atomic condition in a composed condition plus the outcome of the decision. 100% coverage requires that each atomic condition be evaluated at least once to both true and false, plus the requirement that the decision takes both possible outcomes.

In addition to coverage measurements, we documented differences between the behaviours of model and implementation, and grouped these failures into 26 classes. Since the elements of a class exhibit a similar erroneous behaviour, we conjecture that the elements of each class correspond to the same fault, i.e., the same cause of the deviation in behaviours. Since the SUT was built by an external third party, we could not verify this conjecture. We use the terms "failure class" and "fault" interchangeably. The failure class can be seen as a sample of the failure region that is used in Sec. 2.2 to define a fault. When talking about numbers of detected faults, we always mean *distinct* faults.

Different test suites were applied in order to assess (1) the use of models vs. hand-crafted tests, (2) the automation of test case generation with models, and (3) the use of explicit test case specifications. We also provide a comparison with randomly generated tests.

Test Suites

This section describes the seven different test suites that we compared, and explains with what means we built them. The length of all test cases varies between 8 and 25 steps. To all test cases, a postambule of 3–12 steps is automatically added that is needed to judge the internal state of the SUT (registry download). We do not directly access the internal state of the NetworkMaster because we perform a black-box test of it.

Table 6.2: Test suites

suite	automation	model	TC specs
A	manual	yes	yes
B	auto	yes	yes
C	auto	yes	no
D	auto	no	n/a
E	manual	no	n/a
F	manual	no	n/a
G	manual	no	n/a

We investigated the following test suites:

A A test suite that was *manually* created by *interactively simulating the model*: $|A| = 105$ test cases.

B Test suites that were generated *automatically*, based on the model, *by taking into account functional test case specifications*. Tests were generated at random,

with additional constraints that reflect the test case specifications. The number of test cases in each suite varies between 40 and 1000. We refer to these test suites as “automatically generated”.

C Test suites that were generated at random, *automatically* on the grounds of the model, *without taking into account any functional test case specifications*. $|C| = 150$.

D Test suites that were *randomly* generated, *without referring to the model*. $|D| = 150$.

E A *manually* derived test suite that represents the *original requirements message sequence charts* (MSCs). This test suite contains $|E| = 43$ test cases.

F A *manually* derived test suite that, in addition to the original requirements MSCs, contains some further MSCs. These are a result of *clarifying the requirements* by means of the model. The test suite itself was derived without the model. This test suite contains $|F| = 50$ test cases.

G A test suite that was *manually developed* with traditional techniques, i.e., without a model: 61 test cases.

All these test suites are summarised in Tab. 6.2. The difference between test suites $\{E, F\}$ and G is that $\{E, F\}$ directly stem from requirements documents only whereas G is is based on test documents. The difference between A and F is similar: F is a direct result of requirements engineering activities, and A results from testing activities, in particular building the test model.

Functional Test Case Specifications (Suite B)

We defined functional test case specifications in order to specify sets of test cases to be generated for suite B. Each test specification is related to one functionality of the NetworkMaster, or to a part of the behaviour it exhibits in special situations. We identified seven classes of functional test case specifications that we state informally.

TS1 Does the NetworkMaster start up the network to normal operation if all devices in the environment answer correctly?

TS2 How does the NetworkMaster react to central registry queries?

TS3 How does the NetworkMaster react if nodes do not answer?

TS4 Does the NetworkMaster recognise all situations when it must reset the MOST network?

TS5 Does the NetworkMaster register signals that occur spontaneously?

TS6 Does the NetworkMaster reconfigure the network correctly if one node jumps in or out of the network?

TS7 Does the NetworkMaster reconfigure the network correctly if a node jumps in or out of the network more than once?

We implemented and refined TS1–TS7 into 33 test case specifications by stipulating that specific signals must or must not occur in a certain ordering or frequency in traces of the NetworkMaster model.

Automatic Generation

Automatic generation of test cases was done as follows. For suite B, we translated the test case specifications into constraints, and added them to the CLP translation of the model. The resulting CLP program was executed for test cases of a length of up to 25 steps. Computation was stopped after a given amount of time, or, as a consequence of state storage and test case specification, when there were no more test cases to enumerate. For each of the 33 specifications, this yielded suites that satisfy them. During test case generation, choosing transitions and input signals was performed at random. In order to mitigate the problems that are a result of the depth first search we use, we generated tests with different seeds for the random number generator: for each test case specification, fifteen test suites with different seeds were computed. Out of each of the fifteen suites, a few tests were selected at random. We hence generated test suites that were randomly chosen from all those test cases that satisfy the test case specifications.

Suite C was generated in a similar manner, but without any functional test case specifications. Suite D was derived by randomly generating input signals that obeyed some sanity constraints (e.g., switch on the device at the beginning of a test case) but did not take into account any logics whatsoever. In order to get the expected output part of a test case, we applied the randomly generated input to the model and recorded its output.

6.2.4 Observations and Interpretations

This section describes our findings in terms of fault detection, model coverage, and implementation coverage.

Fault Detection

26 faults were detected by the various tests. In addition, 3 major inconsistencies, 7 omissions, and 20 ambiguities were found in the specification documents while the model was built. Two of the 26 faults are faults in the model, a consequence of mistaken requirements. The remaining faults compromise 13 implementation faults and 11 requirements faults. The latter are defined by the fact that their removal involved changing the user requirements specifications. Note that these did not include the model itself. Even the updated requirements MSCs – on which test suite F is based – contained omissions and ambiguities. Changing requirements specifications was not necessary for implementation faults. Out of the 24 faults in the implementation, 15 were considered severe by the domain experts, and 9 were considered non-severe. In this context, severity means that the occurrence of such a fault at runtime would lead the whole system to fail.

Because we could not automatically assign a failure to its class we had to manually check the results of running the test cases. This restricts the number of tests. In terms

of suites B, we picked 4 times 5 tests and once 10 tests for each of the 33 refined test case specifications, which adds to $4 \cdot 165 + 330 = 990$ tests. For suites C, we picked 4 times 150 tests, and 3 times 150 tests for suites D. Fig. 6.7 shows the faults (classes of failures) that were detected with different test suites; the first bar for suite B is the one that consists of 330 tests. The AF bar represents the test suite that consists of $A \cup F$; these two together seemed a natural reference candidate for assessing automated tests.

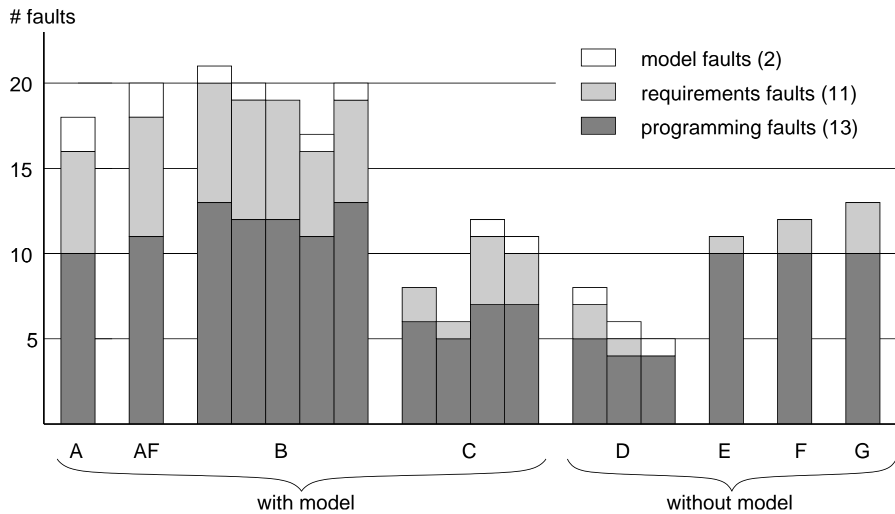


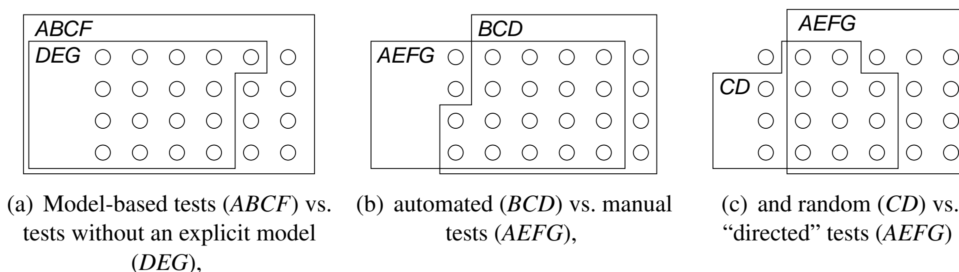
Figure 6.7: Detected faults

The major observation is that model-based and hand-crafted tests both detect approximately the same number of implementation faults. Requirements faults are predominantly detected by model-based tests. This is because building the model involved a thorough review of the requirements documents, and these are directly reflected in the model. None of the test suites detected all 26 faults, and there is no correlation between test suites and the severity status of the respective detected faults. Test suite A (105 tests, 18 faults) detects slightly fewer faults than the combined *AF* (148 tests, 20 faults). Approximately the same amount of faults are detected by the test suites B, 23 when cumulating the suites. Note that the they then consist of 990 tests whereas *AF* only has 148 test cases.

Randomly generated model-based tests (suites C, cumulated: 15 faults) detect roughly as many faults as manually designed tests (E-G). The latter detect more implementation faults, and almost the same number of requirements faults. For many of the faults detected only by the randomly generated tests it holds that they were judged unlikely by the domain experts. Hence, the failure probability in the field (cf. Chap. 4) seems to be low. Suites D (cumulated: 8 faults) exhibit the smallest number of detected faults. All of them are also detected by B; two faults not detected by AF correspond to traces that, once more, appear abstruse to a human because of the involved randomness. The use of functional test case specifications hence ensures that respective tests perform better than purely randomly generated tests.

We come back to three of of our initial questions now and compare different test approaches based on diversity. For this, there are three Venn diagrams shown in Fig. 6.8. The circles represent the 24 faults in the NetworkMaster. In the following comparisons

we always used a single representative when for a specific type of test suite more than one has been built. Some more detailed comparisons can be found in [168]. The diagram in Fig. 6.8(a) compares model-based tests with tests that were derived without an explicit model. For this comparison we consider the test suites A, B, C, and F to be model-based. The latter is part of this set because although it is based on MSCs, the MSCs in turn have been updated by the model. We observe that the model-based tests are a superset of the other test. In particular, the tests that do not use an explicit model have not been able to detect 7 of the faults. Hence, the use of models seems to make other tests superfluous.



(a) Model-based tests (*ABCF*) vs. tests without an explicit model (*DEG*),

(b) automated (*BCD*) vs. manual tests (*AEFG*),

(c) and random (*CD*) vs. "directed" tests (*AEFG*)

Figure 6.8: Venn diagram showing the diversity of different approaches

The second diagram in Fig. 6.8(b) concentrates on the comparison of automatic and manual test case derivation. The automatically derived test suites are B, C, and D. The others are considered to be built manually. Note, that in both sets there are model-based tests. We see that there is a stronger diversity between these sets. The automated tests detected four faults that the manual tests did not find whereas the manual tests could find two additional faults. This suggests that a combination of automatic and manual test case generation is most beneficial.

Finally, the diagram in Fig. 6.8(c) compares the two random test suites C and D with the "directed" tests A, E, F, and G. The directed tests are those tests that were designed with a special purpose, i.e., testing special situations. The random tests only cover stochastically the input space. We deliberately excluded test suite B here because the distinction is not obvious. Although test case specification direct the test case generation, we still pick specific test cases randomly from the large possibilities. Again, we observe that there is a strong diversity between both sets. There is even one fault that was not detected by both sets of test suites. The combination of both approaches seems again to be beneficial.

Model Coverage

For the model coverage as well as the implementation coverage it holds that they are not necessary for the economics model discussed in Chap. 4. However, the analysis is important to extend the empirical body of knowledge and it can be used in a more detailed testing model that uses coverage information, e.g. [125].

The model contains 1722 C/D evaluations in transition guards and functional programs used by the component `RegistryManager`. The implementation contains

916 C/D evaluations. Fig. 6.9 shows C/D coverage at the level of the Java simulation code generated from the model.

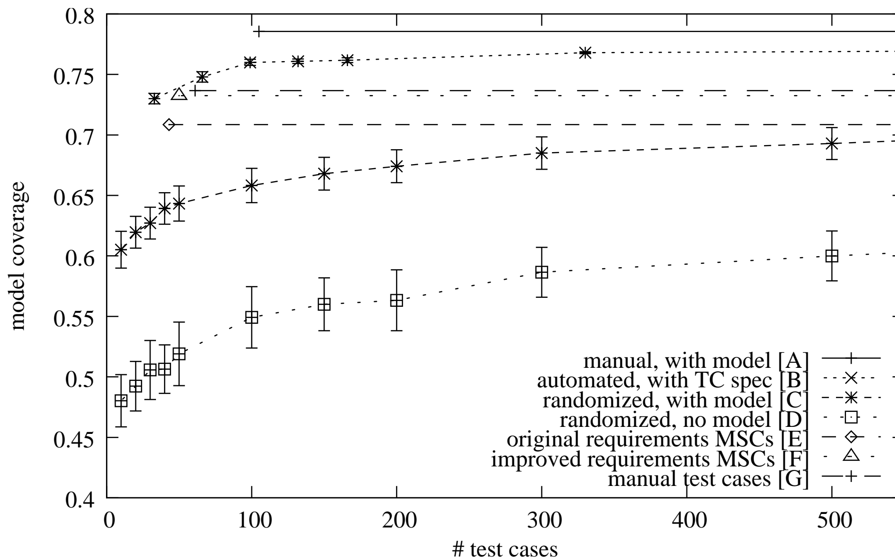


Figure 6.9: Model coverage

For test suites with varying numbers of test cases, we display the mean that was computed from 25 experiments, i.e., 25 times a choice of n test cases out of original sets that range from 6,000 to 10,000 tests. The error bars denote the 98% confidence interval for the mean under the assumption that the data is approximately normally distributed. For the sake of graphical representation, we do not display any numbers for more than 550 test cases.

Coverage does not exceed 79%. The reason is the handling of pattern matching in the generated Java code with trivially true conditional statements. Except for the test cases that have been generated without a model, the 98% confidence intervals for the given means are rather small. This implies a likelihood that the displayed trends are not subject to random influences.

A yields the highest coverage which is unmatched by the second best suite B. That A yields such a high coverage is explained by the fact that the same person built the model and the test case specifications. This person intuitively tried to match the structure of the model. In our case study, automation could hence not match the coverage of manually generated model-based tests. A does not include all covered C/Ds of suites B to G: even though the absolute coverage of A is the highest, it turns out that the latter yield up to 14 additional evaluations of atomic conditions. Furthermore, generated tests covered more possible input signals, a result of randomisation. Manually derived test cases included some special cases that the randomly generated tests did not cover.

Suites F and G are the next best suites; this is explained by the fact that the improved requirements documents contain some "essential" runs of the model. Suite C, i.e, randomly generated model-based tests, match the coverage of F at about 500 test cases. The comparison of test suites $\{C, D\}$ and B shows that the use of functional test case specifications leads to higher coverage with fewer test cases. This is explained

with the fact that test case specifications “slice” the model. Thus, with this smaller state space it is more likely that all elements are covered.

Implementation Coverage

Technical constraints of the test executions made it impossible to run the same set of experiments on the implementation. Because of the limited number of evaluated test suites we cannot display the evaluation of coverage with an increasing number of test cases. Instead, we display the relationship between model coverage and implementation coverage (Fig. 6.10) for test suites with a fixed number of elements. These test suites form a superset of those regarded in Fig. 6.7. That not all of them were considered in the fault analysis is a consequence of the effort that is necessary to assign failures to failure classes.

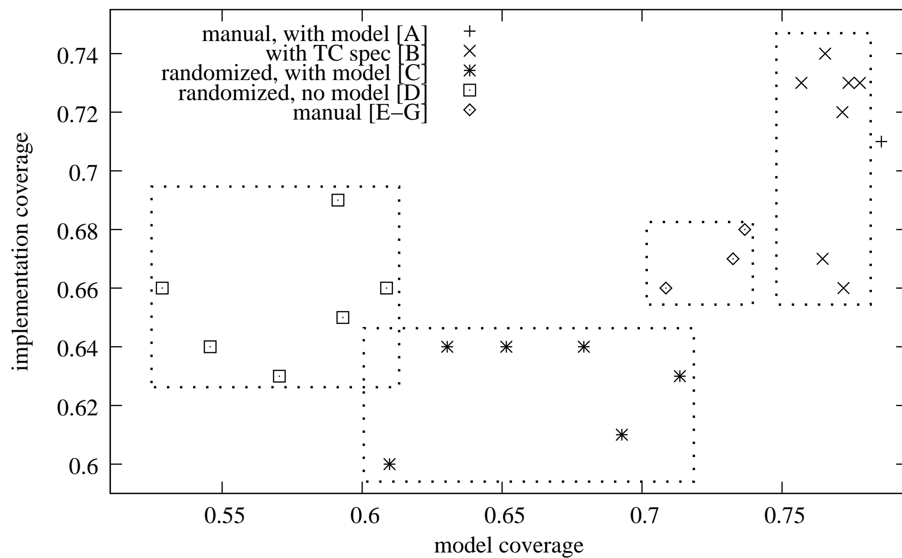


Figure 6.10: Coverages

That implementation coverage does not exceed 75% is a result of the abstractions applied to the model. We excluded most C functions from the measurements that had no counterparts. However, as mentioned above, some of the behaviour abstracted in the model is scattered through the code, and we did not touch these parts. One can see that test suites that were built with randomness (B, C, D) yield rather different coverages in their own classes. This is likely due to random influences: as our measurements and the 98% confidence intervals in Fig. 6.9 indicate at least for the model, test suites from one category tend to yield rather constant coverages.

On average, the random suites C and D yield roughly the same implementation coverage. As in the case of the model, coverage tends to increase for suite B. There is a moderate positive correlation between coverages (correlation coefficient $r = .63$; $P \leq .001$). We expected to see a stronger correlation on the grounds of the argument that the “main” threads of functionality are identical in the model and an implementation. This was not confirmed. The figure suggests that there is a rather strong (the small

number of measurements forbids a statistical analysis) correlation of coverages if only the manually derived suites $\{E, F, G, A\}$ are regarded.

While the manually built model-based test suite A yields higher model coverage than the tests in B – as explained above – it exhibits a lower implementation coverage than B. This, again, is a result of the fact that the implementation ran into some branches that were not modelled.

Coverage vs. Fault Detection

A combination of the data from the sections above is given in Figs. 6.11 and 6.12. Both figures suggest a positive correlation between C/D coverage and fault detection. Data is more scattered in the case of implementation coverage: correlation coefficient $r = .68$ ($P \leq .001$) for the implementation. Correlation is $r = .84$ ($P \leq .0001$) for the model with a logarithmically transformed ordinate. We observe in Fig. 6.11 that test suite D yields a comparatively high coverage but finds few faults. As above, this is explained by the fact that implementation coverage includes functionality that is not implemented in the model, most importantly, timing issues. Furthermore, one can see that at high coverage levels, increasing coverage does not necessarily increase the number of detected faults.

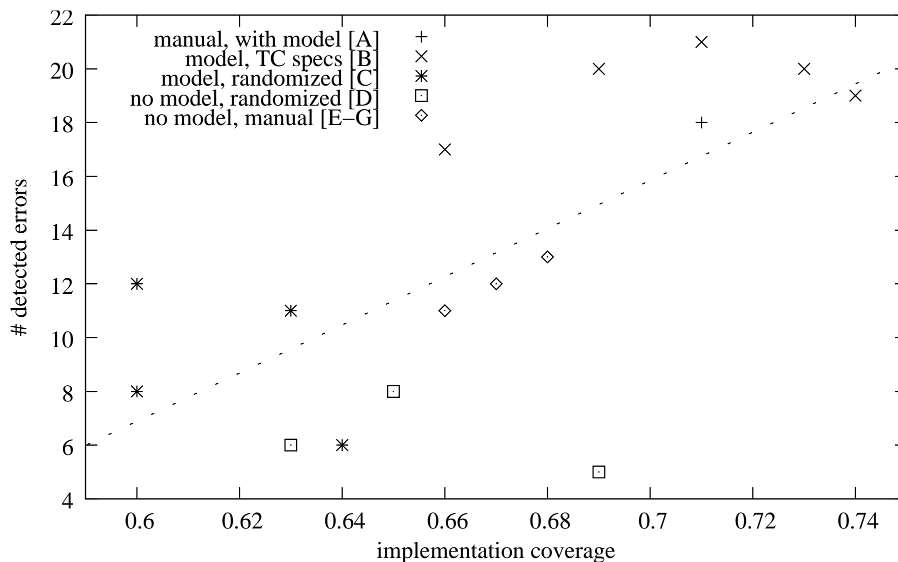


Figure 6.11: Implementation coverage vs. faults

Defect-Proneness

The metrics suite was applied to the component diagrams and state machines of the NetworkMaster. The resulting values for the metrics are summarised in Table 6.3.

The data from the table shows that the *RegistryMgr* has the highest complexity in most of the metrics. Therefore, we classify it as being highly fault-prone. As discussed above, 24 faults were identified by the test activities of which 13 are implementation

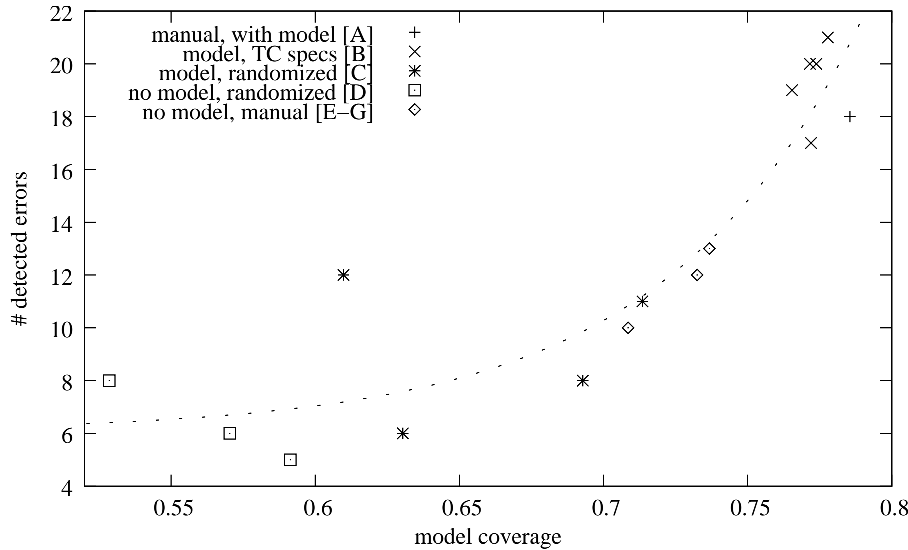


Figure 6.12: Model coverage vs. faults

Table 6.3: The results of the metrics suite for the NetworkMaster.

Class	DIT	NOC	NOP	NRI	NPI	CCS
NetworkMaster	0	0	5	4	5	0
Divide	0	0	0	1	3	11
Merge	0	0	0	3	1	8
MonitoringMgr	0	0	0	2	1	0
RequestMgr	0	0	0	2	1	14
RegistryMgr	0	0	0	4	7	197

faults, 9 requirements defects, and 2 model faults. Of these faults, 21 can be attributed to the *RegistryMgr* and 3 to the *RequestMgr*. This assignment was done based on the responsibilities of the components because we had no direct access to the faults. There were no faults revealed in the other components. Hence, the high fault-proneness of the *RegistryMgr* did indeed result in a high number of faults detected during testing.

Discussion

This case study shows that building precise models in itself is able to reveal inconsistencies and omissions in the requirements as well as faults in the implementation. Especially executable behaviour models can be seen as an abstract prototype of the system. It is remarkable, however, that the number of detected implementation faults is independent of the use of models.

In accordance with earlier studies [157, 169], the benefit of automation is not clear. In our approach, we depend on humans that identify the "interesting" cases and formulate them as test case specifications. Whether structural criteria alone would be sufficient is disputable. Furthermore, the current generation technology needs manual optimisations in the generated CLP code. To this end, it has not been shown that a

complete automation ("push-button technology") is possible at all.

Nevertheless, automation is indeed helpful when changes in the model have to be taken into account. Provided that test case generation is a push-button technology, it is obviously simpler to automatically generate new tests than to hand-craft them. It is possible to conceive and hand-craft 100 tests in a few hours, but this becomes more complicated for 1,000 tests. Recall how a significant increase in automatically generated model-based tests revealed some additional faults. Moreover, the consistency of changes to the test suite can be assured easily as the model acts as a single source of information. In manual test cases it is difficult to ensure that changes are made consistently in all test cases. The length of the test cases – the number of steps that must be performed – matters as well. The number of test cases must be restricted because they not only have to be applied but also to be evaluated: if there is a deviation in behaviours, then the test run must be manually inspected. In addition, in the case of the software-in-the-loop simulation of the embedded system of our study, each test takes at least 10 seconds because of hardware limitations. This naturally restricts the number of tests that can be run. Furthermore, we found that purely randomly generated tests are difficult to interpret because they correspond to highly "non-standard" behaviour.

Counting failures for reactive systems is non-trivial. When the behaviours of model and implementation differed at a certain moment in time, they tended to differ for the rest of the test case, too. We tried to associate a maximum number of faults to a test run, but were in doubt sometimes: in our statistics, the majority of test cases revealed not more than one fault.

It is difficult to draw conclusions from the moderate correlation between model coverage and implementation coverage. Using coverage criteria as test case specifications for automated test case generation relies on their suspected ability to detect faults. In addition to the ongoing controversy on this subject, our results suggest some care with directly transferring findings on *implementation coverage* to *model coverage*. Model coverage, as we define it, is clearly dependent on the simulation code generator that is used.

There are certain threats to validity. When comparing test suites built by different teams, which is the case for our test suites A and G, one must take into account the fact that different people in different contexts with different knowledge of the system conceived them. Hamlet [72] comments on that and the findings of Hutchins et al. [85] indicate that test suites derived by different test engineers – or even different test suites derived by the same engineer – vary w.r.t. effectiveness. While we consider it possible to generalise our findings to other event-discrete embedded devices with almost no ordered data types, we cannot say whether the same is true for discrete-continuous embedded systems or business information systems. As mentioned above, it is, in general, likely that the benefits of automation are greater if significantly more tests could be run. This is not always the case for embedded systems.

We are also aware that we used one specific modelling language, and tested an implementation at a certain stage of development. We do not know if our findings generalise for implementations in a more mature state. Furthermore, we cannot judge whether or not different coverage criteria, particularly those based on data flow, exhibit the same characteristics. In terms of test case generation technology, we do not think that our approach is fundamentally different from others.

Also for the validation of the suite of model metrics there are two specific threats

to the validity of the results: (1) The NetworkMaster had already been tested by the third-party supplier. We do not know the locations and number of the faults detected there. (2) We have so far no field data about faults of the NetworkMaster that could be used to confirm the results. Hence, the total number of faults is unknown.

6.2.5 Summary

After describing and discussing the different aspects of the MOST NetworkMaster case study, we put it into relation to the analytical model and the metrics suite. For the analytical model, the case study provides values to calibrate it w.r.t. model-based testing. Furthermore, we are able validate the metrics suite to be able to identify defect-prone components.

Difficulty

This case study provides, besides the insights discussed above, also the opportunity to calculate first estimates for the average difficulty $\bar{\theta}$ for the defect detection of different model-based tests. These results are used in the review of the empirical studies in Sec. 4.2.5. The review is the basis of the sensitivity analysis of the model and hence this study contributed valuable data. We add all found defects and approximate the difficulty by the ratio of defects not detected by all defects. For the model-based test suites this gives average difficulties of 0.25, 0.12, 0.17, and 0.67 for A, AF, B, and C, respectively. For the functional tests derived from MSC specifications we get difficulties of 0.54, 0.50, and 0.67. The purely random tests (D) have a difficulty of 0.67.

Defect-Proneness

Furthermore, we validated the metrics suite from Chap. 5 as suitable predictor for defect-prone components. The suite identified the component correctly as defect-prone that during testing contained the most defects. In combination with the results from the automatic collision notification case study, we can further analyse the metrics.

Correlation of Metrics. A main problem of software metrics is that different metrics might not be independent. We analyse our proposed metrics suite concerning the correlation of the different metrics based on the data from the case studies on the automatic collision notification system of Sec. 6.1 and the NetworkMaster. The sample size is still small therefore the validity is limited but we can give first indications.

We are not able to analyse DIT and NOC because they were not used in the case studies. Also it does not make sense to analyse NOP with only two non-null data points. Therefore we concentrate on NRI, NPI, and CCS. The correlation between NRI / CCS and NPI / CCS is low with a a correlation coefficient $r = -0.17$ and $r = -0, 13$, respectively. Only the correlation between NRI and NPI is more interesting. The correlation coefficient is 0.55 but the Chi-test and F-test only yielded probabilities of 0.35 and 0.17, respectively, for both data rows coming from the same population. Hence, we have a good indication that the metrics of our suite are not interdependent.

Correlation of Metrics and Faults. As we use the classification approach with our metrics, we cannot estimate numbers of faults and therefore a correlation between estimated and actual faults is not possible. Also a correlation analysis between the single metrics and the number of found faults has a limited meaning because only the combined suite can provide a complete picture of the complexity of a component. However, the statistical correlation between the metrics and the number of faults is not as low as expected. For NRI the coefficient is 0.35, for NPI 0.58, and for CCS 0.53 but Chi- and F-tests show a very low significance probably because of the small sample size.

Discussion. By looking at the case studies it seems that the CCS metric has the most influence on the fault-proneness. However, there are components that do not have a state machine but their behaviour is described by its parts and still might contain several faults. It can also be rather trivial to see that a specific component is fault-prone as in the case of the *RegistryMgr* of the *NetworkMaster*. This component has such a large state machine in comparison to the other components that it is obvious that it contains several faults. In larger models with a large number of components it might not be that obvious. Finally, there is no evident influence of the application type on the metrics visible from the case studies as both have components with a rather small number of interfaces and parts and a few components with quite large state machines.

6.3 Mobile Services Backend Systems

Extensive research has been done on finding defects in code by automated static analysis using tools called *bug finding tools*, e.g. [6,57,77]. Although the topic is subject of ongoing investigations, there are only few studies about how these tools relate among themselves and to other established defect-detection techniques such as testing or reviews.

We address the question of how automated static analysis using bug finding tools relates to other types of defect-detection techniques and if it is thereby possible to reduce the effort for defect-detection using such tools. In detail, this amounts to three questions.

1. Which types and classes of defects are found by different techniques?
2. Is there any overlap of the found defects?
3. How large is the ratio of false positives from the tools?

This case study has also been published in [212].

Analysed Tools

The three bug finding tools that we used for the comparison are described in the following. We only take tools into account that analyse Java programs because the projects we investigated, as described below, are all written in that language. All three tools are published under an open source license. We used these three tools as representatives for tools that mainly use bug patterns, coding standards, and dataflow analysis,

respectively. We deliberately ignored tools that need annotations in the code because they have quite different characteristics.

FindBugs. The tool *FindBugs* was developed at the University of Maryland and can detect potentially problematic code fragments by using a list of bug patterns. It can find faults such as dereferencing null-pointers or unused variables. To some extent, it also uses dataflow analysis for this. It analyses the software using the bytecode in contrast to the tools described in the following. The tool is described in detail in [77]. We used the Version 0.8.1 in our study.

PMD. This tool [161] concentrates on the source code and is therefore especially suitable to enforce coding standards. It finds, for example, empty try/catch blocks, overly complex expressions, and classes with high cyclomatic complexity. It can be customised by using XPath expressions on the parser tree. The version 1.8 was used.

QJ Pro. The third tool used is described in [170] and analyses also the source code. It supports over 200 rules including ignored return values, too long variable names, or a disproportion between code and commentary lines. It is also possible to define additional rules. Furthermore, checks based on code metrics can be used. The possibility to use various filters is especially helpful in this tool. We evaluated version 2.1 in this study.

6.3.1 Projects

We want to give a quick overview of the five projects we analysed to evaluate and compare bug finding tools with other defect-detection techniques.

General

All but one of the projects chosen are development projects from the telecommunications company O_2 Germany for backend systems with various development efforts and sizes. One project was done by students at the Technische Universität München. All these projects have in common that they were developed using the Java programming language and have an interface to a relational database system. The O_2 projects furthermore can be classified as web information systems as they all use HTML and web browsers as their user interface.

Analysed Projects

The projects are described in more detail in [107]. For confidentiality reasons, we use the symbolic names A through D for the industrial projects.

Project A. This is an online shop that can be used by customers to buy products and also make mobile phone contracts. It includes complex workflows depending on the various options in such contracts. The software has been in use for six months. It consists of 1066 Java classes that consist of over 58 KLOC (kilo lines of code).

Project B. The software allows the user to pay goods bought over the Internet using a mobile phone. The payment is added to the mobile bill. For this, the client sends the mobile number to the shop and receives a transaction number (TAN) via short message service (SMS). This TAN is used to authenticate the user and authorises the shop to bill the user. The software has not been put into operation at the time of the study. Software B has 215 Java classes with over 24 KLOC in total.

Project C. This is a web-based frontend for managing a system that is used to convert protocol files between different formats. The analysed tool only interacts with a database that holds administration information for that system. The software was three months in use at the time it was analysed. It consists of over 3 KLOC Java and JSP code.

Project D. The client data of O_2 is managed in the system we call *D*. It is a J2EE application with 572 classes, over 34 KLOC and interfaces to various other systems of O_2.

EstA. The only non-industrial software that we used in this case study is *EstA*. It is an editor for structuring textual requirements developed during a practical course at the Technische Universität München. It is a Java-based software using a relational database management system. The tool has not been extensively used so far. It has 28 Java classes with over 4 KLOC.

6.3.2 Approach

In this section, the approach of the case study is described. We start with the general description and explain the defect classification and defect types that are used in the analysis.

General

We use the software of the five projects introduced in Sec. 6.3.1 to analyse the interrelations between the defects found by bug finding tools, reviews, and tests. For this, we applied each of these techniques to each software as far as possible. While a review was only made on project C, black-box as well as white-box tests were done on all projects. We ran the bug finding tools with special care to be able to compare the tools as well. To have a better possibility for comparison with the other techniques, we also checked each warning from the bug finding tools if it is a real defect in the code or not. This was done by an inspection of the corresponding code parts together with experienced developers. The usage of the techniques was completely independent, that is, the testing and the review was not guided by results from the bug finding tools.

The external validity is limited in this case study. Although we mostly considered commercially developed software that is in actual use, we only analysed five systems. For better results more experiments are necessary. Furthermore, the tests on the more mature systems, i.e. the ones that are already in use, did not reveal many faults. This can also limit the validity. Moreover, the data from only one review is not representative but can only give a first indication. Finally, we only analysed three bug finding

tools, and these are still under development. The results might be different if other tools would have been used.

In the following we call all the warnings that are generated by the bug finding tools *positives*. *True positives* are warnings that are actually confirmed as defects in the code, *false positives* are wrong identifications of problems.

Defect Severity

For the comparison, we use a five step categorisation of the defects using their severity. Hence, the categorisation is based on the effects of the defects rather than their cause or type of occurrence in the code. We use a standard categorisation for severity that is slightly adapted to the defects found in the projects. Defects in category 1 are the severest, the ones in category 5 have the lowest severity. The categories are:

1. *Defects that lead to a crash of the application.* These are the most severe defects that stop the whole application from reacting to any user input.

2. *Defects that cause a logical failure.* This category consists of all defects that cause a logical failure of the application but do not crash it, for example a wrong result value.

3. *Defects with insufficient error handling.* Defects in this category are only minor and do not crash the application or result in logical failures, but are not handled properly.

4. *Defects that violate the principles of structured programming.* These are defects that normally do not impact the software but could result in performance bottlenecks etc.

5. *Defects that reduce the maintainability of the code.* This category contains all defects that only affect the readability or changeability of the software.

This classification helps us (1) to compare the various defect-detection techniques based on the severity of the defects they find and (2) analyse the types of defects that they find.

Defect Types

Additionally to the defect classification we use defect types. That means that the same or very similar defects are grouped together for an easier analysis. We do not use a standard classification of defect types as described in Sec. 2.2.2 but was defined specifically for the tools and programming language to allow a more fine-grained comparison.

The defect types that we use for the bug finding tools can be seen as a unification of the warning types that the tools are able to generate. Examples for defect types are “Stream is not closed” or “Input is not checked for special characters”.

Table 6.4: Summary of the defect types found by the bug finding tools

Defect Type	Category	FindBugs	PMD	QJ Pro
Database connection is not closed	1	8/54	8/8	0/0
Return value of function ignored	2	4/4	0/0	4/693
Exception caught but not handled	3	4/45	29/217	30/212
Null-pointer exception not handled	3	8/108	0/0	0/0
Returning null instead of array	3	2/2	0/0	0/0
Stream is not closed	4	12/13	0/0	0/0
Concatenating string with + in loop	4	20/20	0/0	0/0
Used "==" instead of "equals"	4	0/1	0/0	0/29
Variable initialised but not read	5	103/103	0/0	0/0
Variable initialised but not used	5	7/7	152/152	0/0
Unnecessary if-clause	5	0/0	16/16	0/0
Multiple functions with same name	5	22/22	0/0	0/0
Unnecessary semicolon	5	0/0	10/10	0/0
Local variable not used	5	0/0	144/144	0/0
Parameter not used	5	0/0	32/32	0/0
Private method not used	5	17/17	17/17	0/0
Empty finally block	5	0/0	1/1	0/0
Unnecessary comparison with null	5	1/1	0/0	0/0
Uninitialised variable in constructor	5	1/1	0/0	0/0
For- instead of simple while loop	5	0/0	2/2	0/0

6.3.3 Analysis

This section presents the results of the case study and possible interpretations. At first, the bug finding tools are compared among each other, then the tools are compared with reviews, and finally with dynamic tests.

Bug Finding Tools

We want to start with comparing the three bug finding tools described in Sec. 2.3.4 among themselves. The tools were used with each system described above.

Data. Tab. 6.4 shows the defect types with their categories and the corresponding positives found by each tool over all systems analysed. The number before the slash denotes the number of true positives, the number after the slash the number of all positives.

Observations and Interpretations. Most of the true positives can be assigned to the category *Maintainability of the code*. It is noticeable that the different tools predominantly find different positives. Only a single defect type was found by all tools, four types by two tools each.

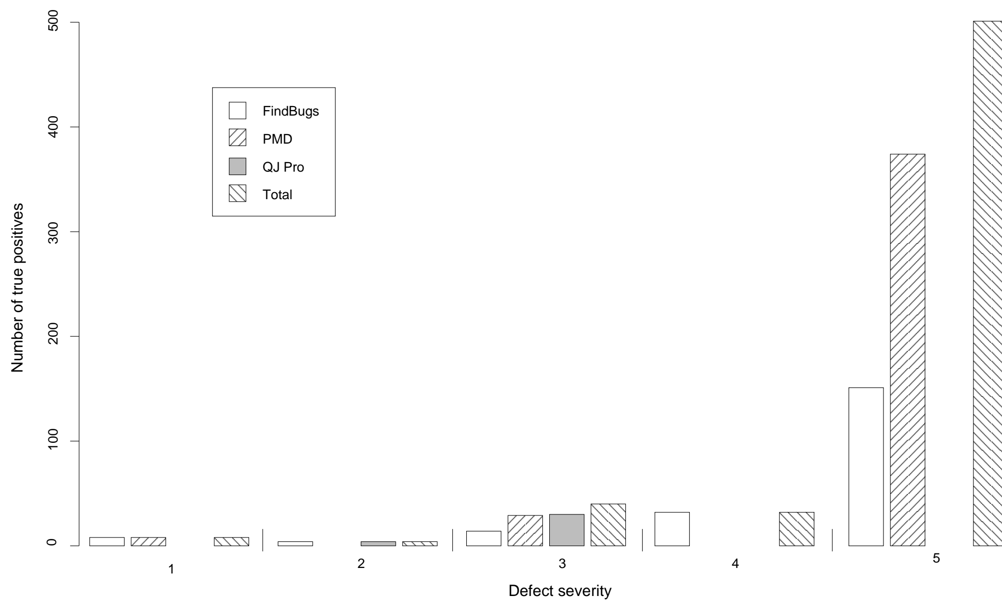


Figure 6.13: A graphical comparison of the number of true positives found by each tool and in total

Considering the categories, FindBugs finds in the different systems positives from all severity levels and PMD only from the severities *Failure of the application*, *Insufficient error handling*, and *Maintainability of the code*. QJ Pro only reveals positives from the severity levels *Logical failure of the application*, *Insufficient error handling*, and *Violation of structured programming*. The number of faults found in each category from each tool is graphically illustrated in Fig. 6.13. Also the number of types of defects varies from tool to tool. FindBugs detects defects of 13 different types, PMD of 10 types, and QJ Pro only of 4 types.

The accuracy of the tools is also diverse. We use the defect type "Exception is caught but not handled" that can be found by all three tools as an example. While FindBugs only finds 4 true positives, PMD reveals 29 and QJ Pro even 30. For this, the result from QJ Pro contains the true positives from PMD which in turn contain the ones from FindBugs. A reason for this is that QJ Pro is also able to recognise a single semicolon as a non-existent error handling, whereas the other two interpret that as a proper handling. This defect type is also representative in the way that FindBugs finds the least true positives. This may be the case because it uses the compiled class-files while PMD and QJ Pro analyse the source code.

A further difference between the tools is the ratio of true positives to all positives. PMD and FindBugs have a higher accuracy in indicating real defects than QJ Pro. Tab. 6.5 lists the average ratios of false positives for each tool and in total. It shows that on average, half of the positives from FindBugs are false and still nearly a third from PMD. QJ Pro has the worst result with only 4% of the positives being true positives. This leads to an overall average ratio of 0.66, which means that two thirds of the positives lead to unnecessary work. However, we have to notice that FindBugs and PMD are significantly better than that average.

Table 6.5: Average ratios of false positives for each tool and in total

FindBugs	PMD	QJ Pro	Total
0.47	0.31	0.96	0.66

An illustrative example is the defect type “Return value of function is ignored”. FindBugs only shows 4 warnings that all are true positives, whereas QJ Pro provides 689 further warnings that actually are not relevant. Because all the warnings have to be looked at, FindBugs is in this case much more efficient than the other two tools.

The efficiency of the tools varied over the projects. For the projects *B* and *D*, the detection of the defect type “Database connection not closed” shows only warnings for true positives with FindBugs. For project *A*, it issued 46 warnings for which the database connection is actually closed. Similarly, the detection rate of true positives decreases for the projects *D* and *A* for the other two tools, with the exception of the well recognised positives from the maintainability category by PMD. This suggests that the efficiency of the defect detection depends on the design and the individual programming style, i.e. the implicit assumptions of the tool developers about how “good” code has to look like.

A recommendation of usage of the tools is difficult because of the issues described above. However, it suggests that QJ Pro, although it finds sometimes more defects than the other tools, has the highest noise ratio and therefore is the least efficient. FindBugs and PMD should be used in combination because the former finds many different defect types and the latter provides very accurate results in the maintainability category. Finally, PMD as well as QJ Pro can be used to enforce internal coding standards, which was ignored in our analysis above.

Bug Finding Tools vs. Review

An informal review was performed only on project *C*. The review team consisted of three developers, including the author of the code. The reviewers did not prepare specifically for the review but inspected the code at the review meeting.

Data. The review revealed 19 different types of defects which are summarised in Tab. 6.6 with their severities and number of occurrences.

Observations and Interpretations. All defects found by bug finding tools were also found by the review. However, the tools found 7 defects of type “Variable initialised but not used” in contrast to one defect revealed by the review. On the other hand, in project A the review detected 8 defects of type “Unnecessary if-clause”, whereas the tools only found one. The cause is that only in the one defect that was found by both there was no further computation after the if-clause. The redundancy of the others could only be found out by investigating the logics of the program.

Apart from the two above, 17 additional types of defects were found, some of which could have been found by tools. For example, the concatenation of a string with “+” inside a loop is sometimes not shown by FindBugs although it generally is able to

Table 6.6: Summary of the defect types and defects found by the review

Defect Type	Severity	Occurrences
Database connection is not closed	1	1
Error message as return value	1	12
Further logical case ignored	2	1
Wrong result	2	3
Incomplete data on error	2	3
Wrong error handling	2	6
ResultSet is not closed	4	1
Statement is not closed	4	1
Difficult error handling	4	10
Database connection inside loop opened and closed	4	1
String concatenated inside loop with “+”	4	1
Unnecessary parameter on call	5	51
Unnecessary parameter on return	5	21
Complex for loop	5	2
Array initialised from 1	5	21
Unnecessary if clauses	5	8
Variable initialised but not used	5	1
Complex variable increment	5	1
Complex type conversion	5	7

detect this defect type. Also, the defect that a database connection is not closed was not found, because this was done in different functions. Furthermore it was not discovered by the tools that the ResultSet and the corresponding Statement was never closed. Other defect types such as logical faults or a wrong result from a function cannot be detected by bug finding tools. These defects, however, can be found during a review by following test cases through the code.

In summary, the review is more successful than bug finding tools, because it is able to detect far more defect types. However, it seems to be beneficial to first use a bug finding tool before inspecting the code, so that the defects that are found by both are already removed. This is because the automation makes it cheaper and more thorough than a manual review. However, we also notice a high number of false positives from all tools. This results in significant non-productive work for the developers that could in some cases exceed the improvement achieved by the automation.

Bug Finding Tools vs. Testing

We used black box as well as white box tests for system testing the software but also some unit tests were done. The black box tests were based on the textual specifications and the experience of the tester. Standard techniques such as equivalence and boundary testing were used. The white box tests were developed using the source code and path testing. Overall several hundred test cases were developed and executed. A coverage tool has also been used to check the quality of the test suites. However, there were no

Table 6.7: Summary of the defect types and defects found by the tests

Defect Type	Severity	Occurrences
Data range not checked	1	9
Input not checked for special characters	1	6
Logical error on deletion	1	1
Consistency of input not checked	2	3
Leading zeros are not ignored	2	1
Incomplete deletion	2	2
Incomprehensible error message	3	7
Other logical errors	2	3

stress tests which might have changed the results significantly. Only for the projects *EStA* and *C*, defects could be found. The other projects are probably too mature so that no further defects can be found by normal system testing.

Data. The detected defect types together with their severities and the number of occurrences are summarised in Tab. 6.7. We also give some information on the coverage data that was reached by the tests. We measured class, method, and line coverage. The coverage was high apart from project *C*. In all the other projects, class coverage was nearly 100%, method coverage was also in that area and line coverage lied between 60 and 93%. The low coverage values for project *C* probably stem from the fact that we invested the least amount of effort in testing this project.

Observations and Interpretations. The defects found by testing are in the severity levels *Failure of the application*, *Logical failure*, and *Insufficient error handling*. The analysis above of the defects showed that the bug finding tools predominantly find defects in the severity level *Maintainability of the code*. Therefore, the dynamic test techniques find completely different defects.

The software systems for which defects were revealed had no identical defects detected with testing or bug finding tools. Furthermore, the tools revealed several defects also in the systems for which the tests were not able to find one. These are defects that can only be found by extensive stress tests, such as database connections that are not closed. This can only result in performance problems or even a failure of the application, if the system is under a high usage rate and there is a huge amount of database connections that are not closed. The most defects, however, are really concerning maintainability and are therefore not detectable by dynamic testing.

In summary, the dynamic tests and the bug finding tools detect different defects. Dynamic testing is good at finding logical defects that are best visible when executing the software, bug finding tools have their strength at finding defects related to maintainability. Therefore, we again recommend using both techniques in a project.

Table 6.8: The effectiveness per defect-detection technique

Technique	Number of defects	Effectiveness
Bug Finding Tools	585	81%
Review	152	21%
Tests	32	4%
Total w/o dup.	769	100%

Table 6.9: The effectiveness for each category

Category	Bug Finding Tools	Reviews	Tests	Total
1	22% (8)	36% (13)	44% (16)	100% (37)
2	20% (4)	65% (13)	45% (9)	100% (26)
3	85% (40)	0% (0)	15% (7)	100% (47)
4	70% (32)	30% (14)	0% (0)	100% (46)
5	88% (501)	20% (112)	0% (0)	100% (613)

Effectiveness

The effectiveness is an important aspect of the costs and benefits of SQA. This factor can be used to estimate the difficulty of defect detection and is important to be analysed. It is also called defect removal efficiency is by Jones in [94]. It denotes the fraction of all defects that were detected by a specific defect-detection technique. The main problem with this metric is that the total number of defects cannot be known. In our case study we use the sum of all different defects detected by all techniques under consideration as an estimate for this number. The results are shown in Tab. 6.8. The metric suggests that the tools are the most efficient techniques whereas the tests where the least efficient.

However, we also have to take the defect severities into account because this changes the picture significantly. The Tab. 6.9 shows the effectiveness for each technique and severity with the number of defects in brackets. It is obvious that tests and reviews are far more effective in finding defects of severity 1 and 2 – the most severe defects – than the bug finding tools.

6.3.4 Discussion

The result that bug finding tools mainly detect defects that are related to the maintainability of the code complies with the expectation an experienced developer would have. Static analysis only allows to look for certain patterns in the code and simple dataflow and controlflow properties. Therefore only reviews or tests are able to verify the logic of the software (as long as the static analysis is not linked with model checking techniques). The tools do not “understand” the code in that sense. The prime example for this is the varying efficiency over the projects. In many cases, the tools were not capable to realise that certain database connections are not closed in the same

Java method but a different one. They only search for a certain pattern. Therefore, the limitation of static analysis tools lies in what is expressible by bug patterns, or in how good and generic the patterns can be.

However, it still is surprising that there is not a single overlapping defect detected by bug finding tools and dynamic tests. On the positive side, this implies that the two techniques are perfectly complementary and can be used together with great benefit. The negative side is that by using the automated static analysis techniques we considered, it may not be possible to reduce costly testing efforts. That there is only little overlapping follows from the observation above that the tools mainly find maintenance-related defects. However, one would expect to see at least some defects that the tests found also detected by the tools, especially concerning dataflow and controlflow. The negative results in this study can be explained with the fact that most of the projects analysed are quite mature, and some of them are already in operation. This resulted in only a small number of defects that were found during testing which in turn could be a reason for the lack of overlapping.

A rather disillusioning result is the high ratio of false positives that are issued by the tools. The expected benefit of the automation using such tools lies in the hope that less human intervention is necessary to detect defects. However, as on average two thirds of the warnings are false positives, the human effort could be even higher when using bug finding tools because each warning has to be checked to decide on the relevance of the warning. Nevertheless, there are significant differences between the tools so that choosing the best combination of tools could still pay off.

Bug finding tools that use additional annotations in the code for defect-detection could be beneficial considering the overlap of defects with other techniques as well as the false positives ratio. The annotations allow the tool to understand the code to a certain extent and therefore permits some checks of the logic. This deeper knowledge of the code might reduce the false positives ratio. However, to make the annotations requires additional effort by the developers. It needs to be analysed if this effort is lucrative.

The effort and corresponding costs of the determination of defects using the tools (including checking the false positives) was not determined in this study. This is however necessary to find out if the use of bug finding tools is beneficial at all.

There are only few studies about how bug finding tools relate among themselves and to other established defect-detection techniques such as testing or reviews. In [181] among others PMD and FindBugs are compared based on their warnings which were not all checked for false positives. The findings are that although there is some overlap the warnings generated by the tools are mostly distinct. We can support this result with our data.

Engler and Musuvathi discuss in [50] the comparison of their bug finding tool with model checking techniques. They argue that static analysis is able to check larger amounts of code and find more defects but model checking can check the implications of the code not just properties that are on the surface.

In [93] a static analysis tools for C code is discussed. The authors state that sophisticated analysis of, for example, pointers leads to far less false positives than simple syntactical checks.

An interesting combination of static analysis tools and testing in described in [46]. It is proposed to use static analysis to find potential problems and automatically generate

test cases to verify if there is a real defect. However, the approach obviously does not work with maintenance-related defects.

6.3.5 Summary

The work presented is not a comprehensive empirical study but a case study using a series of projects mainly from an industrial environment giving first indications of how the defects found by bug finding tools relate to other defect-detection techniques.

The main findings are that the bug finding tools revealed completely different defects than the dynamic tests but a subset of the types of the review. The defect types that are detected by the tools are analysed more thoroughly than with reviews. The effectiveness of the tools seems to strongly depend on the personal programming style and the design of the software as the results differed strongly from project to project. Finally, a combination of the usage of bug finding tools together with reviews and tests would be most advisable if the number of false positives were lower. It probably costs more time to resolve the false positives than is saved by the automation using the tools.

Therefore, the main conclusion is that bug finding tools can save costs when used together with other defect-detection techniques, if the tool developers are able to improve the tools in terms of the false positives ratio and tolerance of different programming styles.

This study is only a first indication and needs further empirical validation to be able to derive solid conclusions. For this, we plan to repeat this study on different subjects and also taking other tools into account, e.g. commercial tools or tools that use additional annotations in the source code. Also, the investigation of other types of software is important, since we only considered web applications in this study.

Difficulty. We use the results of this case study also to determine the average difficulty $\bar{\theta}$ of bug finding tools as used in the economics models of Chap. 4. As we discussed above, this can only be coarse-grained but this study is the only one available so far. After eliminating the false positives, the tools were able to find 81% of the known defects over several projects. Hence, the difficulty is only 0.19. However, the defects had mainly a low severity where severity described the impact on the execution of the software. For the severest defects – which would have high effect costs – the difficulty increased to 78%, for the second severest defects even to 80%. For lower severities the difficulty lies between 12% - 30%.

6.4 Telecommunication Systems

One input parameter of our economics model is the failure probability of faults in the field, i.e., the probability that residual faults actually lead to inaccurate behaviour. We describe a case study in the following that tests the assumption that the failure probabilities of the faults are related by an geometric progression.

6.4.1 Geometric Progression of Failure Probabilities

A validated assumption about the relationship between the failure probabilities of faults would help to judge the importance and influence of this factor on the economics

of quality assurance. This all leads back to the established work on software reliability models. However, most of them do not deal with the notion of faults explicitly but rather consider failures only. A useful classification by Miller [134] shows that most existing models are exponential order statistic (EOS) models. He further divided the models into deterministic and doubly stochastic EOS models arguing that the failure rates either have a deterministic relationship or are again randomly distributed. For the deterministic models, Miller presented several interesting special cases. The well-known Jelinski-Moranda model [92], for example, has *constant rates*. He also stated that *geometric rates* are possible as documented by Nagel [146, 147].

This geometric sequence (or progression) between failure rates of faults was also observed in projects of the communication networks department of the Siemens AG. In several older projects which were analysed, this relationship fitted well to the data. Therefore, a software reliability model based on a geometric sequence of failure rates is proposed. This tendency is also in accordance with experiences at IBM reported by Adams in [2] although he did not observe a geometric progression directly.

6.4.2 Approach

The assumed basic relationship of a geometric progression of the failure probabilities of faults is also used in a derived reliability growth model. Details on that are published in [209, 210]. This model will be called the *Fischer-Wagner* model in the following. Using this model we analysed how well the assumption of the geometric progression fitted onto real project data. For this we relied on several data sets publicly available collected during testing and on original data from the telecommunications department of Siemens from the field trial of two products.

We follow [141] and use the *number of failures approach* to analyse the validity of our model for the available failure data. We assume that there have been q failures observed at the end of test time (or field trial time) t_q. We use the failure data up to some point in time during testing$t_e (\leq t_q)$ to estimate the parameters of the mean number of failures $\mu(t)$. The substitution of the estimates of the parameters yields the estimate of the number of failures $\hat{\mu}(t_q)$. The estimate is compared with the actual number at q. This procedure is repeated with several t_es.

For a comparison we can plot the relative error $(\hat{\mu}(t_q) - q)/q$ against the normalised test time t_e/t_q. The error will approach 0 as t_e approaches t_q. If the points are positive, the model tends to overestimate and the other way round. Numbers closer to 0 imply a more accurate prediction and hence a better model.

We use as comparison four well-known models from the literature: Musa basic, Musa-Okumoto, Littlewood-Verall, and NHPP. All these models are implemented in the tool SMERFS [53] that was used to calculate the necessary predictions. We describe each model in more detail in the following.

Musa basic. The Musa basic execution time model assumes that all faults are equally likely to occur, are independent of each other and are actually observed. The execution times between failures are modelled as piecewise exponentially distributed. The intensity function is proportional to the number of faults remaining in the program and the fault correction rate is proportional to the failure occurrence rate.

Musa-Okumoto. The Musa-Okomoto model, also called logarithmic Poisson execution time model, was first described in [142]. It also assumes that all faults are equally likely to occur and are independent of each other. The expected number of faults is a logarithmic function of time in this model and the failure intensity decreases exponentially with the expected failures experienced. Finally, the software will experience an infinite number of failures in infinite time.

Littlewood-Verall Bayesian. This model was proposed for the first time in [121]. The assumptions of the Littlewood-Verall Bayesian model are that successive times between failures are independent random variables each having an exponential distribution. the distribution for the i-th failure has a mean of $1/\lambda(i)$. The $\lambda(i)$s form a sequence of independent variables, each having a gamma distribution with the parameters α and $\phi(i)$. $\phi(i)$ has either the form: $\beta(0) + \beta(1) \cdot i$ (linear) or $\beta(0) + \beta(1) \cdot i^2$ (quadratic). We used the latter.

NHPP. Various models based on a non-homogeneous Poisson process are described in [156]. The particular model used also assumes that all faults are equally likely to occur and are independent of each other. The cumulative number of faults detected at any time follows a Poisson distribution with mean $m(t)$. That mean is such that the expected number of faults in any small time interval about t is proportional to the number of undetected faults at time t. The mean is assumed to be a bounded non-decreasing function with $m(t)$ approaching in the limit, A (The expected total number of faults to be, eventually, detected in the testing process), as the length of testing goes to infinity. It is possible to use NHPP on time-between-failure data as well as failure counts. We used the time-between-failure version in our evaluation.

We apply the reliability models to several different sets of data to compare the predictive validity. Three data sets are provided by *The Data & Analysis Center for Software* of the US-American Department of Defense, the other three projects were done at Siemens. The former are called *DACS* together with their system number, the latter were given the name *Siemens* and are consecutively numbered.

6.4.3 Results

We give for each analysed project a brief description and show a plot of the relative errors of the different models. Finally, the results are combined to allow generalisations of the results.

DACS 1

The first project data under consideration had the aim of developing a real time command and control application. The size of the software measured in delivered object code instructions is $21,700$. 136 failures were observed during the system test. The system code of this project is 1. The data is based on execution time, therefore all models were easily applicable.

The plot in Fig. 6.14 shows the relative error curves for all considered models. The predictions become more and more accurate as increasingly more sample data is available which was expected. The NHPP, Musa basic, and Musa-Okumoto models all tend

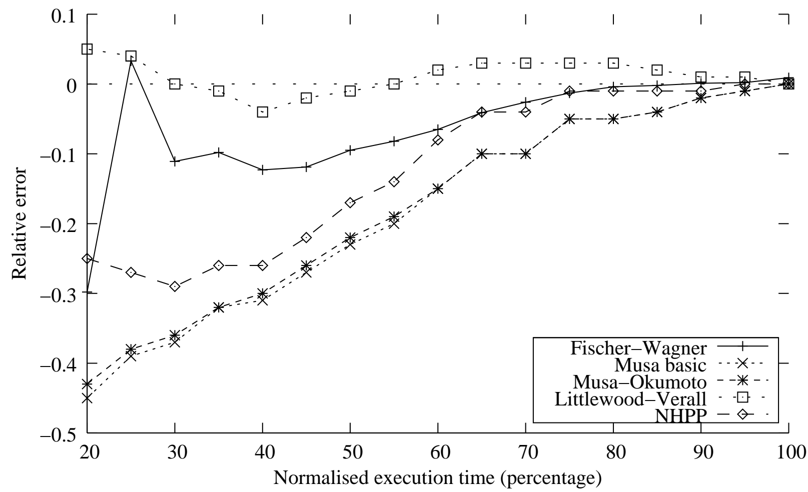


Figure 6.14: Relative error curves for the models based on the DACS 1 data set

to predict too little failures whereas the Littlewood-Verall model mostly overestimates the number of future failures. The latter model also yields the best predictions from early stages on. The Fischer-Wagner model has a quite similar predictive validity. From 80% of the time on the predictions are even more accurate than all the others although the predictions are in the beginning worse than the ones from the Littlewood-Verall model.

DACS 6

This data set comes from the subsystem test of a commercial subsystem with 5,700 delivered object code instructions. In total 73 failures occurred. We tried to use all models on this data as well but the models were not applicable at all stages. Especially the NHPP model was only able to make predictions for about half of the analysed parts of the data set. The results can be found in Fig. 6.15.

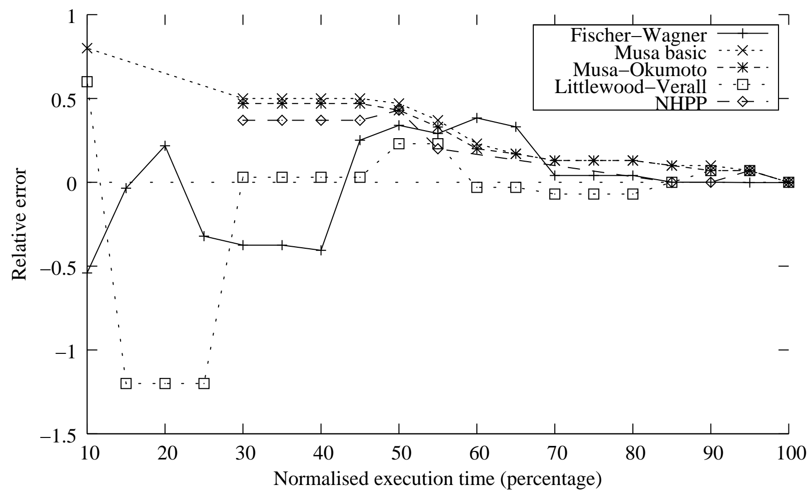


Figure 6.15: Relative error curves for the models based on the DACS 6 data set

It is obvious that the models behave strongly different. Note that the Littlewood-Verall model gives the most accurate predication although it is the worst before 30% of the execution time is over. The Fischer-Wagner model is similar to the other models. Sometimes it is able to predict more accurately (between 45% and 55%, and after 70%), sometimes the predictions are worse (between 55% and 70%).

DACS 40

The DACS 40 test data describes the results of the system test of a military application with 180,000 delivered object code instructions. The Musa basic model was not applicable to this data. All the other models perform well with relative errors not bigger than 0.25. The results are again illustrated in a diagram in Fig. 6.16.

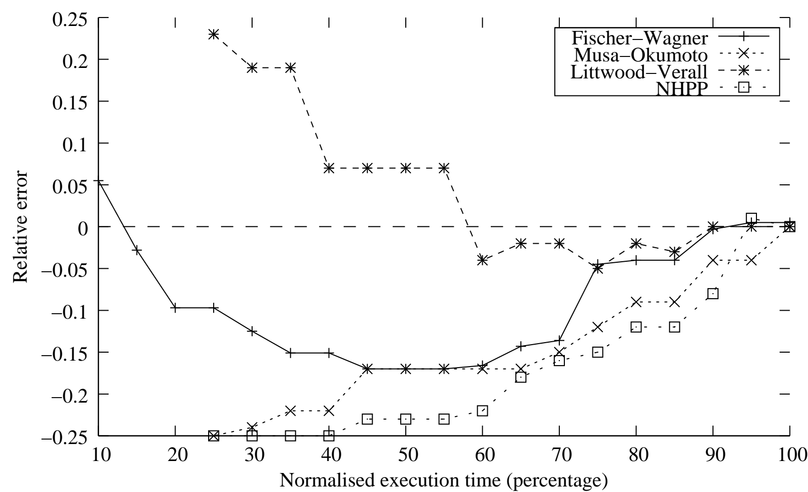


Figure 6.16: Relative error curves for the models based on the DACS 40 data set

For this data the Musa-Okumoto model is able to make the best predictions with an relative error of about 0.05 after 40% of the execution time. The Fischer-Wagner model is able to outperform the others in the early stages but is worse than the Musa-Okumoto model from 35% onwards. However, beginning at 75% the predictive validity is similar.

Siemens 1

This data comes from a large Siemens project that we call *Siemens 1* in the following. The software is used in a telecommunication equipment.

We only look at the field trial because this gives us a rather accurate approximation of the execution time which is the actually interesting measure regarding software. It is a good substitute because the usage is nearly constant during field trial. Based on the detailed dates of failure occurrence, we cumulated the data and converted it to time-between-failure (TBF) data. This was then used with the Fischer-Wagner, Musa-basic, Musa-Okumoto, and NHPP models. The results can be seen in Fig. 6.17. In this case we omit the Littlewood-Verall that made totally absurd predictions of over a thousand future failures.

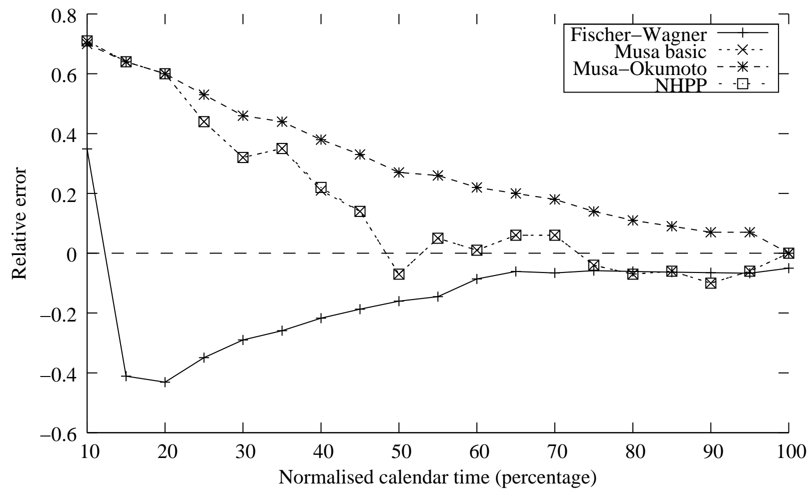


Figure 6.17: Relative error curves for the models based on the Siemens 1 data set

The Musa-basic and the NHPP models yield similar results all the time. They overestimate in the beginning and slightly underestimate in the end. The Musa-Okumoto model overestimates all the time, the Fischer-Wagner model underestimates. All models make again reasonably well predictions. The Fischer-Wagner model has a relative error below 0.2 from 45% on, the Musa basic and the NHPP models even from 40% on.

Siemens 2

Siemens 2 is a web application for which we only have a small number of field failures. This makes predictions more complicated as the sample size is smaller. However, it is interesting to analyse how the different models are able to cope with this. For this, we have plotted the results in Fig. 6.18.

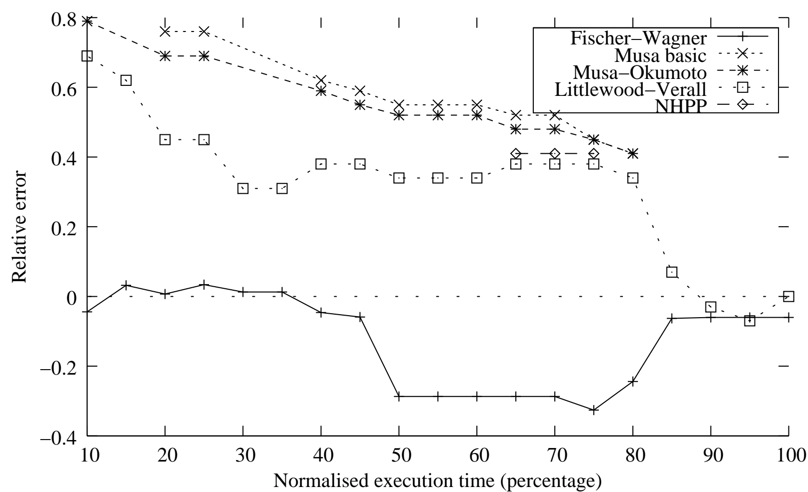


Figure 6.18: Relative error curves for the models based on the Siemens 2 data set

Again not all models were applicable to this data set. The NHPP model only made predictions for a small number of data points, the Musa basic and the Musa-Okumoto models were usable mainly in the middle of the execution time. All models made comparably bad predictions as we expected because of the small sample size. Surprisingly, the Fischer-Wagner model did extremely well in the beginning but worsened in the middle until its prediction became accurate in the end again. Despite this bad performance in the middle of the execution time it is still the model with the best predictive validity in this case. Only the Littlewood-Verall model comes close to these results. This might be an indication that the Fischer-Wagner model is good suited for smaller sample sizes.

Combined Results

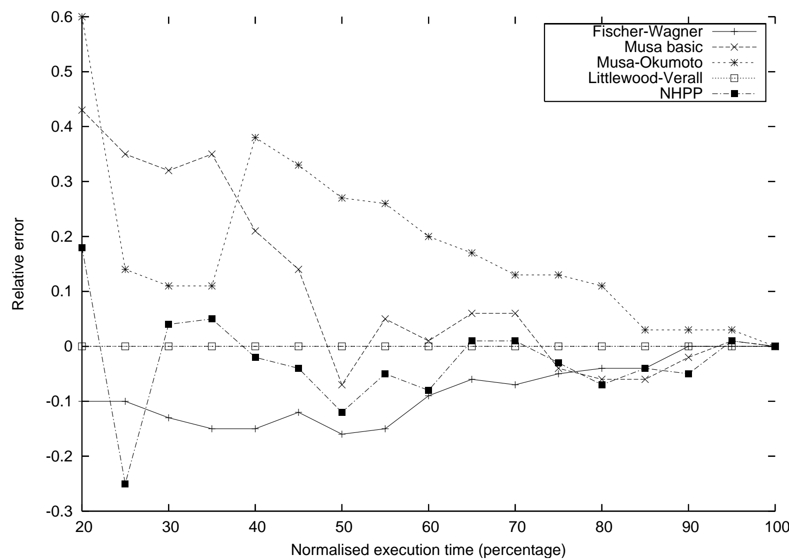


Figure 6.19: Median relative errors for the different models based on all analysed data sets

The usage of the number of failures approach for each project resulted in different curves for the predictive validity over time. For a better general comparison we combined the data into one plot which can be found in Fig. 6.19. This combination is straight-forward as we only considered relative time and relative errors. To avoid that strongly positive and strongly negative values combined give very small errors we use medians instead of average values. The plot shows that with regard to the analysed projects the Littlewood-Verall model gives very accurate predictions, also the NHPP and the proposed model are strong from early on.

6.4.4 Summary

Several data sets from DACS and Siemens are used to evaluate the predictive validity of the model in comparison to well-established models. We find that the model based on the geometric progression of failure probabilities of faults often has a similar predictive

validity as the comparison models and outperforms most of them. However, there is always one of the models that performs better than ours. Nevertheless, we are able to validate the assumption that a geometric progression is reasonable because we have a good performance over all analysed projects in contrast to other models.

Failure Probability in the Field. We can use this geometric progression for calibrating the analytical model from Chap. 4. The input factor *failure probability in the field* per fault is hard to determine because it involves careful measurement. This basic relationship between the different faults simplifies this determination because only the values for some of the faults needs to be determined. The other failure probabilities can then be calculated using the geometric progression.

7 Summary and Outlook

In this final chapter we summarise the contributions of this dissertation and put them into context. We also describe our current work and directions for possible further research.

7.1 Summary

Costs and benefits are clearly a central factor in planning software quality assurance. In the end, all commercial software needs to generate profit for the company that developed it. As we saw, the analytical quality assurance typically accounts for about 50% of the costs during development. Hence, we need to plan carefully and based on solid facts. However, those decisions are still mainly made using intuition [187].

To be able to transfer the intuitive process to a more formal, fact-based approach, we need to describe the relationships of the influencing factors in an economic model. This would help decision-maker to plan SQA and analyse the effects of their decisions. We propose an analytical model of the quality economics of defect-detection techniques. The reason for the focus on defect-detection techniques (or analytical QA) is that constructive QA has a different characteristic because it *prevents* defects from being introduced. This makes the measurement process more difficult and other mechanisms would have to be introduced into the model. Moreover, defect-detection techniques constitute about 50% of the total development costs. Hence, an improvement or optimisation in this area can already be seen as beneficial.

The proposed model is more detailed than available economics models for SQA but more general than technique-specific efficiency models. The former are derived from other industrial areas – mainly manufacturing – and concentrate on factors such as the cost of capital and use only coarse-grained parameters of the software. The latter stem from detailed and technical observations using specific techniques. These are mainly reliability growth models for system testing and efficiency models for inspections. The problem here is that they do not suffice to plan the whole quality assurance.

Our economics model actually consists of two versions: (1) an ideal or theoretical model and (2) a simple or practical model. The ideal model incorporates many important factors that we identified of having an influence on the costs and benefits of SQA. The aim is to analyse the differences between defect-detection techniques with respect to these factors and to analyse which are the most important factors. The derived practical model has a reduced set of input factors and is based on defect types to be applicable in planing and optimising the quality assurance in real projects. For all the input factors of the model, we reviewed the available empirical literature and synthesised the data to have average values for the model parameters. These values can be used (1) as first estimates in practical application and (2) for sensitivity analysis. A global sensitivity analysis of the models shows that the most important factors w.r.t.

the variance in the output of the model are the removal costs of defects in the field, the distribution of document (or artefact) types, the average difficulty (or effectiveness) of the used techniques, and the execution costs. Hence, further investigations on theses factors are most beneficial. The sensitivity analysis also has the result that the average labour costs, the sequence of application, and the failure probability in the field do not seem to have a large influence. Therefore, approximations of those factors are sufficient for precise predictions.

All the discussion so far viewed the system of which the quality is assured as a whole. However, there is a possibility to optimise the quality assurance on the architectural level. In particular, defect-detection techniques can be concentrated on components that are most defect-prone, i.e., are likely to contain the most defects. There are existing approaches to that but we propose a *model-based* identification of defect-prone components. This involves a metrics suite that is applied to detailed design models of the software to be built. The applicability of this suite to models allows an early planning of quality assurance, i.e., we do not need to wait until we can collect code metrics.

Several case studies were carried out with three different aims: (1) investigation of different factors of the model (mainly difficulty) of different defect-detection techniques, (2) validation of the metrics suite to identify defect-prone components, and (3) analyse the factor failure probability in the field of the model. In particular, we validated the metrics suite on a model of an automatic collision notification system for automobiles and a model of the network controller of an infotainment network for modern automobiles. In both studies the approach was able to predict the most fault-prone components correctly. The network controller case study was mainly aiming at analysing model-based testing. Hence, we evaluated its effectiveness (with and without automation) and compared it to hand-crafted and random tests. The effectiveness of simple static analysis tools for Java was analysed over several projects of a mobile communication company. Finally, based on the data of a telecommunications company we showed that a geometric progression is a good model for the failure probability of faults in the field. All this information was used in the above mentioned synthesis of the available empirical work. Especially on static analysis tools and model-based testing there is only little empirical knowledge so far.

7.2 Outlook

We first discuss some possible practical and theoretical improvements of the analytical model of SQA and the metrics suite for the prediction of defect-prone components. Finally, we propose – on a high level of abstraction – an integrated approach to quality modelling and management.

7.2.1 Practical and Theoretical Improvements

The analytical model as well as the metrics suite can be improved theoretically and also need more practical application to analyse their usefulness.

Practical Application

The main goal for future research is to apply the model in practice; in particular to evaluate its predictive validity. However, this is a major project because the model relies on historical data and probably needs to be calibrated over several projects before reasonable predictions are possible. Nevertheless, this is the most important validation of the practical model. Currently a cost analysis of static analysis tools is done in cooperation with a company extending the case study that evaluated the effectiveness and defect types.

As we identified the factors of the model that are most beneficial to evaluate further, we plan to do empirical studies on these factors. In the mentioned study on static analysis tools, we concentrate on the removal costs of defects in the field, as there is clearly a lack of empirical data. Also the distribution of defects over different artefact types has been largely neglected so far. So, studies on this factor would have a strong impact.

A further validation of the metrics suite for identifying defect-prone components is needed. Two case studies merely show the applicability of the approach and give hints but are no total empirical validation. Therefore, we currently carry out another case study that evaluates this predictive validity.

Theoretical Improvement

On the theoretical side, an incorporation and analysis of further influencing factors would be interesting. Especially, the time to market and the costs of capital are two factors that often play an significant role in economics models. By incorporation and a repeated sensitivity analysis we could show whether these factors are really important in our context and we need to investigate them further. Also a consideration of false positives as in Sabaliauskaite et al. [182,183] might be beneficial for a better evaluation of inspections and static analysis tools. Moreover, it would be interesting to move from simple expected values to full distributions in the model. It would allow to assess also the best and the worst cases and hence contribute to risk analysis.

To further improve the benefit of applying the defect-proneness approach we plan to incorporate other factors such as the severity of the potential failures from the components. Also more sophisticated statistical analyses, such as the ones used for code metrics in [4], might be used to improve the predictions.

7.2.2 An Integrated Approach

Parallel to improving and validating the proposed approaches, it is necessary to develop an integrated view on quality modelling and management. The research area is currently dominated by isolated solutions that all have a value on their own but for an useful application there should be a coordinated, or better an integrated, approach. For this, we identify important characteristics of quality models, develop a general approach to integrated quality modelling, and give a concrete example how different models could be used together.

Characteristics

We discussed in Sec. 2.4 different important dimensions of quality models. One dimension is the *quality view* they support such as user-based or value-based. Furthermore, we distinguished between three main types of quality models:

- Constructive: We see constructive models as explanations of relationships between constructive actions and some aspects of software quality. For example, the use of a different programming language might influence the reliability of the system in a certain way. These relationships are probably very coarse-grained but help to understand and to choose from the possibilities during development.
- Predictive: The predictive models help to plan the future development of some quality aspects and hence are used to plan the quality assurance.
- Assessing: The assessing models allow to estimate the current state of the software to control the quality assurance.

Finally, quality models can have several dimensions of specificness. We identified that models can be *general* or *product-specific*. Models can also concentrate on a specific defect-detection technique (technique-specific), a specific phase in the development process (phase-specific), or a specific quality attribute (attribute-specific).

Integrated Quality Modelling and Management

The difficulty is (1) to choose the right models and (2) to integrate them usefully. For both purposes, we believe that an abstract quality model or a quality *meta-model* could help. This meta-model describes the commonalities and abstract concepts of the possibly used concrete quality models. Based on this abstract quality model we can define the process of quality modelling and management as depicted in Fig. 7.1.

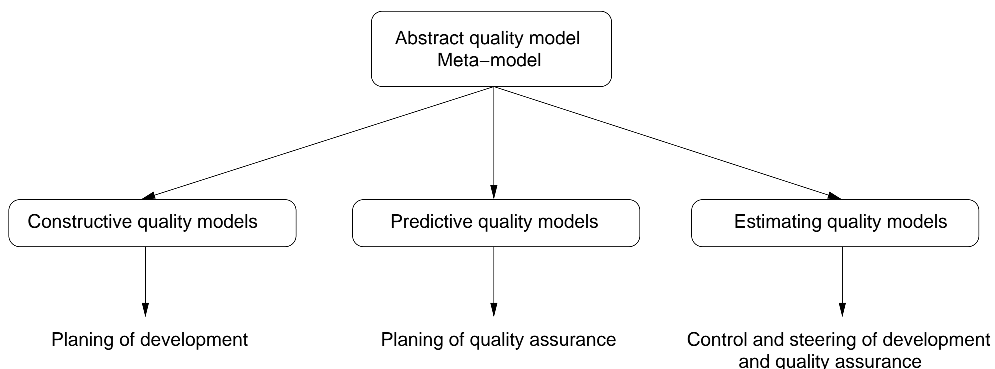


Figure 7.1: The integrated process of quality modelling and management

Using the abstract quality model, we perform three activities:

1. Identification of relevant views on quality
2. Identification of important quality attributes

3. Defining the desired quality goals

The first activity is related to the views on quality as discussed above. We have to analyse the market of our product and identify the views on quality that are most important there, e.g., a user-based view. We also need to use market analysis to identify the quality attributes that have the strongest influence and to find out which attributes are of no importance. For example, in some applications security or safety is not an issue. Finally, based on the two previous decisions, we define for the quality attributes the goals that we want to reach with our product. Using this information in the meta-model, we can identify the concrete quality models that fit to our needs. A further important decision is also to choose whether a product-specific model is needed or a general model is sufficient. For attributes and views that are considered less important, general models can give enough information to control the project. For important attributes and views, there should be product-specific models in place.

Firstly, constructive quality models give guidelines for the development that help to reach the quality goals. For example, the decision of which programming language to use could be based on such a model. In general, these models are a tool that helps in planing the development. Furthermore, predictive models – possibly for each relevant quality attribute – need to be identified using the meta-model. They are used to plan the quality assurance. Early in the development process, these might be more general models such as the one proposed in Chap. 4 but later more specific models such as reliability growth models are more useful. Finally, during the development and quality assurance, we use estimating quality models that assess the current state of the product and process. This enables us to control and steer the development project based on real data. The integration of the different models is always based on the meta-model that needs to define how these different predictions and estimates fit together. This is probably one of the main challenges for research in the integrated approach we propose here.

Bibliography

[1] A. Frank Ackerman, Lynne S. Buchwald, and Frank H. Lewski. Software Inspections: An Effective Verification Process. *IEEE Software*, 6(3):31–36, 1989.

[2] Edward N. Adams. Optimizing Preventive Service of Software Products. *IBM Journal of Research and Development*, 28(1):2–14, 1984.

[3] David S. Alberts. The economics of software quality assurance. In Edward Miller and William E. Howden, editors, *Tutorial: Software Testing & Validation Techniques*. IEEE Computer Society, 1978.

[4] Hirohisa Aman, Naomi Mochiduki, and Hiroyuki Yamada. A Model for Detecting Cost-Prone Classes Based on Mahalanobis-Taguchi Method. *IEICE Transactions on Information and Systems*, E89-D(4):1347–1358, 2006.

[5] Algirdas Avižienis, Jean-Claude Laprie, and Brian Randell. Fundamental concepts of dependability. Technical Report N01145, LAAS-CNRS, 2001.

[6] Thomas Ball and Sriram K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *Proc. 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3. ACM Press, 2002.

[7] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.

[8] Victor R. Basili and Richard W. Selby. Comparing the Effectiveness of Software Testing Strategies. *IEEE Transactions on Software Engineering*, SE-13(12):1278–1296, 1987.

[9] Boris Beizer. *Software Testing Techniques*. Thomson Learning, 2nd edition, 1990.

[10] Stefan Biffl. Hierarchical Economic Planning of the Inspection Process. In *Proc. Third International Workshop on Economics-Driven Software Engineering Research (EDSER-3)*. IEEE Computer Society Press, 2001.

[11] Stefan Biffl, Bernd Freimut, and Oliver Laitenberger. Investigating the Cost-Effectiveness of Reinspections in Software Development. In *Proc. 23rd International Conference on Software Engineering (ICSE '01)*, pages 155–164. IEEE Computer Society, 2001.

[12] Stefan Biffl and Michael Halling. Investigating the Defect Detection Effectiveness and Cost Benefit of Nominal Inspection Teams. *IEEE Transactions on Software Engineering*, 29(5):385–397, 2003.

[13] Stefan Biffl, Michael Halling, and Monika Köhle. Investigating the effect of a second software inspection cycle: Cost-benefit data from a large-scale experiment on reinspection of a software requirements document. In *Proc. First Asia-Pacific Conference on Quality Software (APAQS '00)*, pages 194–203. IEEE Computer Society, 2000.

[14] C. Billings, J. Clifton, B. Kolkhorst, E. Lee, and W. B. Wingert. Journey to a Mature Software Process. *IBM Systems Journal*, 33(1):46–61, 1994.

[15] Robert V. Binder. *Testing Object-Oriented Systems. Models, Patterns, and Tools*. The Addison-Wesley Object Technology Series. Addison-Wesley, 2000.

[16] Aaron B. Binkley and Stephen R. Schach. Validation of the Coupling Dependency Metric as a Predictor of Run-Time Failures and Maintenance Measures. In *Proc. 20th International Conference on Software Engineering (ICSE '98)*, pages 452–455. IEEE Computer Society, 1998.

[17] Leland T. Blank and Anthony J. Torquin. *Engineering Economy*. Series in Industrial Engineering and Management Science. McGraw-Hill, 4th edition, 1998.

[18] James Kenneth Blundell, Mary Lou Hines, and Jerrold Stach. The Measurement of Software Design Quality. *Annals of Software Engineering*, 4:235–255, 1997.

[19] Barry Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[20] Barry Boehm and Victor R. Basili. Software Defect Reduction Top 10 List. *IEEE Computer*, 34(1):135–137, 2001.

[21] Barry Boehm, LiGuo Huang, Apurva Jain, and Ray Madachy. The ROI of Software Dependability: The iDAVE Model. *IEEE Software*, 21(3):54–61, 2004.

[22] Barry W. Boehm, John R. Brown, Hans Kaspar, Myron Lipow, Gordon J. MacLeod, and Michael J. Merrit. *Characteristics of Software Quality*, volume 1 of *TRW Series of Software Technology*. North-Holland Publishing, 1978.

[23] Philip J. Boland, Harshinder Singh, and Bojan Cukic. Comparing Partition and Random Testing via Majorization and Schur Functions. *IEEE Transactions on Software Engineering*, 29(1):88–94, 2003.

[24] Andrea Bondavalli, Mario Dal Cin, Diego Latella, István Majzik, András Pataricza, and Giancarlo Savoia. Dependability Analysis in the Early Phases of UML Based System Design. *Journal of Computer Systems Science and Engineering*, 16(5):265–275, 2001.

[25] Andrea Bondavalli, Ivan Mura, and István Majzik. Automated Dependability Analysis of UML Designs. In *Proc. Second IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 139–144. IEEE Computer Society, 1999.

[26] Jewgeni Botaschanjan, Leonid Kof, Christian Kühnel, and Maria Spichkova. Towards Verified Automotive Software. In *Proc. 2nd International Workshop on Automotive Software Engineering*. ACM Press, 2005.

[27] Karen V. Bourgeois. Process Insights from a Large-Scale Software Inspections Data Analysis. *CrossTalk. The Journal of Defense Software Engineering*, 9(10):17–23, 1996.

[28] Max Breitling. *Formale Fehlermodellierung für verteilte reaktive Systeme*. PhD Dissertation, Technische Universität München, 2001.

[29] Lionel Briand, Khaled El Emam, Oliver Laitenberger, and Thomas Fussbroich. Using Simulation to Build Inspection Efficiency Benchmarks for Development Projects. In *Proc. 20th International Conference on Software Engineering (ICSE '98)*, pages 340–349. IEEE Computer Society, 1998.

[30] Manfred Broy, Florian Deißenböck, and Markus Pizka. Demystifying Maintainability. In *Proc. 4th Workshop on Software Quality (4-WoSQ)*, pages 21–26. ACM Press, 2006.

[31] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *LNCS*. Springer, 2005.

[32] Bundesrepublik Deutschland. *V-Modell XT*, 1.2 edition, 2006. http://www.v-modell-xt.de/.

[33] Marilyn Bush. Improving Software Quality: The Use of Formal Inspections at the JPL. In *Proc. 12th International Conference on Software Engineering (ICSE '90)*, pages 196–199. IEEE Computer Society, 1990.

[34] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A Static Analyzer for Finding Dynamic Programming Errors. *Software: Practice and Experience*, 30(7):775–802, 2000.

[35] David N. Card and William W. Agresti. Measuring Software Design Complexity. *The Journal of Systems and Software*, 8:185–197, 1988.

[36] Michelle Cartwright and Martin Shepperd. An Empirical Investigation of an Object-Oriented Software System. *IEEE Transactions on Software Engineering*, 26(8):786–796, 2000.

[37] María Victoria Cengarle. Inspection and Testing – Towards combining both approaches. Technical Report 024.02/E, Fraunhofer IESE, 2002.

[38] T.Y. Chen and Y.T. Yu. On the Relationship Between Partition and Random Testing. *IEEE Transactions on Software Engineering*, 12(12):977–980, 1994.

[39] Shyam R. Chidamber and Chris F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[40] Ram Chillarege. Orthogonal Defect Classification. In Michael R. Lyu, editor, *Handbook of Software Reliability Engineering*, chapter 9. IEEE Computer Society Press and McGraw-Hill, 1996.

[41] Sunita Chulani and Barry Boehm. Modeling Software Defect Introduction and Removal: COQUALMO (COnstructive QUALity MOdel). Technical Report USC-CSE-99-510, University of Southern California, Center for Software Engineering, 1999.

[42] CMMI Product Team. CMMI^{SM} for Systems Engineering/Software Engineering/Integrated Product and Process Development/Supplier Sourcing, Version 1.1, Continuous Representation. Technical Report CMU/SEI-2002-TR-011, Carnegie-Mellon University, 2002.

[43] James S. Collofello and Scott N. Woodfield. Evaluating the Effectiveness of Reliability-Assurance Techniques. *Journal of Systems and Software*, 9(3):191–195, 1989.

[44] Deutschland schlampt bei Softwarequalität. *Computer Zeitung*, 35(21), 2005. In German.

[45] Harris Cooper. *Synthesizing Research. A Guide for Literature Reviews*, volume 2 of *Applied Social Research Methods Series*. SAGE Publications, third edition, 1998.

[46] Christoph Csallner and Yannis Smaragdakis. Check 'n' Crash: Combining Static Checking and Testing. In *Proc. 27th International Conference on Software Engineering (ICSE '05)*, pages 422–431. ACM Press, 2005.

[47] Sunita Devnani-Chulani. *Bayesian Analysis of Software Cost and Quality Models*. PhD dissertation, University of Southern California, 1999.

[48] Bruce P. Douglass. Computing Model Complexity. Available at http://www.ilogix.com/whitepapers/whitepapers.cfm, 2004.

[49] João Durães and Henrique Madeira. Definition of Software Fault Emulation Operators: A Field Data Study. In *Proc. 2003 International Conference on Dependable Systems and Networks (DSN '03)*, pages 105–114. IEEE Computer Society, 2003.

[50] Dawson Engler and Madanlal Musuvathi. Static Analysis versus Model Checking for Bug Finding. In *Proc. Verification, Model Checking and Abstract Interpretation (VMCAI '04)*, volume 2937 of *LNCS*, pages 191–210. Springer, 2002.

[51] Michael E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 15(3):182–211, 1976.

[52] Michael E. Fagan. Reviews and Inspections. In Manfred Broy and Ernst Denert, editors, *Software Pioneers – Contributions to Software Engineering*, pages 562–573. Springer, 2002.

[53] William H. Farr and Oliver D. Smith. Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS) Users Guide. Technical Report NAVSWC TR-84-373, Naval Surface Weapons Center, 1993.

[54] Norman Fenton. Software Measurement: A Necessary Scientific Basis. *IEEE Transactions on Software Engineering*, 20(3):199–206, 1994.

[55] Norman E. Fenton and Niclas Ohlsson. Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Transactions on Software Engineering*, 26(8):797–814, 2000.

[56] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics. A Rigorous & Practical Approach*. International Thomson Publishing, 2nd edition, 1997.

[57] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *Proc. 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245. ACM Press, 2002.

[58] Phyllis Frankl, Dick Hamlet, Bev Littlewood, and Lorenzo Strigini. Choosing a Testing Method to Deliver Reliability. In *Proc. 19th International Conference on Software Engineering*, pages 68–78. ACM Press, 1997.

[59] Phyllis G. Frankl and Elaine J. Weyuker. A Formal Analysis of the Fault-Detecting Ability of Testing Methods. *IEEE Transactions on Software Engineering*, 19(3):202–213, 1993.

[60] Phyllis G. Frankl and Elaine J. Weyuker. Provable Improvements on Branch Testing. *IEEE Transactions on Software Engineering*, 19(10):962–975, 1993.

[61] Louis A. Franz and Jonathan C. Shih. Estimating the Value of Inspections and Early Testing for Software Projects. *Hewlett-Packard Journal*, 45(6):60–67, 1994.

[62] Bernd Freimut. Developing and Using Defect Classification Schemes. Technical Report 072.01/E, Fraunhofer IESE, 2001.

[63] Bernd Freimut, Lionel C. Briand, and Ferdinand Vollei. Determining Inspection Cost-Effectiveness by Combining Project Data and Expert Opinion. *IEEE Transactions on Software Engineering*, 31(12):1074–1092, 2005.

[64] Bernd Freimut, Christian Denger, and Markus Ketterer. An Industrial Case Study of Implementing and Validating Defect Classification for Process Improvement and Quality Management. In *Proc. 11th IEEE International Software Metrics Symposium (METRICS '05)*. IEEE Computer Society, 2005.

[65] Daniel Galin. *Software Quality Assurance*. Pearson, 2004.

[66] Daniel Galin. Toward an Inclusive Model for the Costs of Software Quality. *Software Quality Professional*, 6(4):25–31, 2004.

[67] David A. Garvin. What Does “Product Quality” Really Mean? *MIT Sloan Management Review*, 26(1):25–43, 1984.

[68] Christophe Gaston and Dirk Seifert. Evaluating Coverage Based Testing. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *LNCS*, pages 293–322. Springer, 2005.

[69] Tom Gilb and Dorothy Graham. *Software Inspection*. Addison-Wesley, 1993.

[70] Robert B. Grady. *Practical Software Metrics for Project Management and Process Improvement*. Prentice-Hall, 1992.

[71] Robert B. Grady and Tom Van Slack. Key Lessons in Achieving Widespread Inspection Use. *IEEE Software*, 11(4):46–57, 1994.

[72] Richard G. Hamlet. Theoretical Comparison of Testing Methods. In *Proc. 3rd ACM SIGSOFT Symposium on Testing, Analysis, and Verification*, pages 28–37. ACM Press, 1989.

[73] Mary E. Helander, Ming Zhao, and Niclas Ohlsson. Planning Models for Software Reliability and Cost. *IEEE Transactions on Software Engineering*, 24(6):420–434, 1998.

[74] Sallie Henry and Calvin Selig. Predicting Source-Code Complexity at the Design Stage. *IEEE Software*, 7:36–44, 1990.

[75] William C. Hetzel. *An Experimental Analysis of Program Verification Methods*. PhD thesis, University of North Carolina at Chapel Hill, 1976.

[76] Gerard J. Holzmann. Economics of Software Verification. In *Proc. 2001 ACM SIGPLAN–SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, pages 80–89. ACM Press, 2001.

[77] David Hovemeyer and William Pugh. Finding Bugs is Easy. *SIGPLAN Notices*, 39(12):92–106, 2004.

[78] William E. Howden. Theoretical and Empirical Studies of Program Testing. *IEEE Transactions on Software Engineering*, SE-4(4):293–298, 1978.

[79] Chin-Yu Huang and Michael R. Lyu. Optimal Release Time for Software Systems Considering Cost, Testing-Effort, and Test Efficiency. *IEEE Transactions on Reliability*, 54(4):583–591, 2005.

[80] LiGuo Huang and Barry Boehm. Determining How Much Software Assurance is Enough? A Value-based Approach. In *Proc. Seventh Workshop on Economics-Driven Software Research (EDSER-7)*, 2005.

[81] Franz Huber, Bernhard Schätz, and Geralf Einert. Consistent Graphical Specification of Distributed Systems. In *Proc. Formal Methods Europe*, volume 1641 of *LNCS*, pages 122–141. Springer, 1997.

[82] Franz Huber, Bernhard Schätz, Alexander Schmidt, and Katharina Spies. AutoFocus: A Tool for Distributed Systems Specification. In *Proc. 4th International Symposium Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT '96)*, volume 1135 of *LNCS*, pages 467–470. Springer, 1996.

[83] Watts S. Humphrey. *A Discipline for Software Engineering*. The SEI Series in Software Engineering. Addison-Wesley, 1995.

[84] Gábor Huszerl, István Majzik, András Pataricza, Konstantinos Kosmidis, and Mario Dal Cin. Quantitative Analysis of UML Statechart Models of Dependable Systems. *The Computer Journal*, 45(3):260–277, 2002.

[85] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. 16th International Conference on Sofware Engineering (ICSE '94)*, pages 191–200. IEEE Computer Society Press, 1994.

[86] G. H. Hwang and E. M. Aspinwall. Quality costs models and their application: A review. *Total Quality Management*, 7(3):267–281, 1996.

[87] IEEE Std 1044-1993. *IEEE Standard Classification for Software Anomalies*, 1993.

[88] IEEE Std 610.12-1990. *IEEE Standard Glossary of Software Engineering Terminology*, 1990.

[89] Peter In, SangEun Kim, and Matthew Barry. UML-based Object-Oriented Metrics for Architecture Complexity Analysis. In *Proc. Ground System Architectures Workshop (GSAW '03)*. The Aerospace Corporation, 2003.

[90] ISO/IEC TR 15504:1998. *Software Process Assessment*, 1998.

[91] Pankaj Jalote and Bijendra Vishal. Optimal Resource Allocation for the Quality Control Process. In *Proc. 14th International Symposium on Software Reliability Engineering (ISSRE '03)*. IEEE Computer Society, 2003.

[92] Z. Jelinski and Paul B. Moranda. Software Reliability Research. In W. Freiberger, editor, *Statistical Computer Performance Evaluation*. Academic Press, 1972.

[93] Rob Johnson and David Wagner. Finding User/Kernel Pointer Bugs With Type Inference. In *Proc. 13th USENIX Security Symposium*, pages 119–134, 2004.

[94] Capers Jones. *Applied Software Measurement: Assuring Productivity and Quality*. McGraw-Hill, 1991.

[95] Capers Jones. *Software Assessments, Benchmarks, and Best Practices*. Addison-Wesley Information Technology Series. Addison-Wesley, 2000.

[96] W. D. Jones and M. A. Vouk. Field Data Analysis. In M. R. Lyu, editor, *Handbook of Software Reliability Engineering*, chapter 11. IEEE Computer Society Press and McGraw-Hill, 1996.

[97] Joseph M. Juran and A. Blanton Godfrey. *Juran's Quality Handbook*. McGraw-Hill Professional, 5th edition, 1998.

[98] Natalia Juristo, Ana M. Moreno, and Sira Vegas. Reviewing 25 Years of Testing Technique Experiments. *Empirical Software Engineering*, 9:7–44, 2004.

[99] Natalia Juristo and Sira Vegas. Functional Testing, Structural Testing and Code Reading: What Fault Type Do They Each Detect. In Reidar Conradi and Alf Inge Wang, editors, *Empirical Methods and Studies in Software Engineering: Experiences from ESERNET*, volume 2765 of *LNCS*, pages 208–232. Springer, 2003.

[100] Jan Jürjens and Stefan Wagner. Component-based Development of Dependable Systems with UML. In C. Atkinson, C. Bunse, H.-G. Gross, and C. Peper, editors, *Component-Based Software Development for Embedded Sytems. An Overview on Current Research Trends*, volume 3778 of *LNCS*, pages 320–344. Springer, 2005.

[101] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley, 2nd edition, 2002.

[102] Stephen H. Kan, S. D. Dull, D. N. Amundson, R. J. Lindner, and R. J. Hedger. AS/400 software quality management. *IBM Systems Journal*, 33(1):62–88, 1994.

[103] Cem Kaner. Quality Cost Analysis: Benefits and Risks. *Software QA*, 3(1):23–27, 1996.

[104] John C. Kelly, Joseph S. Sherif, and Jonathan Hops. An Analysis of Defect Densities found during Software Inspections. *Journal of Systems and Software*, 17(2):111–117, 1992.

[105] Taghi M. Khoshgoftaar and Timothy G. Woodcock. Predicting Software Development Errors Using Software Complexity Metrics. *IEEE Journal on Selected Areas in Communications*, 8(2):253–261, 1990.

[106] Stephen T. Knox. Modeling the costs of software quality. *Digital Technical Journal*, 5(4):9–16, 1993.

[107] Claudia Koller. *Vergleich verschiedener Methoden zur analytischen Qualitätssicherung*. Diploma Thesis, Technische Universität München, 2004. In German.

[108] Herb Krasner. Using the Cost of Quality Approach for Software. *CrossTalk. The Journal of Defense Software Engineering*, 11(11):6–11, 1998.

[109] Mayuram S. Krishnan. *Cost and Quality Considerations in Software Product Management*. PhD thesis, Carnegie Mellon University, 1996.

[110] Philippe Kruchten. *The Rational Unified Process. An Introduction*. Addison-Wesley, 2nd edition, 2000.

[111] Shinji Kusumoto. *Quantitative Evaluation of Software Reviews and Testing Processes*. PhD dissertation, Osaka University, 1993.

[112] Shinji Kusumoto, Ken ichi Matasumoto, Tohru Kikuno, and Koji Torii. A New Metric for Cost Effectiveness of Software Reviews. *IEICE Transactions on Information and Systems*, E75-D(5):674–680, 1992.

[113] Oliver Laitenberger. Studying the Effects of Code Inspection and Structural Testing on Software Quality. Technical Report 024.98/E, Fraunhofer IESE, 1998.

[114] Oliver Laitenberger. Studying the Effects of Code Inspection and Structural Testing on Software Quality. In *Proc. Ninth International Symposium on Software Reliability Engineering (ISSRE '98)*, pages 237–246. IEEE Computer Society Press, 1998.

[115] Oliver Laitenberger. A Survey of Software Inspection Technologies. In *Handbook on Software Engineering and Knowledge Engineering*, volume 2, pages 517–555. World Scientific Publishing, 2002.

[116] Jean-Claude Laprie. Dependable Computing and Fault Tolerance: Concepts and Terminology. In *Proc. 15th International Symposium on Fault-Tolerant Computing (FTCS-15)*, pages 2–11. IEEE, 1985.

[117] Marek Leszak, Dewayne E. Perry, and Dieter Stoll. A Case Study in Root Cause Defect Analysis. In *Proc. International Conference on Software Engineering (ICSE '00)*, pages 428–437. ACM Press, 2000.

[118] Peter Liggesmeyer. *Wissensbasierte Qualitätsassistenz zur Konstruktion von Prüfstrategien für Software-Komponenten*. PhD thesis, Ruhr-Universität Bochum, 1993. In German.

[119] Peter Liggesmeyer. A Set of Complexity Metrics for Guiding the Software Test Process. *Software Quality Journal*, 4(4):257–273, 1995.

[120] Bev Littlewood, Peter T. Popov, Lorenzo Strigini, and Nick Shryane. Modeling the Effects of Combining Diverse Software Fault Detection Techniques. *IEEE Transactions on Software Engineering*, 26(12):1157–1167, 2000.

[121] Bev Littlewood and J.L. Verall. A Bayesian Reliability Growth Model for Computer Software. *Applied Statistics*, 22(3):332–346, 1973.

[122] Robyn R. Lutz and Inés Carmen Mikulski. Empirical Analysis of Safety-Critical Anomalies During Operations. *IEEE Transactions on Software Engineering*, 30(3):172–180, 2004.

[123] Michael R. Lyu, editor. *Handbook of Software Reliability Engineering*. IEEE Computer Society Press and McGraw-Hill, 1996.

[124] Michael R. Lyu, Sampath Rangarajan, and Aad P. A. van Moorsel. Optimal Allocation of Test Resources for Software Reliability Growth Modeling in Software Development. *IEEE Transactions on Reliability*, 51(2):183–192, 2002.

[125] Yashwant K. Malaiya. The Relationship Between Test Coverage and Reliability. In *Proc. International Symposium on Software Reliability Engineering (ISSRE '94)*, pages 186–195. IEEE Computer Society, 1994.

[126] William A. Mandeville. Software Costs of Quality. *IEEE Journal on Selected Areas in Communications*, 8(2):315–318, 1990.

[127] Tobias Mayer and Tracy Hall. A Critical Analysis of Current OO Design Metrics. *Software Quality Journal*, 8:97–110, 1999.

[128] R. G. Mays, C. L. Jones, G. J. Holloway, and D. P. Studinski. Experiences with Defect Prevention. *IBM Systems Journal*, 29(1):4–32, 1990.

[129] Thomas J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 5:45–50, 1976.

[130] Thomas McGibbon. A Business Case for Software Process Improvement Revised. A DACS State-of-the-Art Report, Data & Analysis Center for Software, September 1999. http://www.dacs.dtic.mil/techs/roispi2/ (December 2005).

[131] Austin C. Melton, David A. Gustafson, James M. Bieman, and Albert L. Baker. A Mathematical Perspective for Software Measures Research. *IEE/BCS Software Engineering Journal*, 5:246–254, 1990.

[132] Atif M. Memon. Empirical Evaluation of the Fault-detection Effectiveness of Smoke Regression Test Cases for GUI-based Software. In *Proc. 20th IEEE International Conference on Software Maintenance (ICSM '04)*, pages 8–17. IEEE Computer Society, 2004.

[133] Mary A. Meyer and Jane M. Booker. *Eliciting and Analyzing Expert Judgement. A Practical Guide*, volume 5 of *Knowledge-Based Systems*. Academic Press, 1991.

[134] Douglas R. Miller. Exponential Order Statistic Models of Software Reliability. *IEEE Transactions on Software Engineering*, 12(1):332–346, 1986.

[135] K.-H. Möller. Ausgangsdaten für Qualitätsmetriken. Eine Fundgrube für Analysen. In C. Ebert and R. Dumke, editors, *Software-Metriken in der Praxis*. Springer, 1996.

[136] Sandro Morasca and Stefano Serra-Capizzano. On the Analytical Comparison of Testing Techniques. In *Proc. International Symposium on Software Testing and Analysis (ISSTA '04)*, pages 154–164. ACM Press, 2004.

[137] Max D. Morris. Factorial Sampling Plans for Preliminary Computational Experiments. *Technometrics*, 33:161–174, 1991.

[138] MOST Cooperation. MOST Media Oriented System Transport—Multimedia and Control Networking Technology. MOST Specification Rev 2.2, Ver 2.2-00. 2002.

[139] John C. Munson and Taghi M. Khoshgoftaar. Software Metrics for Reliability Assessment. In Michael R. Lyu, editor, *Handbook of Software Reliability Engineering*, chapter 12. IEEE Computer Society Press and McGraw-Hill, 1996.

[140] John D. Musa. *Software Reliability Engineering: More Reliable Software Faster and Cheaper*. AuthorHouse, 2nd ed., 2004.

[141] John D. Musa, Anthony Iannino, and Kazuhira Okumoto. *Software Reliability: Measurement, Prediction, Application*. McGraw-Hill, 1987.

[142] John D. Musa and Kazuhira Okumoto. A Logarithmic Poisson Execution Time Model for Software Reliability Measurement. In *Proc. Seventh International Conference on Software Engineering (ICSE '84)*, pages 230–238, 1984.

[143] Glenford J. Myers. A controlled Experiment in Program Testing and Code Walkthroughs/Inspections. *Communications of the ACM*, 21(9):760–768, 1978.

[144] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, 1979.

[145] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining Metrics to Predict Component Failures. In *Proc. 28th International Confernce on Software Engineering (ICSE '06)*, pages 452–461. ACM Press, 2006.

[146] P. M. Nagel, F. W. Scholz, and J. A. Skrivan. Software Reliability: Additional Investigations into Modeling with Replicated Experiments. NASA Contractor Rep. 172378, NASA Langley Res. Center, Jun. 1984.

[147] P. M. Nagel and J. A. Skrivan. Software Reliability: Repetitive Run Experimentation and Modeling. NASA Contractor Rep. 165836, NASA Langley Res. Center, Feb. 1982.

[148] Simeon C. Ntafos. The Cost of Software Failures. In *Proc. IASTED Software Engineering Conference*, pages 53–57. IASTED/ACTA Press, 1998.

[149] Simeon C. Ntafos. On Comparisons of Random, Partition, and Proportional Partition Testing. *IEEE Transactions on Software Engineering*, 27(10):949–960, 2001.

[150] Object Management Group. UML 2.0 Superstructure Final Adopted specification, August 2003. OMG Document ptc/03-08-02.

[151] Hiroshi Ohtera and Shigeru Yamada. Optimal Allocation & Control Problems for Software-Testing Resources. *IEEE Transactions on Reliability*, 39(2):171–176, 1990.

[152] Don O'Neill. Software Maintenance and Global Competitiveness. *Journal of Software Maintenance: Research and Practice*, 9(6):379–399, 1997.

[153] Jens Palsberg. Type-Based Analysis and Applications. In *Proc. 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '01)*, pages 20–27. ACM Press, 2001.

[154] Amit Paradkar. Case Studies on Fault Detection Effectiveness of Model Based Test Generation Techniques. In *Proc. First International Workshop on Advances in Model-Based Testing (A-MOST '05)*, pages 1–7. ACM Press, 2005.

[155] Mark C. Paulk, Charles V. Weber, Bill Curtis, and Mary Beth Chrissis. *The Capability Maturity Model: Guidelines for Improving the Software Process*. The SEI Series in Software Engineering. Addison Wesley Professional, 1995.

[156] Hoang Pham. *Software Reliability*. Springer, 2000.

[157] J. Philipps, A. Pretschner, O. Slotosch, E. Aiglstorfer, S. Kriebel, and K. Scholl. Model-Based Test Case Generation for Smart Cards. In *Proc. 8th International Workshop on Formal Methods for Industrial Critical Systems*, pages 168–192. Elsevier, 2003.

[158] Gergely Pintér and István Majzik. Program Code Generation based on UML Statechart Models. In *Proc. 10th PhD Mini-Symposium*. Budapest University of Technology and Economics, Department of Measurement and Information Systems, 2003.

[159] Robert M. Pirsig. *Zen and the Art of Motorcycle Maintenance*. Bantam Books, reissue edition, 1999.

[160] Plato and Robin Waterfield (Translator). *Symposium*. Oxford World's Classics. Oxford University Press, reprint edition, 1998.

[161] PMD. http://pmd.sourceforge.net.

[162] Adam A. Porter, Harvey P. Siy, Carol A. Toman, and Lawrence G. Votta. An Experiment to Assess the Cost-Benefits of Code Inspections in Large Scale Software Development. *IEEE Transactions on Software Engineering*, 23(6):329–346, 1997.

[163] D. Potier, J. L. Albin, R. Ferreol, and A. Bilodeau. Experiments with Computer Software Complexity and Reliability. In *Proc. 6th International Conference on Software Engineering (ICSE '82)*, pages 94–103. IEEE Computer Society Press, 1982.

[164] Wolfgang Prenninger. *Inkrementelle Entwicklung von Verhaltensmodellen zum Test von reaktiven Systemen*. PhD Dissertation, Technische Universität München, 2005.

[165] Wolfgang Prenninger and Alexander Pretschner. Abstractions for Model-Based Testing. In *Proc. Test and Analysis of Component-based Systems (TACoS '04)*, 2004.

[166] Alexander Pretschner. *Zum modellbasierten Test reaktiver Systeme*. PhD Dissertation, Technische Universität München, 2004.

[167] Alexander Pretschner. Zur Kosteneffektivität des modellbasierten Testens. In *Proc. Dagstuhl-Workshop MBEES 2006*, pages 85–94, 2006.

[168] Alexander Pretschner, Wolfgang Prenninger, Stefan Wagner, Christian Kühnel, Martin Baumgartner, Bernd Sostawa, Rüdiger Zölch, and Thomas Stauner. One Evaluation of Model-Based Testing and its Automation. In *Proc. 27th International Conference on Software Engineering (ICSE '05)*, pages 392–401. ACM Press, 2005.

[169] Alexander Pretschner, Oscar Slotosch, Ernst Aiglstorfer, and Stefan Kriebel. Model-Based Testing for Real. *Software Tools for Technology Transfer*, 5(2–3):140–157, 2004.

[170] QJ Pro. http://qjpro.sourceforge.net.

[171] Ronald A. Radice. *High Quality Low Cost Software Inspections*. Paradoxicon Publ., 2002.

[172] David Raffo, Warren Harrison, John Settle, and Nancy Eickelmann. Understanding the Role of Defect Potential in Assessing the Economic Value of Process Improvements. In *Proc. Second Workshop on Economics-Driven Software Engineering Research*, 2000.

[173] Arun Rai, Haidong Song, and Marvin Troutt. Software Quality Assurance: An Analytical Survey and Research Prioritization. *Journal of Systems and Software*, 40:67–83, 1998.

[174] Andreas Rausch and Manfred Broy. *Das V-Modell XT. Grundlagen, Erfahrungen und Werkzeuge*. dpunkt.verlag, 2006. In German.

[175] Horst Remus. Integrated Software Validation in the View of Inspections / Reviews. In *Proc. Symposium on Software Validation*, pages 57–64. Elsevier, 1983.

[176] Jan Rooijmans, Hans Aerts, and Michiel van Genuchten. Software Quality in Consumer Electronics Products. *IEEE Software*, 13(1):55–64, 1996.

[177] Linda Rosenberg, Ted Hammer, and Jack Shaw. Software Metrics and Reliability. In *Proc. 9th International Symposium on Software Reliability Engineering (ISSRE '98)*, 1998.

[178] Peter Rösler. Warum Prüfen oft 50 mal länger dauert als Lesen und andere Überraschungen aus der Welt der Software Reviews. *Softwaretechnik-Trends*, 25(4):41–44, 2005. In German.

[179] RTI. The Economic Impacts of Inadequate Infrastructure for Software Testing. Planning Report 02–3, National Institute of Standards & Technology, 2002.

[180] Glen W. Russell. Experience with Inspection in Ultralarge-Scale Development. *IEEE Software*, 8(1):25–31, 1991.

[181] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. A Comparison of Bug Finding Tools for Java. In *Proc. 15th IEEE International Symposium on Software Reliability Engineering (ISSRE'04)*, pages 245–256. IEEE Computer Society, 2004.

[182] Giedre Sabaliauskaite. *Investigating Defect Detection in Object-Oriented Design and Cost-Effectiveness of Software Inspection*. PhD dissertation, Osaka University, 2004.

[183] Giedre Sabaliauskaite, Shinji Kusumoto, and Katsuro Inoue. Extended Metrics to Evaluate Cost Effectiveness of Software Inspections. *IEICE Transactions on Information and Systems*, E87-D(2):475–480, 2004.

[184] Andrea Saltelli. Global Sensitivity Analysis: An Introduction. In *Proc. 4th International Conference on Sensitivity Analysis of Model Output (SAMO '04)*, pages 27–43. Los Alamos National Laboratory, 2004.

[185] Andrea Saltelli and Ricardo Bolado. An alternative way to compute Fourier amplitude sensitivity test (FAST). *Computational Statistics & Data Analysis*, 26:445–460, 1998.

[186] Andrea Saltelli, Stefano Tarantola, Francesca Campolongo, and Marco Ratto. *Sensitivity Analysis in Practice – A Guide to Assessing Scientific Models*. John Wiley & Sons, 2004.

[187] Hans Sassenburg. Optimal Release Time: Numbers or Intuition? In *Proc. 4th Workshop on Software Quality (4-WoSQ)*, pages 57–62. ACM Press, 2006.

[188] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.

[189] Bran Selic and James Rumbaugh. Using UML for Modeling Complex Real-Time Systems. Available at http://www.ibm.com/developerworks/rational/library/, 1998.

[190] Carl Shapiro. Premiums for High Quality Products as Returns to Reputations. *The Quarterly Journal of Economics*, 45(6):659–666, November 1983.

[191] Martin L. Shooman and Morris I. Bolsky. Types, Distribution, and Test and Correction Times for Programming Errors. In *Proc. International Conference on Reliable Software*, pages 347–357. ACM Press, 1975.

[192] SimLab 2.2. http://webfarm.jrc.cec.eu.int/uasa/primer/index.asp, 2005.

[193] Sandra A. Slaughter, Donald E. Harter, and Mayuram S. Krishnan. Evaluating the Cost of Software Quality. *Communications of the ACM*, 41(8):67–73, 1998.

[194] Qinbao Song, Martin Sheperd, Michelle Cartwright, and Carolyn Mair. Software Defect Association Mining and Defect Correction Effort Prediction. *IEEE Transactions on Software Engineering*, 32(2):69–82, 2006.

[195] Mark Sullivan and Ram Chillarege. A Comparison of Software Defects in Database Management Systems and Operating Systems. In *Proc. 22nd International Symposium on Fault-Tolerant Computing (FTCS-22)*, pages 475–484. IEEE Computer Society, 1992.

[196] The ENTERPRISE Program. Mayday: System Specifications, 1997. Available at http://enterprise.prog.org/completed/ftp/mayday-spe.pdf.

[197] The ENTERPRISE Program. Colorado Mayday Final Report, 1998. Available at http://enterprise.prog.org/completed/ftp/maydayreport.pdf.

[198] Thomas Thelin, Per Runeson, and Claes Wohlin. An Experimental Comparison of Usage-Based and Checklist-Based Reading. *IEEE Transactions on Software Engineering*, 29(8):687–704, 2003.

[199] Thomas Thelin, Per Runeson, Claes Wohlin, Thomas Olsson, and Carina Andersson. Evaluation of Usage-Based Reading—Conclusions after Three Experiments. *Empirical Software Engineering*, 9:77–110, 2004.

[200] Jeff Tian. Quality-Evaluation Models and Measurements. *IEEE Software*, 21(3):84–91, 2004.

[201] Stefan Wagner. Efficiency Analysis of Defect-Detection Techniques. Technical Report TUMI-0413, Institut für Informatik, Technische Universität München, 2004.

[202] Stefan Wagner. Reliability Efficiency of Defect-Detection Techniques: A Field Study. In *Suppl. Proc. 15th IEEE International Symposium on Software Reliability Engineering (ISSRE '04)*, pages 294–301. INRIA, Rennes, France, 2004.

[203] Stefan Wagner. Software Quality Economics for Combining Defect-Detection Techniques. In *Proc. Net.Object Days 2005*, pages 559–574. tranSIT GmbH, 2005.

[204] Stefan Wagner. Towards Software Quality Economics for Defect-Detection Techniques. In *Proc. 29th Annual IEEE/NASA Software Engineering Workshop (SEW-29)*, pages 265–274. IEEE Computer Society, 2005.

[205] Stefan Wagner. A Literature Survey of the Quality Economics of Defect-Detection Techniques. In *Proc. 5th ACM-IEEE International Symposium on Empirical Software Engineering (ISESE '06)*. ACM Press, 2006.

[206] Stefan Wagner. A Literature Survey of the Software Quality Economics of Defect-Detection Techniques. Technical Report TUM-I0614, Institut für Informatik, Technische Universität München, 2006.

[207] Stefan Wagner. A Model and Sensitivity Analysis of the Quality Economics of Defect-Detection Techniques. In *Proc. International Symposium on Software Testing and Analysis (ISSTA '06)*, pages 73–83. ACM Press, 2006.

[208] Stefan Wagner. Modelling the Quality Economics of Defect-Detection Techniques. In *Proc. 4th Workshop on Software Quality (4-WoSQ)*, pages 69–74. ACM Press, 2006.

[209] Stefan Wagner and Helmut Fischer. A Software Reliability Model Based on a Geometric Sequence of Failure Rates. Technical Report TUMI-0520, Institut für Informatik, Technische Universität München, 2005.

[210] Stefan Wagner and Helmut Fischer. A Software Reliability Model Based on a Geometric Sequence of Failure Rates. In *Proc. 11th International Conference on Reliable Software Technologies (Ada-Europe '06)*, volume 4006 of *LNCS*, pages 143–154. Springer, 2006.

[211] Stefan Wagner and Jan Jürjens. Model-Based Identification of Fault-Prone Components. In *Proc. Fifth European Dependable Computing Conference (EDCC-5)*, volume 3463 of *LNCS*, pages 435–452. Springer, 2005.

[212] Stefan Wagner, Jan Jürjens, Claudia Koller, and Peter Trischberger. Comparing Bug Finding Tools with Reviews and Tests. In *Proc. 17th International Conference on Testing of Communicating Systems (TestCom'05)*, volume 3502 of *LNCS*, pages 40–55. Springer, 2005.

[213] Stefan Wagner and Tilman Seifert. Software Quality Economics for Defect-Detection Techniques Using Failure Prediction. In *Proc. 3rd Workshop on Software Quality (3-WoSQ)*, pages 11–16. ACM Press, 2005.

[214] Stefan Wagner and Tilman Seifert. Software quality economics for defect-detection techniques using failure prediction. *SIGSOFT Software Engineering Notes*, 30(4), 2005.

[215] Wen-Li Wang, Ye Wu, and Mei-Hwa Chen. An Architecture-Based Software Reliability Model. In *Proc. Pacific Rim International Symposium on Dependable Computing (PRDC '99)*, pages 143–150. IEEE Computer Society, 1999.

[216] Christian Weitzel. *Statistische Prozesssteuerung in der Softwareentwicklung. Untersuchung statistischer Methoden und Auswertung von Review-Daten*. Diplomarbeit, Technische Universität München, 2006.

[217] Elaine J. Weyuker. Evaluating Software Complexity Measures. *IEEE Transactions on Software Engineering*, 14(9):1357–365, 1988.

[218] Elaine J. Weyuker. More Experience with Data Flow Testing. *IEEE Transactions on Software Engineering*, 19(9):912–919, 1993.

[219] Elaine J. Weyuker and Bingchiang Jeng. Analyzing Partition Testing Strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, 1991.

[220] Ron R. Willis, Bob M. Rova, Mike D. Scott, Martha I. Johnson, John F. Ryskowski, Jane A. Moon, Ken C. Shumate, and Thomas O. Winfield. Hughes Aircraft's Widespread Deployment of a Continuously Improving Software Process. Technical Report CMU/SEI-98-TR-006, Carnegie-Mellon University, 1998.

[221] Misha Zitser, Richard Lippmann, and Tim Leek. Testing Static Analysis Tools using Exploitable Buffer Overflows from Open Source Code. In *Proc. 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT'04/FSE-12)*, pages 97–106. ACM Press, 2004.