T U M

I N S T I T U T F Ü R I N F O R M A T I K

A Literature Survey of the Software Quality Economics of Defect-Detection Techniques

Stefan Wagner

TUM-I0614
Juli 06

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-07-I0614-0/1.-FI
Alle Rechte vorbehalten
Nachdruck auch auszugsweise verboten

©2006

Druck: Institut für Informatik der
Technischen Universität München

A Literature Survey of the Software Quality Economics of Defect-Detection Techniques*

Stefan Wagner
Institut für Informatik
Technische Universität München
Boltzmannstr. 3, 85748 Garching b. München
Germany

Abstract

Over the last decades, a considerable amount of empirical knowledge about the efficiency of defect-detection techniques has been accumulated. Also a few surveys have summarised those studies with different focuses, usually for a specific type of technique. This work reviews the results of empirical studies and associates them with a model of software quality economics. This allows a better comparison of the different techniques and supports the application of the model in practice as several parameters can be approximated with typical average values. The main contributions are the provision of average values of several interesting quantities w.r.t. defect detection and the identification of areas that need further research because of the limited knowledge available.

*This research was supported by the DFG within the priority program *SoftSpez* (SPP 1064) under project name *InTime*.

Contents

1 Introduction **4**
1.1 Problem 4
1.2 Contribution 4
1.3 Organisation 4

2 Software Quality Economics **6**
2.1 Software Quality Costs 6
2.2 An Analytical Model 7
2.2.1 General 7
2.2.2 Components 9
2.2.3 ROI 12
2.3 Practical Model 12
2.3.1 General 12
2.3.2 Components 13

3 Empirical Knowledge **15**
3.1 Experiments and Field Studies 15
3.2 Approach 15
3.3 Difficulty 16
3.4 Dynamic Testing 17
3.4.1 Classification 17
3.4.2 Setup and Execution Costs 18
3.4.3 Difficulty 18
3.4.4 Removal Costs 21
3.5 Review and Inspection 23
3.5.1 Classification 23
3.5.2 Setup and Execution Costs 24
3.5.3 Difficulty 26
3.5.4 Removal Costs 29
3.6 Static Analysis Tools 31
3.6.1 Classification 31
3.6.2 Setup and Execution Costs 31
3.6.3 Difficulty 32
3.7 Defects 33
3.7.1 Defect Introduction 33
3.7.2 Removal Costs 34
3.7.3 Effect Costs 35
3.7.4 Failure Probability 36

4 Discussion **38**

5 Related Work **39**

6 Conclusions **40**
6.1 Summary 40
6.2 Further Research 40

1 Introduction

The economics of software quality assurance (SQA) are a highly relevant topic in practice. Many estimates assign about half of the total development costs of software to SQA of which defect-detection techniques, i.e., analytical SQA, constitute the major part. Moreover, an understanding of the economics is essential for project management to answer the question how much quality assurance is enough. For example Rai, Song, and Troutt [57] state, that a better understanding of the costs and benefits should be useful to decision-makers.

However, the relationships regarding those costs and benefits are often complicated and the data is difficult to obtain. Ntafos discusses in [50] that cost is a central factor but “it is hard to measure, data are not easy to obtain, and little has been done to deal with it”. Nevertheless, there is a considerable amount of empirical studies regarding defect-detection techniques. The effectiveness and efficiency of testing and inspections has been investigated intensively over the last decades. Yet, we are not aware of a literature survey that summarises this empirical knowledge with respect to an economics model.

1.1 Problem

The main practical problem is how we can optimally use defect-detection techniques to improve the quality of software. Hence, the two main issues are (1) in which order and (2) with what effort the techniques should be used. This paper concentrates on the subproblem that the collection of all relevant data for a well-founded answer to these questions is not always possible.

1.2 Contribution

We review and summarise the empirical studies on various aspects of defect-detection techniques and software defects in general. The results of those studies are assigned to the different input factors of an economics model of analytical SQA. In particular, mean and median values of the input factors are derived to allow an easier application of the model in practice when not all factors are collectable. Furthermore, the found distributions can be used in further analyses of the model. Finally, the review reveals several areas that need further empirical research.

1.3 Organisation

We discuss software quality costs in general and the used economics model for SQA in Sec. 2. The empirical knowledge of defect-detection techniques is then analysed in Sec. 3 structured in test techniques, review techniques, static tools, and defects. In Sec. 4 some issues and interpretation of the

empirical knowledge is discussed. Related work is given in Sec. 5 and Sec. 6 summarises the paper with final conclusions and future work.

2 Software Quality Economics

In this section, we introduce the general concept of quality costs for software. Based on that, we describe an analytical, stochastic model of the costs and benefits – the economics – of analytical SQA and finally possibilities of its practical application.

2.1 Software Quality Costs

Quality costs are the costs associated with preventing, finding, and correcting defective work. Based on experience from the manufacturing area quality cost models have been developed explicitly for software. These costs are divided into *conformance* and *nonconformance* costs, also called *control costs* and *failure of control costs*. The former comprises 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 the usage of various types of tests and reviews.

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 occur during development, the latter describes costs that result from failures at the client. A graphical overview is given in Fig. 1. Because of the distinction between prevention, appraisal, and failure costs this is often called *PAF* model.

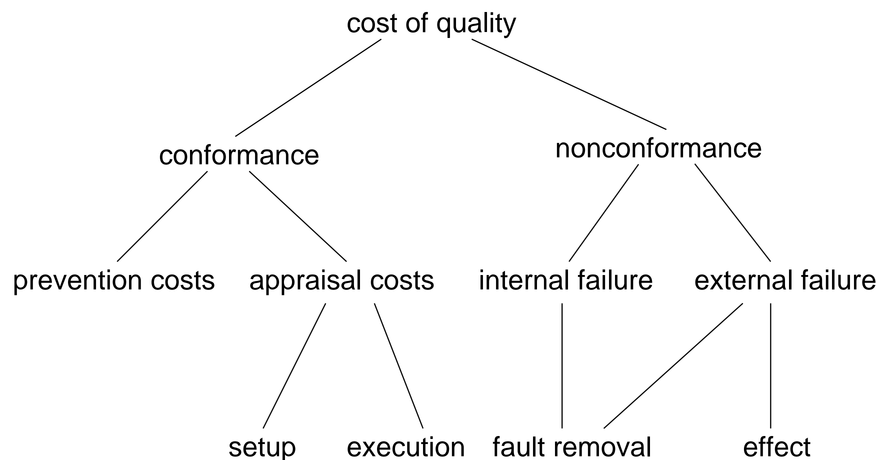

Figure 1: Overview over the costs related to quality

We add further detail to the PAF model by introducing the main types of concrete costs that are important for defect-detection techniques. Note

that there are more types that could be included, for example, maintenance costs. However, we concentrate on a more reliability-oriented view. 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.

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. The reason is that the removal of a detected defect always results in costs no matter whether it caused 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. It is also a good example that 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.

External failures also cause *effect* costs. These are all further costs 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 further costs such as loss of sales because of bad reputation in the effect costs but do not consider it explicitly because its out of scope of this paper.

2.2 An Analytical Model

We describe a general, analytical model of defect-detection techniques in the following based on [75, 76]. General is meant with respect to the various types of techniques. This allows us to review the literature of all of those techniques combined. We mainly analyse different types of testing which essentially detect failures and static analysis techniques that reveal faults in the code or other documents. 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. 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.

2.2.1 General

In this section, we concentrate on an ideal model of quality economics in the sense that 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 in the sense that it is based on expected values as basis for decision making. This approach is already common in other engineering fields [9] to compare different alternatives.

Components. We divide the model in three main components:

- Direct costs d_A
- Future costs o_A
- Revenues / saved costs r_A

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.

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 [41] 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, the refined difficulty function is defined as $\theta_A(i, t_A)$ denoting the difficulty of A detecting 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 $1 - \theta_A(i, t_A)$.

Using this additional dimension we can also analyse different functional forms of the difficulty functions depending on the spent effort. This is similar to the informal curves shown by Boehm [10] 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. In [75, 76] we considered several possible forms of the difficulty functions such as exponential or linear.

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 especially 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. There are various dependencies that could be considered but most importantly there is dependency in terms of propagation. Defects from earlier phases propagate to later phases and over process steps. We actually 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.

2.2.2 Components

We give an equation for each of the three components with respect to single defect-detection techniques first and later for a combination of techniques. Note that the main basis of the model are expected values, i.e., we combine cost data with probabilities.

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. 2 shows systematically the components of the direct costs.

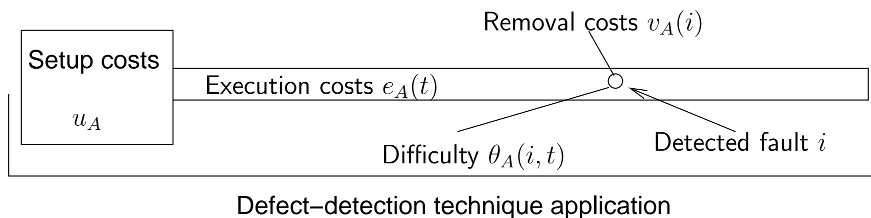

Figure 2: The components of the direct costs

From this we can derive the following definition for the direct costs d_A

containing the three cost types setup costs, execution costs, and removal costs for a technique.

$$d_A = u_A + e_A(t) + \sum_i (1 - \theta_A(i,t)) v_A(i), \tag{1}$$

where u_A are the setup costs, $e_A(t)$ 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 the technique and removal costs for each fault in the software if the technique is able to find it.

Future Costs. In case we were not able to find defects, these will result in costs in the future denoted by o_A. We divide these costs into the two parts fault removal costs in the field $v_F(i)$ and failure effect costs $f_F(i)$. The latter contain all support and compensation costs as well as annoyed customers as far as possible.

$$o_A = \sum_i \pi_i \theta_A(i,t)(v_F(i) + f_F(i)), \tag{2}$$

where $\pi_i = P$(fault i is activated by randomly selected input and is detected and fixed) [41]. Hence, it describes the probability that the defect leads to a failure in the field.

Revenues. It is necessary to consider not only costs with defect-detection techniques but also revenues. These revenues 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 r_A.

$$r_A = \sum_i \pi_i (1 - \theta_A(i,t))(v_F(i) + f_F(i)) \tag{3}$$

Because the revenues are saved future costs this equation looks similar to Eq. 2. The difference is only that we consider the faults that have not been found and hence use the probability of the negated difficulty, i.e., $1 - \theta_A(i,t)$.

Combination. Typically, more than one technique is used to find defects. The intuition behind that is that they find (partly) different defects. These dependencies are often ignored when the efficiency of defect-detection techniques is analysed. Nevertheless, they have a huge influence on the economics and efficiency. In our view, the notion of diversity of techniques from Littlewood et al. [41] is very useful in this context. The covariance

of the difficulty functions of faults describes the similarity of the effectiveness regarding fault finding. We already use the difficulty functions in the present model and therefore are able to express the diversity implicitly.

For the direct costs it means that we sum over all different applications of defect-detection techniques. We define that X is the ordered set of the applied defect-detection techniques. In each application we use Eq. 1 with the extension that we not only take the probability that the technique finds the fault into account but also that the ones before have not detected it. Here also the defect propagation needs to be considered, i.e., that not only the defect itself has not been detected but also its predecessors R_i.

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

$$\Theta(x, R_i) = \prod_{y<x} \theta_y(i, t_y) \prod_{j \in R_i} \theta_y(j, t_y), \tag{4}$$

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 combined direct costs d_X of a sequence of defect-detection technique applications X is then defined as follows:

$$d_X = \sum_{x \in X} \left[u_x + e_x(t_x) + \sum_i \Big((1 - \theta_x(i, t_x)) \Theta(x, R_i) \Big) v_x(i) \right] \tag{5}$$

Note that by using the $\Theta(x, R_i)$ the difference to Eq. 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 $\Theta(x, R_i)$.

The equation for 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.

$$r_X = \sum_{x \in X} \sum_i \left[\Big(\pi_i (1 - \theta_x(i, t_x)) \Theta(x, R_i) \Big) (v_F(i) + f_F(i)) \right] \tag{6}$$

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 $\Theta(x, R_i)$ 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 $\Theta'(x, R_i)$ that describes only the product of the difficulties of detecting the predecessors of i is hinted in the following equation for the future cost o_X of several technique applications X.

$$o_X = \sum_i \left[\pi_i \prod_{x \in X} \theta_x(i, t_x) \underbrace{\prod_{y<x} \prod_{j \in R_i} \theta_y(j, t_y)}_{\Theta'(x, R_i)} (v_F(i) + f_F(i)) \right] \tag{7}$$

2.2.3 ROI

One interesting metric based on these values 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. [11] use the equation (Benefits – Costs)/Costs. To calculate the total ROI with our model we have to use Eqns. 5, 7, and 6.

$$\text{ROI} = \frac{r_X - d_X - o_X}{d_X + o_X} \tag{8}$$

This metric can be used for two purposes: (1) an up-front evaluation of the quality assurance plan as the *expected* 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. Hence, also the post evaluation metric can be seen as an *estimated* ROI.

2.3 Practical Model

As we discussed above, the theoretical model can be used for analyses but is too detailed for a practical application. Hence, we need to simplify the model to reduce the needed quantities.

2.3.1 General

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 assumed that the defects can be grouped in "useful" classes or defect types. For reformulating the equation it was 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.

For our economics model we consider the defect classification approaches from IBM [33] and HP [26] 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 lose 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 [31] or product-specific models such as COQUALMO [66]. To simplify further estimates other approaches can be used. For example, the defect removal effort for different defect types can be predicted using a association mining approach of Song et al. [69].

2.3.2 Components

Similar to Sec. 2.2.2 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. Now we do not consider the ideal quantities but use average values for the cost factors. We denote this with a bar over the cost name.

$$d_A = \bar{u}_A + \bar{e}_A(t) + \sum_i (1 - \theta_A(\tau_i, t))\bar{v}_A(\tau_i), \tag{9}$$

where $\bar{u}_A$ is the average setup cost for technique A, $\bar{e}_A(t)$ is the average execution cost for A with length t, and $\bar{v}_A(\tau_i)$ is the average removal cost in defect type τ_i. Apart from using average values, the main difference is that we consider defect types in the difficulty functions. The same applies to the revenues.

$$r_A = \sum_i \pi_{\tau_i}(1 - \theta_A(\tau_i, t))(\bar{v}_F(\tau_i) + \bar{f}_F(\tau_i)), \tag{10}$$

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

$$o_A = \sum_i \pi_{\tau_i}\theta_A(\tau_i, t)(\bar{v}_F(\tau_i) + \bar{f}_F(\tau_i)). \tag{11}$$

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

$$\theta_A(\tau_i, t_a) = mt_A + 1, \tag{12}$$

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

Combined Economics. Similarly as in the ideal model, the extension to more than one technique can be done. We omit the details here and refer to [75, 76].

3 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 in general.

Empirical research in software engineering is often not as developed as necessary to be able to judge the value of technique and method proposals. The field of quality assurance and defect-detection techniques in particular has nevertheless been subject to a number of empirical studies over the last decades. These studies were used to assess specific techniques or to validate certain laws and theories about defect-detection. Several of these validated laws are compiled in the book of Endres and Rombach [20]. Our focus lies more on the economic relationships in the following.

3.1 Experiments and Field Studies

We can mainly distinguish two types of studies that are relevant in providing data and knowledge for our economics model: (1) experiments and (2) field studies. Experiments are typically performed with a group of students that simultaneously perform similar tasks. This allows great control over the experiment in total and the data collection and other factors in particular. The problem often is that it is not clear if the results can be generalised to be valid in an industrial context.

Field studies on the contrary collect data from industrial projects and analyse the data. Hence, we have better generalisable results in the sense that the data comes from real-world projects with trained developers. However, we are often not able to replicate those studies easily and performing the same task several times is often not possible because of time and money constraints. Finally, the control of other factors influencing the study is largely limited.

As a conclusion of this, empirical research has to consist always of both types of studies. Student experiments need to be done to analyse effects that can only be tested using replication whereas industrial field studies are necessary to generalise results to real-world projects.

3.2 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. 2.2. Literature reviews, also called meta-analysis, is a common technique in social sciences or medicine. A book with details on such a review can be found, for example, in [18]. 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.

3.3 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. [41], which is the origin of the idea of difficulty functions, *effectiveness* is actually defined as

$$1 - E_{p^*}(\theta_A(i)), \tag{13}$$

where E_{p^*} denotes a mean obtained with respect to the probability distribution p^*, i.e., the probability distribution of the presence of faults.

As a simple approximation we then define the following for the average difficulty functions.

$$\bar{\theta}_A = 1 - \text{effectiveness}_A \tag{14}$$

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.

3.4 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.

3.4.1 Classification

There are various possibilities to classify different test techniques. We base our classification on standard books on testing [47, 4] and the classification in [32]. 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. 3 shows these two dimensions and contains some concrete examples and how they can be placed according to these dimensions.

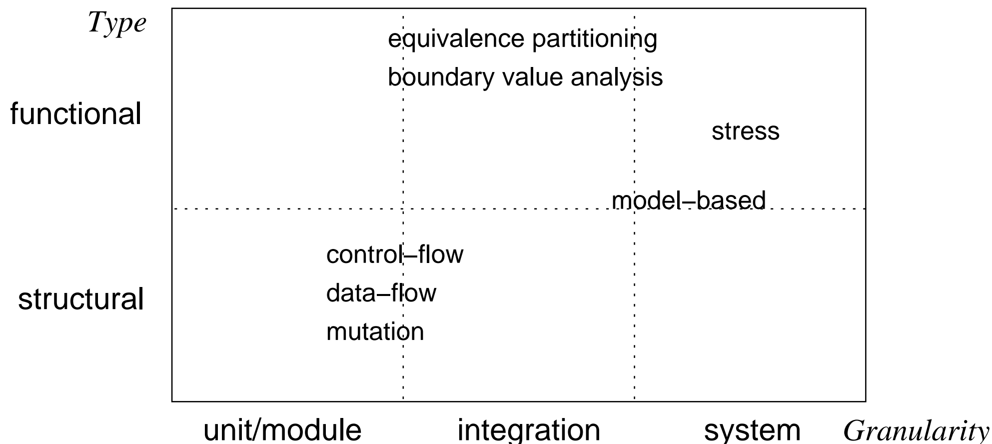

Figure 3: The two basic dimensions of test techniques

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 normally 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 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. Various other types of testing can be found in the literature.

3.4.2 Setup and Execution Costs

We look at the setup and execution costs in more detail in the following. For both cost types the empirical data is limited. However, this is not a great problem because this data can be easily collected in a software company during projects.

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 [31]. 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 typical 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 its even easier than for setup costs as apart from the personnel 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 typical, average costs for the staff. However, we also have average values per function point from [31]. There the average effort for unit tests is 0.25 h/fp, for function tests, system tests, and field tests 0.50 h/fp.

3.4.3 Difficulty

As discussed in Sec. 3.3, 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.

Effectiveness. 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 [46] resulted in an average percentage of defects found for functional testing of 36.0 and for structural of 38.0.
- Jones states in [31] that most forms of testing have an effectivity of less than 30%.
- He also states in [31] 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 [44] 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 [37, 38] 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 [28] on the average percentage of defects found for functional testing as 47.7 and for structural as 46.7.
- The study published in [3] 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 [29] on an older experiment regarding some different testing techniques. Path testing detected 18 of 28 faults, branch testing 6 of 28 faults. The combined use of different structural testing techniques revealed 25 of 28 faults.
- Weyuker reports in [80] on empirical results about flow-based testing. 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 [53]. In two case studies the generated test suites were able to kill between 82% and 96% of the mutants.
- In [8] testing detects 7.2% of the defects.
- In [55] 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. 1. We can observe that the mean and median values are all quite close which suggests that there are no strong outliers. However, the range in general is rather large, especially when considering all test techniques including those studies which do not state the test case derivation technique. When comparing functional and structural testing, there is no significant difference visible.

Table 1: 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

Efficiency. The effectiveness gives a good approximation of the difficulty θ. The efficiency measures the number of detected defects per effort unit (staff-hours, for example). It cannot be used directly in the economics model but is also summarised as it is an important metric itself.

- The classical experiment by Myers [46] resulted in a mean effort per defect of 37 staff-minutes for functional testing and 29 staff-minutes for structural testing.
- The average number of hours to find a defect with testing is given in [68] as 24.3.
- The study in [82] reports that functional testing found 2.47 defects per hour with a std. dev. of 1.10 and structural testing 2.20 with a std. dev. of 0.94.
- The successive usage of functional testing described in [3] revealed defects with a mean efficiency (std. dev. in braces) of 1.38 (0.90), 1.22 (0.91), and 1.84 (1.06) defects per hour.
- 6 hours of effort per failure for detection is given in [79].
- The article [1] reports that in a banking computer-service firm it took 4.5 hours to eliminate a defect by unit testing.
- It is also stated in [1] that another organisation reported the average effort to find a defect by testing to be 8.5 staff-hours.
- Runeson and Andrews describe an experiment [63] where the mean detection efficiency of unit testing was 1.80 defects per hour.

The found efficiencies of functional and structural test techniques are summarised in Tab. 2. We assume that one staff-hour consists of 60 staff-minutes. The results show that the data is quite homogenous because the means and medians are all equal or nearly equal and the ranges are rather small. Especially for functional testing it is only slightly above 1 defect/hour difference between the lowest and the highest value.

Difficulty. The approximation of the average difficulty functions is given in Tab. 3. We used the results of the effectiveness summary above. Hence, the observations are accordingly.

Table 2: Summary of the efficiency of test techniques (in defects per staff-hour)

Type	Lowest	Mean	Median	Highest
Functional	1.22	1.72	1.71	2.47
Structural	0.22	1.5	2.07	2.2
All	0.04	1.26	1.5	2.47

Table 3: Average difficulty functions for testing (in percentages)

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

Defect Types. Finally, for an application to our practical economics model we need to differentiate between different fault types. Basili and Selby analysed the effectiveness of functional and structural testing regarding different defect types in [3]. Tab. 4 shows the derived difficulties using the first approximation.

Table 4: 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. As we are only aware of this single study it is difficult to generalise the results.

3.4.4 Removal Costs

The removal costs are dependent on the second dimension of testing (cf. Sec. 3.4.1): the phase in which it is used. This 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, in comparison with static techniques, is 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 [67] 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.
- Möller [45] reports of removal costs during unit testing of $2,000$ DM and during system testing of $6,000$ DM.
- Jones [31] 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 [17] 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 [24] 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. [35] state that it takes up to 17 hours to fix defects during testing.
- A study [61] found for the financial domain that to fix a defect during coding and unit testing needs 4.9 hours, during integration 9.5 hours, and during beta-testing 12.1 hours.
- The same study [61] reports these measures for the transportation domain. The necessary hours to fix a defect during coding and unit testing is 2.4, during integration 4.1, and during beta-testing 6.2.
- Following [81] 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. [59] 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.

Some statistics of the data above on the removal costs are summarised in Tab. 5. 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 averages 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.

Table 5: 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

3.5 Review and Inspection

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

3.5.1 Classification

Similar as for test techniques in Sec. 3.4, we need also a classification of the large amount of different flavours of reviews and inspections. We mainly base our classification on [39] where four main dimensions of software inspections are identified. (1) The technical dimension for the methodological variations, (2) the economic dimension of the economic effects on projects, (3) the organisational dimension for the effects on the organisation, and (3) the tool dimension that characterises the tool support.

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.

A prominent example is the formal or Fagan inspection [22] 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.

3.5.2 Setup and Execution Costs

Setup Costs. The first question is whether reviews and inspections do have setup costs. We considered those 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 [27] the experience of Hewlett-Packard with inspections. They give typical 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 personnel costs as long as there is no supporting software used. Hence, the execution costs are dependent on the factor t in our model. Nevertheless, there are some typical values for the execution costs of inspections.

- Grady and van Slack describe in [27] 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 typically 0.5 staff-hours are needed for each part.
- Jones has published average efforts in relation to the size in function points in [31]. 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 [71] 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. [54] 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 [31] typical rates for source code inspections as 150 LOC/h during preparation and 75 LOC/h during the meeting.
- Rösler describes in [60] 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. 6. 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 6: 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 [60] argues for an optimal inspection rate of about 0.9 pages per hour.
- In [25] an optimal average rate is one page per hour.
- In [25] also the optimal bandwidth of the inspection rate is 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 models such as the one presented in Sec. 2.2.

3.5.3 Difficulty

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 [31] states that formal design and code inspections tend to be the most effective, and they alone can exceed 60%.
- Basili and Selby [3] 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 [8]. Prebuild inspections are able to find 85.4% and other inspections further 7.3% of the total defects.
- Biffl et al. [7] 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. [37, 38]
- Biffl et al. analysed in [5] 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 [27] it is reported that typically 60 to 70 percent of the defects were found by inspections.
- In [3] 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. [72] report on several experiments on usage-based reading. The effectiveness was 0.29, 0.31, 0.34, and 0.32
- Thelin et al. [71] compare different reading techniques. The effectiveness was 0.31 for UBR and 0.25 for CBR.
- Biffl and Halling [6] 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).

Reading time (h)	0–2	2–4	4–6	6–8
CBR	8.5	17.7	20.9	19.5
SBR-U	9.4	14.6	18.8	20.9
SBR-D	8.7	13.7	18.0	20.0
SBR-T	9.0	14.2	20.0	14.0

We also summarise these results using the lowest, highest, mean, and median value in Tab. 7. 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 7: Summary of the effectiveness of inspection techniques (in percentage)

Lowest	Mean	Median	Highest
8.5	34.14	30	92.7

Efficiency. The efficiency relates the effectiveness with the spent effort. Again, this is not directly usable in the analytical model but nevertheless can give further insights into the relationships of factors.

- Myers [46] reports 75 man-minutes per defect for walkthroughs / inspections.
- Wood et al. [82] found that code reading detects 1.06 defects per hour with a std. dev. of 0.75.
- In three phases of code reading [3] the mean efficiency was 1.9 defects per hour (1.83), 0.56 (0.46), and 3.33 (3.42).
- In [68] it was found that it takes on average 3.8 hours to find a defect with Fagan inspections and 6.4 hours with code reading.
- For inspections it in [79] took 1.43 hours per defect, later less than 1 hour per defect to be detected.
- The effort per defect in the experiment described in [7] is 2.2 staff hours, std.dev. 2.8. For a reinspection it is 3.2 with std.dev. 3.5.
- Ackerman reports from two studies with design and code inspections in [1]. In one study the mean effort for a design inspection was 1.0 hours per defect found, and 4.8 hours per major defect found. For code inspection this increased to 1.2 hours per defect found. In the second study for detailed design inspection 0.58 h/defect were needed and 0.67 h/defect for code.
- In [1] it is also stated that others reported 3 to 5 staff-hours per major defect over several different applications and programming languages.
- Also in [1] the average effort to eliminate a defect by inspection in a banking computer-service firm is given as 2.2 hours.

- An organisation reported the average effort to find a defect by inspections to be 1.4 staff-hours. [1]
- In [36] experiences with the cost of finding a defect in design inspections is reported to be 1.58 hours.
- In [37, 38] the efficiency of individual inspections is reported as 0.19 with a std. dev. of 0.06.
- In a inspection and reinspection study [5] the mean efficiency decreased from 0.82 to 0.53 defects per person-hour for all defects and from 0.35 to 0.24 for level 2 and 3 defects (with 3 being the highest severity).
- In [3] also the efficiency of three usages of inspections were analysed. It was 1.40 defects per hour (0.87), 1.18 (0.84), and 1.82 (1.24), respectively.
- Solingen et al. compare normal inspections with computer-supported inspections in [74]. The measured an average efficiency of the normal inspections of 1.6 defects per hour and 2.0 defects per hour for the computer-supported inspections.
- Van Genuchten et al. analyse inspections with and without the use of a group support system in [73]. In the first field study the efficiency during preparation was 0.81 and during meeting 0.09 defects/staff-hour with paper-based inspections. Electronic inspections had a preparation efficiency of 2.70 and a meeting efficiency of 1.45.
- Rooijmans et al. [59] published data on the effort for preparation, logging meeting and rework effort in three projects. These were 0.5, 0.7, and 2.1 hours per defect, respectively.
- Shooman [67] reports an average effort to find a defect of 0.6 hours.
- Russel [64] analysed software inspections at BNR and found an average efficiency of 0.8 – 1 defect per staff-hour.
- Bush reports in [14] of an average effort to find, fix and verify the correction of 1.5 and 2.1 hours per defect. This corresponds to a cost of $90-$120.
- The experiment of Runeson and Andrews [63] resulted in a mean detection efficiency of inspection of 1.49 defects per hour.
- Thelin et al. [72] report on three experiments on usage-based reading. The efficiency was 5.3, 5.6, 6.0, and 5.6 defects per hour.
- Thelin et al. [71] compare different reading techniques. The efficiency was 5.6 (UBR) and 4.1 (CBR) defects per hour.

The statistics for the efficiency of reviews and inspections can be found in Tab. 8. We do not distinguish different processes and reading techniques here because then we would not have enough information on these in most studies. The mean and median are close, therefore the data set is reasonable. We also observe a large range from 0.16 to 6 defects/staff-hour.

Table 8: Summary of the efficiency of inspection techniques (in defects per staff-hour)

Lowest	Mean	Median	Highest
0.16	1.87	1.18	6

Difficulty. Using the first, simple approximation, we can derive statistics for the difficulty of inspections in reviews in Tab. 9.

Table 9: Derived average difficulty of inspections (in percentages)

Lowest	Mean	Median	Highest
7.3	65.86	70	91.5

Defect Types. Analogous to the test techniques, we have one major study about effectiveness and defect types [3]. The derived difficulty functions are given in Tab. 10. Also for inspections large differences between the defect types are visible but a single study does not guarantee generalisability.

Table 10: 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

O'Neill reports [51] on the average distribution of defects types from a large inspection study. Unfortunately, different defect types are used. The results can be found in Tab. 11 with the ratios given in percentages.

3.5.4 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.

Table 11: Average defect type distribution

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

- Möller [45] reports of removal costs during analysis, design, and coding of 500 DM.
- During requirements reviews a removal effort of 1.00 hours per defect is needed. Design and code inspections have 1.50 h/defect. [31]
- In the book [56] the average effort to find and fix a major problem is given with 0.8 to 0.9 hours.
- In the report [43] the effort for rework for a defect is given as 2.5 staff-hours for in-house defects.
- Collofello and Woodfield [17] 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.
- Bourgeois published in [12] 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. [24]
- Kelly et al. [35] 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 [61] 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. [81]

The summary of the the removal costs can be found in Tab. 12. 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.

Table 12: Summary of the removal costs of reviews (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

3.6 Static Analysis Tools

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

3.6.1 Classification

The term *static analysis tools* denotes a huge field of software tools that are able to find (potential) defects in software code without executing it. The spectrum ranges from simple compiler-like or style checks to sophisticated dataflow analyses or formal verifications. Another common term is *bug finding tools* that often does not include formal techniques.

Those analysis tools use various techniques to identify 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 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 [23] to allow an extended static checking and a combination with model checking.

The results of 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 bug finding tools.

There are only few studies about static analysis tools and hence we can only present limited empirical knowledge.

3.6.2 Setup and Execution Costs

There are no studies with data about the setup and execution costs of using static analysis tools. Still, we try to analyse those costs and their influence in the context of such tools.

Setup Costs. 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 much 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 reviews of the code. Hence, the ratio of false positives is an important measure for the efficiency and execution costs of a tool.

- In [78] we found that the average ratio of false positives over three tools for Java was 66% ranging from 31% up to 96%.
- In [83] 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 [30] 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.

3.6.3 Difficulty

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

- In [29] are also some static analysis techniques evaluated. Interface consistency rules and anomaly analysis revealed 2 and 4 faults of 28, respectively.
- In [65] 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 [21] the comparison of their bug finding tool with model checking techniques. They argument 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.
- Bush et al. report in [15] 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 [52] some bug finding tools that use type-based analysis. He shows that they are able to detect race conditions or memory leaks in programs.

Table 13: 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

- We also analysed the effectiveness of three Java bug finding tools in [78]. 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. 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%.

3.7 Defects

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

3.7.1 Defect Introduction

The general probability that a specific possible fault is introduced into a specific program cannot be determined in general without replicated experiments. However, we can give some information when considering defect types. 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 [70]. The distributions (in percentages) can be found in Tab. 13. Interestingly, the trend in this distributions was confirmed in [19] where several open source projects were analysed.

Lutz and Mikulski used for defects in NASA software a slightly different classification of defects in [42] 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.

Tab. 14 shows types and severities of defects following [62]. We observe that logical and data access defects account for most of the serious defects. Furthermore, most of the defects were defects in the specification.

We can see that the defect types are strongly domain- and problem-specific and general conclusions are hard to make.

Table 14: Software faults by category and severity

Category	Serious	Moderate	Minor	Total
Incomplete or erroneous spec.	19	82	239	340
Intentional deviation from spec.	9	61	75	145
Violation of progr. std.	2	22	94	118
Erroneous data accessing	36	72	12	120
Erroneous decision logic	41	83	15	139
Invalid timing	14	25	5	44
Improper handling of interrupts	14	31	1	46
Wrong constants and data values	14	19	8	41
Inaccurate documentation	0	10	86	96
Total	171	478	553	1202
Percentage	14	40	46	100

3.7.2 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.

- Möller [45] gives 25,000 DM as typical removal costs which constitutes a nearly exponential growth over the phases.
- The ratio of the cost of finding and fixing a defect during design, test, and field use is: 1 to 13 to 92 [34] or 1 to 20 to 82 [58]
- The report [43] states that removal costs are 250 staff-hours per field-defect.
- The survey [17] resulted in the effort to detect and correct a defect in the field of 13.5 hours for a defect discovered.
- Following [61] 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. [81] report that the rework per requirements defect in staff-days during maintenance was 0.65.
- In [40] 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.

- In [64] 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 [14].

For the removal costs we have enough data to give reasonably some statistics in Tab. 15. Note that in this summary the mean and median are extremely different. 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 in Fig. 4.

Table 15: Summary of the removal costs of field defects (in staff-hours per defect)

Lowest	Mean	Median	Highest
3.9	57.42	27.6	250

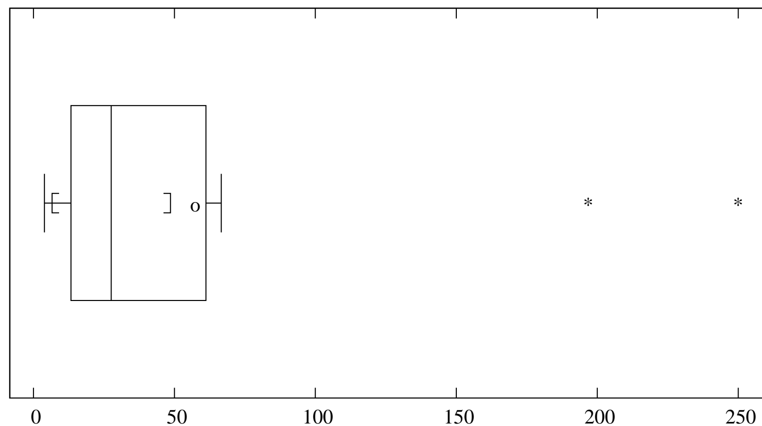

Figure 4: Box plot of the removal costs of field defects in staff-hours per defect

The box plot in Fig. 4 shows two strong outliers that we can eliminate to get a more reasonable mean value. 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. Fig. 5 shows the box plot of the data set without the eliminated outliers.

3.7.3 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 to assign such costs to a single software fault. The highly complex

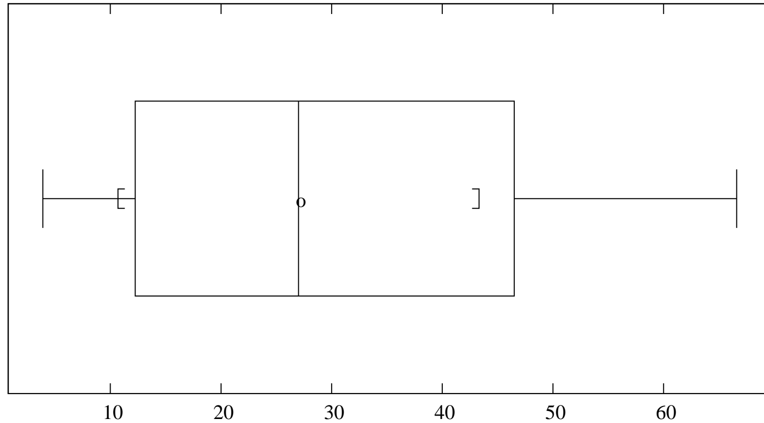

Figure 5: Box plot of the reduced data set of the removal costs of field defects in staff-hours per defect

configurations and the combination with hardware and possibly mechanics of software 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.

Jones [31] 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% or 3%
2. 40% or 15%
3. 30% or 60%
4. 20% or 22%

In [16] is reported that six error types accounted for nearly 80% of the highest severity defects. Nine error types accounted for about 80% of the defects exposed by recovery procedures or exception handlers. They used ODC for classification.

3.7.4 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 [49, 48]. This

relationship can also be supported by data from Siemens when used in a reliability model [77].

4 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 but 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 [31] 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.

5 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, (2) economic-oriented, abstract models for quality assurance in general, and (3) literature surveys of defect-detection techniques. 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 comes 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. A more detailed analysis of the state of the art can be found in [75, 76].

Other surveys on defect-detection techniques are rare but for testing and inspections general literature reviews have been performed. Juristo et al. summarise in [32] 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 [39]. 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. [13] 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 an inspection model.

6 Conclusions

We summarise the main results and contribution of the paper in the following and give directions for further research.

6.1 Summary

We reviewed and summarised the relevant empirical studies on defect-detection techniques that can be used to determine the input factors of an economics model of software quality assurance. The results of the studies were structured with respect to the technique they pertained and the corresponding input factor of the model. The difficulty function is the most complex factor to determine. We introduced two methods to obtain approximation of the factors for the three groups of techniques.

We observed that test techniques tend to be more efficient in defect detection having lesser difficulties but to have larger removal costs. A further analysis in the model might reveal which factor is more important. Furthermore, the removal costs increase also strongly considering different types of tests or reviews, i.e., during unit tests fault removal is considerably cheaper than during system tests. This suggests that unit-testing is very cost-efficient.

6.2 Further Research

We discussed an optimal inspection rate, i.e., the optimal effort per LOC regarding the efficiency of the inspection, and noted that it is not well understood how a deviation from this optimal rate has effects on other factors in defect detection. Hence, further studies and experiments on this would be needed to refine the economics model and improve the analysis and prediction of the optimal quality assurance.

The difficulty of detecting different defect types with different detection techniques should be investigated more thoroughly. The empirical knowledge is extremely limited there although this would allow an improved combination of diverse techniques.

The effect costs are a difficult part of the failure costs. They are a highly delicate issue for most companies. Nevertheless, empirical knowledge is also important there to be able to estimate the influence on the total quality costs.

The collected empirical knowledge on the input factors can be used to refine the sensitivity analysis of the model that was done in [75]. A sensitivity analysis can be used to identify the most important input factors and their contribution to the variation in the output. The mean value and knowledge on the distribution (if available) can be used to generate more accurate input data to the analysis.

We will also extend the economics models by a size metric for better

predictions because several factors, such as the execution costs, are dependent on the size of software.

References

[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] 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.

[4] Boris Beizer. *Software Testing Techniques*. Thomson Learning, 2nd edition, 1990.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] Leland T. Blank and Anthony J. Torquin. *Engineering Economy*. Series in Industrial Engineering and Management Science. McGraw-Hill, 4th edition, 1998.

[10] Barry Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[11] Barry Boehm, LiGuo Huang, Apurva Jain, and Ray Madachy. The ROI of Software Dependability: The iDAVE Model. *IEEE Software*, 21(3):54–61, 2004.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] Jörgen Christmansson and Peter Santhanam. Error Injection Aimed at Fault Removal in Fault Tolerance Mechanisms – Criteria for Error Selection using Field Data on Software Faults. In *Proc. Seventh International Symposium on Software Reliability Engineering (ISSRE '96)*, pages 175–184. IEEE Computer Society, 1996.

[17] James S. Collofello and Scott N. Woodfield. Evaluating the Effectiveness of Reliability-Assurance Techniques. *Journal of Systems and Software*, 9(3):191–195, 1989.

[18] Harris Cooper. *Synthesizing Research. A Guide for Literature Reviews*, volume 2 of *Applied Social Research Methods Series*. SAGE Publications, third edition, 1998.

[19] 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.

[20] Albert Endres and Dieter Rombach. *A Handbook of Software and Systems Engineering. Empirical Observations, Laws and Theories*. The Fraunhofer IESE Series on Software Engineering. Pearson, 2003.

[21] 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.

[22] Michael E. Fagan. Reviews and Inspections. In Manfred Broy and Ernst Denert, editors, *Software Pioneers – Contributions to Software Engineering*, pages 562–573. Springer, 2002.

[23] 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.

[24] 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.

[25] Tom Gilb and Dorothy Graham. *Software Inspection*. Addison-Wesley, 1993.

[26] Robert B. Grady. *Practical Software Metrics for Project Management and Process Improvement*. Prentice-Hall, 1992.

[27] Robert B. Grady and Tom Van Slack. Key Lessons in Achieving Widespread Inspection Use. *IEEE Software*, 11(4):46–57, 1994.

[28] William C. Hetzel. *An Experimental Analysis of Program Verification Methods*. PhD thesis, University of North Carolina at Chapel Hill, 1976.

[29] William E. Howden. Theoretical and Empirical Studies of Program Testing. *IEEE Transactions on Software Engineering*, SE-4(4):293–298, 1978.

[30] Rob Johnson and David Wagner. Finding User/Kernel Pointer Bugs With Type Inference. In *Proc. 13th USENIX Security Symposium*, pages 119–134, 2004.

[31] Capers Jones. *Applied Software Measurement: Assuring Productivity and Quality*. McGraw-Hill, 1991.

[32] Natalia Juristo, Ana M. Moreno, and Sira Vegas. Reviewing 25 Years of Testing Technique Experiments. *Empirical Software Engineering*, 9:7–44, 2004.

[33] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley, 2nd edition, 2002.

[34] 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.

[35] 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.

[36] Barbara A. Kitchenham, A. Kitchenham, and J. Fellows. The Effects of Inspections on Software Quality and Productivity. *ICL Technical Journal*, 5(1):112–122, 1986.

[37] Oliver Laitenberger. Studying the Effects of Code Inspection and Structural Testing on Software Quality. Technical Report 024.98/E, Fraunhofer IESE, 1998.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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).

[44] 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.

[45] K.-H. Möller. Ausgangsdaten für Qualitätsmetriken. Eine Fundgrube für Analysen. In C. Ebert and R. Dumke, editors, *Software-Metriken in der Praxis*. Springer, 1996.

[46] Glenford J. Myers. A controlled Experiment in Program Testing and Code Walkthroughs/Inspections. *Communications of the ACM*, 21(9):760–768, 1978.

[47] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, 1979.

[48] 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.

[49] P. M. Nagel and J. A. Skrivan. Software Reliability: Repetitive Run Experimentation and Modeling. NASA Contractor Rep. 165836, NASA Langley Res. Center, Feb. 1982.

[50] Simeon C. Ntafos. On Comparisons of Random, Partition, and Proportional Partition Testing. *IEEE Transactions on Software Engineering*, 27(10):949–960, 2001.

[51] Don O'Neill. Software Maintenance and Global Competitiveness. *Journal of Software Maintenance: Research and Practice*, 9(6):379–399, 1997.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] Ronald A. Radice. *High Quality Low Cost Software Inspections*. Paradoxicon Publ., 2002.

[57] 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.

[58] Horst Remus. Integrated Software Validation in the View of Inspections / Reviews. In *Proc. Symposium on Software Validation*, pages 57–64. Elsevier, 1983.

[59] Jan Rooijmans, Hans Aerts, and Michiel van Genuchten. Software Quality in Consumer Electronics Products. *IEEE Software*, 13(1):55–64, 1996.

[60] 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.

[61] RTI. The Economic Impacts of Inadequate Infrastructure for Software Testing. Planning Report 02–3, National Institute of Standards & Technology, 2002.

[62] Raymond J. Rubey. Quantitative Aspects of Software Validation. In *Proc. International Conference on Reliable Software*, pages 246–251. ACM Press, 1975.

[63] Per Runeson and Anneliese Andrews. Detection or Isolation of Defects? An Experimental Comparison of Unit Testing and Code Inspection. In *Proc. 14th International Symposium on Software Reliability Engineering (ISSRE '03)*, pages 3–13. IEEE Computer Society, 2003.

[64] Glen W. Russell. Experience with Inspection in Ultralarge-Scale Development. *IEEE Software*, 8(1):25–31, 1991.

[65] 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.

[66] Giedre Sabaliauskaite. *Investigating Defect Detection in Object-Oriented Design and Cost-Effectiveness of Software Inspection*. PhD dissertation, Osaka University, 2004.

[67] 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.

[68] Sun Sup So, Sung Deok Cha, Timothy J. Shimeall, and Yong Rae Kwon. An Empirical Evaluation of Six Methods to Detect Faults in Software. *Software Testing, Verification and Reliability*, 12:155–171, 2002.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

[73] Michiel van Genuchten, Cor van Dijk, Henk Scholten, and Doug Vogel. Using Group Support Systems for Software Inspections. *IEEE Software*, 18(3):60–65, 2001.

[74] Rini van Solingen, Michiel van Genuchten, and Rob J. Kusters. The Impact of EMS Support on Inspections: Description of an Experiment. In *Proc. Thirty-First Annual Hawaii International Conference on System Science (HICSS '98)*, volume 1, pages 575–579. IEEE Computer Society, 1998.

[75] 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)*. ACM Press, 2006.

[76] Stefan Wagner. *Cost-Optimisation of Analytical Software Quality Assurance*. PhD Dissertation, Technische Universität München, 2006. To appear.

[77] 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*. Springer, 2006.

[78] 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.

[79] Edward F. Weller. Lessons from Three Years of Inspection Data. *IEEE Software*, 10(5):38–45, 1993.

[80] Elaine J. Weyuker. More Experience with Data Flow Testing. *IEEE Transactions on Software Engineering*, 19(9):912–919, 1993.

[81] 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.

[82] Murray Wood, Marc Roper, Andrew Brooks, and James Miller. Comparing and Combining Software Defect Detection Techniques: A Replicated Empirical Study. In *Proc. 6th European Conference held jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC '97/FSE-5)*, pages 262–277. Springer, 1997.

[83] 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.