MAGISTERARBEIT

Titel der Magisterarbeit

“Temporal behavior of defect detection performance in design documents: an empirical study on inspection and inspection based testing”

Verfasser

Faderl Kevin, Bakk.

Angestrebter akademischer Grad
Magister der Sozial- und Wirtschaftswissenschaften (Mag. rer. soc. oec.)

Wien, 2009

Studienkennzahl lt. Studienblatt:	A 066 926
Studienrichtung lt. Studienblatt:	Wirtschaftsinformatik
Betreuer:	ao. Prof. Dr. Stefan Biffl
Mitwirkung:	Dipl.-Ing. Dietmar Winkler

Abstract

The quality of software requirements and design documents are success critical issues in software engineering (SE) practice. Organizational measures, e.g., software processes, help structuring the development process along the project life-cycle, constructive approaches support building software products, and analytical approaches aim at investigating deliverables with respect to defects and product deviations. Software inspection and testing are well-known and common techniques in Software Engineering to identify defects in code documents, specifications, and requirements documents in various phases of the project life-cycle.

A major goal of analytical quality assurance activities, e.g., inspection and testing, is the detection of defects as early as possible because rework effort and cost increase, if defects are identified late in the project. Software inspection (SI) focuses on defect detection in early phases of software development without the need for executable software code. Thus, SI is applicable to written text documents, e.g., specification and requirements documents. Traditional testing approaches focus on test case definition and execution in later phases of development because testing requires executable code. Thus, we see the need to combine test case generation and software inspection early in the software project to increase software product quality and test cases.

Bundling benefits from early defect detection (SI application) and early test case definition based on SI results can help identifying (a) defects early and (b) derive test cases definitions for systematic testing based on requirements and use cases. Our approach – inspection-based testing – leads to a test-first strategy on requirements level.

This thesis focuses on the investigation of an inspection-based testing approach and software inspection with respect to the temporal behavior of defect detection with emphasis on critical defects in requirements and specification documents.

The outcomes concerning the temporal behavior showed up some interesting results. UBR performs in the time interval of the first 120 minutes very effective and efficient. UBT-i in contrary needs more time, about 44 % for its testing duration to achieve as good defect detection results as UBR. The comparison of these two software fault detection techniques showed that UBR is on the whole not the superior technique. Because of the inconsistent findings in the experiment sessions a clear favorite cannot be named. Concerning the results for the fault positives the expected temporal behavior, which was that the fewest false positives were found in the first 120 minutes, could not be investigated and the hypothesis on this had to be rejected.

A controlled experiment in an academic environment was made to investigate defect detection performance and the temporal behavior of defect detection for individuals in a business IT software solution.

The results can help project and quality managers to better plan analytical quality assurance activities, i.e., inspection and test case generation, with respect to the temporal behavior of both defect detection approaches.

Kurzfassung

Die Qualität der Software ist natürlich ein erfolgskritischer Faktor im Software Engineering (SE), genauso wie die Design Dokumente in den frühen Softwareentwicklungsphasen. Organisatorische Faktoren, wie etwa der verwendete Software-Entwicklungsprozess, helfen den Prozeß an sich besser zu Strukturieren und zu Optimieren. Entwicklungsansätze unterstützen diesen Prozeß, während analytische Ansätze darauf abzielen Fehler und Produktabweichungen zu vermeiden. Software Inspektionen (SI) und Tests sind bereits bekannte und anerkannte Techniken im SE um Fehler im Software Code, in Spezifikationen oder Design Dokumenten, während verschiedenster Phasen des Produktlebenszykluses, zu identifizieren.

Ein Hauptaugenmerk von analytischen Qualitätssicherungen wie SI und Tests liegt auf der frühen Entdeckung von Fehlern. Denn je später ein Fehler im Produktentwicklungsprozess gefunden wird, desto aufwendiger und teurer ist dessen Entfernung. SI fokussieren auf eine Fehlerfindung in einer sehr frühen Phase des gesamten Prozesses ohne die Notwendigkeit eines Ausführbaren Software Codes. Deshalb ist SI anwendbar auf geschriebene Text Dokument wie Design Dokumente. Traditionelle Testansätze fokussieren auf die Erstellung von Testfällen und deren Exekution in späteren Phasen des Prozesses, weil sie im Gegensatz zu SI auf ausführbaren Code angewiesen sind. Folgernd ist es notwendig Testfallerstellung und SI zu kombinieren, um in noch frühen Phasen die Qualität weiter verbessern zu können.

Die Vorteile beider Ansätze zu vereinen wird helfen um (a) Fehler sehr früh zu finden und (b) Testfälle zu definieren, welche ein systematisches Testen erlauben, daß wiederum auf Anforderungen und Use-Cases basiert. Der Ansatz in dieser These - auf Inspektionen basiertes Testen – wird zu einer „Zuerst Testen“ Strategie auf Anforderungsbasis führen

Diese These konzentriert sich auf einen auf Inspektionen basierten Test Ansatz, sowie auf SI generell mit einer genaueren Untersuchung des zeitlichen Verhaltens dieser Techniken in Design Dokumenten mit Hauptaugenmerk auf sehr kritische und kritische Fehler.

Die Ergebnisse der Untersuchungen des zeitlichen Verhaltens ergaben, daß UBR in dem Zeitintervall der ersten 120 Minuten äußerst effektiv und effizient agiert. UBT-i hingegen benötigt mehr Zeit, ca. 44 % um ein gleichwertiges Ergebnis erzielen zu können. Der Vergleich der beiden Software Fehlerfindungstechniken zeigte weiters, daß UBR ganzheitlich gesehen nicht die überlegene Technik ist. Wegen der inkonsistenten Resultate der Experiment Sessions kann jedoch auch keine überlegene Technik definitiv genannt werden. Betreffend den Ergebnissen der False Positives, konnte das erwartete zeitliche Verhalten, daß die wenigsten False Positives in den ersten 120 Minuten gefunden werden, nicht beobachtet werden. Deshalb mußte die betreffende Hypothese verworfen werden.

Die These basiert auf einem Experiment, welches in einer kontrollierten akademischen Umgebung durchgeführt wurde um die Fehlerfindungseffizienz Einzelner zu untersuchen.

Die Ergebnisse werden Projekt- und Qualitätsmanagern helfen, um deren Qualitätsmaßnahmen besser planen zu können und es weiters ermöglichen deren zeitliche Dauer und daraus folgende Effizienz und Effektivität besser abschätzen zu können.

Table of Content

Abstract I

Kurzfassung II

Table of Content III

1 Introduction 1

2 Product and Process Improvement 8

2.1 Capability Maturity Model (CMM) 9

2.2 The Process of Software Inspection 11

2.3 Roles in inspections 17

2.4 Inspection Team Size 18

2.5 Selection of Inspectors 19

3 Best-Practice Software Inspection 20

3.1 Ad-hoc reading 21

3.2 Checklist-based reading 21

3.3 Perspective-based reading (PBR) 23

3.4 Usage-based reading (UBR) 24

3.5 Comparison of reading techniques 28

3.6 Temporal behavior 30

4 Software Testing and Test-First Development 32

4.1 Black-Box Testing 33

4.2 White-Box Testing 33

4.3 Unit Testing 34

4.4 Test-First Development 35

4.5 Usage-based testing (UBT) 37

5 Research Approach 40

5.1 Variables 41

5.2 Hypotheses 43

5.2.1 Is UBR more Effective and Efficient than UBT-i? 44

5.2.2 Are the Techniques basically effective and efficient in the first 120 minutes? 44

5.2.3 During which time intervals will the fewest False Positives be found? 45

6 Experiment 46

6.1 Experiment Description 46

6.2 Planning and preparation 48

6.2.1 Software Artifacts 48

6.2.2 Reference Defects ... 50
6.3 Operation ... 52
6.4 Evaluation ... 53
6.5 Threats to validity ... 55
7 Results of the Experiment ... 57
7.1 Effort ... 57
7.2 Effectiveness ... 58
7.2.1 Combined Sessions – Combined Techniques ... 59
7.2.2 Temporal behavior of combined sessions and techniques ... 60
7.2.3 Temporal behavior of separated sessions and techniques ... 62
7.3 Efficiency ... 71
7.3.1 Combined Sessions – Combined Techniques ... 73
7.3.2 Temporal behavior of combined sessions and techniques ... 74
7.3.3 Temporal behavior of separated sessions and techniques ... 75
7.4 False positives ... 81
7.4.1 Combined Sessions – Combined Techniques ... 81
7.4.2 Temporal behavior of combined sessions and techniques ... 82
7.4.3 Temporal behavior of separated sessions and techniques ... 84
8 Discussion ... 91
8.1 Is UBR more Effective and Efficient than UBT-i? ... 91
8.2 Are the Techniques basically effective and efficient in the first 120 minutes? ... 93
8.3 During which time intervals will the fewest False Positives be found? ... 94
9 Conclusions and Follow-Up ... 97
References ... 99
Table of Figures ... 104
List of Tables ... 106
Curriculum Vitae ... 107
Appendix ... 108

1 Introduction

Software is an important part of many technical products available on the market in these days and it will become even more important in the future. Software is used in a variety of things, for example, mobile phones, cars, TV sets, coffee machines etc. More complex software is used in more complex systems, like computers and the used software is of course sensitive to any kind of defects made in any development phase. The errors which come from human faults are making the software product fault-prone. This lack of quality ends often in a lost of money as well as reputation, because customers naturally don't want to spend money for low quality software products. But until yet many software products still ship late, with a fewer functionality than originally arranged, higher production costs and with poor quality. A number of factors exist, leading to such unwanted project results. The main contributor is of course the lack of controls for removing defects. Faults are created and injected throughout the whole software development project life cycle into several kinds of artifacts, which seems to be an unfortunate fact of software development. Quality control is therefore very important for organizations developing software products.

The removal of defects with inspections or tests can be a very expensive task, but when the customers find the defects, costs tend to explode and sometime increase by a factor of 100 or more as well as the reputation of the firm and the confidence in the software products are decreased [70]. The costs to remove defects should be calculated just from the beginning and naturally included in the whole cost calculation. As Radice R. [70] states out, that it can happen that these kinds of costs can in some software projects conduct up to 65 % of the total estimated project costs. So there is of course a large economic opportunity in reducing and improving the effectiveness of quality assurance.

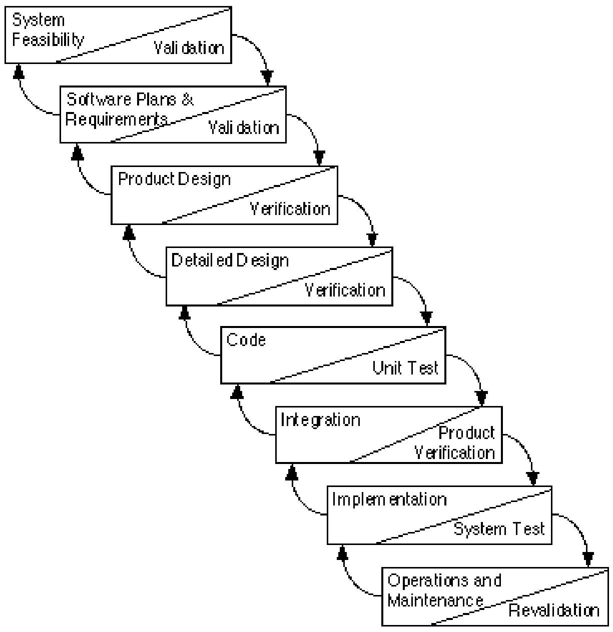
Fagan [32] strongly emphasizes that software inspections have a formal procedure and therefore are able to produce repeatable results. On the contrary walkthroughs are performed not so regularly and thoroughness. He also remarks that in some cases walkthroughs may be identical to formal inspections, but in many cases they are informal and less efficient [48]. Wheeler et al. [101] point out some principal differences between review processes. Knight and Myers [48] suggest that walkthroughs are used to examine the source code and that formal reviews are the presentation of the work product to the rest of the team members and inspections are error detection techniques that ensure particular coding standards and issues are enforced [5]. According to these authors, Fagan's inspection method is a combination of a walkthrough, formal review and inspection [5]. IEEE Standard 1028-1997 [41] provides the following descriptions:

- an inspection is '*a visual examination of a software product to detect and identify software anomalies, including errors and deviations from standards and specifications*';
- a walkthrough is '*a static analysis technique in which a designer or programmer leads members of the development team and other interested parties through a software product, and the participants ask questions and make comments about possible errors, violation of development standards, and other problems'*;
- a review is '*a process or meeting during which a software product is presented to project personnel, managers, users, customers, user representatives, or other interested parties for comment or approval'*.

The software engineering process itself is a process, which has the reputation of being very complex and therefore a number of different models exist, which are trying to improving the process. Conventional models are for example the Waterfall Model, Spiral Model, the V-Model and many others. There are lots of varieties which had been developed and how these models were put into practical work, but all these different approaches of these development models have some activities in their processes in common.

The waterfall model, which was first formally described by Royce W. [72], shown in Figure 1-1, consists of several sequential development phases and each of them can include a verification step which can lead back to the previous phase. These steps backwards give the possibility to correct and so to enhance the product's quality. The weakness of this model is that defect detection in late development phases leads to high expenses.

Figure 1-1 The Waterfall Model [43]

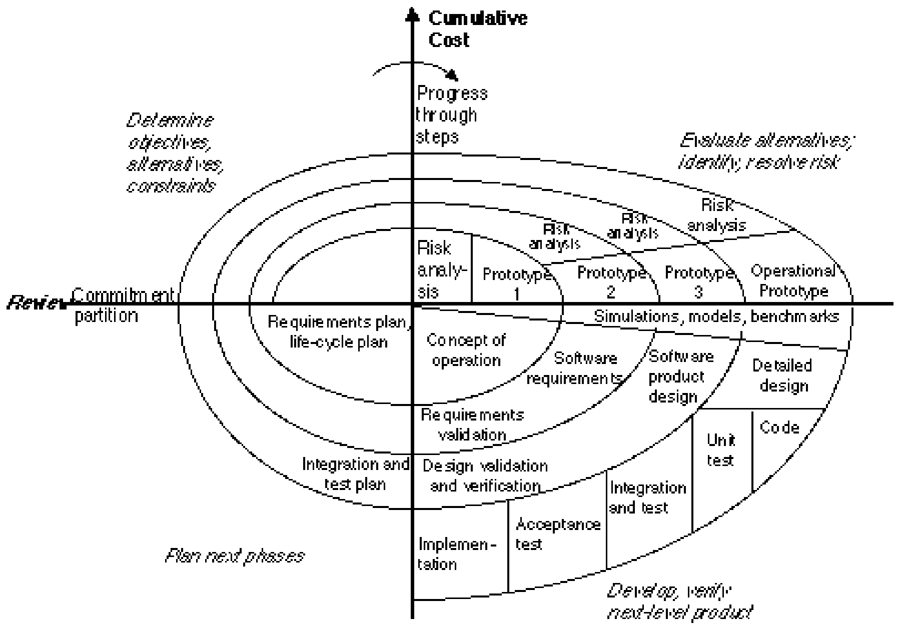


The spiral model, which can be seen in Figure 1-2, was developed by Boehm B. [16] and is like the waterfall model one of the first development models for software engineering. The phases of this model are more complex and have to be passed sequentially whereas in each phase a prototype is developed. The model itself is split into four areas which all phases have to run through:

1. Determine objectives, alternatives and constraints
2. Evaluate alternatives and identify and resolve risks
3. Develop and verify next level product
4. Plan next phase

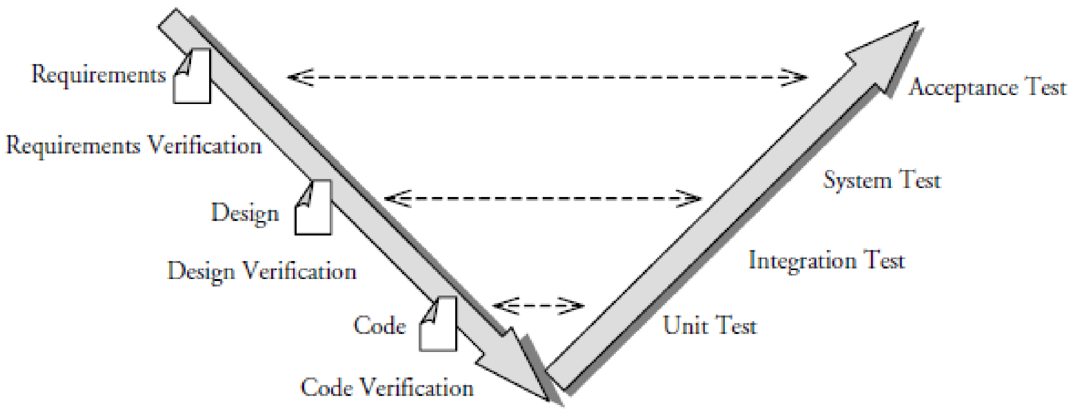
All phases together try to avoid mistakes and wrong decisions in the development process and therefore to enhance the product's quality and at the same time to keep the costs as minimal as possible [16].

Figure 1-2 **The Spiral Model [43]**



The V-Model is shown in Figure 1-3. On the left side of the V the system's specification can be seen and on the right side of the V the verification and validation measurements are listed. Starting from testing each unit step by step the whole system is tested where various verification and validation activities are applied. The model emphasizes the fact, that the activities in the latter part of the project are all about testing implementations of the specifications produces in the earlier part [64].

Figure 1-3 **The V-Model [3]**



The three described models show that quality assurance is always somehow integrated in the development processes. Software inspection is needed and should therefore applied to the process as early as possible to be able to detect defects. Thus in early phases executable software is not present, written text documents, e.g., specification and requirements documents have to be evaluated.

Software inspections and testing methods are mostly relative simple and straightforward to use. Radice R. [70] states out that the most important thing is for sure a belief in its capabilities, application of necessary preconditions and a good management support to make it work to a software organization's best advantage. Because when the management level doesn't support the used software inspection or test method then the programmers and managers will find countless excuses to cause the quality assurance method to fail. When the software managers and software engineers of an organization think that the process will not work, then there is a very good chance that they will fulfill their expectations [70].

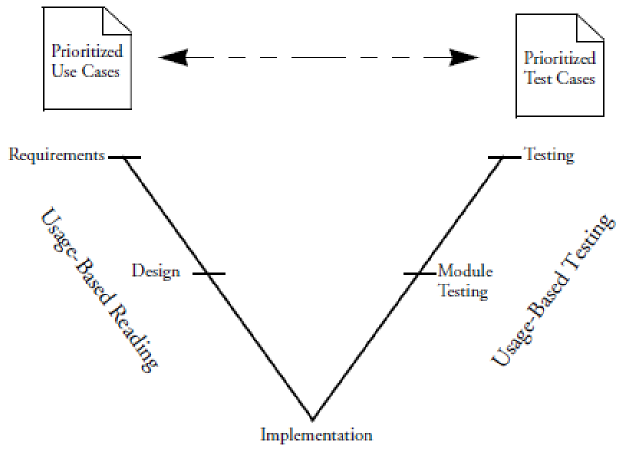
So, when the quality process is given a fair chance by the management and the software engineers and some fundamental things are taken into account, like training of the inspecting participants and a committed time frame for inspections and tests, then the process will work effectively and efficiently [70]:

> *'When practicing inspections one should always work to achieve effectiveness first, then, while maintaining high effectiveness, work to improve the efficiency.'* [70]

Software inspections and tests have the same main goal, which is to detect faults. A lot of different research activities has been made in these areas. They were mostly conducted isolated, but a few studies were made which try to highlight the way on how the methods could benefit from each other [6] [85].

UBR and UBT are focused on detecting the most critical faults from a user's point of view. UBR provides reviewers with prioritized use cases and UBT provides testers with prioritized test cases. Although UBR and UBT are two complementary fault detection techniques, in the software development they have a relationship to each other, which is shown in Figure 1-4.

Figure 1-4 **The connection between UBR and UBT [3]**



For the inspection based testing approach UBT has been improved. Winkler et al. [108] added testing capabilities based on a modification by including inspection methods into the standard usage based testing approach, called "*Usage-based Testing with inspection*" (UBT-i). What means, that the generation of test cases is an additional outcome in contradiction to the standard defect detection. This has some benefits; UBT-i can now also be applied to design specification as well as the generation of test cases has become an integral part of the testing process itself. Now it is therefore possible to compare temporal behavior of the defect detection performance concerning UBR and UBT-i in design documents, which is the main topic of this thesis.

The topic of this master thesis is based on the investigation of an inspection based testing approach and software inspection. The software fault detection techniques UBR and UBT-i will be investigated concerning their temporal behavior of defect detection performance. It should be examined if the most critical defects of the inspected and tested artifacts will be found at the beginning, in the mid or at the end of the inspection and testing duration. This outcome should help project and quality managers to better address and define the needed time to achieve their wanted quality assurance arrangements. Knowing how much time is really needed to detect the most critical defects in software artifacts with the usage of UBR and UBT-i should add a useful and cost reduction benefit to the software development life cycle.

The mentioned techniques will be measured concerning their performance defect detection with effectiveness, efficiency and false positives. All these measures will be investigated in context of the temporal behavior, which will be addressed that the inspections and testing are split into similar time intervals. Each of these time intervals is then examined separately to be able to make conclusions.

The study experiment was made in an academic environment, which provides the base to derive the results from for the software fault detection technique UBR as well as UBT-i and to investigate defect detection performance in context with their temporal behavior.

In section 2, *Product and Process Improvement,* it is explained how the software inspection process works and what is basically needed to go on. Section 3, *Best-Practice Software Inspections,* gives an overview about some often used and well proved inspection techniques as well as section 4 ‚Software Testing and Test-First Development, gives some theoretical background information about the most common testing techniques. In chapter 5, the *Research Approach* explains the variables that exist in the experimental environment as well as the proposed hypotheses. The subsequent chapter 6, *Experiment,* describes all the relevant things about the study design followed by the results made from it in the section 7, *Results of the Experiment*. The *Discussion* in chapter 8, which concerns and addresses al made hypotheses followed by the *Conclusions* in chapter 9 are the final of this master thesis.

2 Product and Process Improvement

Successful software engineering requires the application of engineering principles guided by informed management. The principles must themselves be rooted in sound theory. While it is tempting to search for miracles and panaceas, it is unlikely that they will appear. The best course of action is to stick to age-old engineering principles. There simply are no "silver bullets." [19]

In the early engineering days ships sank and bridges collapsed [68]. Nowadays these accidents occur only rarely because these engineering fields have very well evolved and their procedures are grounded in age-old engineering principles [68].

Software engineering is in comparison a very young discipline and still seeks this kind of evolvement and verified procedures and solutions. A vast majority of scientist research some kind of design patterns to be able to develop proven solutions to common design problems in the software product life cycle. Other computer scientists are also researching in a mathematical way, which addresses methods to verify the correctness and stability of software algorithms. In fact the software engineering community has realized that it is in need of a high-quality software development process to be able to produce high-quality software products [52]. Process standards such as ISO 9000, the Capability Maturity Model (CMM) and the Software Process Improvement and Capability Determination (Spice) have therefore been developed to aid enterprises and people to achieve more predictable results by guiding them to incorporate proven procedures into their process. Normally the companies who adopt the standards advocated in ISO 9000 and CMM have typically shown tremendous improvements in their software quality output.

The term quality is however difficult to define. Therefore, the quality term has been elaborated in terms of six attributes for easier explanation (ISO-9126) [82]. The explanations of the quality attributes below are the ones used by Bass et al [10].

- **Functionality:** *The ability of the software to do work for which it was intended*
- **Reliability:** *The ability of the software to keep operating over time*
- **Efficiency:** *The ability of the software to respond with appropriate speed to a user's requests*

- **Usability:** *The ability of the software to satisfy the user*
- **Maintainability:** *The ability to make changes quickly and cost effectively in the software*
- **Portability:** *The ability of the software to run under different computer environments*

The next chapter describes the Capability Maturity Model (CMM) and how the outcome of this thesis should help to improve the outcome when using a Software Process and Product Improvement reference model.

2.1 Capability Maturity Model (CMM)

The CMM for software is a reference model to examine software process maturity and a normative model for helping software organizations progress along an evolutionary path from ad hoc, chaotic processes to mature discipline software processes [40]. The CMM is organized into five maturity levels as described: [40]

1. **Initial:** *The software process itself can be characterized as ad hoc as well as in some cases chaotic. Few processes are defined, and success depends on individual effort and heroics.*
2. **Repeatable:** *Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.*
3. **Defined:** *The software engineering process for the management as well as the engineering activities are very well documented, standardized, and integrated into the software process for an organization. Projects use an approved, tailored version of the organization's standard software processes for developing and maintaining software.*
4. **Managed:** *Detailed measures of the software engineering process and their quality are collected*.
5. **Optimizing:** *Continuous process improvement is facilitated by quantitative feedback from the process and from piloting innovative ideas and technologies.*

Except for Level 1 each of the described maturity levels is sub-divided into several key process areas that indicate the areas an organization should focus on to improve its software process [40]. These areas are shown in Table 2-1

Table 2-1: CMM Level and Key Process Areas [40]

CMM Level	Focus	Key Process Areas
1 Initial	Competent people and heroics	
2 Repeatable	Project management processes	Requirements management Software project planning Software project tracking and oversight Software subcontract management Software quality assurance Software configuration management
3 Defined	Engineering processes and organizational support	Organization process focus Organization process definition Training program Integrated software management Software product engineering Intergroup coordination Peer reviews
4 Managed	Product and process quality	Quantitative process management Software quality management
5 Optimizing	Continuous process improvement	Defect prevention Technology change management Process change management

The rating components of the CMM, for the purpose of assessing an organizations process maturity, are its maturity levels, key process areas as well as their goals and furthermore every key process area is described by informative components: key practices, sub practices and examples. The key practices are describing as the main infrastructure and activities that contribute most to the effective implementation and institutionalization of the key process area [40].

This thesis affects the CMM **level 2**: *Repeatable in the context the key process areas of software project planning and software quality assurance*, **level 4***: Managed in area software quality management* and in **level 5**: *Optimizing with are defect prevention*. There it should help the management to more precisely define the timely amount, which has to be assigned to inspections and testing durations to get an adequate and acceptable defect detection outcome. To improve the whole software quality management process as well as to improve defect prevention with the capability to detect defects in very early stages in the software process life cycle.

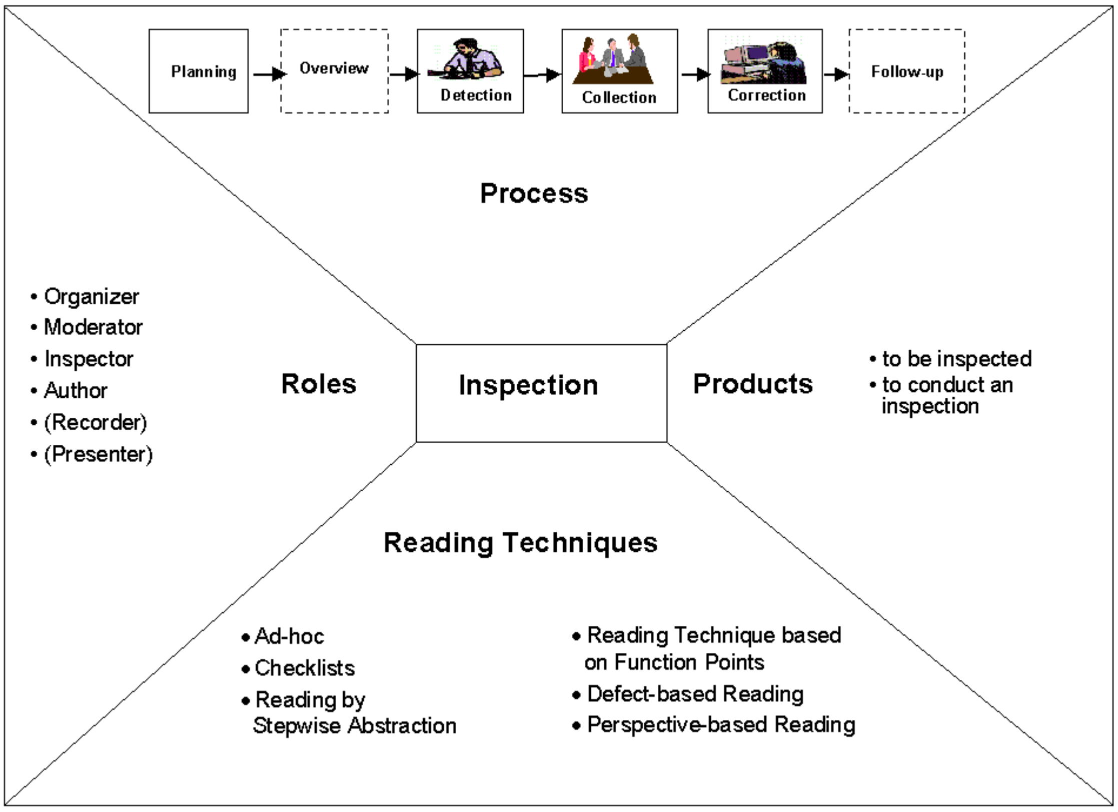
2.2 The Process of Software Inspection

Software inspection is a static method to verify and validate a software artifact manually [34] [83]. Verification means checking if the product is developed correctly or fulfils its specifications. Validation means checking if the correct product is developed or fulfils the customer's needs [58]. It can be applied to hardly any artifact produced during the whole software development life cycle. Unfortunately software inspection is not always applied.

The software inspection is a peer review process, which is normally led by software developers. These developers are normally very well trained in the used techniques [101]. Fagan M. originally developed the software inspection process "*out of sheer frustration*" [31]. It has been more than 30 years since Fagan M. published the inspection process in his famous article in 1976 [32]. Since then the importance of the software inspection process has been raised and many different software firms and developers started using it. Many software developers and researchers engaged in improving the inspection process in the last years. Fagan's inspection method has been studied and presented by many researchers in various forms around the world [5].

The following figure shows the technical dimensions of software inspections. The inspections process, the inspected artifact, the team roles participants as well as their team size and the reading technique. Since the inspections must be tailored to fit many different development situations, it is essential to characterize the technical dimension of current inspection methods and their refinements to grasp the similarities and differences between them.

Figure 2-1 The Technical Dimensions of Software Inspections [62]



A software inspection is a well-structured technique that originally began on hardware logic and moved to design and code, test plans and documentation [30]. The process itself can be characterized in terms of its objective, number of participants, preparation, participants' roles, meeting duration, work product size, work maturity, output products and the process discipline [31]. First it is needed that a very well defined software process has been defined. Is this criterion available and also with an exit-option, then a software product is needed that exactly meets this kind of criterion [12].

A reference model for software inspection processes is needed to be able to explain the various similarities and differences between the inspection methods. To define such a reference model, Laitenberger O. [62] argues, that the purpose of the various activities within an inspection rather than their organization, with which it would be possible to provide a different examination of these approaches. Six major process phases are implemented as depicted in Figure 2-1.

- Planning
- Overview
- Defect Detection
- Defect Collection
- Defect Correction
- Follow-up

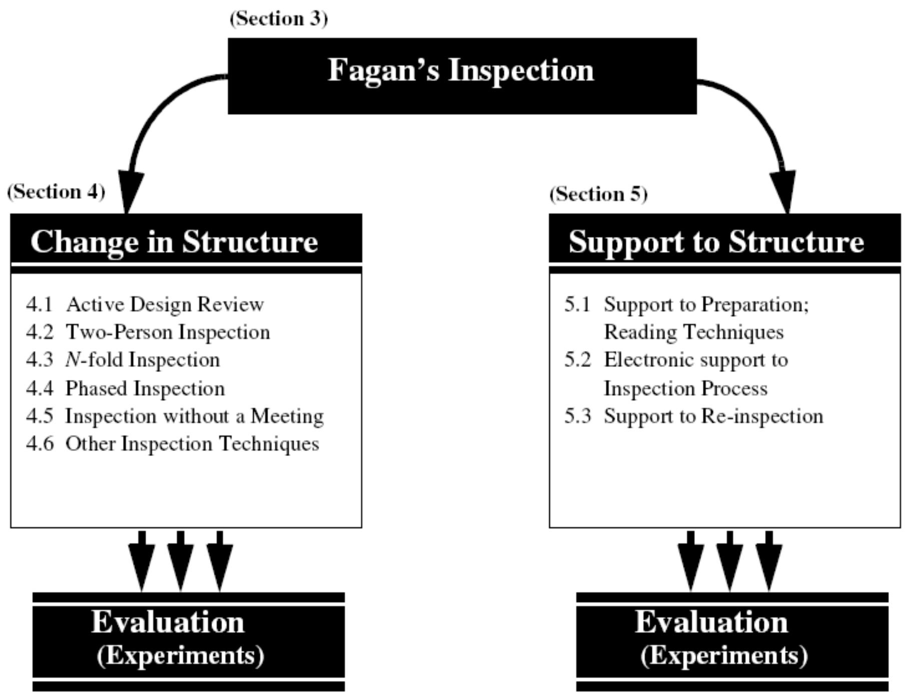
The inspection is performed by a team in which every participant has its well defined role. It is important that the people performing the inspection are familiar with the product as well as having a basic knowledge about the inspection process. If this knowledge is not present they must be trained. The members of the inspection team examine the material individually to learn about the product. After this, the participants attend a meeting in which they have to identify defects. The next step is, that the list of defects found is sent back to the author of the documents. These documents will then be repaired and removed during any of the later stages in the review process [5].

An effective software review process needs to address the relationships of all the required variables in terms of tasks involved, tools and methods used, and the skill, training and motivation of people [5]. Various researchers have made proposals which attempt to improve upon the process of Fagan's inspection method. A literature review reveals two major areas of study, as illustrated in Figure 2-2.

A lot of research of different developers and organizations has been done on the structure of the inspection process. They have developed several new process and models by restructuring the basic processes in Fagan's inspection method [5].

This master thesis focuses on the methods and models that support the structure, preparation of the inspection process.

Figure 2-2: **Evolution of the inspection process with change and support to structure [5].**


(Section 3)
Fagan's Inspection
(Section 4)
Change in Structure
4.1 Active Design Review
4.2 Two-Person Inspection
4.3 N-fold Inspection
4.4 Phased Inspection
4.5 Inspection without a Meeting
4.6 Other Inspection Techniques
(Section 5)
Support to Structure
5.1 Support to Preparation; Reading Techniques
5.2 Electronic support to Inspection Process
5.3 Support to Re-inspection
Evaluation (Experiments)
Evaluation (Experiments)

Planning

In the planning phase the main goal is to organize a particular inspection when artifacts, which have to be inspected, pass specific entry criteria. For example, when a source code successfully compiles without any syntax errors. This phase includes the selection of inspection participants, their assignment to roles, the scheduling of the inspection meeting and the partitioning and distribution of the inspection material [62]. Planning is very important to be a separate phase, because there must be a person within a project or organization who is responsible for planning all inspection activities, even if such an individual plays numerous roles [62].

Overview

The next step is the overview phase. In this phase a first meeting should be made and the author should explain the inspected artifact to the participants. This phase should mainly be used to provide a more transparent view of the inspected artifact to the participants, what makes it easier for them to understand its functionality. Such a first meeting could be particularly valuable for the inspection of early artifacts, such as a requirements or design document, but also for complex source code [62]. On the other hand, does this meeting consumes some effort and therefore increases also the duration of any kind of inspection and it may therefore focus the participants attention on particular issues. These limitations may be one reason why Fagan M. [34] states that an overview meeting for code inspection is not necessary. This statement is supported by Gilb et al. [37], who call the overview meeting the "*Kickoff Meeting*" and point out that such a meeting can be held, if it is desired, but it is not mandatory for each inspection cycle. On the contrary other authors consider this phase essential for effectively performing the subsequent inspection phases. Ackerman et al. [1] for example argued that the overview brings all inspection participants to the point where they can easily read and analyze the inspected artifact.

Laitenberger O. [62] claims, that there are three conditions under which an overview meeting is definitely justified and beneficial:

1. *When the inspected artifact is complex and difficult to understand. In this case, declarations from the author over the inspected artifact make it easier to understand it for the participants*
2. *If the inspected artifact belongs to a large software system, the author should then explain the relationship between the inspected artifact and the whole software system to the other participants.*
3. *When new team members join the inspection team, the author should explain the inspected artifact so that the new team members are also able to inspect it.*

Summarized can be said, that most published applications of inspections report performing an overview meeting, but on the other hand he also says that there are also examples that either did not perform one.

Defect Detection

The defect detection phase can be named as the core of an inspection. The main goal of this phase is to identify the defects of a software artifact. How this phase should be organized best, is still in debate in the literature. Laitenberger O. [62] says that the issue is whether defect detection is more an individual activity and hence should therefore be conducted as part of a group meeting, that is, an inspection meeting. Fagan M. [34] says that a group meeting has very positive influences on the achievement, because participants check the inspection artifact together. He makes the implicit assumption that interaction contributes something to an inspection that is more than the mere combination of individual results. This effect is called the “*phantom*” inspector [34].

In many cases, authors distinguish between a “*preparation*” phase of an inspection, which is performed individually, and a “*meeting*” phase of an inspection, which is performed within a group [1]. However, it is often not really clear for which purpose the preparation phase is performed. It could be for the main goal, which is naturally to detect defect, or just to be able to understand the artifact, which then leads in a later meeting phase to detect defects. For example, Ackerman et al. [1] state that a preparation phase lets the inspectors thoroughly understand the inspected artifact. They say that the main goal of the preparation phase is not explicitly the defect detection.

The literature on software inspection does not really provide a definitive answer on which alternative is best; Laitenberger O. [51] took a look at some literature from the psychology of small group behavior [79] [45] [53]. The conclusion of the psychologists asked, regarding the question if individuals or groups are more effective, depends on the past experience of the persons involved, the kind of task they are attempting to complete, the process that is being investigated, and the measure of effectiveness, because some of these parameters of course vary a little bit in the context of a software inspection [51]. Finally it is recommended that the defect detection activity may be organized as both individual and group activity with a strong emphasis on the individual part [62].

Defect Collection

In most published inspection processes more than one person participates in an inspection and checks a software artifact for defects. Every detected must of course be collected and documented. Also a decision has to be made about every reported defect if it is really a defect, which is the main objective of the defect collection phase. Another objective may be at the end of the phase if the artifact has to be inspected again. The defect collection phase is mostly performed in a group meeting so the decision if the found defect really is a defect or not is often a group decision as well as if to perform a re-inspection. To make the re-inspection decision a more objective one, some authors suggest applying a statistical model, such as a capture-recapture model, for estimating the remaining number of defects in the software product after inspection. If the number is higher than a certain threshold, then the artifact needs to be inspected again [62].

Defect Correction

In the defect correction phase the author has to rework and correct the defects found. To do this the author has to edit the artifact and deals with each reported defect. There is only little discussion in the literature about this activity [60][54].

Follow-up

The main goal of this objective is to check that the author has resolved all defects found in the defect collection phase. To do this, one of the inspection participant has to verify the defect resolution. Apparently do many think, that the follow-up phase is an optional one, like the overview phase [62].

Products

This dimension refers to the product, or artifact which is actually inspected. Boehm B. [15] argues that one of the most prevalent and costly mistakes made in software projects today are deferring the activity of detecting and correcting software problems until late in the project. This statement points out, that software inspections should be made also for early life-cycle documents. Also a look in the literature points out that in most cases inspection was applied to code documents. Code inspection naturally makes the quality of the code a better one and therefore reduces the overall costs, but the reduction can be higher when inspection is used for early life-cycle artifacts [15].

2.3 Roles in inspections

There is not much disagreement regarding the definition of inspection roles in the literature. In the following the different roles are described [62]:

- **Organizer:** *The organizer plans all inspection activities within a project or even across projects.*
- **Moderator:** *The Moderator moderates the inspection meeting and he ensures that the inspection procedures are followed and that team members perform their duties. In this case the, moderator is the key person in a successful inspection as he manages all inspection team and must offer leadership. A special training as well experience for the moderator role is mandatory.*

- **Inspector:** *Inspectors are the backbone of each inspection and are responsible for detecting the defects in the target artifact. Usually all team members can be assumed to be inspectors, regardless of their other roles in the inspection team.*
- **Reader / Presenter:** *If an inspection meeting is made, the reader will present the inspected products at an appropriate pace and lead the team through the material in a complete and logical fashion. The reader should also explain and interpret he material / artifact rather than reading it literally.*
- **Author:** *The author is the one that developed the inspected artifact and is responsible for the correction of defects during rework. During an inspection meeting, the author addresses specific questions the reader is not able to answer. The author must not serve as moderator, reader or recorder.*
- **Recorder:** *The recorder's responsibility is to log all kind of defects in an inspection defect list.*
- **Collector:** *His job is to collect all defects found by the inspectors, if an inspection meeting has not been made.*

2.4 Inspection Team Size

Fagan M. [83] recommends keeping the inspection team quite small, that is, four people and Bisant et al. [12] have found performance advantages in an experiment with two persons: the inspector and the author, who can also be regarded as an inspector. Kusumoto et al. [50] also took a closer look at the two-person approach in an educational environment. Weller [100], on the other hand, uses three to four inspectors in his field study and from Madachy et al. [55] comes out that the optimal size is between three and five people and Bougeois K. [17] confirms these results in a different study. Porter et al.'s [66] experimental results are, that the reduction of the attendant inspectors from four to two significantly reduces the effort but does not increase the effectiveness of the inspection.

It can be seen that in the literature there is unfortunately no definitive answer to the optimal number of inspectors and team size. The size should better be modulated in relationship to the type of the artifact and the environment in which the inspection is performed as well as the costs associated with defect detection and correction in later development phases. Normally it is recommended to start with one team, consisting of three to four people: One must be the author, one or two inspection participants and also one moderator is needed. The Moderator should also play the role of the presenter. When a few inspections are made the benefits of changing the team member size can be empirically evaluated, but the question if the effort for the extra person really pays off [62].

2.5 Selection of Inspectors

The best inspectors are of course the people, who are also involved in the development process of the software artifact itself [96]. Also external inspectors could be taken into account if they have special experience and or knowledge that would have a positive influence on the inspection [69]. The chosen inspectors should also have a good experience as well as knowledge about the artifact [96] [46] [34]. This often limits the possible inspectors to only a small number of developers working on similar artifacts. Also personal with only little experience are mostly not chosen as inspectors although they would learn about the artifact and so could profit a lot from inspections. With the use of reading techniques this problem can widely be avoided.

Managers should mostly not attend or participate in an inspection [61] [69], because they do not really concentrate on the quality of the artifact but more on the quality of the people who created the artifact [96].

3 Best-Practice Software Inspection

There exist a considerable high number of studies that focus on methods and tools to support the preparation of the inspection process. This Section reviews different reading techniques and states out why UBR is mainly used for this investigation.

It is very important that the inspector has an understanding of the artifact, which will be inspected. Otherwise he wouldn't be able detecting defects if the artifact tends to be very complex, which is often the case. On the whole, a reading technique is just a procedural method for the individual inspector to detect defects in the inspected artifact. At least, it is intended that inspectors use the available reading techniques since this makes the result of the defect detection activity less dependent on human factors, for example experience.

Multiple reviewers are able to identify several potential defects in the reviewed artifacts when using a defined reading technique. A few techniques are available that are proven to be more effective to support these kind of activities. Researchers all agree that the choice of the reading techniques has a potential impact on the measured inspection performance and is therefore very import for the whole process [5].

To improve the quality as well as the amount and the fault searching process used for software inspections, a number of different reading techniques have been developed. Some of the most often used reading techniques are [65]:

- ad-hoc reading
- checklist-based reading
- perspective-based reading
- usage-based reading.

As different as these reading techniques are, they have a common general goal, which is to help the reviewers to become and stay focused during the inspection of a certain software document and thereby to detect more faults [65].

Reading techniques are classified as systematic techniques and non-systematic techniques [66] [81]. The systematic reading techniques such as perspective-based reading, apply a highly explicit and structural approach to the process. It provides a set of instructions to reviewers and explains how to read the software document and what they should especially look for [37]. The non-systematic reading techniques, such as ad hoc reading or checklists based reading on the other, apply to an intuitive approach and offer little or no support to the reviewer. A number of empirical studies have also been made to compare the performance of reading techniques by measuring the overall number of defects found from every inspected technique [5].

The following sections gives an overview of the most commonly used forms of reading techniques.

3.1 Ad-hoc reading

Ad-hoc reading, by default, offers only very little reading support at all since a software product is just given to an inspector without any comments, explanations or guidelines on how to proceed through it and as well as on what a special look should be taken. So this reading technique takes a very general viewpoint of reviewers and is denoted when no specific reading technique is used. However, ad-hoc does not mean that inspection participants do not scrutinize the inspected product systematically. The reviewers don’t need to be trained and there is no defined procedure which they can follow. Instead the reviewers have to use their own skill, knowledge and experience to identify faults in the documents.

Laitenberger [62] argues that also training sessions in program comprehension as presented in [28] may help subjects develop some of these capabilities to alleviate the lack of reading support. Also only a few times in the literature the ad-hoc reading approach was really used, but many articles were found in which only very little was mentioned about how an inspector should proceed in order to detect defects. He assumed that in the most of these cases no particular reading technique was provided, because otherwise it would have been stated [5]. Summarized: Ad hoc reading doesn’t have any support to give to the reviewers [5].

3.2 Checklist-based reading

This reading technique is a more systematic and structured one than ad-hoc reading. The original procedure developed by Fagan [32] included the use of checklists. The reviewer works through a list, in which questions has to be answered or ticks a number of predefined issues that have to be checked. The questions are expected to guide the reviewer throughout the whole inspection process [5].

The major goal is defining the responsibilities regarding the reviewers and providing guidance to them helping to identify as many defects as possible. After Gilb et al. [37] have the checklists to be developed from the project itself. The preparation of each individual type of documentation has to be done for each different type of product and also for each process role. The checklist is important, because it helps to concentrate on questions that it is easier for reviewers to identify major defects or prioritize different defects [5]. A checklist should be no more than one single page for each type of documentation [37]. In some cases the length of a checklist may exceed one page. In these cases, it may be possible to make inspectors responsible for different parts of the checklist [62].

Although reading support in the form of a list of questions is better than nothing (such as ad-hoc reading), checklist-based reading has several weaknesses [62]. The given questions are often kept in a general theme and are not sufficiently tailored to a particular development environment. So, the checklist often provides only very little support for an inspector to understand the inspected artifact, which can often be essential to detect, for example, major application logic defects. Also a detailed instruction on how the checklist has to be used is often not made. Therefore in some cases it stays quite unclear when and also based on what kind of information an inspector has to answer a particular question of the list.

Actually several strategies are possible addressing all the questions in a checklist as followed: The participant takes a question and then reads through the complete artifact answering the questions. Afterwards the next question has to be taken. But this procedure is also quite common: The participant reads through the complete document and afterwards the questions of the checklist are answered. It is quite unclear which approach participants mostly follow when using a checklist and how they achieved their results in terms of defects detected. Another problem of checklist-based reading is that checklist questions are often limited to the detection of defects that belong to particular defect types. Inspectors may often not focus on defect types not previously detected and, therefore, may miss whole classes of defects [62].

With the discussed problems we are now able to develop a checklist according to the following principles [62]:

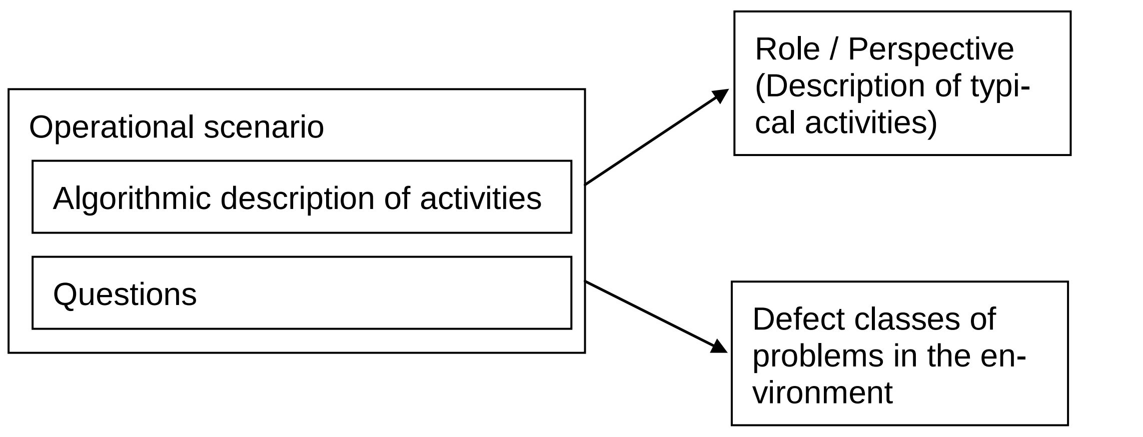
- The length of a checklist should not exceed one page.
- The checklist question should be phrased as precise as possible.
- The checklist should be structured so that the quality attribute is clear to the inspector and the question give hints on how to assure the quality attribute.

Although these actions can be taken, a checklist still provides only little guidance for inspectors on how to perform the various checks. This weakness led to the development of more procedural reading techniques [62].

3.3 Perspective-based reading (PBR)

Perspective-based reading (PBR) was originally developed and experimentally validated at NASA [51]. PBR is an enhanced version of scenario-based reading. The technique focuses on the point of view or needs of the stakeholders [5]. Each scenario consists of a set of questions and a scenario itself is a viewpoint of an algorithmic description. The description shows activities as well as questions of the inspected document and from which an abstraction can be build. Afterwards finally this abstraction has to be analyzed, which is developed based on the knowledge about the environment. In this environment the reading process then is applied: roles in the software development process and defect classes as shown in the Figure 3-1.

Figure 3-1: Description of the PBR-Model [86]


Operational scenario
Algorithmic description of activities
Questions
Role / Perspective (Description of typical activities)
Defect classes of problems in the environment

M. Ciolkowski [86] describes the activity of a scenario should be a description on how to build an abstraction of the inspected document. An activity should be typical for a particular role within the software development process. The role has to determine the perspective from which the reader is to inspect the document, typically a customer or consumer of the corresponding document. A question is an interrogation of the reader about the activity [86], i.e. the process of building the abstraction or the result of the activity. The questions are derived from defect classes or problems that are typical for the product or for the environment. The question on the scenario should not be compared to the tick-list of a checklist.

Basili et al. [51] made a number of different experiments at the NASA. These experiments tried to investigate the effectiveness of PBR on, for example, requirements documents. Unfortunately they found no mentionable difference in the performance and in the number of defects found of reviewers who used their own usual technique and those who were using PBR, but reviewers performed significantly better on the generic the generic documents [5]. Laitenberger et al. [92] also found no significant performance differences when they ran a more detailed experiment using PBR on code documents at Robert Bosch GmbH. Shull et al. [37] pointed out that PBR is suited to reviewers with a certain range of experience. These authors argued that reviewers using PBR on kinds of requirements documents detect more defects, in contrary to those wo use, for example, less structured methods. They also emphasized that PBR has beneficial qualities because it is systematic, focused, goal-oriented, customizable and transferable via training [5].

3.4 Usage-based reading (UBR)

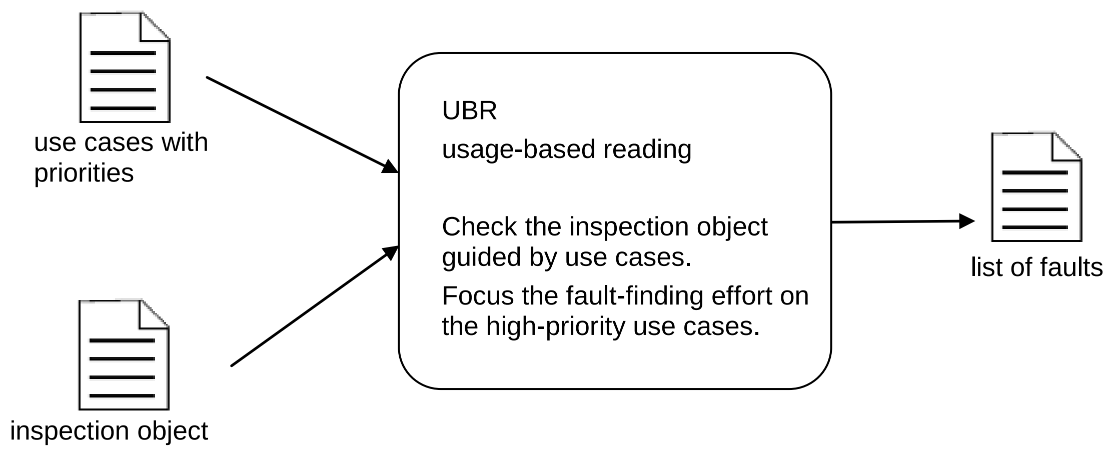
The preparation of software inspections, which is made by individuals, enlarged its focus from only comprehension, initially proposed by Fagan [33] to also comprise fault searching. The aim of many reading techniques is to find as many faults as possible, albeit of their importance. The inspection effectiveness in most cases measured in numbers of faults detected, without taking into account that some defects in the inspected object tend to affect the system quality a lot more than eventually others do [91]. What is again a very important point when costs should not exceed expectations, because critical failures are mostly more complex than non-critical failures and therefore they will need more time to fix. So UBR can help to reduce costs.

The idea behind UBR is to focus on detecting the most critical faults in the inspected artifact. The defects are not assumed to be of equal importance and therefore UBR concentrates on finding the most critical ones from the users’ point of view, which are most dangerous to the overall system quality. The UBR method focuses the reading effort guided by a prioritized, requirements-level use case model [91].

A use case represents how the system can be used, viewed as a set of related transactions performed by an actor and the system in dialogue [34] [24]. The basic idea of modeling usage from an external point of view by describing different usage scenarios is practiced in industrial requirements engineering in various contexts and ways [42]. Industrial software development projects often produce a set of use cases that represents the principal way of using the system, and the set of use cases typically acts as a basis for system design and testing [63].

The background of UBR is from operational profile testing [74] and the user perspective in object-oriented development [9] [63]. UBR utilizes the set of use cases as a vehicle for focusing the inspection effort, much the same way as a set of test cases focuses the testing effort [77]. The use cases should show the inspectors how to inspect the document in a similar way as the test cases show the testers how to test the system [91]. Figure 3-2 shows the input and results of UBR.

Figure 3-2: Input and Output of UBR. [91]


use cases with priorities
inspection object
UBR
usage-based reading
Check the inspection object guided by use cases.
Focus the fault-finding effort on the high-priority use cases.
list of faults

A very important thing concerning the inspection effort in UBR is the prioritization of use cases. UBR assumes that a set of use cases is prioritized in a way which reflects the desired focusing criterion. If the inspection is aimed at finding the faults that are most critical to the system quality, the use cases should be prioritized correspondingly

[91]. The use cases may, for example, be prioritized through pair-wise comparison using the *analytical hierarchy process* (AHP) [33] [96] with the criterion:

"Which use case will impact most negatively

on the system quality if it is not fulfilled?"

The use cases are prioritized before an inspection session and they should be made by some potential users or by someone who is very familiar with the usage of the software. The use cases can be utilized for hardly any kind of inspections, like requirements documents, design documents or source code. This applies to a specific project and has to be done only once for the duration of the whole software project. The inspectors then read through the whole documents and manually execute the use cases in the defined order. During this process they try to detect as many defects, which are most critical and therefore important according to the prioritization and therefore also to the users [65].

As told before, UBR is kind of operational profile testing, which takes the inspector into the user perspective. This is quite the same way as a set of test cases focuses the testing effort. The use cases give the reviewers the guidance how to inspect a design or code document in a similar manner as the test cases tell the testers how to test the system. The individual inspection of a design document using UBR is performed in the following basic steps [65]:

- **Before inspection:** *The use cases have to be prioritized in order of importance from a user's point of view.*
- **Preparation:** *To read through the whole design document to be inspected, the use cases should try to guide the reading. The requirements document is used as a reference to which the design is verified.*
- **Individual inspection:** *Inspect the design document by following the procedure:*
    1. *Select the use case with the highest priority.*
    2. *Trace and manually executing the use case through the design document and use the requirements documents as a reference.*
    3. *Ensure that the document under inspection fulfills the goal of the use case, that the needed functionality is provided, that the interfaces are correct etc. indentify and report the issues found.*

4. *Repeat the inspection procedure using the next use case until all use cases are covered, or until a time limit is reached.*

Two variants of the UBR method are defined, *ranked-based reading* and time-controlled reading [65].

Ranked-based reading, which is the basic form of UBR, prioritizes the use cases with respect to the importance from a user's perspective. A reviewer who uses the ranked-based reading variant follows the use cases in the order in which they appear in the ranked use case document. Time-controlled reading adds a time budget to each use case in order to force a reviewer to utilize a specific use case the specified time. Time budgets are given to each use case and are normally longer for use cases which have a higher rank and less time budgets for use cases with a lower rank. By using this kind of prioritization method, it would be possible to derive the relative priority $p_i, (0 \leq p_i \leq 1, \sum p_i = 1)$, of each use case U_i. Based on this, UBR may be carried out as follows: [91]

[1] Decide on the total time T to be spent on reading of artifact A

[2] Assign the time $T_i = p_i * T$ to each use case U_i

[3] For each use case U_i, inspect A for a period of T_i by "walking through" the events of p_i

and decide if A is correct with respect to U_i [91].

UBR is a novel reading technique which differs a little bit from the other reading techniques. Although UBR is related to PBR there are some differences between these two techniques. The relation to PBR is the utilization of the user perspective. However, UBR focuses only on the users and guides the reviewers based on the users' needs during an inspection by providing the reviewers with developed and prioritized use cases [65]. In PBR on the other hand different perspectives are used to produce artifacts during an inspection. The reviewers that apply the user perspective develop use cases based on the inspected artifact and thereby find faults. In UBR, the use cases are used as a guide through the inspected artifact. The main goal of UBR is naturally to improve the efficiency as well as the effectiveness by directing the inspection effort to the most important use cases form a user's viewpoint. Despite PBR has the goal of improving the effectiveness by minimizing the overlap of defects that the reviewers tend to find. The latter is, however, not always achieved [1].

Another practical difference exists between PBR and UBR [65]. PBR is a reading technique that can be used with hardly all artifacts produced during a software development lifecycle, if the developed scenarios for PBR are general. In PBR, the term scenario is a metalevel concept, denoting a procedure that a reader of a document should follow during an inspection [65]. That means that for example scenarios which have been developed for requirements documents may be used for all requirements documents. However, the same scenarios cannot be used for design or code inspections. On the contrary, UBR scenarios are specific to each project, which means that the used cases can only be utilized within the project they are developed for [65], but on the other hand they can be used for requirements design as well as for code inspections in that project. In addition, they may also be used for test specification development as well as inspection [65]. This is one of the greatest benefit and also the reason why it is used in this master thesis.

3.5 Comparison of reading techniques

This section is about to give an overview about examined experiments and their results as well as a comparison of reading techniques made by Laitenberger [62]. A general prescription about when to use which reading technique cannot really be done. But a comparison between them has been set up following these criteria to provide answers to the following questions [62]:

- **Application Context:** To which software artifact can this reading technique be applied and to which software artifact has this reading technique already been applied?
- **Usability:** Can the reading technique give you guidelines how the software artifact can be checked for detecting defects?
- **Repeatability:** Are the results that the inspector found during inspection repeatable, that means, will another person detect the same defects in the software artifact?
- **Adaptability:** Can the reading technique be adapted to particular aspects, for example the notation of the document, or typical defect profiles in an environment?
- **Coverage:** Are all required quality properties of the software product, such as correctness or completeness, verified in an inspection?
- **Training required:** Is it required that the inspectors are trained in the used reading technique?

- **Validation:** How was the reading technique validated, that is, how broadly has it been applied so far?

Table 3-1 below shows the characteristics of each reading technique according to these criteria. Question marks are used in cases for which no clear answer can be provided at this time.

Table 3-1: Characterization of Reading Techniques [62]

Reading Technique	Characteristics						
	Application Context	Usa-bility	Repeat-ability	Adapt-ability	Cover-age	Training required	Validation
Ad-hoc	All Products	No	No	No	No	No	Industrial Practice
Checklist	All Products	No	No	Yes	Case dependent	No	Industrial Practice
Reading by stepwise Abstraction	All Products allowing abstraction, Funct. Code	Yes	Yes	No	High for correctness defects	Yes	Applied primarily in Clean room projects
Defect-based reading	All Products, Requirements	Yes	Case Dependent	Yes	High	Yes	Experimental Validation
Perspective based reading	All Products, Requirements, Design, Code	Yes	Yes	Yes	High	Yes	Experimental Validation and Industrial Use
Traceability based reading	Design specifications	Yes	No	No	High	Yes	Experimental Validation
Usage based reading	All Products, Requirements, Design, Code	Yes	Yes	Yes	High	Yes	Experimental Validation

It can be seen that UBR is achieving quite good results in all questions. Next, UBR will be compared in already examined experiments and it will be shown, that this inspection technique is making good results here too, see Figure 3-3 below. Normally four different variables are compared: effort, effectiveness, efficiency and false positives. All these studies were conducted in a controlled academic environment.

Figure 3-3: Studies on UBR

Study (author, title, year)	Compared techniques	Superior technique
Thelin T. et al, "*Prioritized Use Cases as a Vehicle for Software Inspections*", 2003 [89]	UBR – CBR	UBR
Thelin T. et al, "*An Experimental Comparison of Usage Based and Checklist-Based Reading*", 2003 [92]	UBR – CBR	UBR
Thelin T. et al, "*A Replicated Experiment of Usage Based and Checklist-Based Reading*", 2004 [88]	UBR – CBR	UBR
Winkler D. et al, "*Investigating the Effect of Expert Ranking of Use Cases for Design Inspection*", 2004 [107]	UBR – UBR-i – CBR	UBR
Winkler D. et al, "*Investigating the impact of Active Guidance on Design Inspection*", 2005 [106]	UBR – CBR	UBR

The investigations of Thelin T. et al. [89], [92] and [88] figured out that UBR is regarding efficiency and effectiveness significantly better than CBR. Defects were also classified by the defect severity classes and inspectors who had to apply UBR found measurable more crucial as well as important defects than inspectors which had to deal with CBR.

Winkler D. et al. observed in both studies [107] and [106] that effort of all investigated techniques is quite similar. But when it comes to effectiveness and efficiency UBR is performing better than CBR. False positives where also examined in these studies were as a result UBR achieved also better results than CBR.

3.6 Temporal behavior

A lot of different investigations about reading techniques have been made so far, but the temporal behavior is a point in which the related work searched, tends to have a gap. Therefore this Thesis tries to find answers on when is which software fault detection technique basically performing at its peak level, meaning during which time intervals, will the most critical defect be found by the participants.

Summarized can be said that UBR achieved good results compared with several different reading techniques as well as compared with them in different experiments investigated in detail in the previous chapter. Thus this technique is worth to have a closer look on its temporal behavior and in comparison with a testing method that also focuses on the users' perspective as well as on the most critical and important defects in design documents. The temporal behavior of the software fault detection techniques will be measured by effectiveness, which is the number of matched defects (= number of seeded defects found by a participant) in relation to the overall number of seeded defects per individual defect severity class in a certain time interval and efficiency, which is the number of matched defects found per certain time interval, for example 60 minutes.

The main outcome of this thesis will be the temporal behavior, meaning in which time interval, UBR and UBT-i are performing most effective and efficient as well as find the most critical defects in the inspected software artifacts. This adds a benefit to the knowledge about these software fault detection techniques, making it possible to better define and more precisely determine the optimal inspection and test duration, or to be able to control which kind of defects the inspectors should mainly search by only altering the duration of the inspection or test.

4 Software Testing and Test-First Development

Testing has of course the same challenge that reading techniques have, to find defects as early as possible in the specified artifacts and therefore to improve the quality of the software product as well as to reduce the overall costs. This section gives an overview about the typical testing approaches like black-box, white-box testing and unit testing, test-first development as well as a detailed view on usage based testing and its adaption.

Normally a test plan is made, which includes several test-cases [15]. These test-cases define the work of the testers and covers the complete functionality of the project. It is also important to say that trial and error testing during the implementing sessions is not really testing. It is also important that in most cases the person who has the role of the implementer not also gets the role of the tester.

The test protocol is the output when running test cases against a defined system. It is of course necessary that the tester writes down the false behavior of the system and, if available, the unique error number for the subsequent bug fixing processes that then have to come.

Test reports are normally produced after testing, for example after one week. If the testing process is automated, such reports can be produced periodically, for example every week. These documents are of great importance for the management to be able to make decisions, as well as for the development team to give them feedback about the quality of their work.

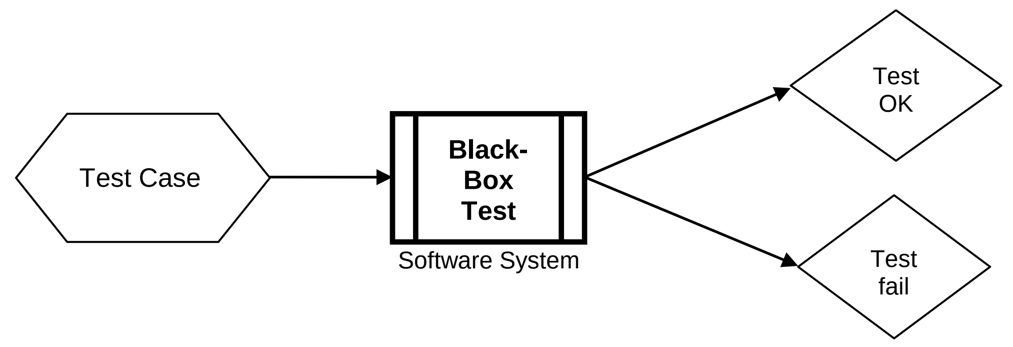
Software testing methods are traditionally divided into black box testing and white box testing. In some cases also the terms behavioral and structural are used, although behavioral test design is a little bit different from black box testing. This is, because a knowledge of the internal the tested system is not forbidden at all, but it is still discouraged. These two different methods are mostly used to describe the point of view that a test engineer uses when designing his test cases. Black box and white box are test design methods, whereas unit testing or usage based testing, which will also be explained in the following chapters, are testing processes which conduct a different level of testing. Also each level of testing can use any test design method. But unit testing is usually associated with white box testing, whereas usage based testing on the other hand is usually associated with black box testing.

4.1 Black-Box Testing

Black-box testing is also known as functional testing. These are testing techniques that have an external view on the system and test cases are generated without knowledge of the interior of the system, see Figure 4-1. Only the input and the output are of importance for the test cases. Therefore is a successful black-box test no guarantee that the software is really faultless, because specifications made in early phases of the software development life cycle cannot be proven if they have been implemented in the right way. The developer of the test cases must not have knowledge about the functionality of the system, therefore a separated team for the creation of the test cases is necessary.

The tester takes for example the role of a user and proves the test cases which were worked out in advance.

Figure 4-1: Black-Box Testing

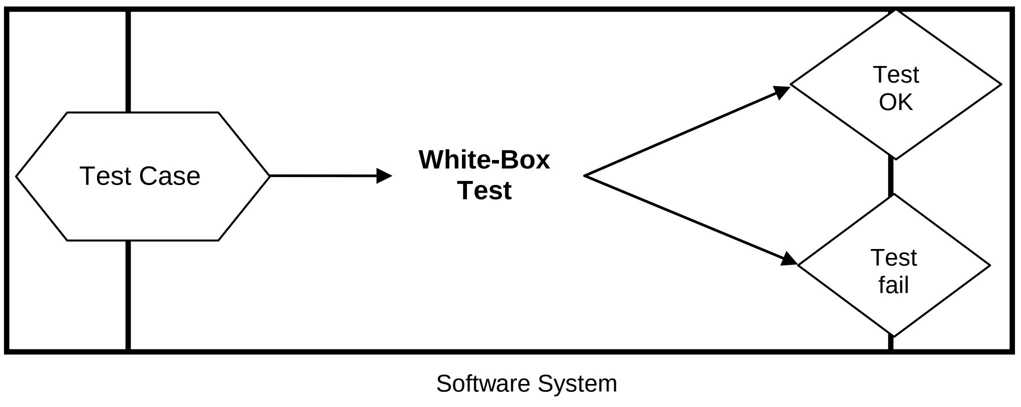


4.2 White-Box Testing

White-box testing techniques take an internal view, as shown in Figure 4-2, and aim at covering all paths in the code or all lines in the code in contrary to black-box tests. White-box tests are made with knowledge about the internal functionality of the system. So they focus on testing source code where the coverage is important.

Should also subparts of the system been tested, is it necessary to know a lot about their functional behavior. So they are also very suitable to localize known defects in those subparts of the system and therefore to identify the component which is responsible for the defect. White-box tests alone are as well as black-box test insufficient to guarantee a failure less software product. A meaningful test series should combine black-box and white-box tests. The programmers of the code have of course a very good knowledge about the system and its functionality and therefore it makes sense that the same persons also develop the white-box tests. So it is normally that there is no separated team needed that makes these test cases. It would also be very extensive to instruct a new team to the software system that should be tested, what is not needed for the system developers [102].

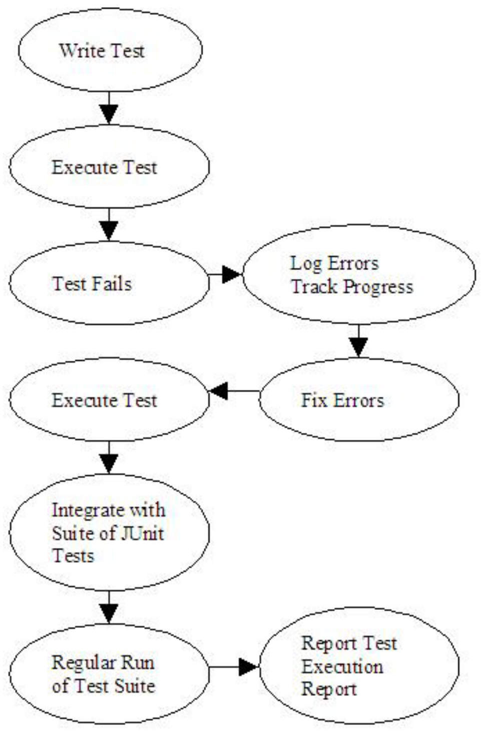
Figure 4-2: White-Box Testing



4.3 Unit Testing

In unit testing, which is traditionally a white box testing method; a programmer tests an individual part or unit of a source if it is faultless. Therefore each unit is viewed and tested isolated. The size of a unit in this correlation can be from the smallest parts of a program to methods or even components [98]. These kind of tests are typically written and run by the software developers itself. The implementation can vary from being completely manual, like paper to being formalized as part of build automation, but commonly it is automated. Normally a strict written contract is provided that the piece of code must satisfy. Also all test cases are independent of each other [97]. The Figure 4-3 below, illustrates the unit testing procedure for the Junit approach.

Figure 4-3: **Unit Testing Process for the Junit Approach [97]**



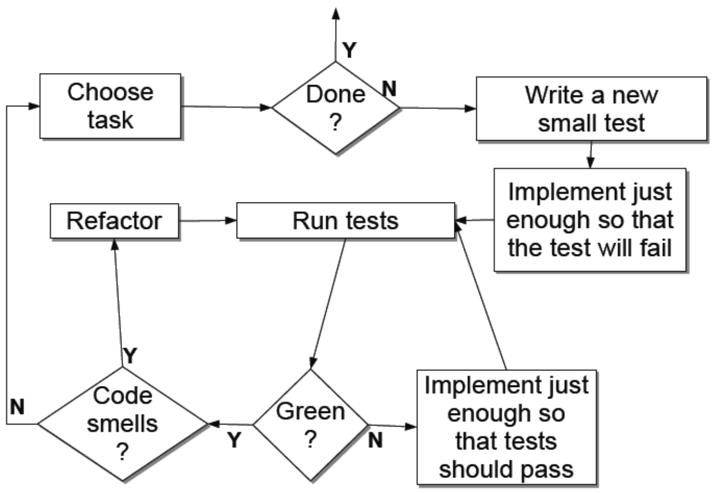
Unit testing even provides a sort of living documentation for the specified system. The software developers can take a look at the unit tests to get knowledge about how to use the unit and also to get a basic understanding of the unit API [98]. The success critical characteristics of the unit can naturally indicate if the use of it was appropriate or inappropriate. On the other hand, an ordinary documentation, which has a kind of a narrative character may sometimes drift away from the implementation of the program and will therefore sooner be outdated. Especially when design changes happen or relaxed practices are common when it comes to keep documents up to date [98].

4.4 Test-First Development

In Test-First Development (TFD), which is often also called Test-Driven Development (TDD) the developer writes automated unit test cases before writing implementation code for the new functionality they are about to produce. Therefore this testing process is also usually associated with the white box testing method. When the developer has written these test cases, which will generally not even be compiled, the developer then starts to write the implementation code to pass these test cases created in advance.

The developer writes some test cases, implements the code, writes some test cases, implements the code, and so on, see Figure 4-4. The whole work is kept within the developer's intellectual control, because he is continuously making small design and implementation decisions and increasing functionality at a relatively consistent rate [56]. A new functionality will not be implemented unless any unit test case has been written for the code and also run properly through the test.

Figure 4-4: Test-First Development [4]


Y
Choose task
Done ?
N
Write a new small test
Implement just enough so that the test will fail
Refactor
Run tests
Y
N
Code smells ?
Y
Green ?
N
Implement just enough so that tests should pass

These are some benefits of test-first development [56]:

- *By using TFD the gap between decision (design developed) and feedback (functionality and performance) can be reduced. Meaning that the fine granular test-then-code cycle would be able to give a constant feedback to the developers.*
- *TFD intends the developers to write a kind of code which is automatically testable, such as having functions or methods returning a value, which can be checked against expected results*
- *With the use of these automated test cases generated in advance, it is easily possible to identify if a new change in the code breaks anything in the existing system. This also allows a smooth integration of new functionality into the code base of the system*

4.5 Usage-based testing (UBT)

Traditional testing is often concerned with the technical details in the implementation, for example: branch coverage, path coverage and boundary-value testing [86]. UBT [44] on the contrary takes the view of the end user, so UBT is a black box testing approach taking the actual operation behavior into account. The focus is not to test how the software is implemented, but how it fulfills its intended purpose from the users' perspective [73]. The same focus as UBR has and therefore the original definition of UBT is similar to UBR. The workflow is defined by the prioritized test-cases in pre-given order. As the v-model of Figure 1-4 on page 6 shows, UBR could be prior to implementing, while UBT is normally conducted after implementing.

Several testing techniques have been empirically evaluated and also compared with different inspection techniques [6] [85]. UBT again was developed to focus on the users and to estimate the reliability [75]. Andersson C. et al. [3] also compared testing and inspection approaches and introduced as well usage-based testing concerning expert prioritized use cases and test cases, which were applied to code documents. But an additional work has to be done, because it is necessary to prioritize the use cases and test cases, which were set up in advance.

UBT is used to certify a particular reliability level and to validate the functional requirements [13] therefore UBT is to exercise the system under the same circumstances as the product is used in production [49].

UBT has two main objectives [73]

1. To find the faults which have the most influence on the reliability of the whole system from the users' point of view.
2. To produce data, which makes it possible to certify and predict the software reliability. Finally to know when testing can be stopped; the product is ready and can be accepted as it is.

Normally when UBT is applied two kinds of models are needed, a model to specify the usage and a reliability model [73].

The usage specification is a model that describes how the software has to used during operation. In the literature different types of models have been presented:

- Tree-structure models, which assign probabilities to sequences of events [57]
- Markov based models, which can specify more complex usage and model single events [103]

The main purpose of such a usage specification is to describe the best way getting a basis for the best practice to select test cases for UBT. This can also be used for two things, first for the analysis of the intended software usage and second to plan the software development itself. Knowing that some parts will have to be reused for sometimes they can be developed in earlier increments and therefore also be certified with higher confidence. The development and the certification of such increments is described in detail by Wohlin C. [109].

A kind of a reliability model will be needed to be able to analyze the defect data collected during the statistical testing. During the last 20 years several different model have been published and described, see Goel A. [38] for an overview, where models of different complexity and possibility to estimate the software reliability have been presented.

In this master thesis a different approach of UBT is used. UBT is typically located in the implementation phase or even later of the software development life cycle. Therefore Winkler et al. [108] improved the testing capabilities of UBT based on a modification by including inspection methods into the standard usage based testing approach – called "*Usage-based Testing with Inspection*" (UBT-i). This approach includes a twofold benefit:

1. UBT may also be applied to design specifications and code documents
2. The generation of test cases is an integral part of the testing process

What means, that the generation of test cases is an additional outcome in contradiction to the standard defect detection.

When executing this UBT-i approach the inspectors have to perform four major steps:

1. Choosing the first prioritized use case
2. Finding equivalence classes as well as test cases equivalent to the selected use case, afterwards applying guidelines for equivalence class derivation.

3. Apply the test cases relating to the prioritized use cases and record the candidates' defects.
4. Go back to step 1 until all use cases and document coverage are executed or the time limit is over.

Using this approach of UBT-i, this software fault detection technique can now be tested on documents of the design specifications. So it is possible to get an impression of its defect detection performance and can also be measured against a software inspection technique like UBR.

This thesis should add knowledge to the basic understanding of UBT-i about the performance in context of its temporal behavior. By knowing in which time intervals UBT-i is performing at its peak level tests can be better planned and organizations are therefore able to reduce efforts and costs for their software quality assurance work.

5 Research Approach

The main focus in this thesis is on the temporal behavior of defect detection effectiveness and efficiency between usage based reading – UBR – and usage based testing with inspection – UBT-i – in design documents.

The investigation of an experiment which was conducted also from Biffl S. et al. [11] will show how the results of these two software fault detection techniques will vary in the asked context of defect detection effectiveness, efficiency and false positives after, for example: 30 minutes, 60 minutes, 90 minutes and so on. Also the kind of defect types and the defect severity classes are important factors and will therefore be taken into analysis. By knowing the effectiveness, efficiency and false positives of each software fault detection technique in the context of certain time intervals a conclusion can be made if UBT-i with a little higher investment of time, because of the creation of the test cases, also leads to a better quality, what should result in a higher effectiveness than UBR.

Depending on the results that the investigation of the experiment will reveal different inferences can be made. Apart of the question which software fault detection technique is the more effective and efficient, the crucial question is to know the time intervals in which UBR and UBT-i are most effective and also efficient as well as in which the least false positives will be found. If the research will give this information the most defect detection effectiveness and efficiency in a temporal context can be identified. By having the knowledge which technique finds between which time intervals for example, the most crucial defects, companies are therefore able to give their inspections and or tests the perfect duration for their individual expected defect-finding outcome. Will the investigation not reveal a precise time intervals in which these software fault detection techniques are highly effective or efficient; than this could for example mean that these measures are tied to each individual inspector. If this happens a further deeper research about the individual skill and experience level of each inspector has to be made and hopefully by comparing participants with similar levels some commonness will be found. But such a deeper investigation of individual skill and experience levels will not be part of this thesis.

To summarize three main research questions are asked:

1. Is UBR more Effective and Efficient than UBT-i?
2. Are the Techniques basically effective and efficient in the first 120 minutes?
3. During which time intervals will the fewest False Positives be found?

5.1 Variables

The types of variables defined for this experiment are independent and dependent variables. They are explained in more detail in the following section.

Independent Variables

The qualification and the document location are the independent variables, so they do not depend on other variables.

The **Qualification** of the subjects was detected by performing an entry assignment. In relation to their results all subjects were divided in qualification classes. High, medium and low qualified inspectors were distinguished. The assignment included in context to reviews, inspection and usage-based reading a corresponding task.

The **document location**, through which the candidates had to go are of a different kind of documents related to the used system. The defects were seeded in the source code and design documents of the experiment. In this master thesis we concentrate on the design documents only.

Dependent Variables

These variables capture the performance of the different software fault detection techniques, which were applied in this experiment study. Following the standard practice in several empirical studies and the specific experiment, the focus is especially on time variables and performance measures. What concerns the time variables it will be analyzed the time spent on inspection and testing in minutes and the clock time when each defect is found (in minutes, starting from the beginning of the inspections and tests).

As far as performance measures are concerned it will be concentrated on the defect detection effectiveness and efficiency as well as false positives in a temporal context (30 minutes, 60 minutes, 120 minutes etc.) what means the share of defects found by each individual inspector and tester in a certain time interval in relation to the sum of the defects of severity classes A+B, which were seeded into the several software artifacts.

The **Effectiveness** is the number of matched defects (= number of seeded defects found by a participant) in relation to the overall number of seeded defects per individual defect severity class in a certain time interval. It is expected that a difference in effectiveness between the inspectors and testers applying one of the two software fault detection techniques UBR and UBT-i will be revealed. Effectiveness is further measured on the severity classes A+B and all seeded defects.

$$Effectiveness\ [\%] = \frac{matched\ defects}{total\ number\ of\ seeded\ defects}$$

The **Efficiency** is the number of matched defects (= number of seeded defects found by a participant) found per certain time interval, for example 60 minutes.

$$Efficiency\ [per\ timeframe] = \frac{matched\ defects}{timeframe}$$

False Positives are recorded defects, but these defects could not be associated to any reference defects, which were seeded by the experts. So False positives are all found but not matched defects.

$$False\ Positives\ [\%] = Found\ Defects - not\ seeded\ Defects$$

The **Effort** records the time needed by all participant to get through the used software fault detection technique and therefore to detect defects. The effort is calculated by adding the preparation time to the working time and subtracting the break time of the candidates.

$$Effort = Preparation\,Time + Working\,Time - Break\,Time$$

Defect Severity Classes are: class A (critical), these would have serious influence on the fundamental functionality of the product. Class B (major), which are defects of medium risk but also have an important influence on the functionality of the software system. Defects which have the class C only have a minor influence on the functionality and quality of the software product.

The **Mann-Whitney Test** is performed to examine if the results of two groups are significantly different.

$$U = n_1 * n_2 + \frac{n_1 * (n_1 + 1)}{2} - R$$

$$U = \frac{n_1 * n_2}{2} - u(\alpha) * \sqrt{\frac{n_1 * n_2 * (n_1 * n_2 + 1)}{12}}$$

The **Kruskal-Wallis Test** is quite the same like the Mann-Whitney Test, with the difference, that it can be used to test if more than only two groups are significantly different.

$$H = \frac{12}{n(n+1)} \sum_h \frac{S_h^2}{n_g} - 3(n+1)$$

5.2 Hypotheses

In the experimental study the performance in temporal behavior of two software fault detection techniques will be observed and investigated: Usage based Reading – UBR – and usage based testing with inspection – UBT-i. As the main goal of this thesis is to reveal which of these two techniques is during which time interval more effective and efficient. The number of false positives found will also be analyzed in a timely manner. The focus on similar research hypotheses regarding effectiveness, efficiency and false positives will be made. The calculation of effectiveness has been adopted to reflect the results in timely manner. In more detail the following research hypotheses will be evaluated:

5.2.1 Is UBR more Effective and Efficient than UBT-i?

This question involves two different measures, effectiveness and efficiency. It will give clearance about which of these two software fault detection techniques will perform better in the first 120 minutes.

H1: Effectiveness (UBR) > Effectiveness (UBT-i) for Design Documents in the first 120 minutes: This hypothesis is based on the prioritized use cases to use with UBR, which were made from experts and on the other hand with the test cases which each individual inspector had to made on their own. Therefore even though the UBT-i inspectors have to make test cases, which also takes some time, the quality i.e. the effectiveness of UBR should be higher for the first 120 minutes of inspection.

H2: Efficiency (UBR) > Efficiency (UBT-i) for Design Documents in the first 120 minutes: UBR inspectors don't have to make test cases prior to start detecting defects and they also have prioritized use cases, which are made by experts. So UBR inspectors have several advantages compared to UBT-i inspectors, which should in investigations be reflected in a higher efficiency of the first 120 minutes.

5.2.2 Are the Techniques basically effective and efficient in the first 120 minutes?

This assumption predicts that in the first 120 minutes of inspection and testing duration the most defects of severity classes A+ B will be found and afterwards only fewer of them.

H3: Are the techniques most effective and efficient in the time interval from 0 to 120 minutes for design documents: This hypothesis is based on the assumption that the inspectors and testers are mostly concentrated for the first 120 minutes. Also because of the prioritized use cases and test cases, which leads them to the defects and therefore not more than 120 minutes should be necessary to achieve an effective and efficient inspection as well as testing performance.

5.2.3 During which time intervals will the fewest False Positives be found?

As according the previous research question it will be assumed that in the first 120 minutes, from which is assumed to have a better effectiveness as well as efficiency, it is further expected that also in these time intervals a smaller number of false positives will be found by the participants.

H4: Will with UBR fewer false positives are found in the first 120 minutes than with UBT-i: This hypothesis predicts that in the first 120 minutes of duration fewer false positives will be found with the software fault detection technique UBR than with UBT-i. This could be again because of the use of the prioritized use cases from which the inspectors should get an advantage.

H5: Will the fewest false positives in UBR and UBT-i be produced in the first 120 minutes of inspection and testing: As approached in the hypothesis H3 it is further assumed, that the most defects of the severity classes A+B will be found in the first 120 minutes of the testing and inspection duration. The logical implication of this would be that also in this time interval the fewest false positive will be produced by the inspectors as well as testers and on the contrary afterwards most of them. This could be because inspectors or testers will find defects as a reason why they think they have to and therefore the more defects they find and the later it is in the inspection or test the more of them could be false positives.

The next section deals with the description and planning of the study experiment and how it was hold and evaluated.

6 Experiment

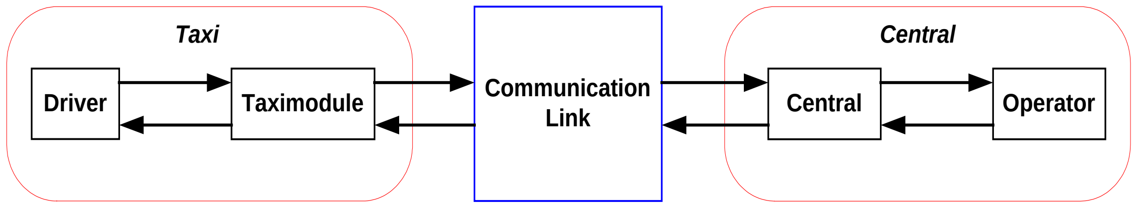
The Experiment itself is an extension of previous Experiments, which were concentrated on the usage based reading technique. They were made at Lund University in Sweden by Thelin et al. [3] [88] [90].

First some of the key aspects of the experiment are described, which form the basis of our empirical study including the overview and also which kind of expectations we have for the experiment. Next the threats to validity we had to define will be explained as well as the planning and preparation, then the operation of the experiment study and finally the evaluation phase.

6.1 Experiment Description

The experiment consists of a taxi management system which was originally provided by Thelin et al. [88] [90] who investigated different reading technique approaches. Before we go into detail a short overview of the system is necessary. The study describes a system which consists of two parts, as shown in Figure 6-1, on the left side the taxi part and on the right side the central part. These parts are connected to each other with the communication link.

Figure 6-1: Taxi Management System – Overview [105]



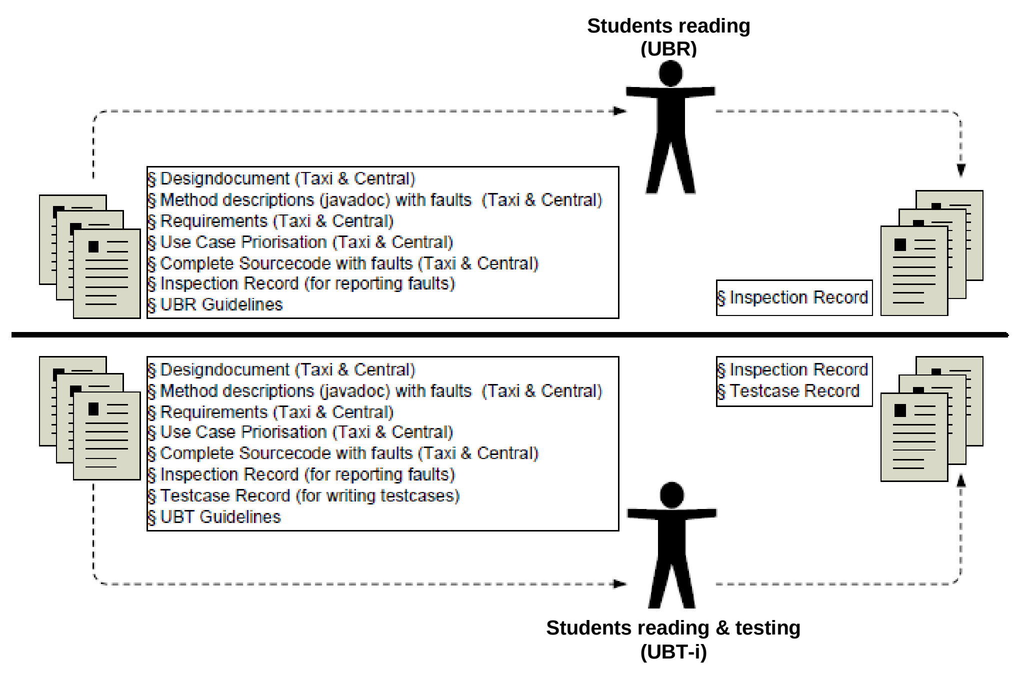
The Taxi module and the Driver represent the Taxi itself, which can be called and/or directly occupied. The Central part is handling the entire number of incoming request, for example: a taxi call. The central part knows also always all the states that each individual taxi has. It consists of Central and the Operator. The two parts of the taxi management system are linked together by the Communication Link.

Each technique, UBR and UBT-i will be introduced to the inspectors separately so they are able to apply the method in a correct way. The experiment is held in two sequential sessions. Each of the sessions has a duration of approximately 5 hours. The complete study design and workflow is visualized in Figure 6-3. In session one, which was the first possibility for the subjects in practicing with UBR and UBT-i, the taxi part is inspected as well as tested and in session two the central part. The main task of the subjects is to detect defects in the source code and design documents. This is of course equal to UBR and UBT-i. The difference for UBT-i is that test cases must be written which intend to be helpful in finding defects. The detailed workflow for UBR and UBT-i, which was also handed out to the participants can be seen in the Appendix. Afterwards a feedback questionnaire was done to bring the inspectors in the possibility to reflect how well the method had been applied and how the inspectors dealt with the tasks. Finally a data registration has to be done, where all paper-based results have to be entered into a Web-tool so the evaluation of the results can be done.

The subjects in the study were 41 graduate software engineering students. At first they made a PairProgrammig qualification test. This was made that we can be sure that for the inspection participants they have sufficient implementation skills to make the tests and inspections. All of the chosen participants were assigned randomly to the techniques to be able to control the influence of inspector capability and to achieve a better external validity. The experiment was integrated in a practical part of a software engineering and quality assurance workshop.

Figure 6-2 shows the configuration concerning the subjects. Each group of participants got the necessary documents, the complete design documents and the source code in a document including all the seeded defects.

Figure 6-2: Configuration of the Experiment


Students reading
(UBR)
§ Designdocument (Taxi & Central)
§ Method descriptions (javadoc) with faults (Taxi & Central)
§ Requirements (Taxi & Central)
§ Use Case Priorisation (Taxi & Central)
§ Complete Sourcecode with faults (Taxi & Central)
§ Inspection Record (for reporting faults)
§ UBR Guidelines
§ Inspection Record
§ Designdocument (Taxi & Central)
§ Method descriptions (javadoc) with faults (Taxi & Central)
§ Requirements (Taxi & Central)
§ Use Case Priorisation (Taxi & Central)
§ Complete Sourcecode with faults (Taxi & Central)
§ Inspection Record (for reporting faults)
§ Testcase Record (for writing testcases)
§ UBT Guidelines
§ Inspection Record
§ Testcase Record
Students reading & testing
(UBT-i)

6.2 Planning and preparation

The used taxi management system was adopted from previous studies [88] [90] and had to be reviewed and controlled. A big part of the artifacts were given but some had to be prepared. This section gives an overview about all used artifacts and a description of seeded defects.

6.2.1 Software Artifacts

Artifacts have to be distinguished, because of their kind of purpose, on the one hand documents for preparation and on the other hand documents which are needed for the use for one of the software fault detection techniques.

The documents for the preparation phase were a tutorial and the guidelines:

- The *guidelines* were partly taken over but had to be reworked. The aim of this document is to provide the subject with a step by step guidance to be able to apply the used software fault detection technique.

- The *tutorial* was a presentation of the used software fault detection technique and how to handle with all other needed documents. With an example it was shown practically how to use the inspection record including one exemplary filled line. Afterward open questions were discussed in the group.

The experiment setup consisting of:

1. a textual description of the requirements defines the terminology and all functional requirements for both modules central and taxi
2. the design documents of the taxi management system presents more precisely the entertained modules as well as the internal activities, which are for example: interface descriptions, data structures and so on
3. the guideline for the techniques applied as well as questionnaires for determining inspector capability and feedback.

The following documents have been used to apply the software fault detection technique on the taxi management system:

- The *textual requirements document* consists of 8 pages including 2 UML2 component diagrams. These documents are describing the basic functionality of the system in a very user-friendly way.
- The *design documents* consists also of 8 pages, which have about 2400 words, 2 component diagrams and 2 UML diagrams. An overview of the software modules have been described as well as their context including the internal representation, which means the relationships between two or more modules and an external representation. This in turn means the relationships between the user and the system. Also a sum of 24 prioritized use case descriptions from the users' point of view and altogether a number of 23 sequence diagrams has been provided. This artifact describes the technical dimension of the taxi management system.
- *Guidelines* for the correct use of the assigned techniques are also handed out to the participants.

Only one form was used in this study which was the inspection record.

- The inspection record is a form in which all detected defects by the subject had to be written down. For each found defect the severity class, the defect type and the document location had to be filled in.

Complementary questionnaires were handed out to all subjects.

- *Feedback questionnaires* handed out after each session, which gave the candidates the possibility to communicate their impressions and estimation about their own detected defects.
- The experience questionnaire was provided online and filled in after the registration for this task. By this questionnaire we wanted to measures the candidates' implementation skills.

6.2.2 Reference Defects

This section gives an overview of all reference defects seeded into the design documents and how they were split in context to experiment sessions, defect severity classes and also document locations.

Experiment Sessions

The reference defects were not randomly seeded into both experiment sessions, central and taxi, but as good as possible equal between them. As the Table 6-1 visualizes in number and in percent in the experiment session central there are 2 more defects than in the taxi session.

Table 6-1: Reference Defects in both experiment sessions

	Number of Defects [num]	**Number of Defects [%]**
Central part	31	51,67
Taxi Part	29	48,33
Summary	**60**	**100**

Defect Severity classes and Document Location

Overall 60 faults have been seeded into the document packages as the Table 6-2 shows below. The figure presents the nominal number of seeded defects according to defect severity classes and document location. These faults have been seeded by highly experienced experts into the design specification and source code documents [105]. In this thesis we focus only on the defect classes crucial and major, which should naturally gain a higher weight.

Table 6-2: Allocation of Seeded Defects [105]

	Design Documents	**Source Code**	**Sum**
Crucial (class A)	10 (17%)	19 (32%)	29 (49%)
Major (class B)	12 (20%)	12 (20%)	24 (40%)
Less important (class C)	5 (8%)	2 (3%)	7 (11%)
Summary	**27 (45%)**	**33 (55%)**	**60 (100%)**

Found defects of the class A (critical), in design documents 10 and in the source code 19, would have serious influence on the fundamental functionality of the product. Defects of the class B (major), in design documents 12 and in the source code also 12, are only rarely occurring but also important defects or less important frequent defects of medium risk. Defects which have the class C are rarely occurring and only have a minor influence on the functionality and quality of the software product. All recorded defects had to be classified by the subjects in the inspection record, which was a subjective classification by the candidates, itself. As Table 6-2 further visualizes, exactly 55 % of all reference defects were strewed in the source code documents and 45 % were strewed in the design specification documents. In this thesis only the defects of classes A+B in the design documents are of importance.

6.3 Operation

The complete study design and workflow is visualized in Figure 6-3. The knowledge and the basic understanding of the subjects was given and proofed with a qualification test so we can act with the assumption that everyone has some related knowledge.

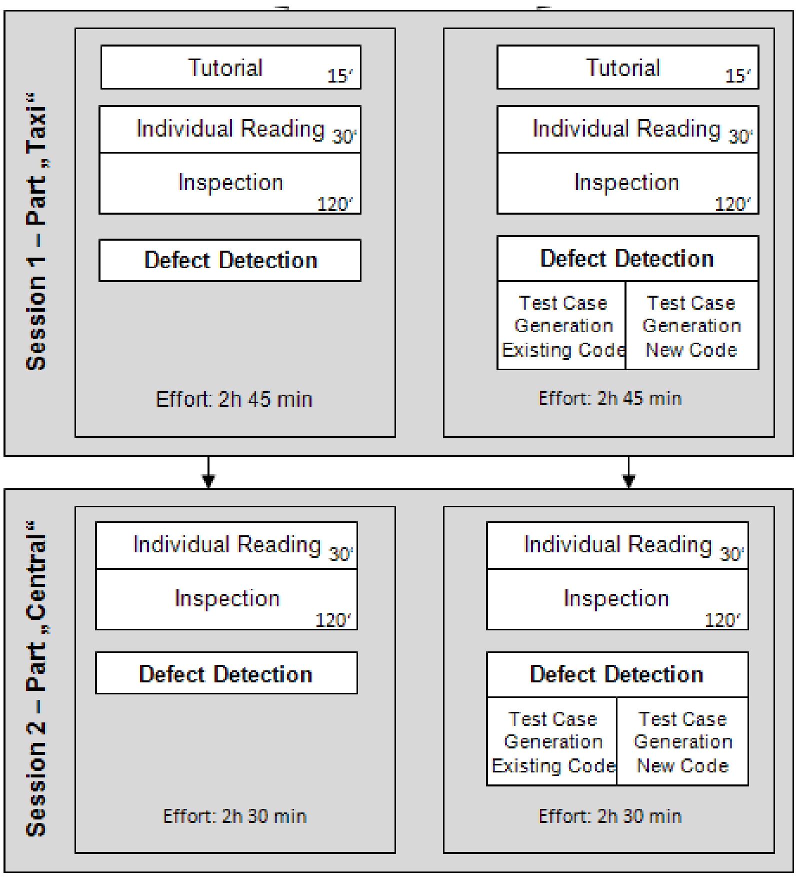
Before the first of the two sessions was held, a tutorial was carried out which gave an introduction to the concept of inspection and testing. All used artifacts presented and explained as well as the inspection record.

The first session, which dealt with was the taxi part, it was also the first possibility for practicing with the software fault technique for the participant as well as for ensuring that all candidates are proceeding in a correct way. A guideline was also handed out to all participants including a step by step instruction. The first session consists of three parts which were the same duration for each used technique:

1. The tutorial lasts 15 minutes and the participants got another short introduction in practicing with their technique and how they should operate with the record sheet and so on.
2. Individual reading took 30 minutes for each candidate where they had to read through all the provided documents.
3. Inspection or test took 120 minutes of the given documents.

In the second session, which was the central part, the same software fault detection technique was used under same conditions without any task modification. Even the same time intervals were maintained. The only difference was that the tutorial at the beginning was passed. Also the same guideline as mentioned before was handed out again to the participants. So it could be avoided that even when the candidate forgot how to perform with the used software fault detection technique he had a detailed guideline to follow.

Figure 6-3: Experiment operation

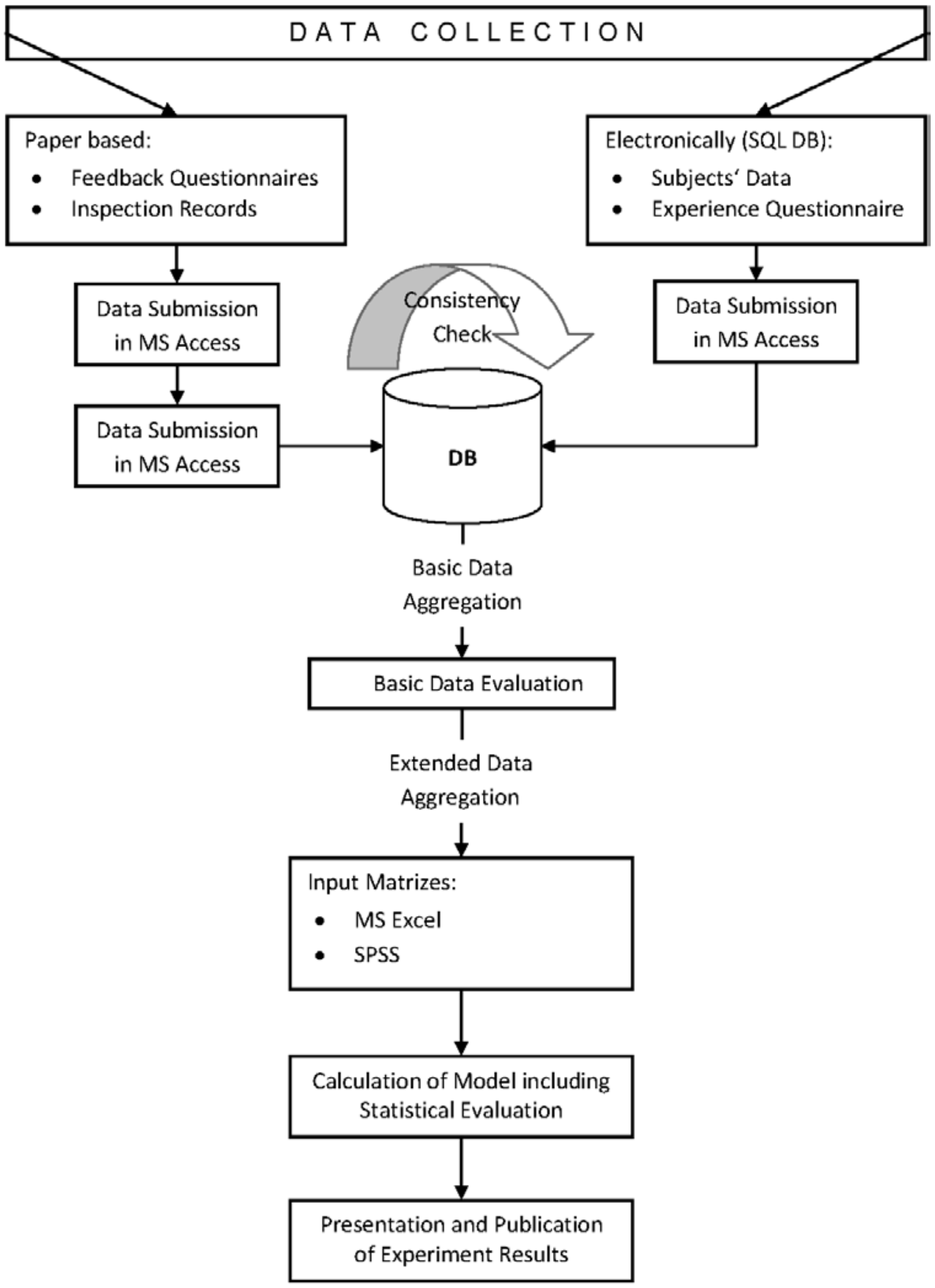


The next chapter gives an overview about the evaluation phase of the experiment study.

6.4 Evaluation

The process of the data evaluation can in detail be seen in Figure 6-4. The overall process was not very complex but temporal quite extensive. The personal data of the candidates and their experience questionnaire were entered into a data gathering tool that was set up especially for this experiment. All paper based documents, which the subjects had to fill in, were collected after each session. These papers were for example the inspection record and the feedback questionnaire. The next step was that the collected feedback questionnaires and the inspection records had to be entered into the experiments Access database. During the entering process some data validation was already made, e.g. some subjects were removed because the sessions were not complete. Afterwards all data from the SQL database had to be converted. Finally when all data was completely available in the Access database various control queries were made to ensure the consistency and plausibility as well as to examine all needed fix values. For example subjects had to be removed who did not finish the task or performed only one session.

Figure 6-4: Data evaluation process


DATA COLLECTION
Paper based:
• Feedback Questionnaires
• Inspection Records
Electronically (SQL DB):
• Subjects' Data
• Experience Questionnaire
Data Submission in MS Access
Consistency Check
Data Submission in MS Access
Data Submission in MS Access
DB
Basic Data Aggregation
Basic Data Evaluation
Extended Data Aggregation
Input Matrizes:
• MS Excel
• SPSS
Calculation of Model including Statistical Evaluation
Presentation and Publication of Experiment Results

The evaluation of all data records was made with Excel, Access queries and SPSS. The Excel calculations were performed partly based on Access queries and visualized or analyzed in SPSS. Depending on the individual purpose these tools were also mixed.

6.5 Threats to validity

A key issue when performing experiments is the validity of the results. Therefore this section contains possible threats to internal and external validity of the experiment setup and possible countermeasures. We tried to reduce all threats as much as possible as in the following described.

Drew [25] defines internal validity as the technical soundness of a study. A study is internally valid when all the potential factors that might eventually influence the data are controlled except the one under study. This would mean that the main concept of control had been successfully implemented. If, for example, two instructional methods were being compared, internal validity would require that all differences between the groups (e.g. intelligence, age) has to be removed except the differences in the instructional method, which is the experimental variable.

To address the **internal validity** some countermeasures have been implemented [104].

- *Communication between Individuals:* The communications between individuals during the study execution phase have been avoided, because this could have an impact on experiment results. To achieve this, the experiment supervisors paid special attention to the work of the work-units (Inspection, Testing). No communication outside the natural work-units was allowed.
- *Individual breaks:* In order to increase inspector performance individual breaks were allowed during the experiment sessions. The participants have to record breaks to identify the real working effort.
- *Duration:* An upper time limit regarding the overall inspection duration has been set. The inspectors were able to finish earlier but not later than the given maximum time limit.
- *Skills:* All candidates had to pass a PairProgramming qualification test to ensure their sufficient programming skills. 41 subjects of about 60 candidates passed this test.

- *Experiment Proceeding:* A feedback questionnaire was made at the end of the experiment to be able to get some knowledge of the individual course of action and to see if the participants followed the study process and guidelines properly.
- *Document Package:* An initial study to initially verify the experiment package has been made. Also intensive reviews by experts of the study package were made to verify the correctness of the document package, including modifications based on the initial study

Drew [25] defines external validity as the generalizability of results from a given study. The External validity describes how the results of the experimental study will eventually apply to the world outside the academically controlled research situation. If a study is externally valid or has considerable external validity, one can expect that the results are generilizable to a considerable degree.

The following points were made to improve the external validity:

- *Application domain:* A well known application domain, the taxi management system has been used to avoid general domain specific interpretation problems.
- *Document Package:* To be able to compare the results with real world settings, the specification of the experiment has been a real world application. The given design specification may be a limitation for IPP application in an industrial setting, where only fragments of a design specification are given.
- *Selection of participants*: Students have been used as participants, so this might not really be representative for industrial environment. Everyone of the students got an intensive training, which was comparable to a real world setting within their course. Furthermore most of the participating students work at least part-time in industrial context. This information was recorded in the experience questionnaire.
- *Arrangement*: A classroom setting has been used to be able to make the experiment in controlled environment.

Several representative defects were seeded in the design specification and source code documents according to different types of defects and defect locations. The seeded defects were representative of defects found during the development of the documents under study.

7 Results of the Experiment

This section of the thesis summarizes the performance results of the empirical study concerning the effort, effectiveness, efficiency and false positives in a temporal context of the two software fault detection techniques usage based reading – UBR – and usage based testing with inspection – UBT-i.

7.1 Effort

With the reported overall effort of the experiment it is possible to illustrate it in the evaluated scope. In the study context, effort is defined as the overall session duration including individual preparation and execution time in minutes. The Individual preparation time contains the time used for reading the documents as well as getting familiar with the software fault detection technique applied of the participating inspectors. For UBR with expert ranking of use cases only little preparation time is needed [107]. The effort of UBT-i should be measurably higher due to the fact that they have to produce test-cases as an output.

The experiment preparation time has not been taken into account, because this has been done by experts as preliminary work packages before the experiment started. In this evaluation both time intervals, what means session one (Taxi part) and session two (Central part) has been summarized for the effort calculation and illustration, because there is no additional effort within the inspection or testing execution. Table 7-1 displays the mean values as well as the standard deviation of the defect detection effort for UBR and the defect detection effort + test case generation for UBT-i in minutes. Also the p-values are shown to investigate significance of difference between the two techniques.

Table 7-1: Defect Detection Effort (UBR) and Defect Detection Effort + Test Case generation (UBT-i) [min]

	UBR	UBT-i
Mean Value	272.5	268.8
Standard Deviation	38.0	29.1

Mann-Whitney-Test	0,497 (-)

It can be seen that both techniques have an average similar effort. A great difference concerning the effort of the two techniques cannot be recognized, but there is a little bit higher mean value for UBR as well as also a higher standard deviation. The Mann-Whitney test shows, that there is no significant difference concerning the effort between UBR and UBT-i.

7.2 Effectiveness

The effectiveness is the number of real defects found in relation to the overall number of seeded defects per individual defect severity class in a certain time interval. Effectiveness is measured on the severity classes important, which are A and B. The experiment setup, as described in more detail in section 6, consists of an overall number of 60 seeded defects. 27 defects are seeded into the design documents, which are authoritative for this investigation and 33 defects in the provided source code. As mentioned earlier, only defect severity classes A and B are taken into account, therefore attention is paid to 10 critical defects (Class A) and 12 important defects (Class B) in design documents. So we are able to view the results in the right context, because for the calculation only these defects concerning the design documents are taken into account. Defects of classes C will not influence the results, because of their unimportance they are not taken into account. The beginning of the analysis is also the real beginning of the inspection or test, which means the "*gross-processing time*" will be used here in contrary to the investigation of efficiency, where the "*net-processing time*" will be used.

The calculation for effectiveness has also been adopted a little bit to be able to evaluate every timeframe independent from each other. Normally the matched defects are divided through the total amount of seeded defects. Using this formula would not be able to give us information which timeframe would be the most effective one. Therefore the found defects in the preceding time interval will be subtracted from the overall number of seeded defects for the next time interval. Doing this, will bring each time interval in the condition to be evaluated with the found defects of its own time interval and the number of the overall seeded defects that can still be detected. With this adaption of the common formula of effectiveness, it is possible to evaluate each time interval with the right number of seeded defects that are responsible.

The conclusions of these results should answer, which of the two used software fault detection techniques is the most effective one UBR or UBT-i in which time intervals. Afterwards UBR and UBT-i will be investigated separately and we will take a closer look at each time interval, each consisting of 30 minutes. The second part of this section will show the investigation of each session of UBR and UBT-i completely separated from each other, which should give clearance about which time interval of which session will be the most effective one.

7.2.1 Combined Sessions – Combined Techniques

The first investigation will clarify which of these two software fault detection techniques performs most effective. Therefore both sessions of UBR and UBT-i of the study experiment are combined and only defect severity classes A and B of both sessions are taken into account.

In the Box plot in Figure 7-1, in which the data of session 1 (taxi part) and session 2 (central part) are aggregated, can clearly be seen that UBR has a somewhat higher median as well a higher maximum than UBT-i. The comparison of the Mean Value in Table 7-2 shows the outcomes. UBR has a somewhat higher Mean Value 18,89 % than UBT-i with 16,91 %. Although the difference is not really great UBR is a little bit more effective than UBT-i. The Mann-Whitney test shows, that there is no significant difference between UBR and UBT-i.

Figure 7-1: Effectiveness, UBR vs. UBT-i [%]

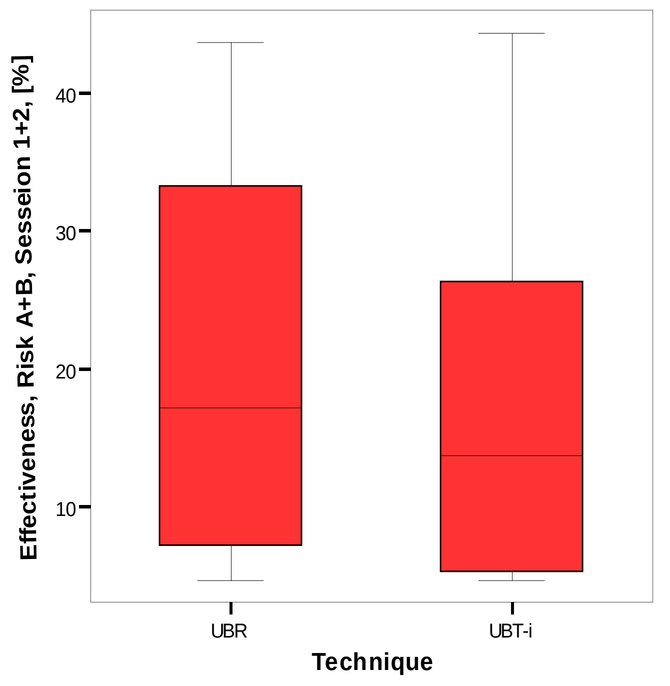


Table 7-2: Effectiveness, UBR vs. UBT-i [%]

	Mean Value	Standard Deviation
UBR	18.9	11.3
UBT-i	16.9	12.6
Mann-Whitney Test	0.317 (-)	

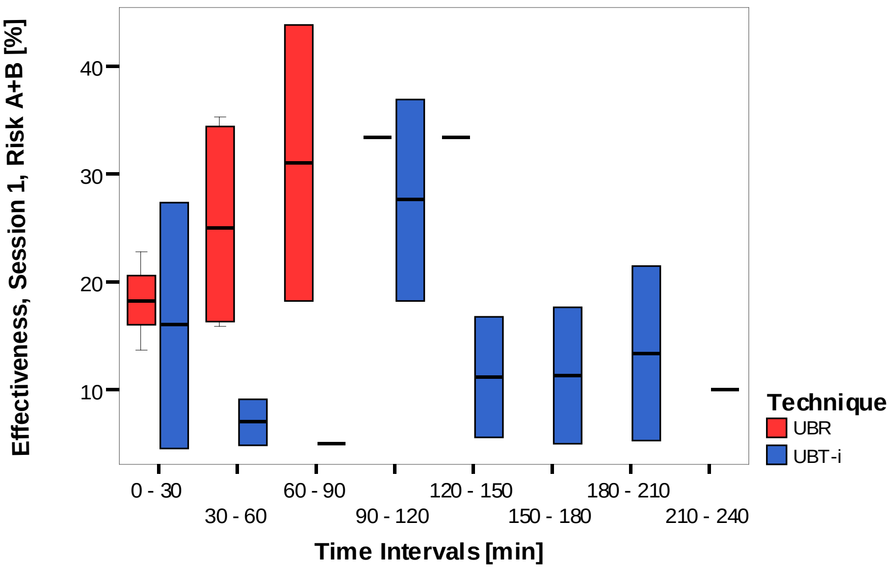
The next section gives an overview about the effectiveness of the software fault detection techniques UBR and UBT-i in a timely matter.

7.2.2 Temporal behavior of combined sessions and techniques

This section gives a detailed overview of both sessions of the study experiment, the taxi and the central part for the techniques UBR and UBT-i. To be able to analyze each session with each technique in a temporal behavior in a very detailed way, each session has again be divided into eight time intervals. Each of these intervals has a duration of 30 minutes. With this kind of investigation it should be possible to determine which time intervals are the most effective one between UBR and UBT-i in session one and two of the study experiment. It will be analyzed which technique is during which session the most effective one. Therefore the mean values of the separated sessions of UBR and UBT-i are opposed to each other.

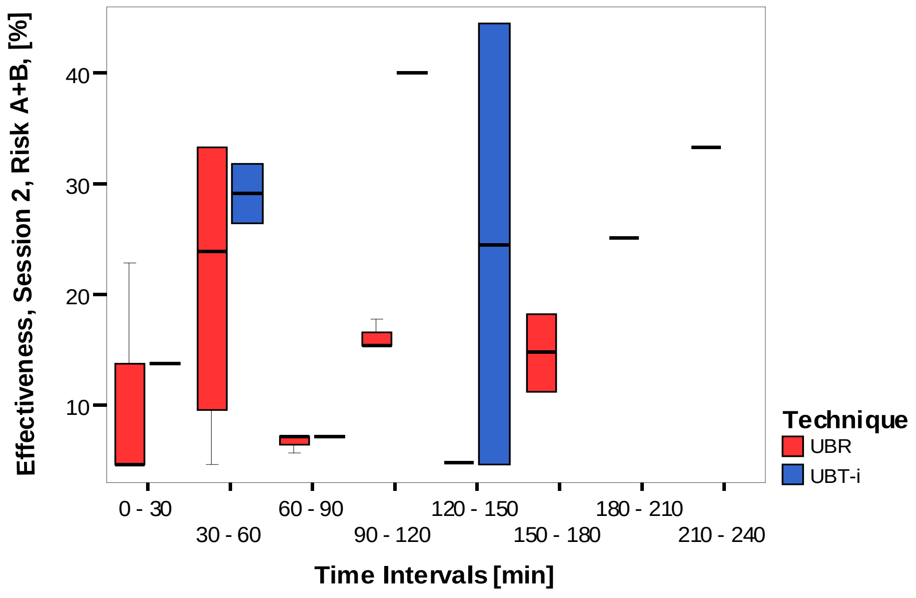
The box plot in Figure 7-2 shows the results of the first session. In this view it can be seen that after the fifth time interval or after 150 minutes of duration the effectiveness of UBR and UBT-i decreases, but it can clearly be seen that UBR in session one is more effective in every timeframe than UBT-i. Whereas time intervals two and three are very effective for UBR, UBT-i has a complete decrease. In the next interval effectiveness rises again for UBT-i to a quite good value, but decreases again as mentioned before in the next time intervals.

Figure 7-2: Effectiveness, Session 1, UBR and UBT-i [%]



Session two is a little bit different in comparison to the first session. Effectiveness varies extremely for the time intervals as well as for the investigated technique, which can be seen in Figure 7-3. But it can be seen that in session two the most effective technique seems to be UBT-i, which stays quite effective until the end of the testing duration. Time interval number 1 and 3 are apparently the one with the least effectiveness for UBT-i as well as for UBR.

Figure 7-3: Effectiveness, Session 2, UBR and UBT-i [%]



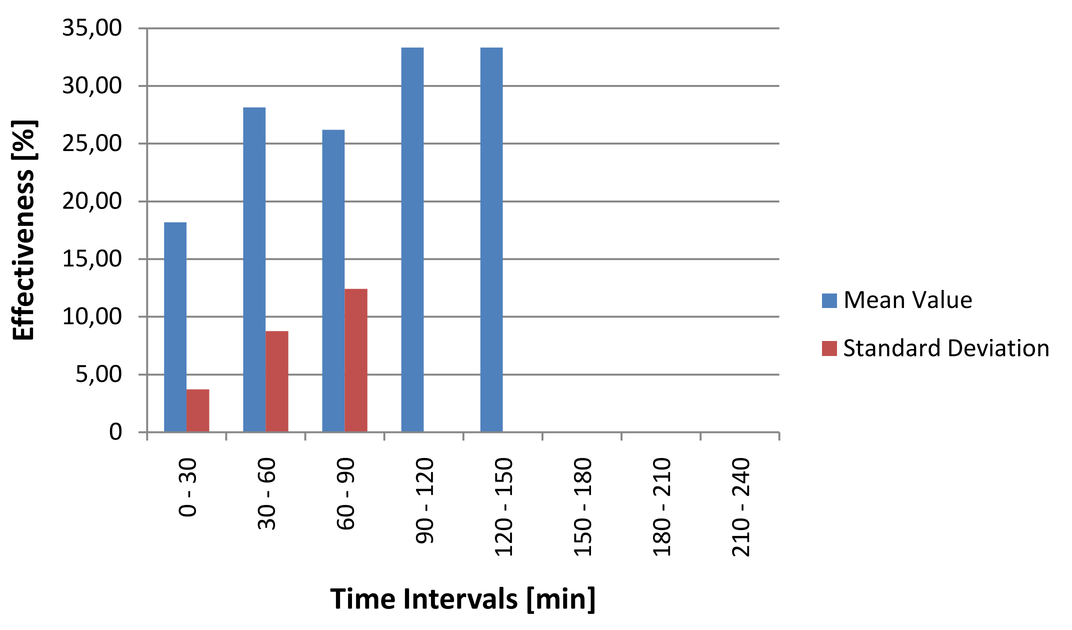
The next investigations concentrate on finding the most effective time intervals for each separated session of the software fault detection techniques UBR and UBT-i.

7.2.3 Temporal behavior of separated sessions and techniques

In this chapter each session will be analyzed separately for UBR and UBT-i to be able to determine which technique is during which time intervals of the considered session the most effective one.

The results of the first separated investigation can be seen in the bar chart in Figure 7-4 as well as in Table 7-3. Remarkable at the first view is of course the growth of the mean value, because it rises higher even in the last two intervals of the inspection duration. The first interval is the most ineffective one with 18.18 % in the first session of UBR, what means that in contrary to the other frames the least defects according to the overall number of defects, which could possibly be found, were detected. The second and third time intervals are quite effective, but intervals four and five are outstanding, because they both have the highest level of mean value with 33.33 % and also no standard deviation, which is remarkable. Why the inspectors were not able to find any defects after this timeframe is not obvious, although there were some seeded defects that they hadn't been detected by any of the participants. So the most effective timeframes for UBR in session one of the experiment are time intervals four and five, which is one hour from 90 to 150 minutes of the inspection duration. Although it is certainly not possible to only hold this time intervals of inspection, so the first five intervals must be declared as the most effective one.

Figure 7-4: Effectiveness, UBR, Session 1, Risk A+B [%]



In Table 7-3 can also be seen that the Kruskal-Wallis Test shows, that there is no significant difference concerning the time intervals in which defects were found. The time intervals where no defects were found were not included.

Table 7-3: Effectiveness, Session 1, UBR [%]

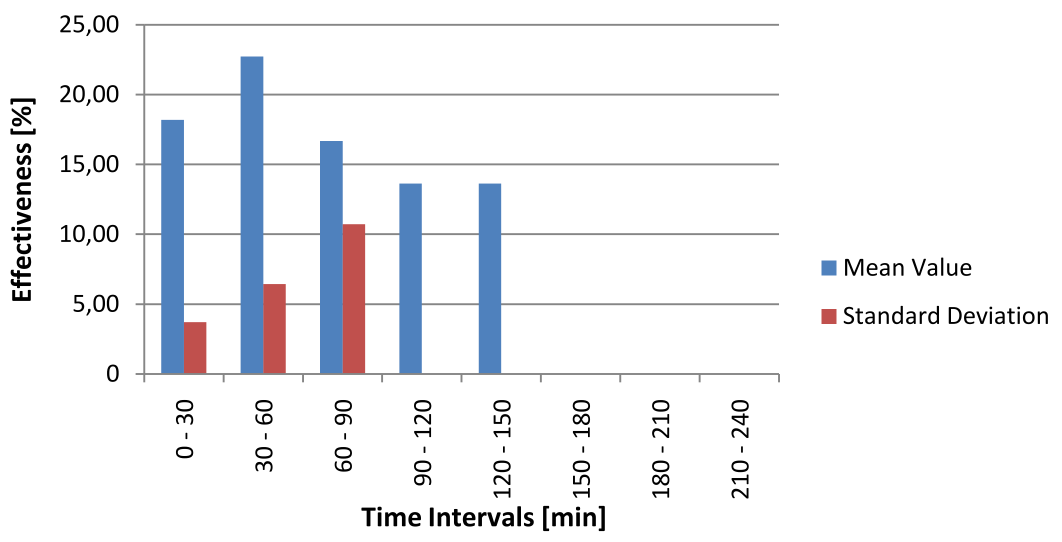
Time Interval [min]	Mean Value	Standard Deviation
0 – 30	18.18	3.71
30 – 60	28.14	8.77
60 – 90	26.20	12.43
90 – 120	33.33	0
120 – 150	33.33	0
150 – 180	0	0
180 – 210	0	0
210 – 240	0	0

Kruskal-Wallis-Test	0.720 (-)

But although Figure 7-4 gives a detailed overview of the effectiveness, it should not be forgotten that, because of the altered calculation method in the second time interval 15 defects were found and in both of the time intervals four and five only 3.

To ensure that with the adopted calculation method of the effectiveness, for these investigations, an accurate outcome has been produced, the standard calculation was also made. Therefore Figure 7-5 shows the effectiveness with the standard calculation method, in which it can be seen that the trend line is absolutely a different one, what concerns only the time intervals number 4 and 5. Here the last two time intervals are the least effective one, because only a minor number of defects were found in contrast to the overall number seeded defects. Although of this different result it can also be stated out, that the first five time intervals are very effective, because the last two intervals are not so ineffective at all.

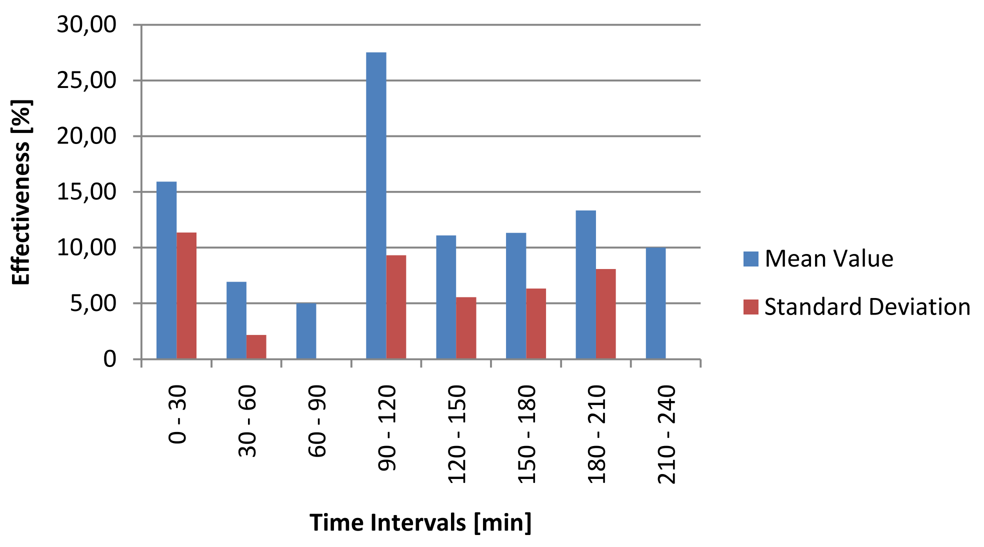
Figure 7-5: Effectiveness (standard calculation), UBR, Session 1, Risk A+B [%]



The results of the first session of the investigated technique UBT-i can be seen in Figure 7-6 and Table 7-4. UBT-i starts with a quite good amount of mean value of 15.91 %, which is the second best value of this session, but also has the highest standard deviation of session one with 11.36 %. The next two time intervals are absolutely ineffective with mean values of only 6.93 % and 5 %. Interval number 4 is by far the most effective time intervals with a mean value 27.51 % and also a very low standard deviation. The rest of the testing duration keeps at a quite ineffective level but also higher than time intervals two and three.

UBT-i is therefore most effective, because of time interval four, in the first two hours in inspection. But the time intervals afterwards should not be sent to coventry because they are not ineffective at all, although they are not able to get a higher mean value than 13.35 %.

Figure 7-6: Effectiveness, Session 1, UBT-i [%]



In Figure 7-3 can also be seen that the Kruskal-Wallis Test shows, that there is no significant difference concerning the time intervals in which defects were found.

Table 7-4: Effectiveness, Session 1, UBT-i [%]

Time Interval [min]	Mean Value	Standard Deviation
0 – 30	15.91	11.36
30 – 60	6.93	2.16
60 – 90	5.00	0
90 – 120	27.51	9.33
120 – 150	11.11	5.56
150 – 180	11.32	6.32
180 – 210	13.35	8.08
210 – 240	10.00	0

Kruskal-Wallis-Test	0.064 (-)

To again ensure that the outcomes are accurate the standard calculation was made another time for monitoring reasons. The standard calculation, see Figure 7-7 below, of the effectiveness for session 1 of UBT-i shows, in contrary to UBR, exactly the same trend line as the adopted calculation method.

Figure 7-7: Effectiveness (standard calculation), UBT-i, Session 1, Risk A+B [%]

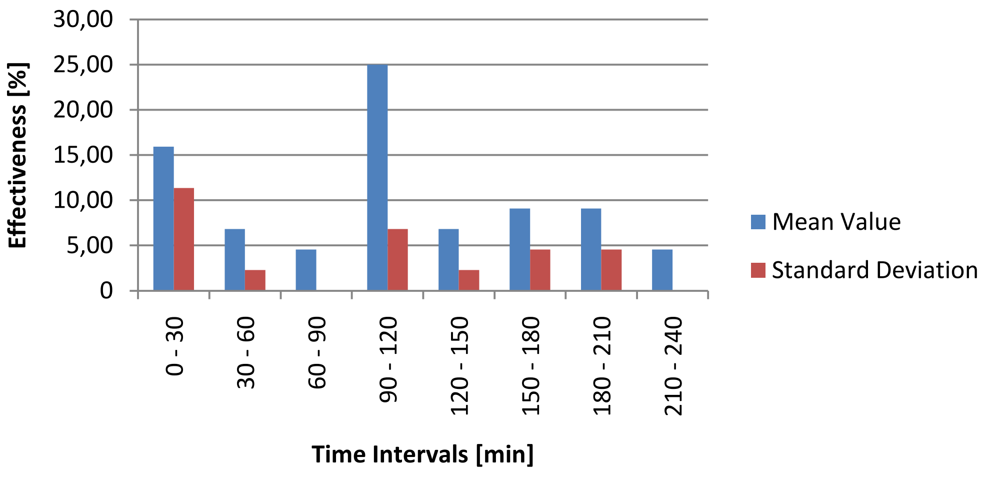
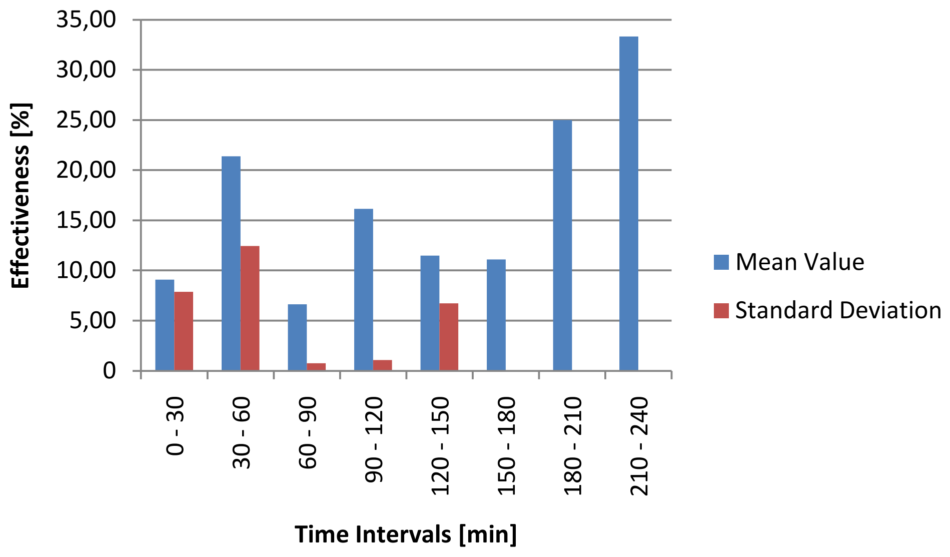


Figure 7-8 and Table 7-5 show the results of UBR of session two of the experimental study. The results of UBR are also very interesting, because it can again clearly be seen in the bar chart of Figure 7-8 that the last two time intervals are the most effective one, what is again caused by the method of calculation for the effectiveness and therefore the standard calculation method will also be taken into account. The first interval is not very effective with a mean value of 9.09 % and also a very high value of standard deviation of 7.87 %, which is hardly the same as the mean value. The second time interval of the inspection is very effective with a mean value of 21.37 %. Effectiveness falls down in the third interval to a very low level of mean value, which is 6.61 %. In the fourth time interval effectiveness rises again to a very good mean value of 16.14 %. The next two intervals of inspection are not very effective and are therefore not really mentionable. The last two time intervals have again a very high amount of mean value. The most effective intervals are for session two of UBR the first four timeframes of testing duration.

Figure 7-8: Effectiveness, Session 2, UBR [%]



In Table 7-5 can also be seen that the Kruskal-Wallis Test shows, that there is no significant difference concerning the time intervals.

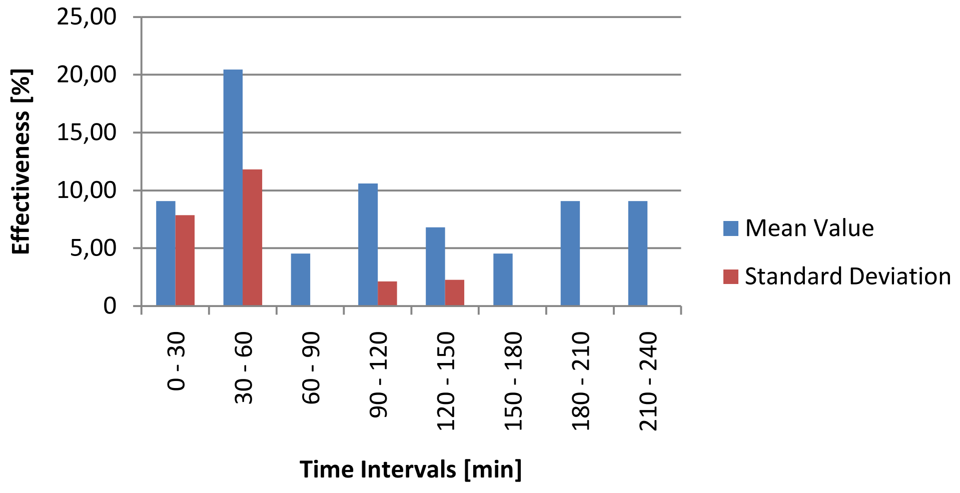
Table 7-5: Effectiveness, Session 2, UBR [%]

Time Interval [min]	Mean Value	Standard Deviation
0 – 30	9.09	7.87
30 – 60	21.37	12.44
60 – 90	6.61	0.75
90 – 120	16.14	1.07
120 – 150	11.47	6.71
150 – 180	11.11	0
180 – 210	25.00	0
210 – 240	33.33	0

Kruskal-Wallis-Test	0.296 (-)

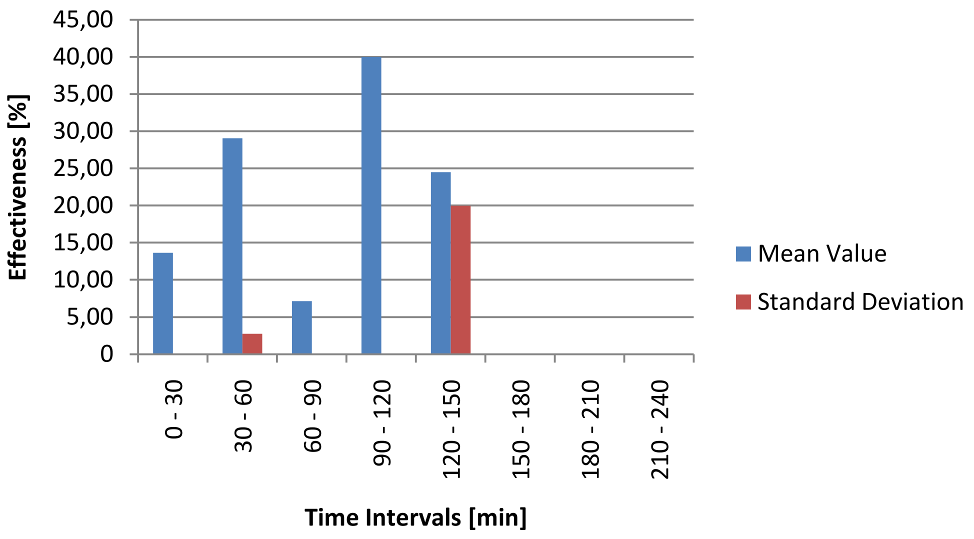
Figure 7-7 shows the bar chart for the standard calculation of effectiveness for UBR of session two and has the same change in the trend line as session one. The last two time intervals are not highly effective, although they are not ineffective at all. The first four time intervals are still the most effective one, but this investigation changes the results for intervals number 7 and 8 dramatically.

Figure 7-9: Effectiveness (standard calculation), UBR, Session 2, Risk A+B [%]



The results of the second session of UBT-i can be seen in the bar chart of Figure 7-10 as well as Table 7-6, which shows the exact outcomes. The first time interval is quite ineffective and reaches therefore only a mean value of 13.64 %. Effectiveness rises in time interval two to a very high mean value of 29.07 %, but standard deviation stays at a remarkable low level of 2.75 %. Interval three is absolute an outlier and has only a mean value of 7.14 %. Interval four has a very good amount of mean value, which is 40 % and this time interval is therefore very effective. The fifth interval of testing is not very effective and reaches only a mean value of 24.49 %. Therefore are the most effective time intervals one to four, although interval five is also not completely ineffective.

Figure 7-10: Effectiveness, Session 2, UBT-i [%]



In Table 7-1 below can be seen that the Kruskal-Wallis test stated out, that there is no significant difference between the time intervals.

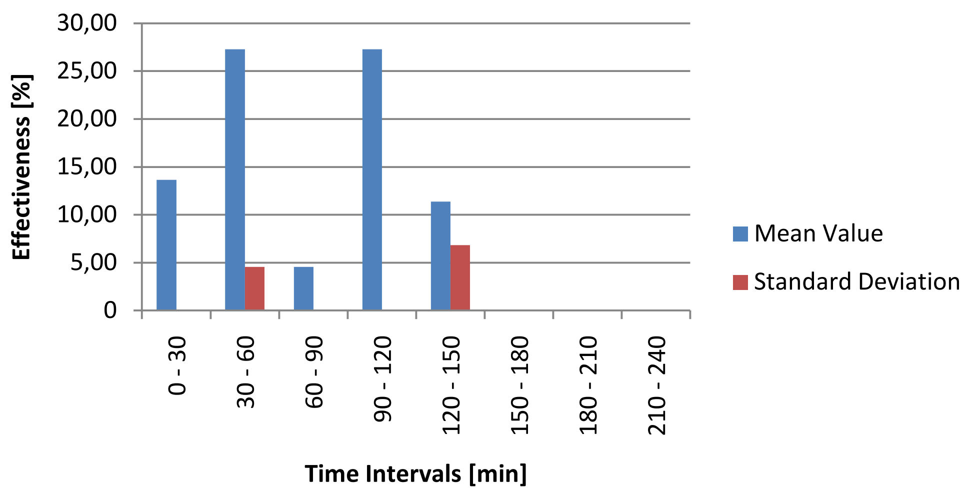
Table 7-6: Effectiveness, Session 2, UBT-i [%]

Time Interval [min]	Mean Value	Standard Deviation
0 – 30	13.64	0
30 – 60	29.07	2.75
60 – 90	7.14	0
90 – 120	40.00	0
120 – 150	24.49	19.95
150 – 180	0	0
180 – 210	0	0
210 – 240	0	0

Kruskal-Wallis-Test	0.729 (-)

The standard calculation below in Figure 7-11, shows the same trend line as the adopted calculation method. Exactly as in the session one of UBT-i the trend line of the adopted calculation does not vary from the standard calculation.

Figure 7-11: Effectiveness (standard calculation), UBT-i, Session 2, Risk A+B [%]



In the next section of this master thesis the efficiency of the software fault detection techniques will be analyzed in a temporal behavior. This section will give clearance about in which time intervals the most seeded defects will be found by the inspection and testing candidates.

7.3 Efficiency

The efficiency is the number of real defects found per certain time interval. Several different intervals will be investigated, the overall time for the inspection and test as well as the time divided into 4 time intervals which consist of each one hour. All investigations concerning the efficiency will be made only with the defect severity classes A+B. Defects of classes C will not be taken into account, because of their unimportance. This section is also sub classified into the investigation of the efficiency of UBR vs. UBT-i with sessions 1 and 2 combined. The second part of this section will show the investigation of each session of UBR and UBT-i separately, which should give clearance about which interval of which session will be the most efficient one.

An additional aspect is also the investigation of the average of the time when the candidates recorded their first found matched defect. In Figure 7-12 the box plot can be seen when the average of the candidates of each session for UBR and UBT-i found their first matched defect. In the Table 7-7 the exact data of mean value and standard deviation give a more detailed view. It can be seen that the subjects using UBR are able to find their first matched defect earlier in every experiment session than the other participants using UBT-i. Mentionable is also that the participants using UBR were not able to find their first defect earlier in the second session of the experiment. The exact opposite occurred; they found their first defect later. UBT-i on the contrary showed an outcome as expected, the subjects were able to reduce the time when the first defect was found. Whereas the standard deviation does not reveal any mentionable difference between the two software fault detection techniques. The Mann-Whitney test, which was made for each session separately does not show any significant difference between UBR and UBT-i.

Figure 7-12: First defect found

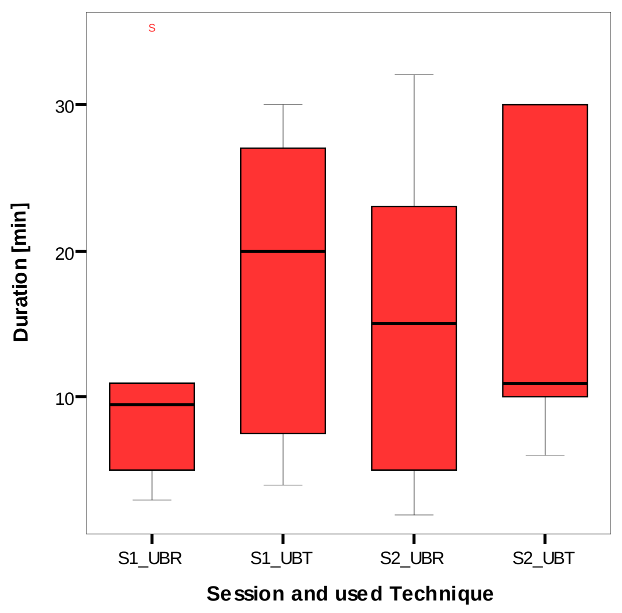


Table 7-7: First defect found [min]

	Mean Value	**Standard Deviation**
S1 UBR	12.17	10.59
S1 UBT-i	17.57	10.39
S2 UBR	15.44	10.93
S2 UBT-i	17.40	10.42

Mann-Whitney-Test *Session 1*	0.473 (-)
Mann-Whitney-Test *Session 2*	0.639 (-)

These outcomes are further used for the calculation of the efficiency in that way that not the whole first hour of inspection or test duration7 will be used for calculation - "*gross-processing time*", but until the first defect is found – "*net-processing time*". Therefore the mean value of each session from UBR and UBT-i will be used, which gives a more exact view on the efficiency of technique, most notably of course on the first time interval.

7.3.1 *Combined Sessions – Combined Techniques*

First let us take a look at which of these two software fault detection techniques performs most efficient. Therefore both sessions of UBR and UBT-i are combined and defect severity classes A and B are taken into account. As mentioned before only the net-processing time is used for this analysis too.

Figure 7-13: Efficiency, UBR vs. UBT-i [%]

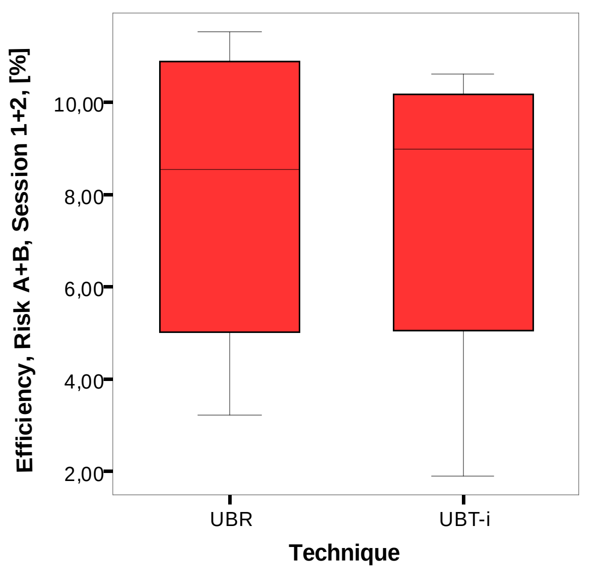


Table 7-8: Efficiency, UBR vs. UBT-i [%]

	Mean Value	**Standard Deviation**
UBR	7.96	2.35
UBT-i	7.62	2.63

Mann-Whitney-Test	0.773 (-)

As Figure 7-13 depicts UBR has a higher maximum than UBT-i and the same minimum level, although UBT-i on the other hand has a somewhat higher median. In Table 7-8 it can also be seen that the difference in efficiency between the two software fault detection techniques is only marginal. UBT-i has 0.34 % higher efficiency than UBR, which is really not very great. The Mann-Whitney test shows also that there is no significantly difference between UBR and UBT-i.

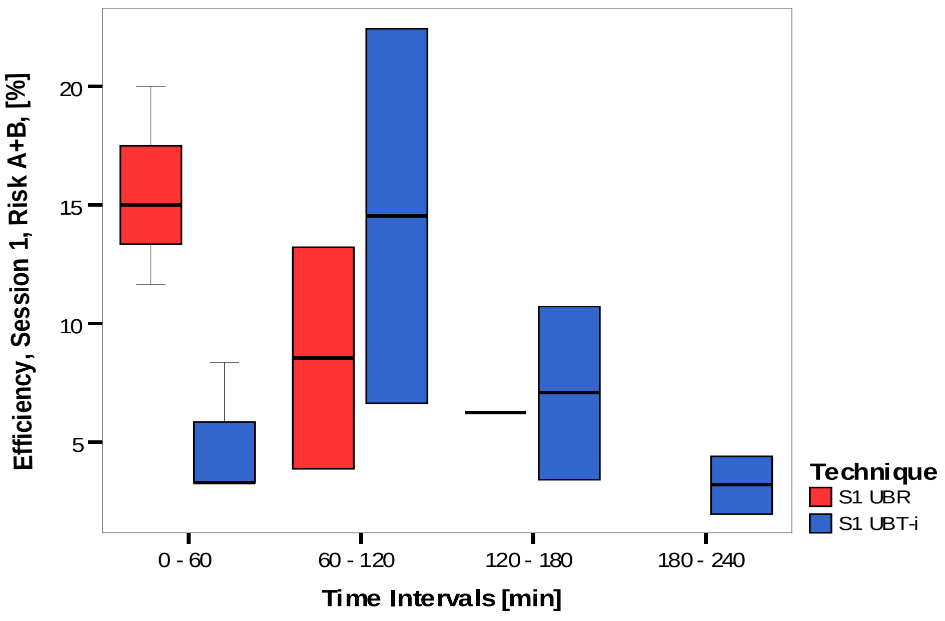
7.3.2 Temporal behavior of combined sessions and techniques

This section of the paper gives a very detailed view over both sessions, taxi and central, of UBR and UBT-i and its temporal behavior concerning the efficiency. This analysis uses again four timeframes and each of these timeframes consists of a duration time of one hour. Also the net-processing time is used for this investigation.

First it will be analyzed which used technique is during which session the most efficient one. Figure 7-14 shows the combined results.

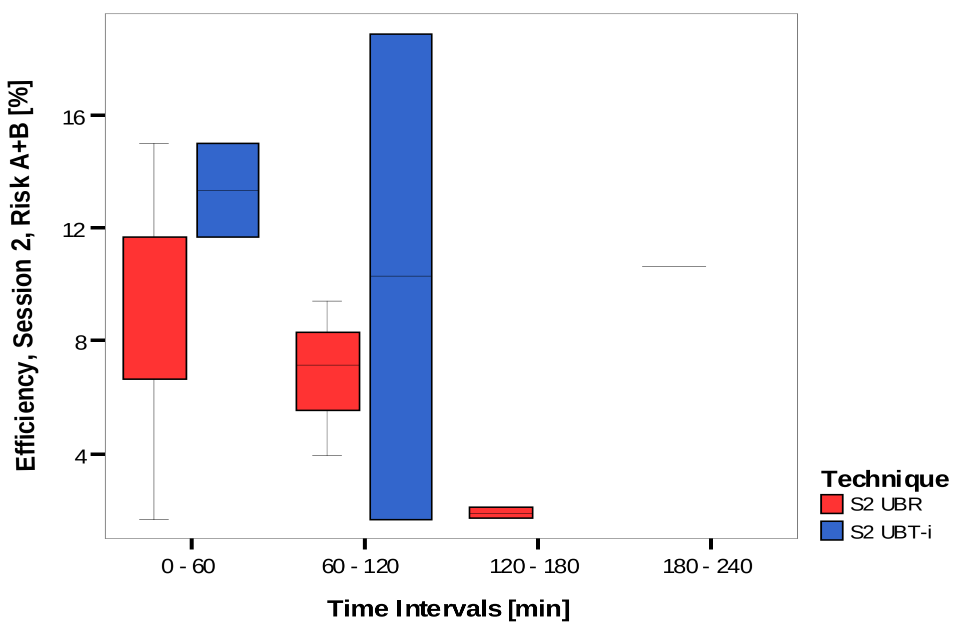
Afterwards each timeframe of each used software fault detection technique is analyzed in detail separately, to be able to determine which timeframe is the most efficient one of the investigated technique.

Figure 7-14: Efficiency, Session 1, UBR and UBT-i [%]



The box plot in Figure 7-14 shows clearly, that the most efficient timeframes for session number 1 are the first two time intervals, or the first 120 minutes of inspection or test duration. To declare an overall winner for session one is quite difficult, because in interval number 1 UBR is much more efficient than UBT-i whereas UBT-i is performing better than UBR in time interval number 2. In the third interval the techniques have hardly the same mean value of efficiency and the last time interval is on the whole not very efficient. So a definite winner cannot really be determined.

Figure 7-15: Efficiency, Session 2, UBR and UBT-i [%]



In session two, which can be seen in the box plot in Figure 7-15, the situation is a different one. UBT-i performs in the first two time intervals very efficient and at a higher mean value than UBR does. UBR surprisingly raises the mean value to a quite high level in the last of the four time intervals. So UBT-i is quite clear the more efficient software fault detection technique in session two, which was the central part of the study experiment.

7.3.3 Temporal behavior of separated sessions and techniques

The next investigations take a closer look at the efficiency of every session and technique separated from each other. This is done to be able to say which time interval of which technique is the most efficient one.

The Figure 7-16 gives an isolated view on the technique UBR of session one and shows the mean value of efficiency as well as the standard deviation in a bar chart. It can be seen that the first hour of inspection is the most efficient one and has also a very low value of standard deviation. In the next two time intervals the mean value decreases. Whereas the second hour has only the half of the mean value of the first hour, it on the contrary has also a higher standard deviation, which is quite remarkable. In the next time interval efficiency falls again until in the last hour it reaches zero. It can also be said, that the first three time intervals of UBR are mainly efficient, whereas in the last hour not even one defect were found by the participants.

Figure 7-16: Efficiency, Session 1, UBR [%]

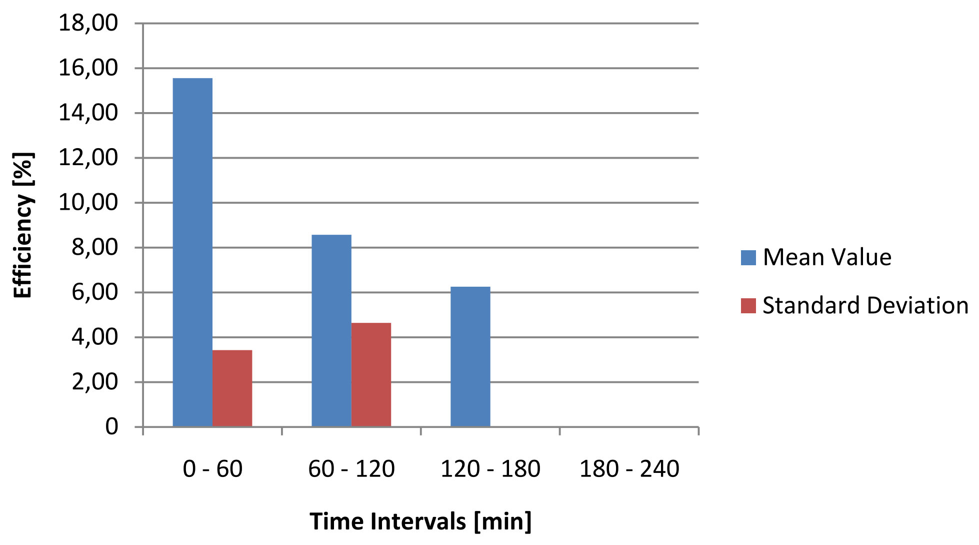


Table 7-9 shows the detailed outcomes of the calculation and depicts with the Kruskal Wallis test, that there is no significant difference between the records of the time intervals.

Table 7-9: Efficiency, Session 1, UBR [%]

Time Intervals [min]	Mean Value	Standard Deviation
0 – 60	15.56	3.42
60 – 120	8.56	4.64
120 – 180	6.25	0
180 - 240	0	0

Kruskal-Wallis-Test	0.304 (-)

Next we take a closer look at the first session of UBT-i. Figure 7-17 gives in the form of a bar chart overview about the progression of efficiency in the four time intervals. It declares that the most efficient intervals are number two and three or the second and third hour of testing duration. The value of standard deviation changes quite proportionally with the mean value and is therefore unremarkable. It can therefore be said, that the first three time intervals, or first three hours of UBT-i in session one are the highly efficient.

Figure 7-17: Efficiency, Session 1, UBT-i [%]

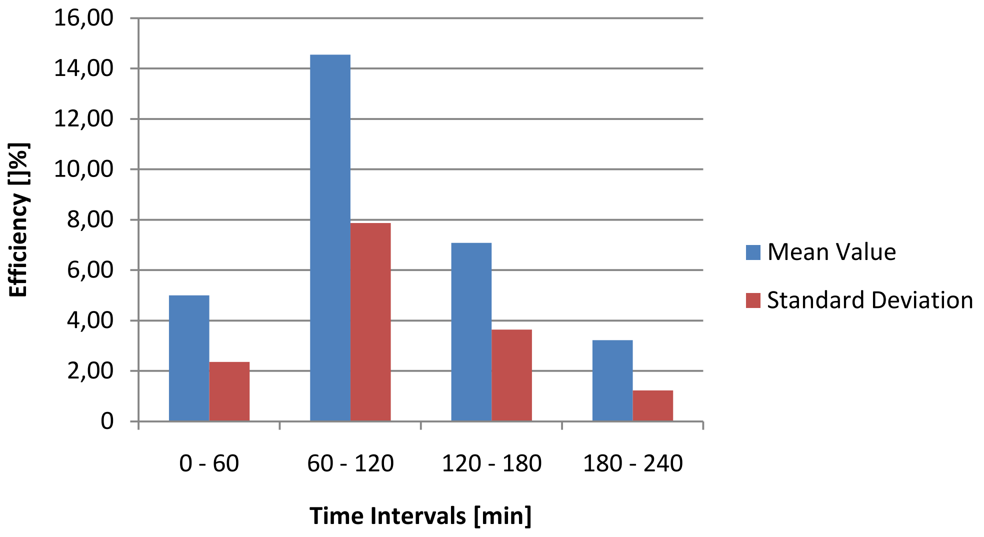


Table 7-10 shows the exact data of the analysis and states out that whit the Kruskal-Wallis test it can be declared, that there is no significant difference between the records of the investigated time intervals.

Table 7-10: Efficiency, Session 1, UBT-i [%]

Time Intervals [min]	Mean Value	Standard Deviation
0 – 60	5	2.36
60 – 120	14.54	7.87
120 – 180	7.08	3.63
180 - 240	3.22	1.22

Kruskal-Wallis	0.330 (-)

The next two investigations concentrate on the efficiency of session two, the central part of the experiment study.

Figure 7-18 shows the outcomes of UBR for session two in form of bar chart, of the most interesting investigation concerning the efficiency of this experiment study, although it can easily be explained. Session two of UBR, which has similarly to session one, two very efficient time intervals at the beginning of the inspection. What is highly remarkable about this part is the last hour of inspection – time interval number 4. This interval is called an outlier, because it has an abnormal high mean value and also no standard deviation, which is a little bit curious by itself. Because all investigations made, have a standard deviation when the mean value has a minimum of 6 %. This circumstance can of course be declared too. It is the consequence when only one group finds a quite high number of defects during the concerned time interval. An investigation of the data confirms this assumption. So the most efficient time intervals of session two of UBR are also the first two intervals.

Figure 7-18: Efficiency, Session 2, UBR [%]

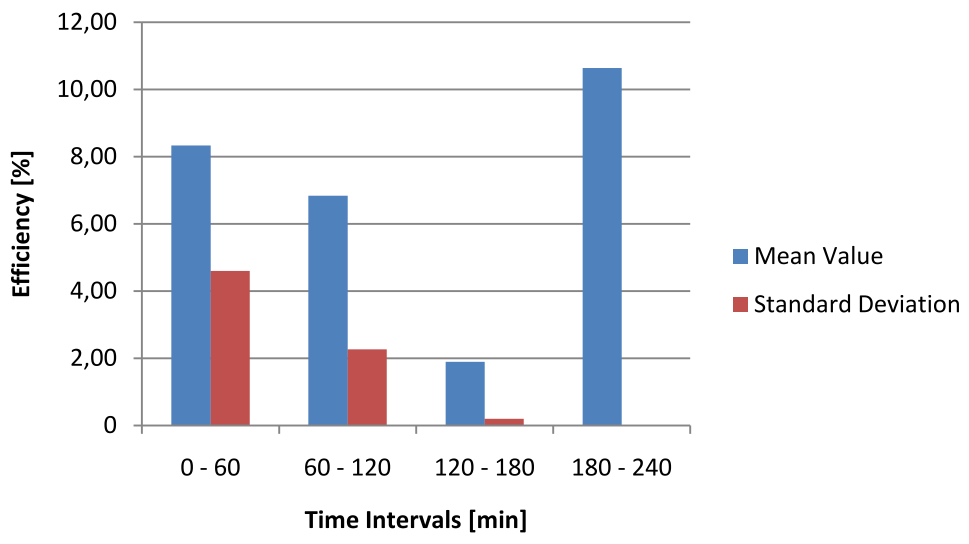


Table 7-11 shows the detailed data of the investigated session and technique and ensures that there is no significant difference between the recorded data of these time intervals.

Table 7-11: Efficiency, Session 2, UBR [%]

Time-Frame	**Mean Value**	**Standard Deviation**
0 – 60	8.33	4.59
60 – 120	6.83	2.26
120 – 180	1.89	0.19
180 - 240	10.64	0

Kruskal-Wallis-Test	0.557 (-)

In Figure 7-19 the bar chart for session two of UBT-i can be seen. This session is a little bit different than the first session of UBT-i. In the first hour of testing the mean value of efficiency reaches a very high level and standard deviation is remarkable low. The second hour has also a quite good level of efficiency, but with a definite higher value of the standard deviation. The outstanding thing is here that from the second hour on, no participant was able to find any matched defect of the classes A or B. A deeper analysis of this part has to be made related to the false positives. But it can of course be said, that for UBT-i the first two hours of inspection are the most efficient one.

Figure 7-19: Efficiency, Session 2, UBT-i [%]

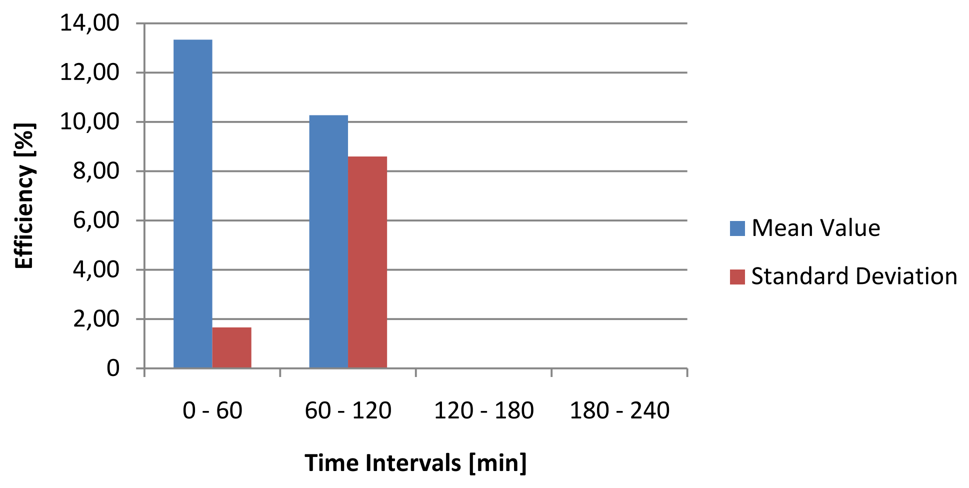


Table 7-12 shows the exact values of the calculations and also depicts with the Kruskal Wallis test, that there is no significant difference between the records of the time intervals.

Table 7-12: Efficiency, Session 2, UBT-i [%]

Time-Frame	Mean Value	Standard Deviation
0 – 60	13.33	1.67
60 – 120	10.27	8.60
120 – 180	0	0
180 - 240	0	0

Kruskal-Wallis-Test	0.439 (-)

The next part of the thesis deals with the number of false positives found, which are also analyzed in a temporal context to find out in which period of time the candidates found least of the matched defects.

7.4 False positives

These are defects which are registered by the inspectors, but do not belong to any defined referenced seeded defect according to the overall number of seeded defects by the experts. This section deals with them and will analyze its spreading in the different time intervals, sessions and used software fault detection techniques of the study experiment. The sessions are again divided in eight time intervals, as used in investigating the effectiveness before, each consisting of 30 minutes. In contrary to the analysis of effectiveness and efficiency, all types of the defect severity classes, which are A, B and C are taken into account. The beginning of the analysis is also the real beginning of the inspection or test, which means the "*gross-processing time*" will be used here.

A good software fault detection technique guides the inspectors or testers in identifying only true defects and therefore it should reduce the overall number of false positives at the same time. The more false positives that were found the more effort for defect removal and post-inspection data analysis will be in the later software development life cycle.

First a comparison between the overall number of false positives between the used techniques UBR and UBT-i, with all data from both sessions will be made. Afterwards a detailed look at every separated session of each technique of the experiment study will be made.

7.4.1 Combined Sessions – Combined Techniques

The first investigation of the False Positives is the comparison between UBR and UBT-i to find out with which of these two software fault detection techniques the fewest false defects were found by the participants.

Figure 7-20: False Positives, UBR vs. UBT-i [%]

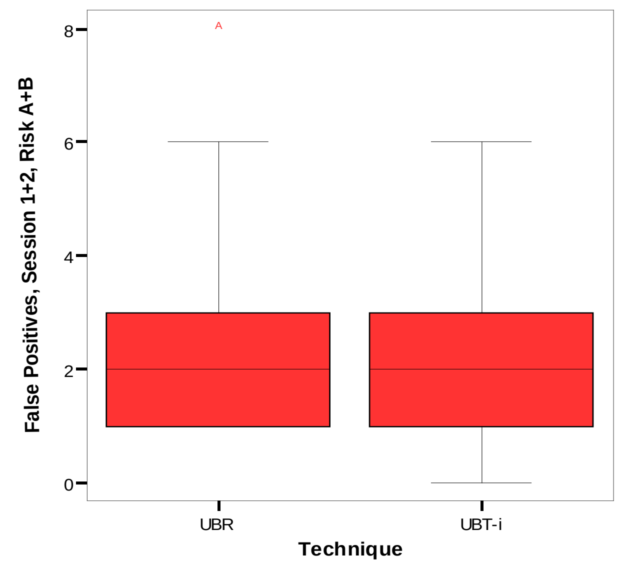


Table 7-13: False Positives, UBR vs. UBT-i [%]

	Mean Value	Standard Deviation
UBR	1.58	0.96
UBT-i	2.34	0.76
Mann-Whitney-Test		0.541 (-)

When considering the outcomes of Figure 7-20 not a clear decision can be made. Although UBR has an outlier the median of the techniques is hardly at the same level. The results of Table 7-13 approve this statement with a somewhat higher mean value of UBT-i 2.34 on the contrary to UBR 1.58. Although the difference is not very big UBR has a lower mean value of 0.76 and therefore performs a little bit better than UBT-i. The Mann-Whitney test shows, that there is no significant difference between these two techniques.

7.4.2 Temporal behavior of combined sessions and techniques

This section of the thesis gives a very detailed overview of the sessions one and two, taxi and central, of the used techniques UBR and UBT-i as well as its temporal behavior concerning fault positives. So it should be possible to determine in which time interval the most fault positives of the investigated techniques will be found. This analysis uses again eight time intervals and each of these timeframes consists of duration of 30 minutes.

First it will be analyzed which technique performs best during which session, i.e. who finds the least false positives. Afterwards each timeframe of each used software fault detection technique is analyzed in detail separately, to be able to determine in which timeframe the least false positives will be found

To be able to investigate also which of the two techniques performs best concerning false positives in which timeframe of session one or two, the next figures are consulted.

Figure 7-21 concerns the session one, the taxi part of the study experiment and reveals that UBR performs better in the first three time intervals, or 90 minutes than UBT-i, but the tide is turning in the fourth time interval. In this special timeframe UBT-i performs much better than it's counterpart. But this change doesn't take very long. It can be seen that after this interval only by candidates, who are using the UBT-i technique, a number of false positives were found. That happens, because the inspectors using UBR were not able to find in these three time intervals any kind of defects and also no fault positives, what can be seen in Figure 7-4. Therefore can be said, that in session one of the experiment, UBR performs better than UBT-i.

Figure 7-21: False Positives, Session 1, UBR and UBT-i [%]

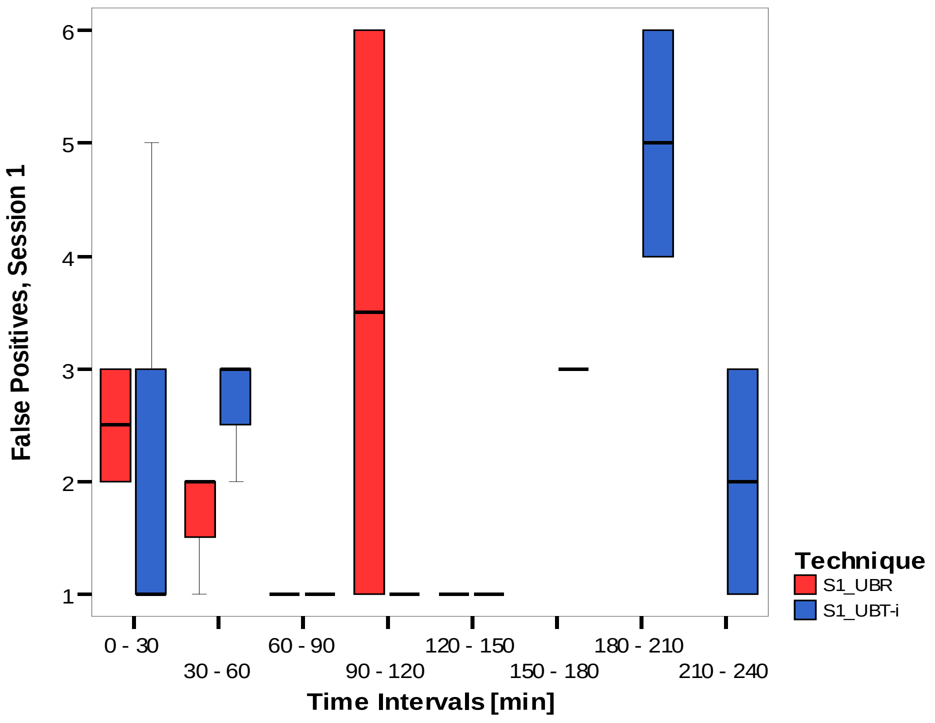
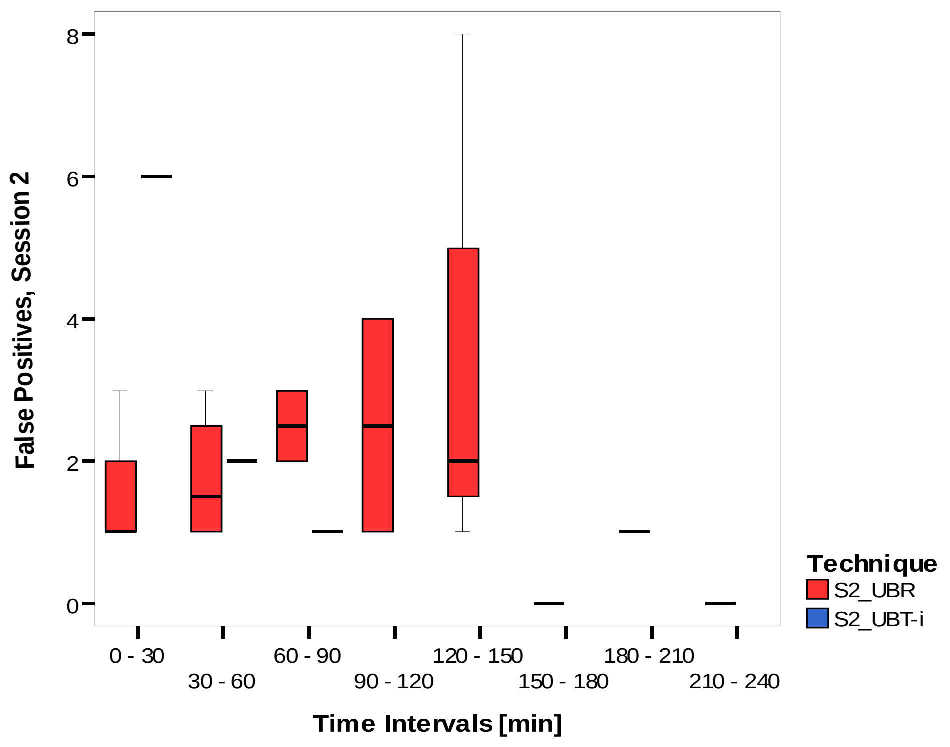


Figure 7-22 shows the results of session two, the central part. Which is different than the results of session one. The amount of mean value of the technique UBR keeps on a quite low level only until the first hour of inspection. Whereas UBT-i starts with a very high number of faults positives it falls down to zero in the next four timeframes. Candidates using UBT-i were after this time intervals not able to find any kind of defects and also no fault positives, what explains the rest of the time intervals. This can be seen in detail in Figure 7-10. Although UBR does not perform as good as in session concerning the number of false positives found it can again be said that in session two UBR is performing better than UBT-i, at least for the first four time intervals.

Figure 7-22: False Positives, Session 2, UBR and UBT-i [%]

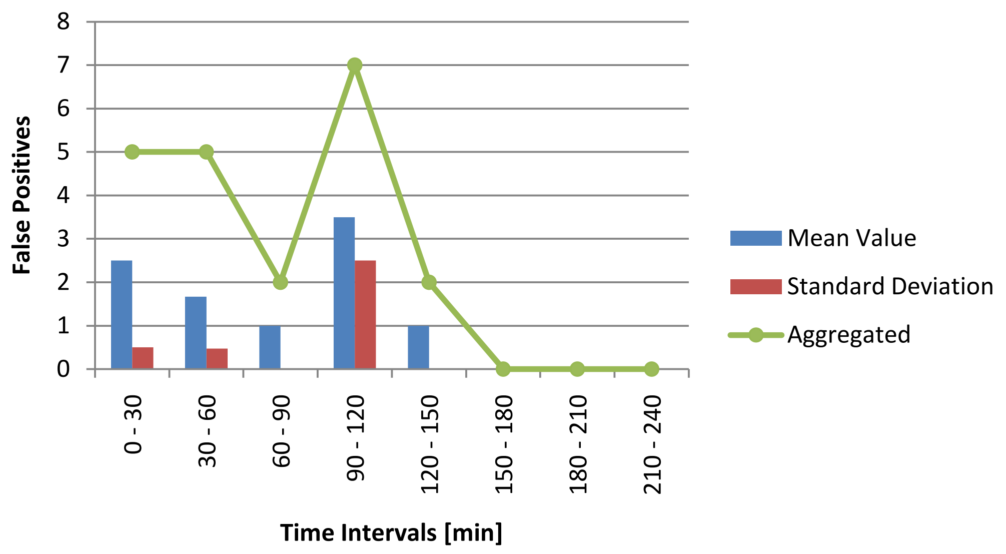


7.4.3 *Temporal behavior of separated sessions and techniques*

The next two investigations take a closer look at the fault positives of session one of every software fault detection technique separated from each other. This is done to be able to say which time intervals of which technique have the least fault positives.

Figure 7-23 shows the mean values, standard deviation and the aggregated number of fault positives found by participants using UBR in the first session of the experiment study in a combined bar and line chart. It can be seen that during the first three time intervals a quite low number of fault positives were found, between 5 and 2. Remarkable is that after the third interval the number of found fault positives rises up to the 7. The conclusion is that UBR is performing at a good level concerning fault positives for the first three time intervals or until 90 minutes of the first session of the experiment.

Figure 7-23: False Positives, Session 1, UBR [%]



The exact values are below in Table 7-14, which also shows the outcome of the Kruskal-Wallis test. The result is, that there is no significant difference between the records of the time intervals of UBR of session one.

Table 7-14: False Positives, Session 1, UBR [%]

Time Interval [min]	Mean Value	Standard Deviation
0 – 30	2.50	0.50
30 – 60	1.67	0.47
60 – 90	1.00	0
90 – 120	3.50	2.50
120 – 150	1.00	0
150 – 180	0	0
180 – 210	0	0
210 – 240	0	0

Kruskal-Wallis-Test	0.269 (-)

The first session of UBT-i is a very interesting one when it comes to investigate the temporal behavior of fault positives. Figure 7-24 below illustrates the results in form of a combined bar and line chart. On the first view can already be seen that the mean value starts good level for time interval one, which is hardly the same as for the first session of UBR. Remarkable is a slump of the number of fault positives found in the timeframes number 3, 4, 5 and 6. It must be said that UBT-i performs very well, especially until the sixth or seventh time interval concerning the number of false positives found, but the least of them were found from the third to the sixth timeframe of testing duration.

Figure 7-24: False Positives, Session 1, UBT-i [%]

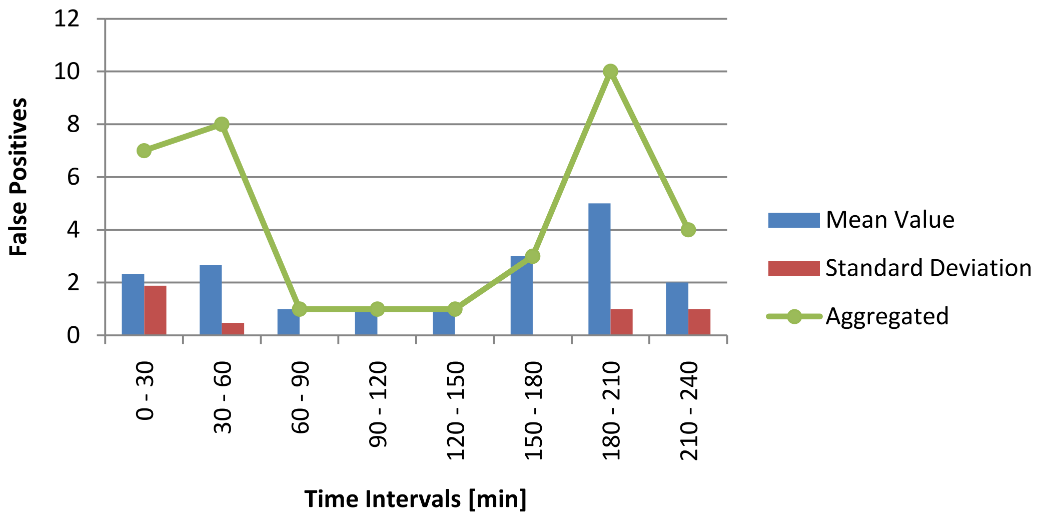


Table 7-15 shows the exact values and the differences of the recorded time intervals are again not significantly different, which can be seen by the result of the Kruskal-Wallis test.

Table 7-15: False Positives, Session 1, UBT-i [%]

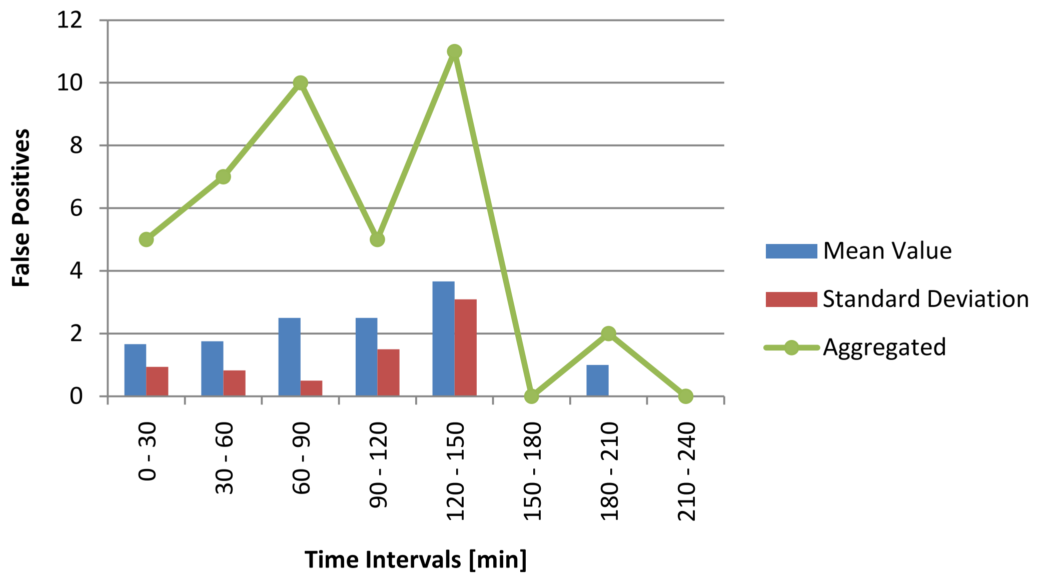
Time Interval [min]	Mean Value	Standard Deviation
0 – 30	2.33	1.89
30 – 60	2.67	0.47
60 – 90	1.00	0
90 – 120	1.00	0
120 – 150	1.00	0
150 – 180	3.00	0
180 – 210	5.00	1.00
210 – 240	2.00	1.00

Kruskal-Wallis-Test	0.364 (-)

The next two analyses concern the number of fault positives found of sessions 2 for UBR and UBT-i. The two techniques are again separated from each other to be able to make conclusions for every investigated time interval of the two sessions from the experiment study.

UBR shows a little bit a different trend line in the bar and line chart in Figure 7-25 as it did in the first session of the experiment. On the first view can already be seen that the mean value of UBR for fault positives keeps a quite low level for the first two timer intervals and then rises consequently until the third and fifth intervals. Although time interval number 3 is an outlier, in which on the whole a quite low number of fault positives were found the mean value keeps at a quite high level. Remarkable is also interval six in which no fault positive were found by the inspectors. So UBR performs quite well for the first 120 minutes of inspection duration.

Figure 7-25: False Positives, Session 2, UBR [%]



In Table 7-16 below the exact values of the investigation can be seen. Also the Kruskal-Wallis test is contained, which shows that there is no significant difference between the investigated records.

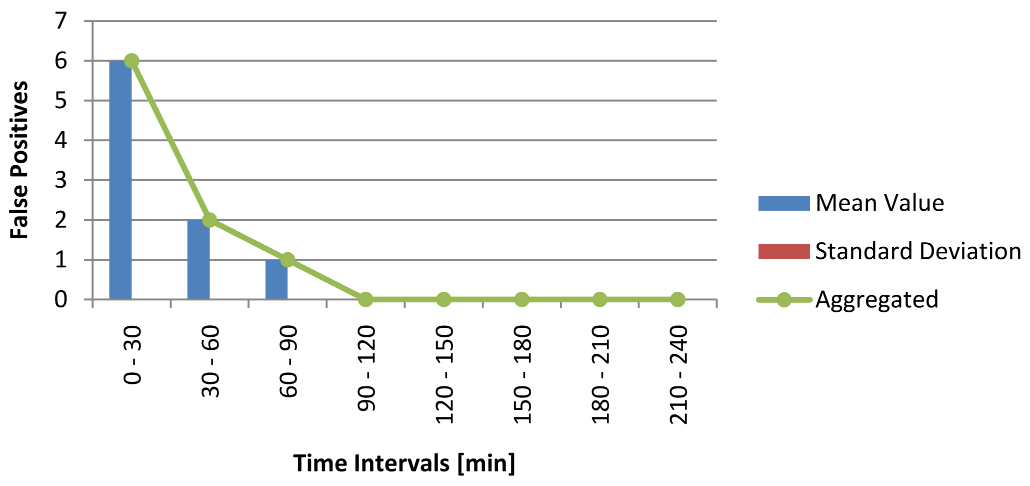
Table 7-16: False Positives, Session 2 UBR [%]

Time Interval [min]	Mean Value	Standard Deviation
0 – 30	1.67	0.94
30 – 60	1.75	0.83
60 – 90	2.50	0.50
90 – 120	2.50	1.50
120 – 150	3.67	3.09
150 – 180	0	0
180 – 210	1.00	0
210 – 240	0	0

Kruskal-Wallis-Test	0.190 (-)

The technique UBT-i is again very special in the second session of this experiment study. The first 30 minutes of testing are performing very bad, which can be seen in the combined bar and line chart of Figure 7-26 below, because of the high number of fault positives found, which is 6. The next time interval is then a better one, only a few fault positives were made, what goes along with a lower number of 2. The third interval is again performing even better with 1 false positive found by the participants. Afterwards no false positives were found by the testers although some found defects were recorded, what can be seen in Figure 7-10.Therefore UBT-i is performing good after the third timeframe of testing duration, but the first time interval is a quite outstanding one.

Figure 7-26: False Positives, Session 2, UBT-i [%]



The data values can be seen in the Table 7-17 below and also the Kruskal-Wallis test, which depicts that there is no significant difference between the investigated records.

Table 7-17: False Positives, Session 2, UBT-i [%]

Time Interval [min]	Mean Value	Standard Deviation
0 – 30	6.00	0
30 – 60	2.00	0
60 – 90	1.00	0
90 – 120	0	0
120 – 150	0	0
150 – 180	0	0
180 – 210	0	0
210 – 240	0	0

Kruskal-Wallis-Test	0.368 (-)

The next chapter of the paper concentrates on the findings made and discusses them. The analyses are also assembled together in a common context to be able to make conclusions about the made investigations and to answer the hypotheses, which were made in chapter 5.2.

8 Discussion

In this section the results of the experiment as well as the practical implications are discussed. The hypotheses of the experiment are summarized and interpreted as follows:

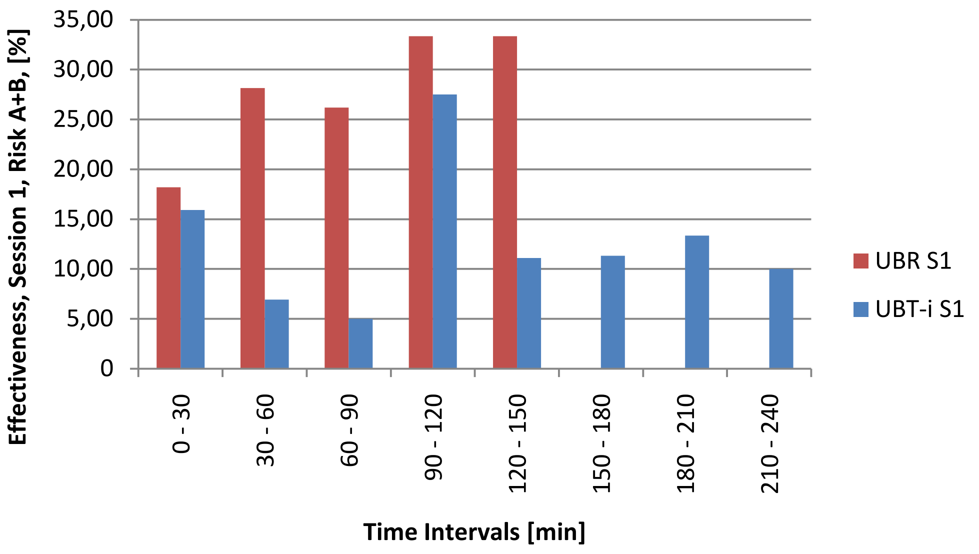
8.1 Is UBR more Effective and Efficient than UBT-i?

This chapter will give information about the performance of the investigated techniques and shows the outcomes of the comparison.

***H1: Effectiveness (UBR) > Effectiveness (UBT-i) for Design Documents in the first 120 minutes*:**

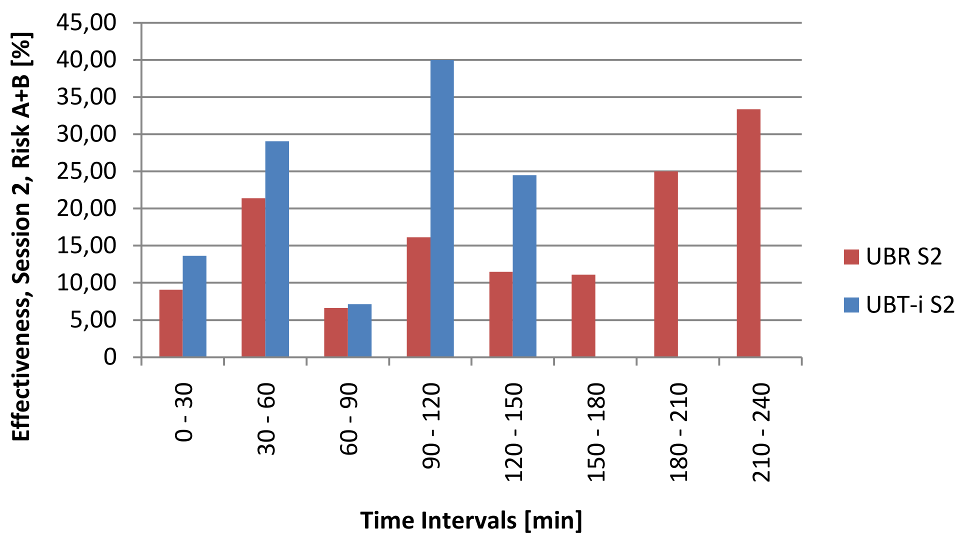
The investigations of the experiment study were able to provide positive results for this hypothesis in session one. The Figure 8-1 shows a combination of the results, which were presented in detail in chapter 7. It can clearly be seen that in the first 120 minutes of inspection and testing duration of session one UBR performs more effective than UBT-i.

Figure 8-1: Mean Value of Effectiveness, Session 1, UBR and UBT-i



The Figure 8-2 below shows the combined results of the investigation of effectiveness for session two from the experiment study. It was therefore not possible to provide a positive result for the hypothesis concerning session two. UBT-i performs more effective than UBR for the first 120 minutes of session two.

Figure 8-2: Effectiveness, Session 2, UBR and UBT-i

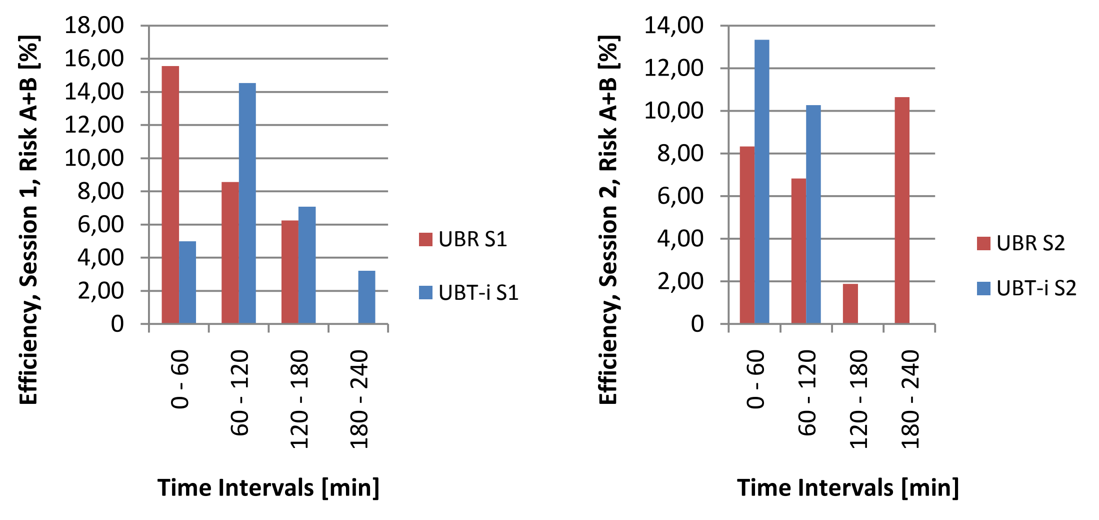


It is therefore not really possible to answer this hypothesis positively or negatively, because it depends on the experiment session. The outcomes of this hypothesis should be analyzed in more detail in future thesis.

***H2: Efficiency (UBR) > Efficiency (UBT-i) for Design Documents in the first 120 minutes of session one and two*:**

This hypothesis must be rejected. It can be seen in the combined bar charts below in Figure 8-3, that UBR only performs more efficient in the first time interval of session one. Afterwards UBT-i performs better in the asked first 120 minutes of inspection and testing duration.

Figure 8-3: Mean Value of Efficiency, Session one and two, UBR and UBT-i



UBT-i is therefore more efficient than UBR, what can bring positive effects on decisions for project and quality managers concerning the choice when UBR or UBT-i should be chosen as the software fault detection technique used.

8.2 Are the Techniques basically effective and efficient in the first 120 minutes?

This research approach should answer the question, if it is possible to shorten the duration of inspections and tests, but to still provide a high level of the defect detection performance of both techniques.

H3: Are the techniques most effective and efficient in the time interval from 0 to 120 minutes for design documents:

For UBR as well as for UBT-i concerning the efficiency this hypothesis is correct, what can be seen in the Figure 8-3. But things get a little bit complicated when effectiveness has to be analyzed, because of the different outcomes of the experiment sessions. UBR is very effective in the requested time interval of session one and session two. The results for UBT-i are not so good for the first 120 minutes of testing duration. It can be said that UBT-i on the whole needs more time to perform really effective.

This hypothesis can therefore not really be answered with yes.

8.3 During which time intervals will the fewest False Positives be found?

With a knowledge of the prediction when the fewest false positives will be found a further prescription can be made about the defect detection performance of UBR and UBT-i concerning their outcome of the first 120 minutes of inspection and testing duration.

H4: Will with UBR fewer false positives are found in the first 120 minutes than with UBT-i:

The result of this hypothesis is also different in the experiment sessions. Whereas UBR performs better concerning the number of false positives found in session one, see Figure 8-4, UBT-i finds fewer false positives in session two, see Figure 8-5. The two figures below are combined from the results of chapter 7. For session one the hypothesis is correct, but for session two it has to be rejected.

Figure 8-4: **False Positives, Session 1, UBR and UBT-i**

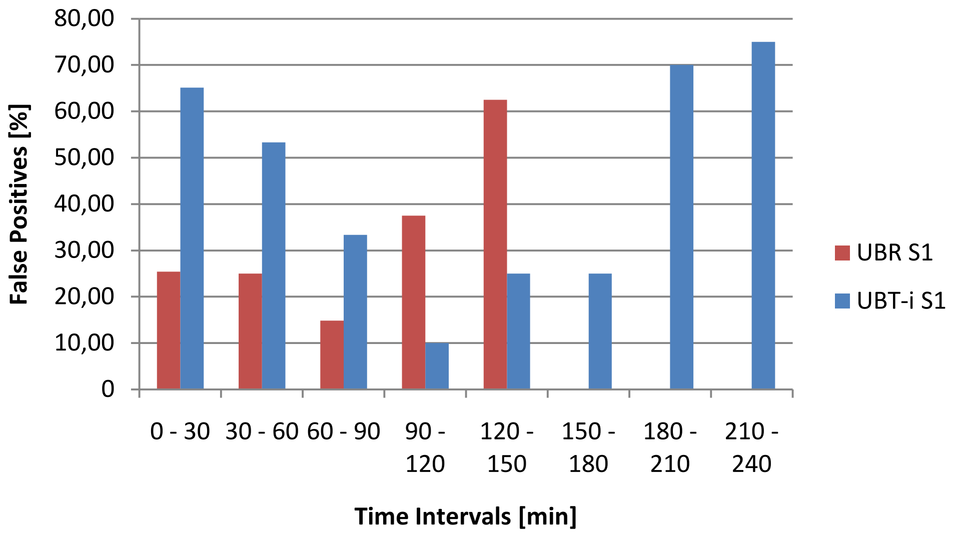
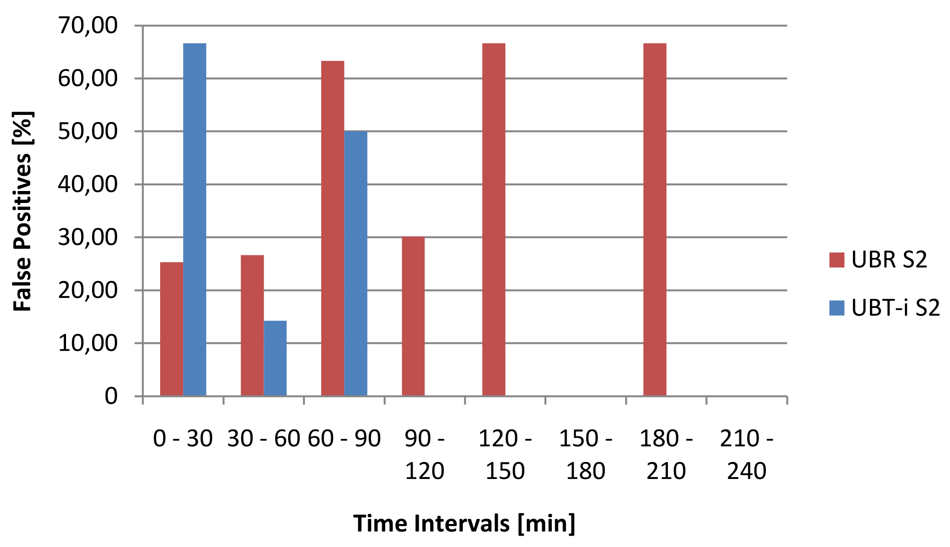


Figure 8-5: False Positives, Session 2, UBR and UBT-i



H5: Will the fewest false positives in UBR and UBT-i be produced in the first 120 minutes of inspection and testing:

For the software fault detection technique UBR this hypothesis has to be rejected. Although in session two the trend line begins at a low level and rises in the time intervals. It breaks in after the fifth timeframe. Session one has a completely different trend line which starts with a higher number of found fault positives and gets lower in the later time intervals.

For UBT-i the hypothesis also has to be rejected because in session two fault positives were only found in the first three time intervals of testing and the trend line in session one is also not very tending increase of found false positives.

Overview of hypotheses

The following Table 8-1 should give an overview about the final status of the made hypotheses.

Table 8-1: Overview of hypotheses

Hypotheses	Description	Status
H1	Effectiveness (UBR) > Effectiveness (UBT-i)	⍓
H2	Efficiency (UBR) > Efficiency (UBT-i)	☒
H3.1	UBR most effective and efficient < 120 min	☑
H3.2	UBT-i most effective and efficient < 120 min	☒
H4	UBR fewer false positives than UBT-i < 120 min	⍓
H5	Fewest false positives of UBR & UBT-i < 120 min	☒

☑ positively, ☒ rejected, ⍓ cannot be answered (distinction in sessions)

9 Conclusions and Follow-Up

In the first part of this thesis an introduction to the basic principles of software fault detection techniques were given. These concepts help to understand how the investigated techniques work and which differences and commons they may have. These things are important to understand, so the different approaches of them are visible to the reader. Afterward the experiment study, on which this thesis relies on, is described in detail and visualized with a number of graphics, helping to get a better knowledge of the planning, preparation and execution of the experiment held in an academically environment. The next chapter is describing the investigated research approach and the basic outcome of this paper. Following with the results of the experiment study and the investigated measures are presented and described. Afterwards the examined results are set in association with the made hypothesis as well as discussed concerning several perspectives of these findings.

Inspection and testing are both very important and also often used approaches in the software engineering practice, which addresses the same main goal – find as many crucial defects in software products as possible. Software Inspection focuses mainly on design specification documents in early phases of the software development lifecycle, whereas traditional testing approaches concentrate more on the implementation phases during the process or even later. Therefore this thesis uses another testing variant, which is called UBT-i, it integrates the benefits of software inspection and software testing. UBT-i is not in the need of executable code and is also a desk test, which is different from traditional testing approaches. Another feature of UBT-i is that the participants generate test cases during their inspection process.

The investigations of this thesis concentrate mainly on the temporal behavior of the software fault detection techniques UBR and UBT-i. The outcomes concerning this temporal behavior showed up some interesting results, but unfortunately not all approaches could be fulfilled concerning the hypotheses. UBR performs in the asked time interval of 120 minutes very effective and efficient. UBT-i in contrary needs more time for its testing duration to achieve as good defect detection results. This delivers an important indicator for the planning of analytical quality assurances in consideration of the scheduled inspection time for UBR as well as UBT-i in a not academically environment. The outcomes of this Thesis should therefore be able to help project as well as quality managers to more precisely define their inspection and testing duration efforts to gain the wanted results.

The comparison of the software fault detection techniques UBR and UBT-i showed that UBR is on the whole not the superior technique as assumed. Concerning the investigated measures, effectiveness and efficiency, the findings were not consistent in the two sessions of the experiment study. Whereas UBR tends to have a better defect detection performance in session number 1 UBT-i did a better job in session number 2. Therefore it cannot clearly state out, which of these techniques is the superior one in the investigation of this thesis.

The assumed hypotheses concerning the number of false positives found in a temporal context were not able to show the expected outcomes. It showed the complete opposite. To clarify these results further studies are needed with a higher number of participants, more seeded defects and a greater number of software artifacts in which defects have to be detected.

Also the differences between the experiment sessions, as mentioned several time before, were partially remarkable, in the context of the investigated measures used like, effectiveness, efficiency and also false positives. To clarify these correlation further studies will be needed. Also the learning effect for these software fault detection techniques should be more investigated, because it was expected that session number 2 of the experiment study should perform better than session number 1 in all asked performance measures.

To proof these results a larger evaluation should be conducted and further experimentation should be planned to provide more understanding about the temporal behavior of UBR and UBT-i. Also a study in a realistic environment or project should be made based on this experiment study in an academically environment.

References

[1] Ackerman, A. F., Buchwald, L.S., Lewsky, F.H., *“Software Inspections: An Effective Verification Process”,* IEEE Software, 6(3): pp. 31-36, 1989.

[2] Andersson C, “*Exploring the Software Verification Process with Focus on Efficient Fault Detection*”, Lund University, 2003

[3] Andersson C., Thelin T., Runeson P., Dzamashvili N.: “*An experimental evaluation of inspection and testing for detecting of design faults*”, ISESE’ 03 – International Symposium of Empirical Software Engineering, pp. 174-184, 2003.

[4] Augustin A, “*Test-Driven Development: Concepts, Taxonomy and Future Direction”*, Proseminar Reliable Systems, Fakultät Informatik, Tehnische Universität Dresden, 2006

[5] Aurum A., Petersson H., Wohlin C., “State-*of-the-art: software inspections after 25 years*”, Softw. Test. Verif. Reliab. 2002; 12: pp. 133–154.

[6] Basili V.R., Selby R.W., “*Comparing the Effectiveness of Software Testing Strategies”,* IEEE Vol. SE-13, Issue 12, pp.: 1278-1296, Dec 1987

[7] Basili VR, Green S, Laitenberger O, Lanubile F, Shull F, S¨orumg°ard S, Zelkowitz M, “*The empirical investigation of perspective-based reading*”, International Journal on Empirical Software Engineering 1996; 1(2): pp. 133–164.

[8] Basili, V. R., “Evolving and Packaging Reading Technologies”, Journal of Systems and Software, 38(1), Cockburn, A., Writing Effective Use Cases, Addison-Wesley, USA, 2001.

[9] Basili, V. R., Shull, F. and Lanubile, F., *„Building Knowledge through Families of Experiments*”, IEEE Transactions on Software Engineering, 25(4): pp. 456-473, 1999.

[10] Bass, L., Clements, P. and Kazman, R., “*Software Architecture in Practice*”, Addison-Wesley, USA, 1998.

[11] Biffl S., Winkler D., Thelin T., Höst M., Russo B., Succi G.: “*Investigating the Effect of V&V and Modern Construction Techniques on Improving Software Quality*”, Poster presented at ISERN 2004.

[12] Bisant DB, Lyle JR, “*A two person inspection method to improve programming productivity*”, IEEE Transactions on Software Engineering 1989; 15(10): pp. 1294–1304.

[13] Björn Regnell, Per Runeson, Claes Wohlin, „*Towards integration of use case modelling and usage-based testing*“, The Journal of Systems and Software 50 (2000) pp. 117±130

[14] Blakely, F. W. and Boles, M. E., *“A Case Study of Code Inspections,”* Hewlett- Packard Journal, 42(4):58-63, 1991.

[15] Boehm, B. W., “*Software Engineering Economics. Advances in Computing Science and Technology”*, Prentice Hall, 1981.

[16] Boem B, “*A Spiral Model of Software Development and Enhancement*”, Computer, IEEE, 21 (5) pp.: 61 – 72, May 1988

[17] Bourgeois, K. V., *“Process Insights from a Large-Scale Software Inspections Data Analysis. Cross Talk,”* The Journal of Defense Software Engineering, 17-23, 1996.

[18] Briand, L., E -Emam, K., Fussbroich, T., and Laitenberger, O., *„Using Simulation to Build Inspection Efficiency Benchmarks for Development Projects”*, Proceedings, 1998.

[19] C. Ghezzi, M. Jazayeri, and D. Mandrioli, “*Fundamentals of Software Engineering*”, Englewood Cliffs, NJ: Prentice Hall, 1991.

[20] Cheng, B. and Jeffrey, R., “*Comparing Inspection Strategies for Software Requirements Specifications*”, Proceedings of the 1996 Australian Software Engineering Conference, pp: 203-211, 1996.

[21] Ciolkowski M, C. Differding, O. Laitenberger and J. Münch, “*Empirical Investigation of Perspective-based Reading: A Replicated Experiment*”, Submitted to 7. Workshop on Empirical Studies of Programmers.

[22] Deck, M., “*Cleanroom Software Engineering to reduce Software Cost*”, Technical report, Cleanroom Software Engineering Associates, 6894 Flagstaff Rd. Boulder, CO 80302, 1994.

[23] Dennis, A. and Valacich, J., *“Computer brainstorms: More heads are better than one.”* Journal of Applied Social Psychology, 78(4): pp. 531-537, 1993.

[24] Wohlin, C., Regnell., B., Wesslén, A. and Cosmo., H, „*User-Centered Software Engineering – A Comprehensive View of Software Development*", Proc. of the Nordic Seminar on Dependable Computing Systems*,* pp. 229-240, 1994.

[25] Drew C, Hardman, Michael L. and Hart, Ann Weaver, "*Designing and Conducting Research: Inquiry in Education and Social Science*", Needham Heights, Massachusetts: Simon and Schuster Company, 1996.

[26] Dunsmore, A., Roper, M., Wood, M., *"Object-Oriented Inspection in the Face of Delocalisation,"* Proceedings of the 22nd International Conference on Software Engineering, Limerick, 2000.

[27] Dyer, M., "*The Cleanroom Approach to Quality Software Development*", John Wiley and Sons, Inc, 1992.

[28] Dyer, M., "*Verification-based Inspection*", Proceedings of the 26th Annual Hawaii International Conference on System Sciences, pp: 418-427, 1992.

[29] Ebenau, R. G. and Strauss, S. H., *Software Inspection Process*, McGraw-Hill*,* USA, 1994.

[30] Fagan ME, "*Advances in software inspections*", IEEE Transactions on Software Engineering 1986; 12(7): pp. 744–751.

[31] Fagan ME, "*Design and code inspections to reduce errors in program development*", IBM Systems Journal 1976; 15(3): pp. 182–211

[32] Fagan ME, Advances in software inspections, "*IEEE Transactions on Software Engineering*", 1986, 12(7): pp. 744–751.

[33] Fagan, M. E. "*Design and Code Inspections to Reduce Errors in Program Development*", IBM System Journal, 15(3): pp. 182-211, 1976.

[34] Fagan, M. E., "Design *and Code Inspections to Reduce Errors in Program Development*", IBM Systems Journal, 15(3): pp. 182-211, 1976.

[35] Freimut B, O. Laitenberger, S. Biffl, "*Investigating the Impact of Reading Techniques on the Accuracy of Different Defect Content Estimation Techniques*", 2001.

[36] Fusaro P, Lanubile F, Visaggio G, "*A replicated experiment to assess requirements inspection techniques*", International Journal on Empirical Software Engineering 1997; 2(1): pp. 39–57.

[37] Gilb T, Graham D, "*Software Inspection*", Addison-Wesley: Wokingham, U.K.

[38] Goel L. A, "*Software Reliability Models: Assumptions, Limitations and Applicability",* IEEE Transaction on Software Engineering Vol. 11, No 12, pp.: 1411 – 1423, 1985

[39] Gough PA, Fodemski FT, Higgins SA, Ray SJ, "*Scenarios—an industrial case study and hypermedia enhancements*", Proceedings 2nd IEEE International Symposium on Requirements Engineering, IEEE Computer Society Press: Los Alamitos, CA, 1995; pp. 10–17.

[40] Herbsleb J., Zubrow D., Goldenson D., Hayes W. and Paulk M., „*Software Quality and the Capability Maturity Model",* Vol 40, No. 6 Communications of the ACM, June 1997.

[41] IEEE Standard, *Standard for software reviews*, 1028-1997, 1998.

[42] Jacobson, I., Christerson, M., Jonsson, P. and Övergaard G. *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, USA, 1992.

[43] Jody P, Software Lifecycle Model, http://www.jodypaul.com/SWE/LCM/, 1999

[44] John D. Musa, "*Operational Profiles in Software* I *Reliability Engineering",* IEEE, 1993

[45] Johnson, p.M., Tjahjono, D., *"Does Every Inspection Really Need a Meeting"*, Journal of Empirical Software Engineering, vol. 3, no. 1, pp. 9-35, 1998

[46] Kaner, C., *"The Performance of the N-Fold Requirement Inspection Method,"* Requirements Engineering Journal, vol. 2, no. 2, pp. 114-116, 1998.

[47] Karlsson, J. and Ryan, K., "*A Cost-Value Approach for Prioritizing Requirements*", IEEE Software, 14(5): pp. 67-74, 1997.

[48] Knight JC, Myers AE, "*An improved inspection technique*", Communications of ACM 1993; pp. 36(11): pp. 50–69

[49] Kouchackjian A, R. Fietkiewicz, "*Improving a product with usage-based testing*", Information and Software Technology 42, pp: 809 – 814, 2000.

[50] Kusumoto, S., Chimura, A., Kikuno, T., Matsumoto, K., Mohri, Y., *“A Promising Approach to Two-Person Software Review in an Educational Environment,”* Journal of Systems and Software, no. 40, pp. 115-123, 1998.

[51] Laitenberger O, DeBaud JM, “*Perspective-based reading of code documents at Robert Bosch GmbH*”, Information and Software Technology 1997; 39(11): pp. 781–791.

[52] Williams L.A., “*The Collaborative Software Process”*, Dissertation, Department of Computer Science, University of Utah, 2000.

[53] Levine, J. M. and Moreland, R. L., *“Progress in Small Group Research,”* Annual Review of Psychology, 41: pp. 585-634, 1990.

[54] Linger RC, Mills HD, Witt BI, “*Structured Programming: Theory and Practice*”, Addison-Wesley: Reading, MA, 1979.

[55] Madachy, R., Little, L., and Fan, S., *“Analysis of a successful Inspection Program,”* Procceding of the 18th Annual NASA Software Eng. Laboratory Workshop, pp: 176-198, 1993.

[56] Maximilien M, Williams L, “*Assessing Test-Driven Development at IBM”,* IEEE, 2003

[57] Musa J.D, “*Operational profiles in software reliability engineering”*, IEEE Software, March pp.: 14 – 32, 1993

[58] Musa, J. D., *Software Reliability Engineering: More Reliable Software, FasterDevelopment and Testing*, McGraw-Hill, USA, 1998.

[59] Myers G. J, “*The Art of Software Testing*”, Wiley Interscience, 1979.

[60] Myers, G. J., *“A controlled experiment in program testing and code walkthroughs/ inspections”*, Communications of the ACM, 21(9): pp: 760-768, 1978.

[61] National Aeronautics and Space Administration, *“Software Formal Inspection Guidebook,”* Technical Report NASA-GB-A302, National Aeronautics and Space Administration. http://satc.gsfc.nasa.gov/fi/fipage.html, 1993

[62] Laitenberger O., “*A Survey of Software Inspection Technologies”*, Handbook on Software Engineering and Knowledge Engineering, Fraunhofer Institute for Experimental Software Engineering (IESE)

[63] Olofsson, M. and Wennberg, M., “*Statistical Usage Inspection”*, Master Thesis, Dept. of Communication Systems, Lund University, CODEN: LUTEDX (TETS-5244)/1-81/(1996), 1996.

[64] Ould M, “*Managing Software Quality and Business Risk”,* John Wiley & Sons Ltd, England,pp.: 105, 1999

[65] Porter A, Votta L, “*Comparing Detection Methods for Software Requirements Inspection: A Replication Using Professional Subjects*” Empirical Software Eng.: An Int’l J., vol. 3, no. 4, pp. 355-380, 1998.

[66] Porter AA, Votta LG, “*An experiment to assess different defect detection methods for software requirements inspections*”, Proceedings 16th International Conference on Software Engineering, Sorrento, Italy, May 1994, IEEE Computer Society Press: Los Alamitos, CA, 1994; pp. 103–112.

[67] Porter AA, Votta LG, Basili V, “*Comparing detection methods for software requirements inspection: A replicated experiment*”, IEEE Transactions on Software Engineering 1995; 21(6): pp. 563–575.

[68] R. L. Baber, “*Comparison of Electrical "Engineering*" of Heaviside's Times and Software "Engineering" of our Times,” IEEE Annals of the History of Computing, vol. 19, pp.: 5-17, 1997.

[69] Rifkin, S. and Deimel, L., “*Applying Program Comprehension Techniques to Improve Software Inspection*”, Proceedings of the 19th Annual NASA Software Eng. Laboratory Workshop. NASA, 1994

[70] Radice R., “*High quality Low Cost Software Inspections,”* issue of Methods & Tools, Summer 2002.

[71] Roper M,Wood M, Miller J, “*An empirical evaluation of defect detection techniques*”, Information and Software Technology 1997; 39(11): pp. 763–775.

[72] Royce W, “*Managing the Development of Large Software Systems*”, *Proceedings of IEEE WESCON 26 (August): 1-9*, 1970

[73] Runeson P, Wohlin C, “*Statistical Usage Testing for Software Reliability Control*”, Informatica Vol. 19 No. 2, pp: 195 – 207, 1995.

[74] Runeson P. and Regnell B., “*Derivation of an Integrated Operational Profile and Use Case Model*”, Proc. of the 9th International Symposium on Software Reliability Engineering, pp. 70-79, 1998.

[75] Runeson P., Regnell B., “*Derivation of an integrated operational profile and use case model”,* Proceedings of the 9th International Symposium on Software Reliability Engineering, pp.: 70-79, 1998

[76] Russell, G. W., *"Experience with Inspection in Ultralarge-Scale Developments"*, IEEE Software, 8(1):25-31, 1991.

[77] Saaty, T. L., "*The Analytic Hierarchy Process"*, McGraw-Hill, USA, 1980.

[78] Sauer, C., Jeffery, R., Lau, L., and Yetton, P., *"The Effectiveness of Software Development Technical Reviews: A Behaviorally Motivated Program of Research"*, IEEE Transactions on Software Engineering, vol. 26, no. 1, 2000.

[79] Seaman, C. B. and Basili, V. R., *"Communication and Organization: An Empirical Study of Discussion in Inspection Meetings"*, IEEE Transactions on Software Engineering, 24(6): pp. 559-572, 1998

[80] Shirey, G. C., *"How Inspections Fail"*, Proceedings of the 9th International Conference on Testing Computer Software, pages 151-159, 1992.

[81] Shull F, Rus I, Basili V, "*How perspective-based reading can improve requirements inspections*", IEEE Computer 2000; 33(7): pp. 73–79.

[82] Software Product Evaluation – General Guide, International Standard 9126, 1991.

[83] Sommerville, I., *Software Engineering*, Addison-Wesley, USA, 2001.

[84] Strauss, S. H. and Ebenau, R. G., *"Software Inspection Process"*, McGraw Hill Systems Design & Implementation Series, 1993.

[85] Sun Sup So, "*An Empirical evaluation of six methods to detect faults in software*", Software Testing, Verification and Reliability, Vol. 12, Issue 3, pp.: 155-172, 2002.

[86] T. A. van Dijk, W. Kintsch, "*Strategies of discourse comprehension*", Academic Press, Orlando, 1983.

[87] Testing definitions: http://www.faqs.org/faqs/software-eng/testing-faq/section-13.html

[88] Thelin T, Andersson C., Runeson P., Dzamashvili-Fogelström N.: „*A Replicated Experiment of Usage-Based and Checklist-Based Reading*", Proceeding of 10th Int. Symp. on Software Metrics, 2004.

[89] Thelin T, Runeson P, Wohlin C, "*Prioritized Use Cases as a Vehicle for Software Inspections*", IEEE Software, vol. 20, no. 4, pp.: 30 – 33, July/Aug. 2003

[90] Thelin T., Runeson, P., Regnell B.: "*Usage-Based Reading – An Experiment to Guide Reviewers with Use Cases*" Information and Software Technology, vol. 43, no. 15, pp. 925-938, 2001.

[91] Thelin T., *„Empirical Evaluations of Usage-Based Reading and Fault Content Estimation for Software Inspections"*, Department of Communication Systems, Lund University, 2002.

[92] Thelin T, Runeson P., Wohlin C., "*An Experimental Comparison of Usage-Based and Checklist-Based Reading*" IEEE transactions on software engineering, vol. 29, no. 8, August 2003

[93] Thelin T, Runeson P., Wohlin C., Olsson T., Andersson C., „*How much Information is Needed for Usage-Based Reading? – A Series of Experiments,*" Proceedings of the 2002 International Symposium on Empirical Software Engineering (ISESE'02).

[94] Thelin T, Runeson P., Wohlin C., Olsson T., Andersson C., *"Evaluation of Usage-Based Reading – Conclusions after Three Experiments"*, Empirical Software Engineering, 9 (2004), pp. 77-110.

[95] Travassos, G., Shull, F., Fredericks, M., and Basili, V.R., *"Detecting defects in object oriented designs: Using reading techniques to increase software quality,"* In the Conference on Object-oriented Programming Systems, Languages & Applications (OOPSLA), 1999.

[96] Travassos, G., Shull, F., Fredericks, M., Basili, V. R., "*Detecting Defects in Object-Oriented Designs: Using Reading Techniques to Increase Software Quality*", Proc. of the International Conference on Object-Oriented Programming Systems, Languages & Applications, 1999.

[97] Unit Testing – Junit Approach, Java Blog, http://javablog.info/2007/04/08/

[98] Unit Tests, Wikipedia*,* http://en.wikipedia.org/wiki/Unit_testing

[99] Weidenhaupt, K., Pohl, K., Jarke, M. and Haumer, P., *„Scenarios in System Development: Current Practice",* IEEE Software, 15(2): pp.: 34-45, 1998

[100] Weller, E. F., *"Lessons from Three Years of Inspection Datal,"* IEEE Software, 10(5): pp: 38-45, 1993.

[101] Wheeler DA, Brykczynski B, Meeson RN, *"Peer review process similar to inspection. Software Inspection: An Industry Best Practice",* IEEE Computer Society Press: Los Alamitos, CA, 1996.

[102] White-Box Tests, Wikipedia, http://de.wikipedia.org/wiki/White-Box-Test

[103] Whittaker J. A, Poore H. J, “*Markov Analysis of Software Specifications”*, ACM Transactions on Software Engineering Methodology, Vol 2, pp.: 93 – 106, 1993

[104] Winkler D, “*Integration of Analytical Quality Assurance Methods into Agile Software Construction Practice*”, IDoEse 2006

[105] Winkler D, Biffl S, “*An Empirical Study on Design Quality Improvement from Best-Practice Inspection and Pair Programming*”, LNCS 4034, pp.: 319 – 333, 2006.

[106] Winkler D, Biffl S, Thurnher B, “*Investigating the Impact of Active Guidance on Design Inspection”*, PROFES, LNCS 3547, 2005

[107] Winkler D, Halling M, Biffl S, “*Investigating the effect of expert ranking of use cases for design inspection*”, Euromicro Conference, Rennes, France IEEE Comp. Soc., 2004

[108] Winkler D., Biffl S., Riedl B.: „*Improvement of Design Specifications with Inspection and Testing*”, Proc. Of Euromicro 05, 2005.

[109] Wohlin C, “*Managing Software Quality through Incremental Development and Certification*”, Bulding Quality into Software, Computations Mechanics Publications, pp.: 187 – 202, 1994

[110] Wohlin C, Runeson P, “*Certification of softare components*”, IEEE Transactions on Software Engineering 20 (6), pp.: 494 – 499, 1994

Table of Figures

Figure 1-1 The Waterfall Model [43] 3
Figure 1-2 The Spiral Model [43] 4
Figure 1-3 The V-Model [3] 4
Figure 1-4 The connection between UBR and UBT [3] 6
Figure 2-1 The Technical Dimensions of Software Inspections [62] 12
Figure 2-2: Evolution of the inspection process with change and support to structure [5]. 14
Figure 3-1: Description of the PBR-Model [86] 23
Figure 3-2: Input and Output of UBR. [91] 25
Figure 3-3: Studies on UBR 30
Figure 4-1: Black-Box Testing 33
Figure 4-2: White-Box Testing 34
Figure 4-3: Unit Testing Process for the Junit Approach [97] 35
Figure 4-4: Test-First Development [4] 36
Figure 6-1: Taxi Management System – Overview [105] 46
Figure 6-2: Configuration of the Experiment 48
Figure 6-3: Experiment operation 53
Figure 6-4: Data evaluation process 54
Figure 7-1: Effectiveness, UBR vs. UBT-i [%] 60
Figure 7-2: Effectiveness, Session 1, UBR and UBT-i [%] 61
Figure 7-3: Effectiveness, Session 2, UBR and UBT-i [%] 62
Figure 7-4: Effectiveness, UBR, Session 1, Risk A+B [%] 63
Figure 7-5: Effectiveness (standard calculation), UBR, Session 1, Risk A+B [%] 65
Figure 7-6: Effectiveness, Session 1, UBT-i [%] 66
Figure 7-7: Effectiveness (standard calculation), UBT-i, Session 1, Risk A+B [%] 67
Figure 7-8: Effectiveness, Session 2, UBR [%] 68
Figure 7-9: Effectiveness (standard calculation), UBR, Session 2, Risk A+B [%] 69
Figure 7-10: Effectiveness, Session 2, UBT-i [%] 70
Figure 7-11: Effectiveness (standard calculation), UBT-i, Session 2, Risk A+B [%] 71
Figure 7-12: First defect found 72
Figure 7-13: Efficiency, UBR vs. UBT-i [%] 73
Figure 7-14: Efficiency, Session 1, UBR and UBT-i [%] 74
Figure 7-15: Efficiency, Session 2, UBR and UBT-i [%] 75
Figure 7-16: Efficiency, Session 1, UBR [%] 76
Figure 7-17: Efficiency, Session 1, UBT-i [%] 77
Figure 7-18: Efficiency, Session 2, UBR [%] 79
Figure 7-19: Efficiency, Session 2, UBT-i [%] 80
Figure 7-20: False Positives, UBR vs. UBT-i [%] 82
Figure 7-21: False Positives, Session 1, UBR and UBT-i [%] 83
Figure 7-22: False Positives, Session 2, UBR and UBT-i [%] 84
Figure 7-23: False Positives, Session 1, UBR [%] 85
Figure 7-24: False Positives, Session 1, UBT-i [%] 86
Figure 7-25: False Positives, Session 2, UBR [%] 88
Figure 7-26: False Positives, Session 2, UBT-i [%] 89
Figure 8-1: Mean Value of Effectiveness, Session 1, UBR and UBT-i 91
Figure 8-2: Effectiveness, Session 2, UBR and UBT-i 92

Figure 8-3: Mean Value of Efficiency, Session one and two, UBR and UBT-i 93
Figure 8-4: False Positives, Session 1, UBR and UBT-i 94
Figure 8-5: False Positives, Session 2, UBR and UBT-i 95

List of Tables

Table 2-1: CMM Level and Key Process Areas [40]. 10
Table 3-1: Characterization of Reading Techniques [7] 29
Table 6-1: Reference Defects in both experiment sessions 50
Table 6-2: Allocation of Seeded Defects [83] 51
Table 7-1: Defect Detection Effort (UBR) and Defect Detection Effort + Test Case generation (UBT-i) [min] 58
Table 7-2: Effectiveness, UBR vs. UBT-i [%] 60
Table 7-3: Effectiveness, Session 1, UBR [%] 64
Table 7-4: Effectiveness, Session 1, UBT-i [%] 66
Table 7-5: Effectiveness, Session 2, UBR [%] 68
Table 7-6: Effectiveness, Session 2, UBT-i [%] 70
Table 7-7: First defect found [min] 72
Table 7-8: Efficiency, UBR vs. UBT-i [%] 73
Table 7-9: Efficiency, Session 1, UBR [%] 76
Table 7-10: Efficiency, Session 1, UBT-i [%] 78
Table 7-11: Efficiency, Session 2, UBR [%] 79
Table 7-12: Efficiency, Session 2, UBT-i [%] 80
Table 7-13: False Positives, UBR vs. UBT-i [%] 82
Table 7-14: False Positives, Session 1, UBR [%] 85
Table 7-15: False Positives, Session 1, UBT-i [%] 87
Table 7-16: False Positives, Session 2 UBR [%] 88
Table 7-17: False Positives, Session 2, UBT-i [%] 90
Table 8-1: Overview of hypotheses 96

Curriculum Vitae

Persönliche Daten:

Name:	Faderl Kevin
Anschrift:	Mariahilfer Gürtel 37/14, 1150 Wien
Telefon:	0676 / 70 95 322
Geboren am/in:	12.01.1981 in Steyr

Schulische Ausbildung:

1991 – 1995	Realgymnasium Steyr
1995 – 2000	Handelsakademie Steyr
2000 – 2005	Wirtschaftsinformatik Bakkalaureatsstudium
2005 – Jetzt	Wirtschaftsinformatik Magisterstidium

Titel der Bakkalaureatsarbeit:

Zusammenführung von mehreren Eclipse Plug-Ins

Berufserfahrung:

09/2000 – 10/2001: Webeditor und Quality Manager bei IDEAL Communications, Neubaugasse 12-14, 1070 Wien.

2001 – 2004: Freelancer als Webeditor, Designer und Quality Manager bei Newton21 Austria (vormals AUnit), Porzellangasse 14/39, 1090 Wien.

2002 – 2005: Freelancer als Tonstudioassistent bei Fa. Home Music, Badgasse 19, 1090 Wien.

2003 - 2009: Freelancer als Webdeveloper bei Media 24, Scheideldorf 61, 3800 Göpfritz.

06/2005 – 12/2005: Freelancer als Webdeveloper und Quality Manager bei Fa. IT-Park, Deutschstraße 1, 2331 Vösendorf

01/2006 bis 05/2007: Selbstständige Tätigkeiten im Rahmen der Fa. NCC

11/2007 bis 04/2009: SAP New Technology Consultant bei Phoron GmbH, Guglgasse 6/3, 1110 Wien

05/2009 bis 08/2009: IT-Projektmanager bei Allianz Versicherungs AG, Hietzinger Kai 101-105, 1130 Wien

08/2009 bis Jetzt: Technischer Projektmanager bei Wyeth Whitehall Export GmbH, Storchengasse 1, 1150 Wien

Appendix

Inspection Record Document:

Inspection Record

Matrnr:	Name:	Date:

Time Log

	Clock Time hh:mm	Break Times (in minutes):	Finishing work:	
Time Log 1			Total work time (minutes) - breaktime =	
Time Log 2			Last use case:	
Time Log 3				
Time Log 4				
Totals (min):				

Issues

Risk	Total number of faults found:	Subjective estimation of the number of faults left after inspection: (do not include the ones that you have found)		
A:		Min:	Expected:	Max:
A+B:		Min:	Expected:	Max:

Explanation of the Inspection Issues

Number:	The number of the fault.
Steps:	The number of the step in the guidelines.
Clock Time	Current clock time.
Position:	Documents (D) / Sourcecode (S): class:method (i.e.: S:classA:methodX)
Risk:	The expected risk of the fault.
Type of Fault:	The expected type of the fault.

Risk Assessment

Risk	Interpretation	Interpretation when inspecting a design document
A	Crucial fault	The functions affected by these faults are *crucial* for the customer, i.e., the functions affected are important for the customer and are often used.
B	Important fault	The functions affected by these faults are *important* for the customer, i.e., the functions affected are either important and rarely used or not as important but often used.
C	Issue/no fault	An issue should be changed in the design, but is not an important or crucial fault. An issue is not counted as a fault in the exercise.

Types of Fault

Type	Interpretation	Interpretation when inspecting a design document
M	Missing	Some information is missing.
W	Wrong	The provided information is wrong.

Inspection Issues

Number	Step	Clock Time hh:mm	Position Documents (D) / Sourcecode (S): class:method (i.e.: S:classA:methodX)	Risk	Type of Fault	Descripion of Fault (Write fault description clear so anyone can read it and understand the fault)

Workflow for UBR:

Steps To Do:	**Purpose and requirements**
1. Log the time.	
2. Read through the textual requirements. Read the 5 first pages and just briefly read the others. **MAX TIME: 20 minutes.**	• Understanding. • Locate the components. • Get familiar with the structure of the document.
3. Log the clock time.	
4. Read through the design document. Read the 2 first pages, and just briefly read the others. **MAX TIME: 20 minutes.**	• Understanding. • Locate the components. • Get familiar with the structure of the document.
5. Log the clock time.	
6. Compare method descriptions and source code to find faults in the method declarations. **Do not yet read the code inside the methods.**	• Detect faults in the method declarations or source code.
7. Start reading the first use case. 8. Follow the required methods for this use case (see method descriptions and sequence diagrams). 9. When reaching a method that has not been checked before, work through the source code, otherwise skip it. 10. Try to detect faults in the method descriptions and the source code while following the use cases and log them.	• The use cases have to be utilized in order. • Detect faults in the method descriptions and the source code. • It is acceptable to return to a use case that you have already worked on.
11. Log the clock time	•
12. When finished inspecting: • Log the last use case used. • **Estimate the number faults left (minimum, most probable, and maximum).** • Answer the feedback questionnaire. • Fill out the individual estimation. • Hand in all material used.	• You are finished when you have worked on each use case or time is up.

Workflow for UBT-i:

Steps To Do:	**Purpose and requirements**
13. Log the time.	
14. Read through the textual requirements. Read the first pages and just briefly read the others. **MAX TIME: 20 minutes.**	• Understanding. • Locate the components. • Get familiar with the structure of the document.
15. Log the time.	
16. Read through the design document. Read the first pages, and just briefly read the others. **MAX TIME: 20 minutes.**	• Understanding. • Locate the components. • Get familiar with the structure of the document.
17. Log the time.	
18. Compare method descriptions and source code to find faults in the method heads. **Do not yet read the code inside the methods.** 19. For each method at the system's border: Find equivalent classes for method parameters and write them next to the method declaration.	• Detect faults in the method declarations or source code. • Find the equivalent classes for each method.
20. Start reading the first use case. 21. Follow the required methods for this use case (see method descriptions and sequence diagrams). 22. When reaching a method that has already been checked, skip it. 23. When reaching a method that has not been checked before, work through its source code: • When the method is at the border of the system (the method is supposed to check passed parameters), create test cases with found equivalent classes. • For ALL methods (also those at the system's border): create test cases for each fork (if/else) using condition chains (e.g.: C1T-C2F). Be sure to check each fork of the code tree. 24. Try to detect faults in the method descriptions and the source code while following the use cases and log them.	• The use cases have to be utilized in order. • Detect faults in the designdocument and the source code. • It is acceptable to return to a use case that you have already worked on. • Create testcases. • Create only testcases that are necessary to cover all equivalent classes. • The use cases have to be utilized in order.
25. Log the time.	•
26. When finished inspecting: • Log the last use case used. • **Estimate the number faults left (minimum, most probable, and maximum).** • Answer the feedback questionnaire. • Fill out the individual estimation. • Hand in all material used.	• You are finished when you have worked on each use case or time is up.