PhD Theses in Experimental Software Engineering

Vol. 40

Frank Elberzhager

A Systematic Integration of Inspection and Testing Processes for Focusing Testing Activities

FRAUNHOFER VERLAG

PhD Theses in Experimental Software Engineering

Volume 40

Contact:
Fraunhofer-Institut für Experimentelles Software Engineering (IESE)
Fraunhofer-Platz 1
67663 Kaiserslautern
Telefon +49 631 6800 - 0
Fax +49 631 6800 - 1199
E-Mail info@iese.fraunhofer.de
www.iese.fraunhofer.de

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliografic data is available in the Internet at <http://dnb.d-nb.de>.
ISBN: 978-3-8396-0445-8

D 386

Zugl.: Kaiserslautern, Univ., Diss., 2012

Printing and Bindery:
Mediendienstleistungen des
Fraunhofer-Informationszentrum Raum und Bau IRB, Stuttgart

Printed on acid-free and chlorine-free bleached paper.

Fraunhofer Information-Centre for Regional Planning and Building Construction IRB
P.O. Box 80 04 69, D-70504 Stuttgart
Nobelstrasse 12, D-70569 Stuttgart
Phone +49 (0) 711 970-2500
Fax +49 (0) 711 970-2508
E-Mail verlag@fraunhofer.de
URL http://verlag.fraunhofer.de

A Systematic Integration of Inspection and Testing Processes for Focusing Testing Activities

Vom Fachbereich Informatik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation
von

Dipl.-Inform. Frank Elberzhager

Fraunhofer Institut für Experimentelles Software Engineering
(Fraunhofer IESE)
Kaiserslautern

Berichterstatter:	Prof. Dr. Dr. h.c. H. Dieter Rombach
	Prof. Dr. Jürgen Münch
	Prof. Dr. Andreas Zeller
Dekan:	Prof. Dr. Arnd Poetzsch-Heffter
Tag der wissenschaftlichen Aussprache:	15. Juni 2012

D 386

"It is the struggle itself that is most important.
We must strive to be more than we are.
It does not matter that we will not reach our ultimate goal.
The effort itself yields its own reward."
Gene Roddenberry

Acknowledgments

Many different people have accompanied and supported me in realizing this thesis.

In the first place, I would like to thank my first supervisor, Prof. Dr. Dr. h.c. H. Dieter Rombach, director of the Fraunhofer Institute for Experimental Software Engineering (IESE) and chair of the Department of Computer Science at the University of Kaiserslautern, Germany. I would like to thank him for his support, the fruitful discussions, and his feedback.

Second, I would like to thank my second supervisor, Prof. Dr. Jürgen Münch, chair at the Department of Computer Science at the University of Helsinki, Finland, and former division head at Fraunhofer IESE. He supported me in finding the right direction, respectively the scope, for this thesis. Furthermore, I could reflect the topic during many discussions with him. In addition, he taught me to stick to the point, and to concentrate on clear and sound presentation and writing. He heavily supported me during a lot of publications. Finally, though I know that he is very busy, he always had time for me when I needed support and motivated me. I would not have completed this thesis without his support.

Third, I would like to thank Prof. Dr. Andreas Zeller, Software Engineering chair at Saarland University, who accepted being my third supervisor at short notice.

At Fraunhofer IESE, many people supported me in many different ways. I would like to thank them all for their help and their continuous support. In particular, I would like to thank my colleagues from the quality assurance departments who discussed the topic with me or supported me during evaluations. I would especially like to thank Thomas Bauer, who started with me in the former Testing and Inspections department about six years ago, who motivated me, and who became a real friend during this time. I would like to thank Alla Rosbach, who supported me during literature surveys and in many inspection projects, and with whom I could always discuss the topic. Furthermore, I would like to thank Sonnhild Namingha for reviewing this thesis and correcting my English.

I am also grateful to the students who supported me, especially Vi Tran, who conducted an extensive literature survey and discussed the state of

the art with me, and Stephan Kremer, who supported me in tool development.

Last but not least, I would like to thank my parents and my wife Irene. My parents gave me the opportunity to study computer science and supported me on that way. Irene, you believed in me, motivated me, and reminded me that there is a life beyond computer science. Thank you very much.

Abstract

Quality assurance effort is often a major cost factor during software development. Especially testing effort can consume more than 50 percent of the overall development effort. Furthermore, it is often unclear how efficient existing quality assurance techniques are and what the potential for savings might be.

Currently, companies often conduct different quality assurance activities, such as inspections and testing, in order to find as many defects as possible before software products are delivered. However, most often such techniques are applied in isolation and do not exploit synergy effects from systematically combining them, such as reduced effort or higher defect detection rates. Moreover, the relations between different static and dynamic quality assurance techniques are widely unclear. In addition, testing activities often have a broad scope and are rarely applied in a focused manner, which results in high costs.

This thesis presents the In^{2}Test approach, which systematically combines inspection and testing processes for focusing testing activities. The main ideas of this integrated approach are (a) to use early inspection results to prioritize testing on parts of a product or on defect types that are expected to be most defect-prone, (b) to consider product metrics and historical data in order to further improve the test focus, (c) to guide the prioritization of system parts and defect types by using rules that are based on explicitly defined assumptions about the relationships between inspection results and remaining defects.

The approach was validated in two case studies: The validation was aimed at (a) showing that In^{2}Test allows for effort reduction during test execution while keeping a comparable level of detected defects during testing, i.e., In^{2}Test allows for improved efficiency compared to non-integrated approaches; (b) revealing underlying assumptions for the prioritization of system parts and defect types for testing based on inspection results; and (c) showing that In^{2}Test is mature for use in industrial applications.

The validation (a) showed an effort reduction in the case studies of between 6% and 34% at a comparable level of detected defects depending on the concrete assumptions and selection rules applied, leading to an efficiency improvement of between 7% and 52%; (b) revealed a set of initial selection rules and assumptions with promising results; (c) showed the applicability of the approach.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Overview

The objective of this chapter is to provide a brief overview of the topic of quality assurance with respect to the problems to be addressed by this thesis. Section 1.2 starts with the general motivation and emphasizes, on the one hand, the need for quality assurance during software development in order to avoid negative consequences, and, on the other hand, the costs of today's quality assurance. Section 1.3 sketches the research scope and the research approach applied in this thesis. Section 1.4 presents the problem statements. Section 1.5 summarizes the goals and hypotheses of this thesis, and shows the relationships between the problems, the goals, and the hypotheses. Section 1.6 gives an overview of the relevance of quality assurance, summarizes software inspections and software testing, and defines some basic terminology. Section 1.7 mentions the research contributions. Finally, Section 1.8 presents the structure of this thesis.

1.2 Motivation

Software and software-intensive systems are part of everyone's life and can be found all around us. Moreover, the size and complexity of such systems are continuously growing. Charette (Charette, 2005), for instance, stated that a typical cellphone in 2005 contained about 2 million lines of code; he expected such phones may contain ten times as many nowadays. Another example are today's top-of-the-range cars with an estimated 100 million lines of code. Consequently, developing high-quality software is becoming ever more challenging and more expensive.

Jackson et al. (Jackson et al., 2007) stated that due to "the growth in complexity and invasiveness of software systems, the risk of a major catastrophe in which software failure plays a part is increasing." Boehm and Basili (Boehm and Basili, 2001) mentioned that between 40 and 50 percent of all delivered software contain non-trivial defects. Hayes (Hayes, 2002) reported from a survey asking 800 business-technology managers about experiences with software defects that 97% of the respondents reported problems due to software defects in the past year, "and nine out of 10 reported higher costs, lost revenue, or both as a result." Humphrey (Humphrey, 2008) confirmed that "today's large-scale systems typically have many defects".

It is an undeniable fact that software and software-intensive systems often contain defects when delivered that may lead to dramatic consequences. For example, a study conducted by the National Institute of Standards and Technology in 2002 showed that software defects cost the U.S. economy about 59.5 billion dollars per year. Furthermore, about 37 percent of these costs could be avoided if quality assurance activities were to be improved (Tassey, 2002). Jones (Jones, 2006) mentioned that one reason for projects that exceed schedules, costs, and time is insufficient quality assurance. Besides economic consequences, a loss of reputation for a company and danger for human beings are further consequences that can result from defect-prone software. Jackson et al. (Jackson et al., 2007) mentioned a lot of accidents and near-accidents from different domains (e.g., aviation, medical devices, infrastructure, defense) caused by software or where defect-prone software was involved. Another source that has been listing software defects and their consequences continuously since 1985 is the Risks Digest (Risks Digest, 2012).

Based on the stated observations, one goal is often to find as many defects as possible in a cost-effective manner before a software product is delivered, which includes considering context factors such as available resources, time, or costs. Consequently, adequate quality assurance activities should be selected and applied in order to reduce the number of defects and thus reduce the impact of failures caused by undiscovered defects within a software product.

A tremendous number of different analytical quality assurance approaches, methods, and techniques have been developed, evaluated, and adapted during the past decades, such as various inspection and testing techniques (Aurum et al., 2002; Burnstein, 2002; Wiegers, 2002; Juristo et al., 2004, 2006; Liggesmeyer, 2009). However, while, on the one hand, costs can dramatically increase if certain defects (especially critical ones) are not found, conducting quality assurance, on the other hand, can also be a major cost driver during software development. This is especially true for testing activities.

Myers (Myers, 1979) already stated that testing can consume approximately 50% of the development time and more than 50% of the overall development costs. Beizer (Beizer, 1990) mentioned that such costs range between 30% and 90%, depending on the concrete method used. Jones (Jones, 1991) estimated the costs for testing activities as being 30 to 40% of the development costs. Moreover, Jones showed that the relative effort for analytical quality assurance increases when the overall development effort of a project increases, i.e., small projects need only about 16% effort for quality assurance and about 70% effort for coding, compared to big projects where the quality assurance effort is about 37% and coding effort is only 12%. Harrold (Harrold, 2000) confirmed effort and cost figures for testing of up to

50% of the total development effort, respectively costs. Hailpern and Santhanam (Hailpern and Santhanam, 2002) stated that the costs for quality assurance activities in typical development organizations range from 50 to 75% of the overall development costs. Juristo et al. (Juristo et al, 2006) stated that testing can exceed half of the overall effort of a project budget. Finally, Liggesmeyer (Liggesmeyer, 2009) concluded that quality assurance activities often consume most of the overall development effort, which Pressman (Pressman, 2009) again calculates as up to 50% of the total development effort. Thus, quality assurance effort, and especially testing effort, has remained high during the past decades.

In order to achieve the desired time, cost, and quality goals, the development approach, including the quality assurance activities, has to be optimized. One reason for insufficient testing, which is one of the essential quality assurance activities today, are inappropriate testing strategies (Kasurinen et al., 2009). Furthermore, Bertolino and Marchetti (Bertolino and Marchetti, 2004) mentioned that test practice currently is performed in a trial-and-error fashion, i.e., systematic and cost-effective strategies are often missing.

Boehm and Basili (Boehm and Basili, 2001) stated that "current software projects spend about 40 to 50 percent of their effort on avoidable rework". One reason for this is that defects have to be corrected that could have been found and corrected during quality assurance activities before distribution. Consequently, mechanisms that support focusing quality assurance activities on defect-prone parts and thus, for example, prioritize parts of a system that are expected to be defect-prone, may decrease rework effort. Humphrey (Humphrey, 2008) stated that due to the growing complexity of today's software and software systems, it is impossible to test all parts and all ways in which such systems can be used. Focusing quality assurance activities is also substantiated by the empirically valid observation of a Pareto distribution for defects that can frequently be observed. For example, Boehm and Basili (Boehm and Basili, 2001) stated that 80% of defects occur in about 20% of the modules, i.e., defects are often not distributed equally.

In conclusion, defects are an inevitable fact of today's software and software systems that software development has to cope with in order to avoid negative consequences such as economic losses, decreased reputation, or risk for human beings. Thus, "software quality is an issue that should concern everyone" (Humphrey, 2008). A variety of different quality assurance activities can address this challenge. However, the costs, respectively the effort for conducting quality assurance activities, are often too high and can consume more than 50% of the total development effort. Some reasons for inadequately high costs are inappropriate quality assurance strategies, high rework effort, and insufficient focusing of quality assurance activities. One important

strategy for addressing these shortcomings is the systematic optimization and integration of quality assurance, which is addressed in this thesis.

1.3 Research Scope

The research scope of this thesis can be summarized according to the following criteria:

- *Domain of software and software systems development:* The results presented here are developed in the area of software engineering. Approaches from other domains such as mechanical productions or chemical industry are not considered, which applies especially for quality assurance processes from these domains, which are not considered. No specific software development process is required to apply the developed approach. However, it is assumed that certain artifacts (e.g., requirements, design, or code) are developed that have to run through certain quality assurance activities until the final software or software system is developed. Furthermore, empirical evidence from the software engineering domain is used to control quality assurance activities.

- *Constructive and analytical quality assurance:* Two directions can be distinguished when performing quality assurance, namely constructive and analytical quality assurance. While constructive quality assurance focuses on preventing defects during the development of a system, analytical quality assurance focuses on finding defects in certain artifacts and in the overall system. This thesis focuses on analytical quality assurance.

- *Quality assurance activities and techniques:* The approach builds explicitly on the application, respectively combination, of two quality assurance activities: software inspections and software testing. However, no specific inspection technique and no specific testing technique is required. Regarding software inspections, various techniques can be applied, e.g., formal inspections, reviews, or peer deskchecks. Regarding software testing, different techniques (e.g., equivalence partitioning, random testing) and different test levels (e.g., unit test, system test) can be addressed. The main reason for not requiring any particular inspection and testing technique is that the approach should be applicable in different environments independent of any concrete quality assurance technique. Indeed, a prerequisite for applying the approach is that a suitable number of defects are found by the applied quality assurance techniques. However, during the evaluations, a Fagan-like inspection process with focused checklist was applied. During unit testing, equivalence

partitioning together with boundary-value analysis was done. For system testing, test cases were derived from tool requirements.

In order to evaluate the performance of quality assurance activities, effectiveness and efficiency are often considered. Effectiveness is the number of defects found; efficiency is the number of defects found per time unit. Three improvement scenarios, respectively goals, are conceivable, which are shown in Figure 1. Consider the initial situation at the top where the effectiveness and the needed effort (time) are shown together with concrete exemplary values and the calculated efficiency. The first improvement scenario aims at an improved effort value and depicts a fixed effectiveness value, i.e., the same number of defects is found in less time. The second scenario illustrates a fixed effort value but a higher number of defects found, i.e., more defects are found in the same time. The third improvement scenario is an improvement of the number of defects found and less time consumed. This thesis, respectively the presented integrated approach, mainly focuses on the first improvement scenario. All three scenarios show an improvement in efficiency, with the third scenario showing the highest value.

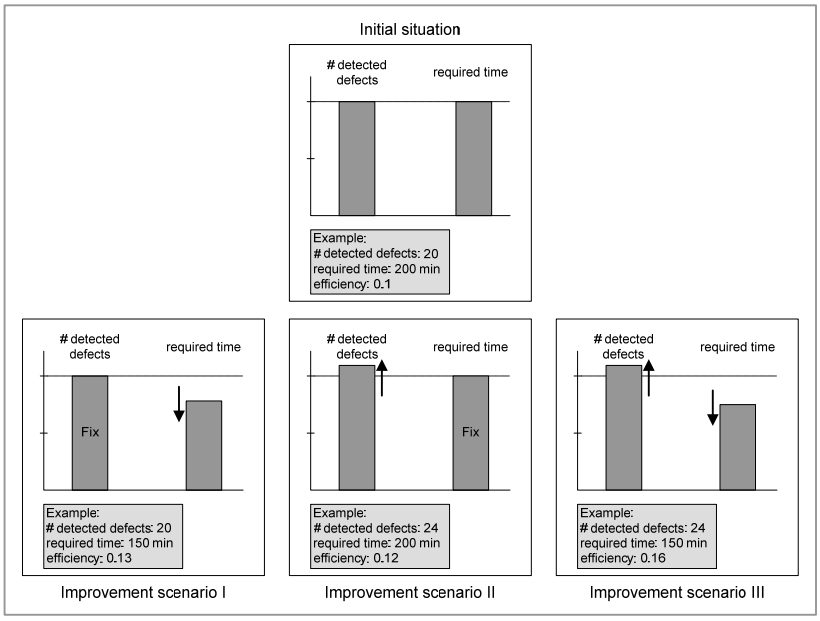


Figure 1 Improvement scenarios.

An overview of the research approach that was pursued in this thesis is shown in Figure 2.

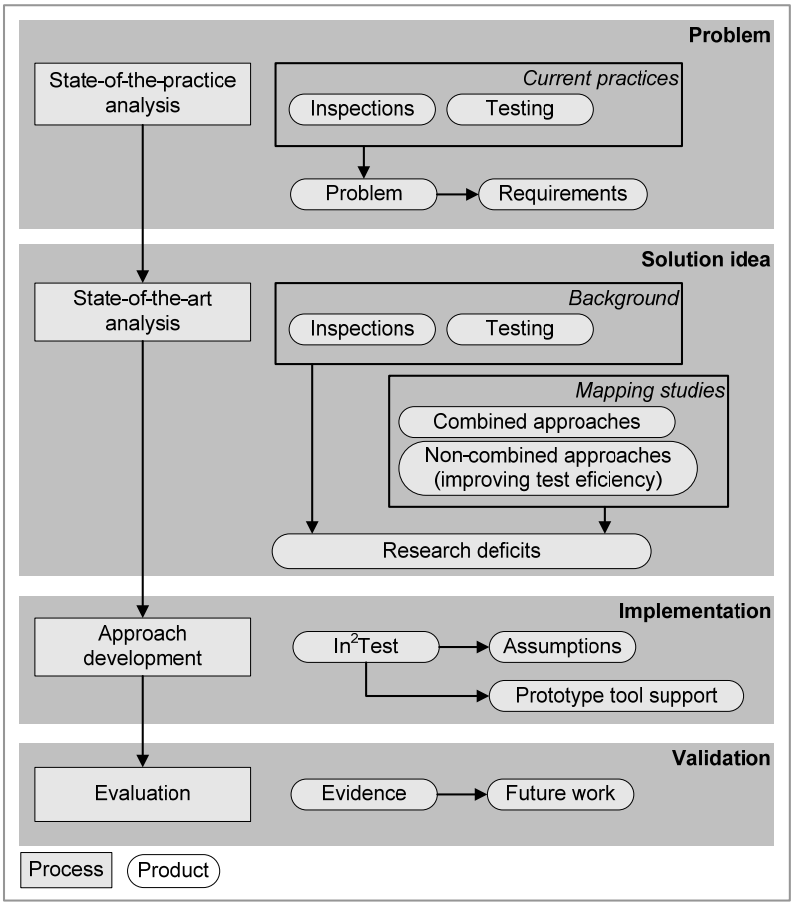


Figure 2 Research approach.

- *State-of-the-practice analysis:* A literature survey was performed with respect to current inspection and testing practices. The focus was put on approaches and techniques applied in practice and on the experiences made with them. Furthermore, problems were identified, and requirements that have to be fulfilled by a new approach in order to overcome the mentioned problems were derived.

- *State-of-the-art analysis:* First, background information regarding software inspections and software testing was gathered through a state-of-the-art survey, as these two quality assurance activities are used in the developed approach. Furthermore, two systematic mapping studies were performed in order to (i) identify existing approaches that already combine static and dynamic quality assurance techniques, and (ii) find further approaches (i.e., non-combined ones) that are able to

improve testing efficiency. Based on these results, the previously derived requirements were used to assess the found approaches, and research deficits were identified.

- *Approach development:* Based on the identified problems, requirements, and research deficits, the **in**tegrated **in**spection and **test**ing approach In²Test was developed, which combines inspection and testing activities in order to improve testing efficiency. Besides a general process, different levels of granularity are implemented, resulting in a one- and a two-staged approach. An initial set of assumptions describing the relationships between inspections and testing and supporting the In²Test approach were stated. Methods for deriving, evaluating, and applying such assumptions are provided. Finally, prototype tool support was developed.

- *Evaluation:* The In²Test approach was evaluated during two case studies which were conducted at Fraunhofer IESE with respect to quality assurance activities regarding the development of two tools.

1.4 Problem Statements

In general, practitioners are well aware of the need to perform quality assurance during software and software system development. However, as already sketched in the motivation, two main problems exist that are addressed within this thesis.

Problem 1: Testing activities often do not use results and insights from early defect detection activities, especially inspection techniques, in order to focus testing: Inspection and testing are two of the most common quality assurance activities performed today during software development. However, if both quality assurance techniques are applied, inspection and testing are usually performed in sequence, without any exchange of data between them to exploit synergy effects. Consequently, testing activities are often not focused based on early defect data. This leads to so-called local inefficiencies, i.e., test-specific effort is wasted. Existing approaches for reducing testing effort are widely based on the use of metrics, risk, or historical data in order to predict fault-prone parts of a product (and thus, to test only these parts) or determine test exit criteria (and thus, to know better when to stop testing). However, they do not make systematic use of the results from inspections, i.e., quantitative defect data from the current software under development is usually not used to control testing processes. Some approaches consider the combination of inspection and testing techniques in a pragmatic and unsystematic manner (Kinochita, 2010) in order to be more effective or to predict the expected number of defects

for testing based on inspection results (Harding, 1998). However, although inspection and testing techniques are sometimes integrated in an informal way in industry, a systematic approach is missing, which would integrate them in order to exploit synergies and to allow controlling testing activities based on inspection defect data. This also means that test strategies are usually not defined or adapted systematically based on early available defect data of a current software development cycle.

Problem 2: Quality assurance activities, especially testing activities, require too much effort. As already stated in the motivation (Section 1.2), quality assurance activities often consume much effort, resulting in high costs. Especially testing may require more than 50 percent of the overall development effort (Harrold, 2000; Hailpern and Santhanam, 2002; Pressman, 2009). Reasons for this include, for example, unsystematic test processes, inappropriate test strategies, or failing to use synergy effects from the systematic combination of different quality assurance activities. Furthermore, the growing size and complexity of software and software systems make it hard to decide which parts of software should be tested with what intensity. Often, no or only poor focusing of testing is conducted, which also leads to high costs for testing, but also for overall quality assurance. As defects are an inevitable fact of today's software and software systems, omitting quality assurance activities often is no option during development in order to save effort. Instead, new strategies and approaches are necessary to reduce quality assurance and testing effort. One important strategy here is an approach for the systematic optimization and integration of quality assurance. However, inspection and testing are usually conducted independent of each other, i.e., they are sometimes applied in sequence in order to find additional defects (Franz and Shih, 1994; Berling and Thelin, 2003) or, based on empirical evidence, a combination is suggested (Runeson et al., 2006), but they do not collaborate in an optimal manner. Bertolino (Bertolino, 2007) concludes that there exist "many fruitful relations between software testing and other research areas", and that many of them were overlooked in the past, which includes the integration of inspection and testing in order to reduce testing effort and, consequently, overall quality assurance effort.

1.5 Goals and Hypotheses

The In^{2}Test approach combines (i.e., integrates) inspection and testing activities in order to focus testing activities. By using inspection defect data and so-called assumptions, a mechanism is provided that allows for prioritizing parts of a system under test that are expected to be defect-prone, or defect types that are expected to appear during testing. The main goals of the In^{2}Test approach are stated as follows:

- *G1 (Effort): Provide an integrated approach that reduces the effort for conducting quality assurance in general, and for testing activities in particular.* An integrated approach should be able to reduce the time, respectively effort, needed for conducting testing activities compared to a non-integrated approach, which may also lead to a reduction of the overall quality assurance effort. Effort reduction should be achieved by a mechanism for focusing testing activities, i.e., prioritizing and selecting parts of a system, respectively defect types, that appear to be relevant for testing activities.

- *G2 (Effectiveness): Provide an integrated approach that is able to find a comparable number of defects compared to non-integrated approaches.* An integrated approach should be able to detect a comparable number of defects when conducting testing activities compared to a non-integrated approach. Two possibilities for focusing testing activities are aimed at. On the one hand, defects can be detected in certain parts of a system under test (i.e., not all parts are focused on for testing). On the other hand, defects of certain defects types can be detected (i.e., not all defect types are focused on for testing).

- *G3 (Evidence): Provide a methodology that allows focusing testing activities based on empirical evidence with the integrated approach.* The integrated approach is applied in order to focus testing activities. In order to be able to prioritize parts of a system or defect types, relationships between inspection and testing have to be known. If such relationships are unknown, assumptions need to be defined that allow focusing testing based on inspection results. Such assumptions should be evaluated with respect to their validity in the given context.

- *G4 (Applicability): Provide an integrated approach that is applicable in industrial contexts.* An integrated approach should be easy to understand and apply in an industrial context, i.e., a light-weight approach is preferred that does not need complete process changes for development or quality assurance activities in a given context. Furthermore, the results of the integrated approach should make sense.

The combination of goals one and two aims at an improved efficiency, while goals three and four aim at showing the feasibility and applicability of the approach. In order to be able to evaluate the goals stated above, the following hypotheses are defined:

- *H1: The effort for applying the integrated inspection and testing approach is at least 20% less compared to applying non-*

integrated inspection and testing processes, with the level of quality of the product under test that can be achieved being at least equal. The In^{2}Test approach will reduce the effort for conducting inspection and testing activities, with the focus being on reducing testing effort by 20%, respectively improving efficiency, and inspection effort remaining constant. Several parts for the evaluation are covered in H1, i.e., efficiency improvement, effectiveness numbers, respectively their improvement, and gaining evidence of the validity of the underlying basis for focusing testing activities. Therefore, a refinement of H1 into several sub-hypotheses can be found in Chapter 5.

- *H2: The integrated inspection and testing approach is applicable*. The In^{2}Test approach will be applicable. This comprises easy understandability, easy applicability, high usefulness, and easy adaptability.

Figure 3 shows the relationships between problems, goals, and hypotheses. Problem one affects all four goals, because all these goals express objectives regarding an integrated approach that uses inspection results. Problem two affects one goal that explicitly aims at reducing effort. Finally, goals one to three are covered in hypothesis one, whereas goal four is covered in hypothesis two.

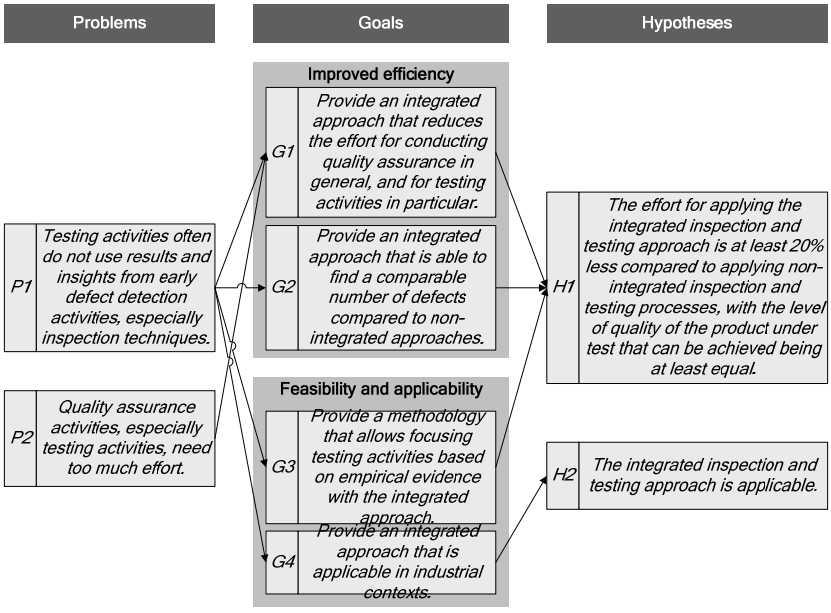


Figure 3 Problems, goals, and hypotheses, and their relationships.

1.6 Background on Software Quality Assurance

This section provides background information on software quality assurance and how it is related to software development. The main quality assurance activities and techniques (i.e., software inspections and testing) focused on in this thesis are emphasized and their importance is clarified. Furthermore, the basic terminology of the terms used most often is stated in order to provide a common understanding.

1.6.1 The Relevance of Quality Assurance

Ever since software and software systems have been developed, quality assurance has been a part of development processes. Different development methodologies originated during the past four decades, for example, the waterfall model (Royce, 1970), the first V-model (Boehm, 1979), or agile models such as extreme programming (Beck, 2000). Depending on the respective development methodology, quality assurance activities are adapted accordingly. Some decades ago, software development paradigms such as "software cannot fail" existed. Such early assumptions were reasonable since software neither ages nor wears out, but they were eventually rejected. One prominent example that built upon that assumption and led to serious consequences was the Therac-25 accident (Leveson and Turner, 1993). However, today, it is well known and widely accepted that quality assurance is a crucial part during software development.

Regarding quality assurance activities, constructive and analytic activities can be distinguished. The former strive to provide systematic techniques and methods that prevent the introduction of defects, for example, by providing patterns, design principles, or coding guidelines. The latter primarily aim at detecting and removing existing defects. Analytic quality assurance activities are also called verification and validation activities, which is defined by the IEEE Standard Glossary of Software Engineering Terminology as "the process of determining whether the requirements for a system or component are complete and correct, the products of each development phase fulfill the requirements or conditions imposed by the previous phase, and the final system or component complies with specified requirements" (IEEE Standard 610.12, 1990). However, the terms "verification" and "validation" are not used consistently in the existing literature and in practice, for example, verification and validation are sometimes considered only as testing activities. Another common understanding is that verification comprises every quality assurance activity before acceptance testing.

Consequently, the terms static quality assurance and dynamic quality assurance are preferred in this thesis. Static quality assurance techniques (e.g., inspections, reviews, walkthroughs, or static analyses such as

program slicing) do not need executable models or executable code, but rather examine artifacts such as requirements documents, design models, or code without running them. In contrast, dynamic quality assurance techniques (e.g., equivalence partitioning, boundary value analysis, control-flow based testing techniques, or dynamic analyses such as program profiling) need executable program parts. Finally, formal quality assurance (which is not in the focus of this thesis) is a third group of analytic quality assurance, which consists of techniques such as formal proofs or symbolic execution.

Today, a large number of well-established static and dynamic quality assurance techniques exist, such as various inspection and testing techniques (Gilb and Graham, 1993; Wiegers, 2002; Burnstein, 2002). In the past, a lot of research has been performed to develop and improve a variety of static and dynamic quality assurance techniques. Juristo et al. (Juristo et al., 2004) examined 25 years of empirical studies with respect to a large number of different testing techniques, classified them, and summarized the main findings. They conclude that the current testing knowledge is very limited. With respect to software inspections, Aurum et al. (Aurum et al., 2002) examined software inspection processes published during the 25 years since inspection as a quality assurance technique was first published by Fagan in 1976 (Fagan, 1976). They identified different inspection processes and support for the inspection, such as reading techniques, tools, and support for deciding whether or not to perform a re-inspection. In conclusion, Aurum et al. (Aurum et al., 2002) stated that the identified studies contribute to the evolution of software inspections, but many research questions remain open. Another examination of software inspection research, covering the period between 1991 and 2005, was performed by Kollanus and Koskinen (Kollanus and Koskinen, 2007). They classified the identified articles into a technical view (e.g., reading techniques, effectiveness factors), a management view (e.g., inspection impact on development process), and other topics (e.g., defect estimation, inspection tools). The two authors concluded that much research has been performed with respect to software inspections, but that empirical knowledge remained low.

One fundamental observation with respect to research on inspection and testing techniques, which are two of the best-established static, respectively dynamic, quality assurance techniques, is that most often, this research is done to improve inspections or testing. In contrast, some studies compare different inspection and testing techniques (Basili and Selby, 1987; Runeson et al., 2006), which often resulted in the conclusion to apply them in combination (Laitenberger, 1998; Endres and Rombach, 2003). Other studies calculated the effectiveness values when applying them in combination to demonstrate the benefit of a joint application (Myers, 1978; Wood et al., 1997; Gack, 2010).

However, except for suggestions to apply both, no concrete process or additional advice is usually provided.

Combining different static and dynamic quality assurance techniques, such as inspections and testing, is a promising way to improve quality assurance and to cope with problems such as high quality assurance costs. Endres and Rombach (Endres and Rombach, 2003) stated that "a combination of different V&V methods outperforms any single method alone". The main rationale for this is that different methods have different strengths and therefore, a combination of quality assurance techniques leads to better results than applying only a single technique. However, most often, inspection and testing are applied in sequence without exploiting additional synergy effects.

The connections between inspection and testing activities seem intuitively clear and obvious, but in practice this is often lost or obscured. The result is poorly prioritized and often redundant quality assurance effort. It is perfectly possible that a strategy combining inspections and testing could have been used in practice already, because the underlying reasoning is grounded on well-known software engineering practices. However, even in this case, it is questionable whether existing approaches rely on explicit, well-grounded and evaluated approaches instead of common sense and unsystematic procedures.

1.6.2 Software Inspections

Next, basic ideas and concepts regarding software inspections will be presented. This comprises the inspection process and a set of variations, inspection reading support, and other research directions. With this, an overview regarding software inspection is given in order to allow the reader to understand the inspection concepts used in this thesis.

A software inspection is a static quality assurance method; it was first published by Michael Fagan in 1976 (Fagan, 1976). The IEEE Standard 1028-1997 (IEEE Standard 1028, 1997) defines an inspection as "a visual examination of a software product to detect and identify software anomalies, including errors and deviations from standards and specifications." The IEEE Standard 610.12-1990 (IEEE Standard 610.12, 1990) defines an inspection as "a static analysis technique that relies on visual examination of development products to detect errors, violations of development standards, and other problems." Three main characteristics of an inspection can be derived from that: First, the main goal of an inspection is to find defects in a software artifact. Second, different kinds of development artifacts, such as requirements, design, or code documents, can be checked during an inspection. Third, an inspection is done by manually checking an artifact and thus, no execution of, for instance, code is necessary to perform an inspection.

Inspection Process

Fagan proposes an inspection process consisting of six main steps in order to perform defect detection (Fagan, 1976, 1986):

- Planning: Organizing the entire inspection process, including selecting participants, determining meeting times and places, and preparing the material.
- Overview: Giving an overview of the artifact to be inspected.
- Preparation: Each inspector has to get familiar with the artifact.
- Inspection: A group session of all inspectors in order to find defects in the corresponding artifact.
- Rework: The author has to correct all found and documented defects.
- Follow-up: Checking the corrected artifact for newly introduced defects during the rework step.

Based on the process defined by Fagan, different process changes and adaptations to the inspection process have been proposed. Parnas and Weiss (Parnas and Weiss, 1985) proposed active design reviews. The main difference is that several brief inspection cycles should be performed instead of one large inspection in order not to overload the participating inspectors and to improve the effectiveness of inspections. A two-person review was defined by Bisant and Lyle (Bisant and Lyle, 1989), where an inspector and an author perform the inspection. Furthermore, Martin and Tsai (Martin and Tsai, 1990) proposed N-fold inspections, where n independent teams conduct the inspection of an artifact. Different evaluations conducted by the authors of the approach showed valuable results.

Several aspects of the inspection processes already shown were used by Knight and Myers (Knight and Myers, 1993) to define phased inspections. Moreover, inspections without a meeting (Votta, 1993; Votta et al., 1995; Johnson and Tjahjono, 1998) or the inspection process given by Gilb (Gilb and Graham, 1993) are further defined and evaluated inspection processes. A summary of these inspection processes was given by Aurum et al. (Aurum et al., 2002).

Laitenberger and DeBaud (Laitenberger and DeBaud, 2000) defined five dimensions of software inspections (technical, managerial, organizational, assessment, tool). With respect to the technical dimension, they propose a process consisting of six inspection steps,

namely planning, overview, defect detection, defect collection, defect correction, and follow-up. One main difference to the Fagan process is that a recommendation for individual defect detection is given rather than for a group session to find defects due to improved effectiveness. However, looking for defects in a team meeting could result in additional defects being found. Besides the process, the technical dimension covers different inspection roles (e.g., organizer, inspector, moderator, author). While the Fagan inspection mainly focuses on code, Laitenberger and DeBaud mentioned different products that can be inspected, such as requirements, design, code, or test cases.

Wiegers (Wiegers, 2002) proposed a classification of inspection processes with respect to the level of formality. The spectrum ranges from least formal processes to most formal processes. Figure 4 shows the inspection processes that are distinguished. The least formal technique is an ad-hoc review in which basically a person discusses with another person if a concrete problem occurs or advice is needed. In a peer deskcheck, some material (e.g., code) is sent to another person, who should read it and comment appropriately. Following a passaround instead, the material to be inspected is sent to more than one person. During a walkthrough, the author of a document presents the complete document or certain parts of it to some people, and discusses the content, the solutions, and the defects. A team review is similar to an inspection process, but less formal, e.g. reading support is not mandatory and metrics do not need to be documented. Finally, an inspection is the most formal process, similar to Fagan inspections. Wiegers mentions five steps: planning, preparation, meeting, correction, and verification of results.

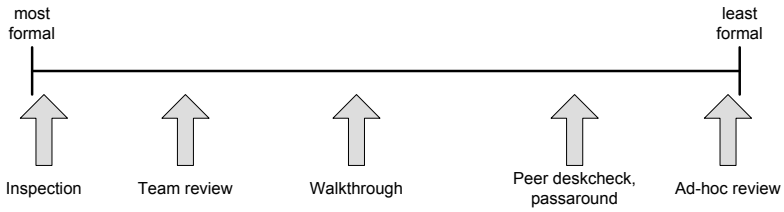


Figure 4 Inspection techniques with respect to different levels of formality.

In conclusion, many different inspection processes have been developed during the past 35 years, ranging from very formal processes to ad-hoc processes. Denger (Denger, 2009) summarized 21 static quality assurance techniques, classifying them as inspection-like, walkthrough-like, and desk-checking-like inspection processes. The terms used, such as inspections, reviews, or walkthroughs, are sometimes used as synonyms and difficult to distinguish in practice, which is also mentioned by Aurum et al. (Aurum et al., 2002). One solution is to define a generic inspection process from which different inspection processes can be

derived based on the context and needs (Denger and Elberzhager, 2007). Another procedure is to carefully select one existing inspection process that is most suitable in a given environment, and adapt it to the extent necessary.

In the following, the term inspection is used as a top-level term covering the mentioned inspection techniques (reviews, walkthroughs, etc.). The technique Wiegers (Wiegers, 2002) classified as inspection is called a formal inspection in order to avoid misunderstandings.

A lot of empirical evidence exists with respect to inspections. Kollanus and Koskinen (Kollanus and Koskinen, 2007) mention that "there are many empirical studies on the effects of inspections". Basili and Boehm (Boehm and Basili, 2001) summarized different studies and claimed that the average effectiveness (i.e., the number of defects found by an inspection) is around 60%. Some issues that have an influence on effectiveness are the size and complexity of a system, the experience of the inspectors, and the type of the applied inspection process. Laitenberger and DeBaud (Laitenberger and DeBaud, 2000) report that based on available quantitative evidence, inspections have a "significant positive effect on the quality of the developed product and that inspections are more cost-effective than other defect detection activities". In addition, performing inspections can reduce maintenance effort (Fagan, 1986).

Most of the mentioned individual inspection techniques have been evaluated by their founders or by different research groups (e.g., N-fold inspections, which were originated and first evaluated by Martin and Tsai (Martin and Tsai, 1990), and further evaluated by Tripp et al. (Tripp et al., 1996) and Kantorowitz et al. (Kantorowitz et al, 1997)).

Consequently, software inspections can be seen as a well-evaluated and mature quality assurance technique. Beside empirical evidence on the effectiveness and efficiency of software inspections, additional improvement characteristics exist. For instance, Wiegers (Wiegers, 2002) stated that different kinds of knowledge are gained and improved during an inspection, for example, knowledge about the product to be checked or about defect types. Laitenberger and DeBaud (Laitenberger and DeBaud, 2000) also talked about learning effects, which have an impact on the quality of the corresponding product or productivity. Doolan (Doolan, 1992) mentioned social aspects such as improved team building and improved communication. Finally, if certain inspection data is measured, projects will become more stable and predictable in terms of number of defects to be expected and effort required.

Inspection Reading Support

Besides research regarding the inspection process itself, another inspection research area is reading support, which is used by inspectors during the preparation phase as support for individual defect detection. Some kind of reading support can also be used to guide defect detection in a meeting (e.g., a checklist) Aurum et al. (Aurum et al., 2002) distinguish between ad-hoc reading, checklist-based reading, stepwise abstraction, defect-based reading, and perspective-based reading. Kollanus and Koskinen (Kollanus and Koskinen, 2007) also mention usage-based reading, abstraction-driven reading, and task-driven inspections. Furthermore, focused checklists and guided checklists exist, as well as traceability-based reading.

When using ad-hoc reading during the preparation step, an inspector does not get any reading support. In this case, the inspector performs defect detection based solely on his knowledge and experience.

Besides a general checklist used by each inspector (Laitenberger et al., 2000), focused checklists (Denger et al., 2004) and guided checklists (Elberzhager et al., 2009) have been developed in order to present different perspectives from which an artifact can be checked, respectively defect classes can be looked for in an inspection. The main advantage of focused and guided checklists is the higher defect coverage within the artifact to be checked and the lower overlap of defects found by the inspectors, i.e., inspectors mainly find different defects, resulting in higher effectiveness. One main problem with checklists is that the checklist questions are often too general, as stated by Brykczynski (Brykczynski, 1999).

Scenarios are another class of reading techniques that comprise, for example, defect-based reading, perspective-based reading, and usage-based reading. The idea is that an inspector should work actively with a document instead of only reading checklist questions in a passive manner. For this, an inspector gets, for example, a description of his perspective (perspective-based reading (Basili et al., 1996; Laitenberger and DeBaud, 1997; Laitenberger et al., 2000)) or a certain defect class (defect-based reading (Porter and Votta, 1998)), which sets the focus. Next, concrete instructions have to be followed and questions have to be answered. For example, imagine an inspector taking the tester perspective. One instruction might be, "Derive a number of test cases from the corresponding document", and a possible question is, "Is all information necessary for deriving test cases stated?" Following traceability-based reading, consistency is checked among other quality aspects within or between different artifacts (Travassos et al, 1999). Thelin et al. (Thelin et al., 2001, 2003, 2004) proposed usage-based reading, in which use cases are manually executed and mainly functional

defects should be found. Abstraction-driven reading, which was presented by Dunsmore et al. (Dunsmore et al., 2001, 2002, 2003), is used to understand code parts and to identify defects when extracting relevant information and abstracting them in objective-oriented environments. Finally, Kelly and Shepard (Kelly and Shepard, 2004) further developed abstraction-driven reading into task-driven inspections by adding three tasks (i.e., steps) an inspector has to perform, namely creating a data dictionary, extracting detailed logic, and deriving cross references.

In conclusion, different kinds of reading support were developed during the past 30 years in order to support inspectors when looking for defects. The most common reading techniques are ad-hoc reading, checklist-based reading, and scenario-based reading. Laitenberger and DeBaud (Laitenberger and DeBaud, 2000) mentioned that more than 25 articles advocate checklists. Brykczynski (Brykczynski, 1999) summarized more than one hundred different checklists addressing various document types and programming languages. At least six different scenario-based reading techniques were developed. Moreover, many experiments were performed in order to compare different reading techniques with, most of them comparing checklist-based reading, perspective-based reading, and ad-hoc reading (for an overview, see, for example, (Laitenberger and DeBaud, 2000; Elberzhager, 2005; Kollanus and Koskinen, 2007; Denger, 2009)). It could be shown that each reading technique can be superior to the others, depending on the concrete context and influence factors. Consequently, it is unclear which one is the most effective or efficient reading technique, respectively if there is even a single best one. However, based on the empirical knowledge on reading techniques, it can be concluded that they support inspectors in a beneficial way, and that a concrete selection has to be decided depending on the concrete context. Finally, Laitenberger and DeBaud (Laitenberger and DeBaud, 2000) stated that "ad-hoc reading and checklist-based reading are probably the most popular reading techniques used today".

Further Research Directions

Beside the inspection process and reading support, approaches for estimating the number of remaining defects have been developed. Such numbers can be used to decide if a re-inspection should be performed or not. One prediction approach, which uses inspection data to predict the defect content, is the capture-recapture method (Eick et al., 1992; Wiel and Votta, 1993; Wohlin et al., 1995; Briand et al., 1997; Miller, 1999; Petersson et al., 2004). The number of remaining defects can be predicted with statistical methods in a software artifact (including code). The detection profile method (Briand et al., 1998) is an alternative prediction approach using a linear regression method. Another prediction approach are subjective estimations, which were investigated by El Emam et al. (El Emam et al., 2000) and Biffl (Biffl, 2000), among

others. Studies performed by them resulted in better predictions compared to objective prediction approaches. Furthermore, curve-fitting methods were developed (Wohlin and Runeson, 1998) or combined with the capture-recapture methods (Briand et al., 1998).

Another aspect when it comes to improving software inspections is the use of tools. Laitenberger and DeBaud (Laitenberger and DeBaud, 2000) compared ten different inspection tools and classified them with respect to multiple criteria such as planning support, defect detection support, automated defect detection, defect collection support, defect correction support, and process measurement support. The authors concluded that the use of inspection tools is limited to particular development steps and may only slightly support the inspection. Hedberg (Hedberg, 2004) classified existing tools with respect to four generations, namely early tools, distributed tools, asynchronous tools, and web-based tools. The author concluded that a variety of different inspection tools have been developed, but "no tool has been widely adopted for practical use or contains all the important features that have proved feasible". The inspection repository presents a list of open-source and commercial inspection tools (Inspection repository, 2011).

Summary

A lot of research has been performed regarding software inspections after inspections were first invented and published by Fagan in 1976. The main research directions have been different inspection techniques and reading techniques, which have been defined, analyzed, compared, and evaluated. Software inspections can be treated as a well-known and highly mature software quality assurance technique.

However, Kollanus and Koskinen (Kollanus and Koskinen, 2007) concluded that although much research has been performed with respect to software inspections, empirical knowledge remains low. Laitenberger and DeBaud (Laitenberger and DeBaud, 2000) mentioned several research questions, all of which focus on specific inspection aspects, such as identification of the most cost-effective inspection variant, determination of stop criteria for inspections, or tool support. None of these questions addresses inspections in a broader sense, for example, integrating inspections with other quality assurance techniques in order to improve the overall effectiveness or efficiency. Aurum et al. (Aurum et al., 2002) also asked several questions to be answered by future research, such as of them asking about the relationship between software inspections and testing. Furthermore, they asked "What is the best way to ensure that the techniques complement each other in the most positive way?". One contribution to answering this question is made by this thesis, which presents an integrated inspection and testing approach.

1.6.3 Software Testing

Next, basic ideas and concepts regarding software testing will be presented. This comprises test levels, test techniques, test processes, and further research directions. With this, an overview regarding software testing is given in order to allow the reader to understand the testing concepts used in this thesis.

Software testing is a dynamic quality assurance activity. The IEEE Standard 610.12-1990 (IEEE Standard 610, 1990) defines testing as (1) "the process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component" and (2) "the process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items". Consequently, one main goal of testing is to find defects in software (e.g., programs, executables) or software systems.

While the presence of defects can be shown with testing (and thus, the quality of the corresponding product can be evaluated), the absence of defects cannot be shown by testing (Dijkstra, 1972). Boehm and Basili (Boehm and Basili, 2001) state that between 40 and 50 percent of software programs contain nontrivial defects. Insufficient testing can result in serious consequences, for example, with respect to costs. Jackson et al. (Jackson et al., 2007) mention many examples of disruptions and accidents from several domains (e.g., aviation, medical devices, infrastructure, defense, voting) due to software defects and describe the consequences in terms of costs, decreased confidence, and even human casualties. Therefore, testing has to be optimized in order to enable products of high quality, i.e., products that contain as few defects as possible before distribution.

One of the first books focusing on testing topics was written by Myers (Myers, 1979), who introduced some fundamental concepts such as black- and white-box testing or a distinction of certain testing levels (e.g., module testing, system testing, acceptance testing). Furthermore, the author stated certain principles, such as not to test a program that is written by oneself, or using valid and invalid input data for testing. In addition, Myers explicitly separated debugging activities from testing activities.

An enormous number of publications (e.g., books, articles in journals and magazines, conference articles) regarding various aspects of software testing has been published, such as test documentation, test procedures, test design, test plans, test phases, or test cases. Consequently, only some basic concepts will be sketched in the following, especially test levels, test methods, test processes, and some other aspects.

Test Levels

In order to conduct testing activities, executable program parts or the entire executable software product are needed. A common distinction of the levels at which testing can be performed is the following (Burnstein, 2002):

- Unit test
- Integration test
- System test
- Acceptance test

Unit or module test comprises testing functions, procedures, classes, and methods. Here, a unit is the smallest possible testable software component. In order to test the interaction of units, an integration test is performed. Different strategies for integrating units exist, for instance, top-down, bottom-up, or big-bang approach. A system test is performed to check the system behavior against its requirements, which includes functional and non-functional requirements. Finally, an acceptance test is done to check the complete system from the perspective of the customer.

Two additional kinds of test levels can be distinguished:

- Alpha test
- Beta test

An alpha test means testing the software in the test environment of the customer, respectively that of a number of customers. A beta test comprises testing or running of the software by one or more customers under real-world conditions.

Test Techniques

Several classifications exist for sorting the multitude of different testing techniques. A common high-level view is a distinction into black-box and white-box testing techniques, respectively test design techniques. White-box techniques use the structure of the program code, i.e., a tester knows concrete implementation details. The structure is often represented by a flowgraph. Black-box techniques do not use the structure of the code (i.e., internal information), but only use external information (e.g., requirements). Gray-box techniques are a third category that is sometimes used to describe methods such as test-first,

where developers write tests for their own code (i.e., they know the internals and thus, this can be seen as a white-box technique), but they write the test cases before the code is developed (i.e., the internals are not known when the test is written, which leads to a black-box technique).

Some concrete testing techniques and their classification are presented next. Burnstein (Burnstein, 2002) distinguishes the following testing techniques regarding black-box and white-box techniques:

- Black-box:
    - Equivalence class partitioning, boundary-value analysis, state-transition testing, cause and effect graphing, and error guessing.
- White-box:
    - Statement testing, branch testing, path testing, data-flow testing, mutation testing, and loop testing.

In addition, the author mentions techniques for testing non-functional properties of a system, such as performance testing, stress testing, configuration testing, security testing, and recovery testing.

Liggesmeyer (Liggesmeyer, 2009) states that a distinction into black-box techniques and white-box techniques is too coarse-grained based on the current state of the art. Therefore, the author suggests a classification of testing techniques into the following categories (the number in brackets show how many concrete testing techniques are mentioned for each category):

- Structure-oriented (control-flow oriented (10), data-flow oriented (9))
- Function-oriented (6)
- Diversifying (3)
- Test of certain areas (3)
- Others (e.g., error guessing, boundary value analysis) (4)

The Standard for Software Component Testing (British Standard, 1998) distinguishes between 13 different testing techniques, such as equivalence partitioning, boundary value analysis, branch testing, modified condition decision testing, or random testing. No classification

is presented. Instead, a description and how to design test cases is given for each technique.

Juristo et al. (Juristo et al., 2004) performed a comprehensive analysis of existing knowledge and empirical evidence on existing testing techniques. The authors classified existing testing techniques into random (3), functional (2), control flow (5), data flow (8), mutation (3), regression (5), and improvement (2). For each category, the results of performed experiments were summarized and conclusions were drawn. Furthermore, analyses of experience with comparisons between different testing techniques were conducted. Their main conclusion is that "more experimentation is needed and much more replication has to be conducted before general results can be stated". However, a lot of useful advice and recommendations for practitioners and research areas for researchers can be found in this work.

In conclusion, a lot of different testing techniques have been developed and classified in different ways. One of the most common classifications, even though a coarse-grained one, is the distinction into black-box and white-box testing techniques. Furthermore, although a lot of empirical evidence exists, Juristo et al. (Juristo et al., 2004) concluded that the knowledge about testing techniques is very limited.

Test Process

A great number of different test processes exists, as a test process has to be adapted to a concrete context in order to be most suitable. However, even if test processes differ in some details, certain general steps exist.

The Standard for Software Component Testing (British Standard, 1998) proposes a generic component test process consisting of five steps. Test planning should comprise a specification that describes how a test strategy is enforced. A set of test cases should be defined using a determined test design technique. Afterwards, the test cases should be executed in the test execution step. The context and the results of the test cases should be documented in the test recording step. Finally, the test results have to be analyzed and checked to see if specified completion criteria are fulfilled. Certain loops exist, indicating that some steps may be repeated. Figure 5 presents an overview of the generic test process.

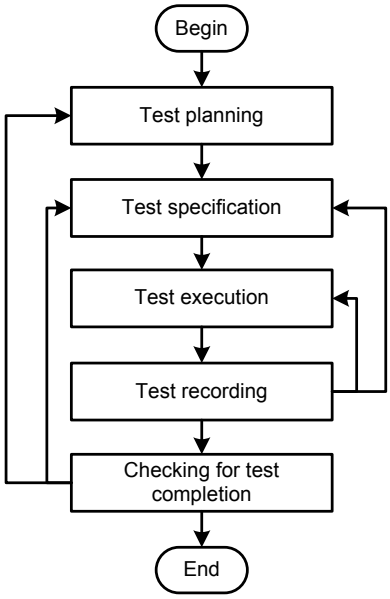


Figure 5 Generic component test process as determined by the Software Component Testing Standard.

Spillner and Linz (Spillner and Linz, 2003) generalized the generic component test process into a fundamental test process, which consists of the same steps. The quasi-standard TMap (TMap, 2011) mentions four phases, respectively steps, that are embedded by a fifth one. Figure 6 shows an overview and some concrete tasks for each step.

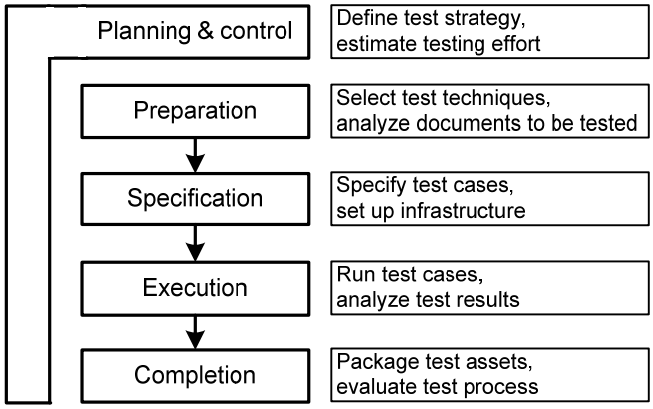


Figure 6 TMap testing process showing steps on the left and tasks on the right.

Different development methodologies have been created, such as the waterfall model (Royce, 1970), the general V-model (Boehm, 1979), the Rational Unified Process (Jacobson et al., 1999), or extreme

programming (Beck, 2000). Depending on the concrete development process, the test process has to be adapted accordingly. For example, in a waterfall model, testing is only performed once at the end of the development lifecycle. For agile development processes such as extreme programming, testing is conduced at a much higher frequency instead.

Further Research Directions

A lot of additional aspects can be considered important regarding testing. Standards such as the ISO/IEC 15504 (ISO/IEC 15504 Standard, 2006), describe among other aspects, how to perform testing with respect to existing state-of-the-art procedures. The standard can be used to assess testing activities in concrete environments, with the focus being made on documentation of testing processes, systematic derivation of test cases, and systematic performance of testing processes. The quasi-standard Test Process Improvement (TPI) (Koomen and Pol, 1999) covers 20 key areas (e.g., test strategy, test specification techniques, test environment, defect management) grouped into the four categories life cycle, techniques, infrastructure and tools, and organization. TPI can be used to evaluate and improve several testing aspects.

With respect to test documentation, the IEEE Standard 829-1998 (IEEE Standard 829, 1998) provides certain guidelines. From high-level test plans via detailed test case specifications to test logs and summaries, a lot of advice is presented on how to document information that arises during the testing process.

Furthermore, testing tools support testing activities. Certain goals can be achieved with tools, such as executing time-consuming activities (e.g., regression testing), creating test logs, improving testing efficiency, or conducting certain measurement activities. One classification of tools is given by Liggesmeyer (Liggesmeyer, 2009) who distinguishes four different classes of test tools:

- dynamic test tools (e.g., structure-oriented test tools, function-oriented test tools, regression test tools)
- static analysis tools (e.g., slicing tools, measurement tools)
- formal verification tools (e.g., symbolic model checking)
- modeling and analyzing tools (e.g., FMECA (Failure Mode, Effects and Criticality Analysis) tools, fault tree tools)

Summary

Several testing aspects have been covered by researchers and practitioners, which is reflected by the huge number of existing publications. The concrete testing process is often dependent on the concrete environment and thus, adaptation is necessary. With respect to empirical knowledge regarding testing, Juristo et al. (Juristo et al., 2004) stated that the knowledge about testing techniques is very limited. Consequently, additional experiments and case studies with respect to certain testing aspects are needed to allow drawing clear conclusions. Moreover, immanent objectives, such as improved efficiency and effectiveness, imply challenges for future research activities, especially with respect to different environments where testing approaches have to be adapted.

1.6.4 Basic Terminology

In order to achieve a common understanding of some basic terms used in this thesis, definitions are provided next.

- *Quality assurance activities (short: quality assurance (QA)):* The IEEE Standard Glossary of Software Engineering Terminology defines quality assurance as "(1) A planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements. (2) A set of activities designed to evaluate the process by which products are developed or manufactured" (IEEE Standard 610.12, 1990). Consequently, quality assurance can be conducted in order to ensure the quality of either products or processes. Here, the term quality assurance activity is considered as a top-level term that covers approaches, methods, techniques, or tools.

    - This thesis focuses on the quality of software and software products.

  A refinement into constructive quality assurance (i.e., quality assurance activities that aim at preventing the introduction of defects by using, for example, patterns and guidelines) and analytical quality assurance (i.e., quality assurance activities that aim at detecting and removing defects) can be made.

    - This thesis focuses on analytical quality assurance activities (i.e., when the term quality assurance is used in this thesis, analytical quality assurance is implicitly meant).

A distinction into static and dynamic quality assurance can also be made. Static quality assurance activities (e.g., inspections) do not need executable models or executable code, but rather examine artifacts such as requirements documents, design models, or code without running them. In contrast, dynamic quality assurance activities (e.g., testing) need executable program parts or software products.

  - This thesis focuses on both static and dynamic quality assurance.

- *Failure:* The IEEE Standard Glossary of Software Engineering Terminology defines a failure as "the inability of a system or component to perform its required functions within specified performance requirements" (IEEE Standard 610.12, 1990). A failure is a kind of misbehavior of software or a software product that is visible to a user. Especially during testing activities, failures are observed (instead of faults).

- *Fault:* The IEEE Standard Glossary of Software Engineering Terminology defines a fault as "an incorrect step, process, or data definition in a computer program" (IEEE Standard 610.12, 1990). A fault is the underlying cause for a failure, i.e., a fault is the de facto reason that software or software systems fail.

- *Defect:* A defect is treated in this thesis as a top-level term comprising faults and failures. While different quality assurance activities tend to rather identify faults or failures, this can be generalized when using a more abstract term such as defect in order to be able to compare and evaluate them.

- *Effectiveness:* The effectiveness of a quality assurance activity is defined as the number of existing defects found, respectively the percentage of defects found with respect to the number of existing defects (Runeson et al., 2006).

  - This thesis uses the first definition.

- *Efficiency:* The efficiency of a quality assurance activity is defined as the number of defects found divided by the time required to detect them (Runeson et al., 2006).

- *Defect content:* Defect content is defined as the number of defects found by one or more quality assurance activities.

- *Defect density:* Defect density is defined as the number of defects found by one or more quality assurance activities divided by a size value (e.g., lines of code).

- *Inspection (inspection activity):* An inspection is a static quality assurance activity. The term is used as a top-level term comprising different concrete inspection techniques, respectively processes.

- *Inspection technique (inspection process):* An inspection technique, respectively process, is a concrete static quality assurance activity, such as a formal inspection, team review, walkthrough, or deskcheck, that comprises different inspection steps (Wiegers, 2002).

- *Inspection reading support (short: reading support or reading technique):* During defect detection in an inspection, inspectors can be supported by reading support, such as checklists.

- *Testing (testing activity):* Testing is a dynamic quality assurance activity. The term is used as a top-level term comprising different test processes, respectively test process steps.

- *Test process:* A test process consists of certain steps, such as test planning, test execution, or test result analysis.

    - The main focus in this thesis is on test execution, i.e., defect detection; a minor focus is placed on test specification.

- *Test technique (synonym: test design technique):* In order to derive and execute test cases in a systematic manner, different test techniques can be applied, such as equivalence partitioning, boundary-value analysis, or control-flow based testing.

- *Test level:* A testing activity can be performed during different development stages, such as at the unit, integration, or system level.

1.7 Research Contributions

The main research contributions (RC) of this thesis can be summarized as follows:

- *RC1 (State-of-the-practice analysis):* The thesis provides an analysis of current inspection and testing practices. An overview

of two major problems that were identified is given, and requirements for an integrated approach are derived.

- *RC 2 (State-of-the-art analysis):* The thesis provides a general overview of existing inspection and testing research contributions. Furthermore, two systematic mapping studies have been conducted. First, an analysis of approaches that combine static and dynamic analysis was performed (Elberzhager et al., 2012a). Second, a mapping study of further approaches that are able to improve testing efficiency was done (Elberzhager et al., 2012b). Finally, a mapping of the requirements to existing approaches concluded this analysis.

- *RC 3 (Development of the In²Test approach):* An integrated inspection and testing approach was developed that uses inspection defect data to focus testing activities. A one-stage approach (i.e., focusing on either defect-prone parts or defect types) and a two-stage approach (i.e., focusing on defect-prone parts and defect types) are provided. Further product metrics and historical data could be combined with the inspection results in order to improve focusing (Elberzhager and Eschbach, 2010; Elberzhager et al., 2010c, 2011d, 2012; Elberzhager and Muench, 2011). Parts of the approach have been implemented as a prototype by the DETECT tool (Elberzhager et al., 2010a).

- *RC 4 (Definition of assumptions):* In order to perform focused testing activities, context-specific knowledge about the relationships between inspections and testing have to be considered. As this is often not available, assumptions have to be stated. This thesis provides a model for the structure of such assumptions, guidelines on how such assumptions can be derived, evaluated, and applied, and a number of exemplary assumptions for applying the In²Test approach (Elberzhager et al., 2011a, 2011c).

- *RC 5 (Evaluation):* Two case studies were performed to evaluate the In²Test approach. These two case studies were performed applying the integrated approach during the development of two tools (Elberzhager et al., 2010c, 2011c, 2011d, 2012; Elberzhager and Muench, 2011).

1.8 Thesis Structure

The first chapter provided an introduction to this thesis. After starting with a motivation of quality assurance in the area of software engineering, the research scope was defined. Furthermore, problems were identified, followed by goals to be achieved and hypotheses to be

checked. Moreover, background information on quality assurance and basic terminology was presented. Finally, the main research contributions were summarized. The remainder of this thesis is structured as follows:

- *Chapter 2: State of the Practice.* Chapter 2 provides an overview of current inspection and testing activities followed in practice. Concretely, current approaches, techniques, and methods are presented and results from studies are summarized. Furthermore, problems with respect to two quality assurance activities are mentioned. Finally, a set of requirements that a new approach has to fulfill is defined.

- *Chapter 3: State of the Art.* Chapter 3 provides an overview of existing approaches that combine static and dynamic quality assurance activities, indicating that there exists no approach that combines inspection and testing techniques in a systematic manner. In addition, an overview of non-combined approaches that improve testing efficiency is given. The chapter concludes with a comparison of requirements fulfillment of the existing approaches in order to identify current gaps in the existing state of the art.

- *Chapter 4: The In²Test Approach.* Chapter 4 presents the basic ideas of the integrated inspection and testing approach. A one-stage and a two-stage approach are presented. In addition, a conceptual model for systematically describing assumptions as well as exemplary assumptions describing the relationships between inspection defects and testing defects are given, together with guidelines for systematic derivation and evaluation. In addition, concepts of the approach implemented in the DETECT tool are shown. Finally, limitations of the approach are mentioned.

- *Chapter 5: Empirical Evaluation.* Chapter 5 presents the results of the empirical evaluations of the In²Test approach. First, the validation strategy is defined and a refinement of the hypotheses is done. Afterwards, two case studies are described, including the context, the design, the execution, the analysis, and limitations of the studies.

- *Chapter 6: Conclusions and Future Work.* Chapter 6 presents a summary of the thesis and provides an outlook on future work with respect to several aspects, such as different improvements of the approach or further evaluations.

- *Appendix A: Checklists.* Appendix A lists the inspection checklists that were used during the evaluations.

- *Appendix B: Experiment Design*. Appendix B presents different designs for future evaluations. Two experiment designs are described that can be used to compare groups using the integrated approach with groups not using the approach.

- *Appendix C: Questionnaire*. Appendix C shows a questionnaire that can be used by practitioners to evaluate the approach. It is based on a standardized model, which has been adapted accordingly.

- *Appendix D: Initial Industrial Evaluation Results.* Appendix D presents some initial insights and results from an industrial context where the approach was applied.

2 State of the Practice

2.1 Overview

The goal of this chapter is to present an overview of the state of the practice regarding software inspections and software testing, and to show that these two quality assurance techniques are most often applied in an isolated manner, without exploiting any synergy effects. Section 2.2 describes how software inspections are currently performed in industry. Section 2.3 gives an overview of testing in industry. Section 2.4 describes two major problems based on the described state of the practice, and derives a set of requirements to overcome these problems. Finally, Section 2.5 summarizes this chapter.

2.2 Software Inspections in Industry

This section presents an overview of how software inspections are applied in the field. A set of empirical results is described that show the performance of software inspections applied in different environments. Based on these observations, software inspections are an effective and efficient static quality assurance activity, which is, when applied, mostly applied in isolation.

The first publication about software inspections by Michael Fagan in 1976 (Fagan, 1976) introduced this kind of static quality assurance and showed initial empirical results from a development project, i.e., results from an industrial context. Since then, software inspections have become a mature and established static quality assurance technique during the past 35 years. The reasons for this include its high applicability in different contexts, its high efficiency and effectiveness, or cost reduction as proven in many empirical studies (summarized, for example, by Laitenberger and DeBaud (Laitenberger and DeBaud, 2000) or Elberzhager (Elberzhager, 2005)).

Ten years after Fagan's first publication, he published results from a project at IBM (Fagan, 1986). A defect detection rate of 93% was achieved by performing inspections over the lifecycle of the product. In general, common defect detection rates (i.e., effectiveness values) for software inspections are between 50% and 70%. Table 1 summarizes some empirical results from different industrial contexts and their effectiveness values.

Table 1 Some empirical results regarding inspection effectiveness from different industrial contexts.

No.	Environment	Artifact	Results	Reference
1	Aetna Life and Casualty	Design / code	82% effectiveness	(Fagan, 1976)
2	IBM Respond	Code	93% effectiveness	(Fagan, 1986)
3	Standard Bank of South Afrika, American Express	Code	~50% effectiveness	(Fagan, 1986)
4	ICL	Design	58% effectiveness	(Kitchenham et al., 1986)
5	Project >700k LoC	Design / code	54% / 64% effectiveness	(Collofello and Woodfield, 1989)
6	Bull HN	Code	~70% effectiveness	(Weller, 1992)
7	AT&T Bell Laboratories	Code	> 70% effectivteness	(Barnard and Price, 1992)
8	Shell Research's Seismic Software Support Group	Requirements	50% effectiveness	(Doolan, 1992)
9	IBM Rochester Labs	Code / pseudocode / module and interface specification	60% / 80% effectiveness	(Gilb and Graham, 1993)
10	HP	Code	60-70% effectiveness	(Grady and van Slack, 1994)
11	Cardiac Pacemaker	n/a	70-90% effectiveness	(McGibbon, 1996)
12	Lockheed Martin's space shuttle onboard software project	n/a	85-90% effectiveness	(Lee, 1997)
13	Robert Bosch GmbH	Code	18-27% effectiveness	(Laitenberger and DeBaud, 1997)
14	Allianz Life Assurance	Requirements / Design	72-100% / 25-58% effectiveness	(Briand et al., 1998b)
15	Bosch Telecom GmbH	Code	55-78% effectiveness	(Laitenberger et al., 2001)
16	Ericsson Microwave Systems AB	Requirements	0.5 - 2.5 defects / page	(Berling and Runeson, 2003)
17	Ericsson Microwave Systems AB	Requirements	0.9 - 1.0 defects / requirement	(Berling and Thelin, 2004)

Table 2 Some results regarding inspection efficiency from different industrial contexts.

No.	Environment	Results	Reference
1	IBM Santa Teresa Lab	Cost ratio inspection defects: testing defects 1:20	(Remus, 1984)
2	Jet Propulsion Laboratory	Cost ratio inspection defects: testing defects 1:10 to 1:34	(Kelly at al., 1992)
3	IBM Rochester Lab	Cost ratio inspection defects: testing defects 1:13	(Kan, 1995)
4	Small warehouse inventory system	1h per defect found with design inspection, 1.2h per defect found with code inspection	(Ackerman et al., 1989)
5	Major government system	0.58h per defect found with design inspection, 0.67h per defect found with code inspection	(Ackerman et al., 1989)
6	Banking computer service firm	2.2h to eliminate defect by code inspection, 4.5h to eliminate defect by testing	(Ackerman et al., 1989)
7	n/a	1.43h to find a defect by inspection, 6h to find a failure with testing	(Weller, 1993)
8	IBM	1h to find a defect by code inspection, 6h to find a defect with testing	(Franz and Shih, 1994)
9	Ericsson Microwave Systems AB	1.2 - 2 defects per hour with requirements inspections	(Berling and Runeson, 2003)
10	Ericsson Microwave Systems AB	0.9 - 1.8 defects per hour with requirements inspections	(Berling and Thelin, 2004)

A number of empirical results also exist with respect to the efficiency of software inspections in industrial contexts. Different kinds of efficiency values are presented. For example, a comparison of costs regarding defects found by inspections and testing shows that inspections are often superior to testing. Furthermore, the time needed to find a defect using inspections is calculated. Table 2 summarizes some empirical results from different industrial contexts and their efficiency values.

Besides effectiveness and efficiency benefits, additional advantages are experienced when performing inspections. Laitenberger and DeBaud (Laitenberger and DeBaud, 2000) mention learning effects, which are worthwhile for educating team members. Gilb and Graham (Gilb and Graham, 1993) summarized different experience data, for example, highly reduced maintenance costs in a banking environment due to rigorous inspections, software projects finished earlier, or reduced testing costs. After inspections were introduced at Allianz Life Insurance, ten percent of testing costs could be saved (Briand et al, 1998b). Results from a software project onboard a space shuttle showed that the ratio between design or code defects found by inspections and such defects found after delivery was 1:92 (Paulk, 1995). Another study revealed that requirements defects found during a requirements inspection can be up to 68 to 110 times less expensive than if found by a customer (Grady, 1999). Results from a telecommunication company showed that, on average, $200 are necessary to find and correct a defect with inspections, compared to $4,200 to correct a defect that is found by a customer (Wiegers, 2002). Experiences from IBM and Toshiba showed ratios for critical defects found during inspection and in the field of 1:117, respectively 1:137 (Shull et al., 2002). The authors concluded that "finding and fixing a severe software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase." Furthermore, they state that "finding and fixing non-severe software defects after delivery is about twice as expensive as finding these defects pre-delivery". Though the concrete cost ratios depend on the specific context and the given numbers are not necessarily true for all environments, they show the potential of software inspections and the benefit of early defect detection.

Based on several observations and on empirical evidence, Laitenberger and DeBaud (Laitenberger and DeBaud, 2000) state that "inspections have had significant positive impact on the quality of the developed software and that inspections are more cost-effective (i.e., efficient) than other defect detection techniques".

Finally, some standards even require performing static quality assurance activities at certain quality gates if standard-compliant software products are to be developed (e.g., ISO/IEC Standard 62304, 2006).

Regarding concrete inspection processes that are applied in practice, these differ in their degree of formality, i.e., both very formal and very informal inspection processes exist. This is reasonable due to a variety of different context and project characteristics, such as available number of people, skills and experience of people, development processes, project size, or maturity of quality assurance processes. Furthermore, a lot of different inspection terms exist, such as formal inspections, (peer) reviews, static testing, or walkthroughs, which are all treated differently. This means that, based on the exact wording used for an inspection

process, it is unclear how it is performed concretely and what the degree of formality is.

Besides some companies that perform inspections in a very formal manner, most companies, especially small- and medium-sized ones, follow a more informal and unsystematic inspection process. Rombach et al. (Rombach et al., 2008) performed an analysis of the application of different static quality assurance techniques in industrial contexts and summarized the following experiences. Based on a large online survey to which 226 people from various domains, company of different sizes, and several countries responded (Ciolkowski et al., 2003), one main conclusion was that many companies perform software inspections unsystematically. In detail, this meant that certain inspection steps or even the entire inspection were optional. Furthermore, about 40 percent of the respondents perform requirements and design inspections, and another 30 percent do code inspections. Consequently, 60 respectively 70 percent do not perform any kind of inspection (neither formal nor informal) during specific development phases. Moreover, only 40 percent perform individual defect detection. Of those, checklists are used most often, followed by experienced-based reading (i.e., ad-hoc reading using no support). Finally, more than 50 percent of the companies either collect no data or do not analyze the collected data in order to assess the performance of the inspection process and to improve the inspection and development process.

Another study was performed by Johnson, who concluded that the adoption of inspections in industry remained rather low and that many companies do not perform inspections or conduct them in an informal way (Johnson, 1998). Some reasons from practitioners for not using inspections are that they are perceived as being too difficult, too costly, or ineffective.

A recent study about quality assurance in practice was conducted by Spillner et al. (Spillner et al., 2011, 2012), who collected feedback from about 1,600 German speaking respondents in the area of quality assurance and project management. While static quality assurance activities were only a minor topic of this study, three fourth of the respondents mentioned that reviews are conducted in their projects. However, they are usually applied in an informal manner (about 70%), and it is unclear how effective or efficient they are in those contexts.

Harjumaa et al. (Harjumaa et al., 2005) analyzed factors that motivate and discourage companies to perform inspections. The study was performed with local companies, but the authors generalized the results for small companies. The main motivator for performing inspections is to reduce defects. Further advantages, such as knowledge sharing or education, are less important. The main obstacles that were identified are lack of time and resources. Overall, the authors stated that

companies agree on and are aware of the potential of software inspections.

Jalote and Haragopal (Jalote and Haragopal, 1998) state that inspections are not widely applied in industry despite considerable evidence. One reason for the authors is the "not-applicable here" syndrome, i.e., the problem that people from a concrete context do not believe that inspections provide certain benefits in their context. The authors provide a solution for overcoming the mentioned problem by performing a small study in the own context, where an inspection was conducted and some data was gathered in order to demonstrate the advantages of an inspection. Results from an experiment demonstrated how an organization changed with respect to conducting inspections in their context.

Garousi and Varma (Garousi and Varma, 2010) performed a replicated study of software testing practices in Alberta, Canada, focusing mainly on testing aspects. However, one question addressed defect detection methods performed in an organization in general. In 2004, about 45% performed informal inspections and about 18% did formal inspections. This changed slightly in 2009, namely from about 45% to 38% for informal inspections, and from about 18% to 28% for formal inspections. However, the overall number of companies performing inspection did not change much during these five years.

In conclusion, a lot of success stories exist that demonstrate the value of software inspections in industry. They mainly comprise overall improved defect detection rates, reduced costs, and higher quality of a product. For example, Grady and van Slack (Grady and van Slack, 1994) report on cost savings by HP of about $21 million due to inspections. Fagan (Fagan, 2001) reported cost savings of a customer of $45 million due to coding defects alone, found by inspections four years after inspections were introduced. One main benefit of a software inspection is that it can be applied early in the development process, for example to inspect requirements or design documents, and consequently, to find defects early. Defects that are introduced early (e.g., during requirements), but are found late (e.g., during system test or after the product is delivered) may lead to high costs for correcting them, especially if they are critical defects. Furthermore, besides defect detection, additional advantages of inspections exist, such as learning and communication effects.

However, despite existing evidence from various industrial environments, inspections are not widely distributed, or are often not applied in a systematic manner. Jones (Jones, 2004) mentioned that unsuccessful projects typically omit inspections. Consequently, certain advantages provided by inspections are not exploited. Data gathered from inspections are often not used to control subsequent quality assurance activities, such as different testing activities. Due to increased challenges

in software development in recent years, such as higher cost pressure, a demand for higher quality, increased complexity, or certain standards that require conducting static quality assurance, inspection techniques will probably be applied more often and more systematically in the future and may play a crucial role in overall quality assurance strategies.

2.3 Software Testing in Industry

This section gives an overview of how software testing is applied in the field. Software testing is widely conducted in various environments. A lot of empirical studies have shown their benefits, but also the need for more efficient test processes.

Software testing is one of the main quality assurance activities in modern software development. Koomen and Pol (Koomen and Pol, 1999) stated that "testing is a must" and an essential prerequisite for developing software and building software systems. Juristo et al. (Juristo et al., 2006) noted that "the importance placed on testing will increase". One reason is that software plays a role in everyone's life (knowingly or unknowingly). Consequently, tolerance for defect-prone software will decrease and quality assurance techniques, such as testing, will have to ensure high quality.

Bertolino and Marchetti (Bertolino and Marchetti, 2004) published an essay on software testing that summarizes various testing aspects and addresses several challenges test practitioners face. Depending on the respective software development process, among which the "V-model" is one of the most popular ones, different test phases are distinguished. The most common ones are unit, integration, system, and acceptance testing. The authors consider regression testing as a parallel testing activity that is performed whenever changes to the system or parts of the system are made in order to check whether these changes did not introduce new defects. Figure 7 gives an overview. Regarding test case selection strategies, the authors distinguished between black-box and white-box methods and summarized several concrete techniques. Black-box techniques are much more dominant in practice than white-box techniques. Furthermore, use-case based testing, functional testing, equivalence partitioning, and boundary-value analysis are some of the most often applied black-box techniques (see, for example, Spillner et al., 2012). Further aspects, such as test execution, test documentation, test management, and test measures, are sketched.

Liggesmeyer (Liggesmeyer, 2009) states that dynamic quality assurance techniques (i.e., testing techniques) are widely applied during software development in industry. According to the results from a survey (Spillner and Liggesmeyer, 1994) focusing on testing phases conducted in practice, module testing is applied most often (about 72%), followed by

integration and system testing (both about 58%). A recent study by Spillner et al. (Spillner et al., 2011, 2012) showed that system testing gained more attention, while the focus on unit testing decreased.

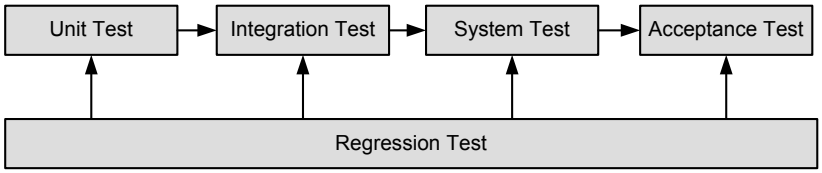


Figure 7 Logical schema of software testing levels according to Bertolino and Marchetti (Bertolino and Marchetti, 2004).

In recent years, new development paradigms have emerged that overcome a) rigorous and cumbersome development processes and b) rich and formal documentation during development. This results, for instance, in agile development processes. Consequently, test processes have been adapted accordingly, resulting in, e.g., test-driven development. Janzen and Saiedian (Janzen and Saiedian, 2005) describe concrete concepts and provide results from empirical evaluations in industry and academia contexts. Some studies showed a positive effect on quality and productivity effects when following a test-driven approach, while other studies showed no effect or even negative effects.

Another study regarding testing in an agile environment describes test planning, design and execution, and defect management with respect to a large-scale project (Talby et al., 2006). The authors could show positive overall results with agile testing methods and were able to perform full regression testing at each iteration despite the huge project. However, the authors state that much more empirical evidence is necessary from different environments and encourage others to perform investigations of long-term agile projects, in particular. Pettichord (Pettichord, 2004) mentions several testing challenges in agile environments, such as testing incomplete code, test stopping criteria, or regression testing in short cycles, and provides some hints on how to address these problems.

Different surveys regarding testing in different industrial environments have focused on various testing aspects. For example, Geras et al. (Geras et al., 2004) conducted a survey of software testing practices in Alberta, Canada. Almost 60 participants took part in this study, which focused on test levels, test techniques, test measures, and test management. Unit and system tests are the test levels that are most often addressed (between 75% and 90%), followed by acceptance, installation, and regression tests. Test automation was low for each level (maximum: about 32% for unit testing, about 21% for system testing). Most often, testers derive test cases based on their experiences and skills, followed by using requirements for test-case derivation. Boundary value or equivalence partitioning are only followed rarely (almost 30%,

respectively 5%). Typically, organizations have more developers than testers. Regarding measures, use case points, McCabe complexity, or lines of code are the most frequently used ones, with between 23% and 37% of the companies using such measures. The authors concluded that immature testing processes (e.g., low number of test levels, lack of testing training) increase the risk of slipped defects found only after delivery.

Garousi and Varma (Garousi and Varma, 2010) replicated the study by Geras et al. (Geras et al., 2004) to analyze what changed and what did not change in the given environment. Their main findings were: almost all companies performed unit tests and system tests with a slight increase since the first study; automation of unit, integration and system tests increased; test case generation techniques did not change much, except that fewer testers used risk and more testers used boundary value analysis; when planning tests, metrics such as McCabe complexity and lines of code were rarely used compared to the first study; instead, use case points were used by more than 70% of the respondents. All in all, the authors state that slight improvements could be noticed, but much more improvement is necessary.

Another survey was conducted to investigate software testing practices in Australia (Ng et al., 2004). Overall, 65 persons or companies participated in the study. About 50%, respectively 70%, performed acceptance and regression testing. Black-box testing techniques (e.g., boundary value analysis, random testing) were followed more often than white-box testing techniques. Most of the organizations performed formal tests in order to check that the requirements are met. However, one third performed testing activities in an unsystematic manner. About two thirds have automated some of their testing activities. Furthermore, different standards were adopted by about 72% of the organizations. In conclusion, the major problems the authors identified from that survey were untrained testers, costs for introducing tools, and time available for testing. Furthermore, based on their observation that about 75 percent of the companies allocated less than 40% of their budget to testing, the authors concluded that organizations "are not allocating realistic budgets to testing".

Runeson (Runeson, 2006) performed a study on unit testing practices and summarized the results from 19 companies that participated in the survey. The author investigated the understanding of unit testing practices and identified strengths and weaknesses of unit testing techniques. Some results were that white-box techniques are preferred, automation is an important goal with respect to unit tests, developers often perform unit tests, and functionality is checked most often with unit tests. The identified challenges include testing graphical user interface units, appropriate documentation, providing training for testers, and the amount of time that should be used for unit testing.

Finally, a study conducted by Otte et al. (Otte et al., 2008) investigated quality assurance methods with respect to open source development. About 400 participants took part in the survey. Overall, testing consumed about 39 percent of the development time and more than half of the projects followed a structured testing approach.

Another observation from practice is that testers are often not trained enough. Myers (Myers, 1979) already mentioned that "students graduate and move into industry without substantial knowledge of how to go about testing a program", which is also substantiated by newer studies such as those mentioned above. Burnstein (Burnstein, 2002), for example, presents eleven principles that can support practitioners and improve their testing knowledge. A positive trend regarding trained developers and testers can be found, but is still improvable (Spillner et al., 2012).

Everett and McLeod (Everett and McLeod, 2007) state that about 300 commercial test tools are available on the market that may support testers during certain testing activities. Furthermore, the authors distinguish between situations in which test tools are advantageous and disadvantageous. Finally, the authors note that no holistic tool exists that covers all relevant aspects within one tool.

One of the first books about software testing, written by Myers (Myers, 1979), indicated that testing consumes approximately 50% of the development time and more than 50% of the overall development costs. As new testing processes, methods, and tools were developed during the next 30 years, one might have expected the effort for testing to decrease. However, Juristo et al. (Juristo et al., 2006) state that "testing can easily exceed half of a project's total effort", and several studies substantiate this observation. For example, Geras et al. (Geras et al., 2004) note that "software testing currently consumes up to 50% of the total cost of software development for the average project". Koomen and Pol (Koomen and Pol, 1999) report that the range for software testing effort is between 25% and 50% of the overall project budget. Liggesmeyer (Liggesmeyer, 2009) concluded that quality assurance activities often consume most of the overall development effort, which Pressman (Pressman, 2009) calculated as up to 50% of the total development effort. This means that testing is still a major cost driver in modern software development.

Determining when to stop testing remains a major problem in industry. Geras et al. (Geras et al., 2004), for example, list coverage criteria that have to be fulfilled, or the passing of acceptance tests. Analyzing the coverage of parts of the system regarding test cases can be used to identify parts that have been tested slightly or not tested at all, and this can lead to additional tests. However, the authors also found out that many companies have only fixed time slots for testing, meaning that a

decision on when to stop testing is often not based on reliable information. The number of companies that followed this principle doubled in the following five years, as shown by Garousi and Varma (Garousi and Varma, 2010). Ng et al. (Ng et al., 2004) state that many practitioners stop testing when the budget is running out or when a certain deadline is achieved. The authors mention metrics (e.g., defect content metrics) as one means for attaining more control over this problem. Finally, Spillner et al. (Spillner et al., 2012) also confirmed that about 60% of the respondents of a recent study stop testing when a delivery date is reached, and about 30% mentioned that they stop testing when budget is consumed.

If not enough time is available to test all parts of a system, respectively if it is too extensive to test the system completely, a decision has to be made regarding which parts to focus on. Runeson (Runeson, 2006) observed that coverage criteria and prioritization of test cases can support making a test stopping decision. Risk-based approaches (Karolak, 1996) can support the decision about which parts to test with which intensity and about when testing is finished (e.g., when test cases covering critical functionality pass). A set of different questions to be answered in order to identify such parts is given by Hower (Hower, 2011). Another idea is to use historical data to identify parts that were defect-prone in past releases. Metrics such as size are also used sometimes to design certain test cases (Garousi and Varma, 2010) and to focus testing activities (Ostrand and Weyuker, 2002).

In conclusion, testing is one of the main quality assurance activities applied in modern software development. Bertolino (Bertolino, 2007) states that "testing is an essential activity in software engineering" and "testing is widely used in industry for quality assurance". However, she also states that "testing is still largely ad hoc, expensive, and unpredictably effective". Juristo et al. (Juristo et al., 2006) concluded that the "state of software testing practices isn't as advanced as software development techniques overall. In fact, testing practices in industry generally aren't very sophisticated or effective." While software testing approaches have been further developed, such improvements are not widely distributed in practice. Bertolino and Marchetti (Bertolino and Marchetti, 2004) mention that "test practice inherently still remains a trial-and-error methodology". The authors further demand "to transform testing from "trial-and-error" to a systematic, cost-effective and predictable engineering discipline".

A lot of different testing techniques have been proposed to support testers. Harrold (Harrold, 2000) states that "testing has been widely used as a way to help engineers develop high-quality systems. However, pressure to produce higher-quality software at lower cost is increasing. Existing techniques used in practice are not sufficient for this purpose." Consequently, existing techniques have to be adapted or new

techniques and approaches have to be developed to master such challenges.

One reason for insufficient testing are bad test strategies. Kasurinen et al. (Kasurinen et al., 2009) analyzed different problems in testing practices. One aspect is testing strategy and planning; most of the investigated companies scale down tests if necessary. Furthermore, the authors concluded that most of the test problems could be overcome with better test strategies. Ng et al. (Ng et al., 2004) also mention that testing strategies will become more important. Moreover, Humphrey (Humphrey, 2008) states that "the current testing-based quality strategy has reached a dead-end" and that "quality improvements required are vast, and such improvements cannot be achieved by merely building ahead with the test-based methods of the past". New testing strategies are necessary.

It is a fact that defects exist in software. For instance, Hayes (Hayes, 2002) reports that 97% of 800 respondents of a survey had problems with software defects in recent years, resulting in most cases in higher costs or lost revenues. Boehm and Basili (Boehm and Basili, 2001) state that 50 to 60 percent of software programs contain non-trivial defects. Testing is one way to find defects. Bertolino (Bertolino, 2007) states that "software testing is and will continue to be a fundamental activity of software engineering; [however], we will need to make the process of testing more effective, predictable and effortless".

This thesis aims at providing a solution to more effectiveness and efficiency during testing. Another goal is to support determining a test strategy. A concrete solution, i.e., a new approach, is presented in Chapter 4 which uses early defect data to help control testing processes.

2.4 Problems and Requirements

Next, two main problems derived from the state of the practice will be described, together with their rationales. Afterwards, requirements that an approach has to fulfill in order to address these problems will be stated.

2.4.1 Problems

Based on the overview of the state of the practice regarding inspections and testing presented above, two major problems have been identified, which will be addressed by the approach presented in this thesis.

Problem 1: Testing activities often do not use results and insights from early defect detection activities, especially inspection techniques. When inspection and testing are used during quality assurance, they are usually

performed in sequence, without any exchange of data between them to exploit synergy effects. Consequently, testing activities are often not focused based on early defect data. This leads to so-called local inefficiencies, i.e., test-specific effort is wasted. Existing approaches for reducing testing effort are widely based on the use of metrics, risk, or historical data to predict fault-prone parts of a product or determine test exit criteria. However, they do not make systematic use of the results from inspections, i.e., quantitative defect data from the software currently under development is not used to control testing processes. Some approaches consider the combination of inspection and testing techniques in a pragmatic and unsystematic manner (Kinochita, 2010) in order to be more effective or to predict the expected number of defects for testing based on inspection results (Harding, 1998). However, although inspection and testing techniques are sometimes integrated in an informal way in industry, no systematic approach could be found that integrates them in order to exploit synergies and allow controlling testing activities based on inspection defect data. This also means that test strategies are usually not defined or adapted systematically based on early defect data of a current software development cycle.

Problem 2: Quality assurance activities, especially testing activities, require too much effort. Effort for conducting software quality assurance activities, especially testing, can consume more than 50% of the overall development effort (Harrold, 2000; Hailpern and Santhanam, 2002; Pressman, 2009). High-quality software is demanded by customers due to the extensive distribution of software. Consequently, software developing companies need to develop software that has high quality. Due to things such as unsystematic test processes or inappropriate test strategies, high costs are an increasing problem in modern software development (Tassey, 2002; Kasurinen et al., 2009). Moreover, software quality assurance techniques rarely consider synergy effects resulting from their systematic combination and integration. This may lead to so-called global inefficiencies, i.e., total quality assurance effort is wasted. The integration of inspection and testing techniques promises different synergy effects such as reduced testing effort. However, inspections and testing are usually conducted independent of each other, i.e., they are sometimes applied in sequence in order to find additional defects (Franz and Shih, 1994; Berling and Thelin, 2003) or, based on empirical evidence, a combination is suggested (Runeson et al., 2006), but they do not collaborate in an optimal manner. Bertolino (Bertolino, 2007) concluded that there exist "many fruitful relations between software testing and other research areas", and that many of them were overlooked in the past. This includes the integration of inspection and testing to reduce testing effort and, as a result, overall quality assurance effort.

2.4.2 Requirements

In the following, a set of requirements is presented that are related to the two problems stated above.

R1: The approach should provide a mechanism for predicting defect-prone parts of a system to be tested. The approach is intended to allow focusing testing activities on those parts of a system that are expected to be defect-prone.

R2: The approach should provide a mechanism for predicting defect types to be tested that appear during testing. The approach is intended to allow focusing testing activities on certain defect types that are expected to show up.

R3: The approach should use early defect data from software inspections. In order to achieve R1 and R2, results (i.e., defect data) from software inspections should be used. Relevant inspection data has to be provided in a suitable manner. Furthermore, it has to be ensured that inspection data gained from a concrete context is reliable, i.e., a mechanism for checking inspection data has to be provided.

R4: The approach should be able to use historical defect data and other metrics that allow for focusing testing. In order to improve the prediction of parts of a system that are expected to be particularly defect-prone and of defect types that are expected to appear during testing, the approach should ensure that established concepts for predictions can be used in addition and can be combined with inspection results.

R5: The approach should make use of empirical evidence for focusing testing activities. Empirical evidence gained from a concrete context should be used by the approach in order to address uncertainty with predictions, adapt the predictions, and focus testing activities more appropriately.

R6: The approach should provide a mechanism for storing experience for later reuse. In order to make predictions in future quality assurance runs based on previous knowledge, it must be possible to store experience and gathered data. One established concept is called an experience base (Basili et al., 1994); a similar solution should be provided.

R7: The approach should be applicable during different lifecycle stages. According to existing software development models (e.g., V-model), and typical software lifecycle stages (e.g., requirements level, design level, code level, unit test level, integration test level, system test level), the approach should be flexible regarding its application during different lifecycle stages.

R8: The approach should be easily and efficiently integrated into existing quality assurance activities, respectively processes. The goal is a light-weight approach that can be applied in a new context with no or only little adaptation of existing inspection and testing activities, respectively inspection and testing processes. Though the approach might give certain recommendations regarding concrete inspection and testing techniques the approach would benefit from, no specific quality assurance activities or processes should be required, as long as inspection and testing defect data is collected. Consequently, better process integration of inspection and testing activities is expected.

R9: The approach should support adaptation to different contexts. Due to the fact that software inspections and testing are applied in a variety of different contexts with various requirements and goals, the approach should be easily adaptable to new environments.

Table 3 shows the relationships between the two stated problems and the derived requirements.

Table 3 Mapping of requirements and problems.

Requirements	**P1:** Inspection defect data is usually not used in a systematic manner to focus testing activities	**P2:** Quality assurance effort, and especially testing effort, is too high
R1: Prediction of defect-prone parts	x	x
R2: Prediction of defect types	x	x
R3: Make use of inspection results	x	
R4: Make use of historical defect data and further metrics	x	
R5: Make use of empirical evidence	x	
R6: Store experience for later reuse	x	
R7: Applicable during different lifecycle stages		x
R8: Able to integrate with different inspection and testing activities		x
R9: Adaptable to different environments		x

With respect to the first problem – inspection defect data is usually not used in a systematic manner to focus testing activities – the first two requirements state how testing activities may be focused and thus, how a new approach may provide a mechanism for supporting a focusing

activity for testing. In order to focus testing activities, a new approach must ensure that inspection results (i.e., defect data) are used explicitly. In addition, further concepts, such as historical defect data or further metrics, may enhance the focusing. Empirical evidence should be considered to improve the focusing. Experience gathered from R3-R5 should be stored in a database for later reuse.

With respect to the second problem – quality assurance effort, and especially testing effort, is too high – the first two requirements regarding a new approach address a way of to reduce effort. When defect-prone parts and defect types are predicted, testing activities can be focused better and thus, test effort may decrease, which may also lead to an overall reduction in effort. Furthermore, requirements R7 to R9 reveal that effort should be saved independent of lifecycle stages (requirement seven), concrete inspection and testing activities (requirement eight), and different environments (requirement nine). This means that the new approach should be as flexible as possible and the set of prerequisites should be as low as possible in order to be able to gain effort reductions with respect to a variety of different context factors.

2.5 Summary

This chapter presented an overview of the state of the practice regarding software inspections and testing. Both quality assurance activities are applied and established in industry, which was demonstrated in several environments. However, synergies between inspections and testing are often not exploited, i.e., those quality assurance activities are usually performed in isolation. Though inspection results may sometimes be used in practice in an informal way to control testing activities, no established concepts, methods, or approaches exist. A systematic approach that describes how a prioritization of testing activities based on inspection results can be done is still missing, i.e., no systematic and explicit approach could be found that describes how results from inspections can be used in a systematic manner to focus testing activities.

Two main problems were derived based on the state of the practice:

- Inspection defect data is often not used to focus testing activities, which leads to local inefficiencies.
- Quality assurance effort, especially testing effort, is often too high, which leads to global inefficiencies.

The connection between the static and dynamic quality assurance activities, i.e., inspection and testing techniques, seems intuitively clear

and obvious, but in practice this is often lost or obscured. The result is often poorly prioritized and redundant quality assurance effort.

Given that the state of the practice tends to assume that results from inspections will somehow magically drive the prioritization of testing activities, this thesis will make a contribution with respect to how this could be done concretely.

Based on the two problems, nine requirements were derived for an integrated approach. Two requirements deal with what should be focused on, three requirements refer to the data that should be used, one requirement explicitly demands storing experience, and three requirements are related to flexibility.

3 State of the Art

3.1 Overview

This chapter presents an overview of approaches that can improve quality assurance. Two main types of approaches are distinguished in this thesis:

- Approaches that combine static and dynamic quality assurance in order to improve quality assurance.
- Other approaches that do not combine static and dynamic quality assurance, but use alternative concepts to improve quality assurance, especially efficiency.

Two systematic mapping studies (Petersen et al., 2008) were conducted to identify relevant articles in a structured manner (Elberzhager et al., 2012b, 2012a). The basic results of these studies are summarized in the following in order to present the state of the art with respect to the scope of this thesis.

3.2 Combination of Static and Dynamic Quality Assurance

The results presented below are based on a systematic mapping study (Petersen et al., 2008), enhanced by some mechanisms from a systematic literature review (Kitchenham and Charters, 2007), such as considering quality criteria or using a protocol. The mapping study was conducted in order to identify existing approaches that combine static and dynamic quality assurance techniques (Elberzhager et al., 2012a). Based on the results of the mapping study, a clear definition of the research contributions could be derived.

On four electronic databases (Compendex, Inspec, ACM Digital Library, and IEEE Xplore) a comprehensive search term was applied, resulting in an initial set of 2498 articles and a total of 51 selected articles. Some articles of relevance that were found independent of the systematic mapping study are also mentioned in this section.

3.2.1 Classification

Different static and dynamic quality assurance techniques can be combined either by compiling or by integrating them.

In this regard, compilation means that different static and dynamic quality assurance techniques are applied to achieve a common goal, but in isolation, without using input from one analysis for the second one. Three sub-categories were identified. The first sub-category comprises the compilation of static and dynamic analyses, i.e., approaches explicitly using static and dynamic analyses in a combined approach were put into this sub-category. Approaches explicitly combining inspection (i.e., static QA) and testing (i.e., dynamic QA) techniques in a compiled manner comprise the second sub-category. Finally, other compilations of static and dynamic QA techniques represent the third sub-category.

In contrast, integration means that the output of one quality assurance technique is used as input for the second quality assurance technique. Two sub-categories were distinguished: first, the integration of static and dynamic analyses, and second, the integration of inspection and testing techniques.

Furthermore, besides the combination of concrete quality assurance techniques, approaches for selecting, combining, and evaluating different static and dynamic QA techniques can be helpful for finding an appropriate quality assurance mix in a given environment.

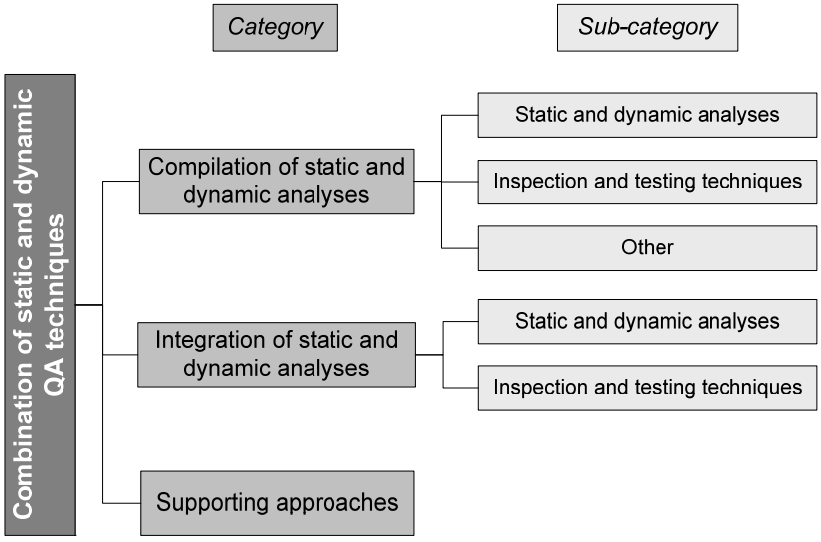


Figure 8 Classification of combined static and dynamic QA techniques.

Compilation

The first sub-category of compilation approaches comprises approaches that apply both static and dynamic analyses to improve a certain quality property, but without using results from each other. For example, Aggarwal and Jalote (Aggarwal and Jalote, 2006) proposed an approach

that combines a static and a dynamic analysis technique in a compiled manner in order to detect certain vulnerabilities. Other examples for the focus of such approaches are thread escape analysis (Chen et al., 2009), atomicity analysis (Chen et al, 2009b), protocol analysis (Gopinathan and Rajamani, 2008), or defects in general that should be found with such combinations (Anderson, 2008; Zimmerman and Kiniry, 2009). All these approaches are tool-supported, either by a combination of existing tools or by proprietary tool prototypes.

With respect to the combination of inspection and testing techniques in a compiled manner, many studies and experiments have been performed to compare them and discuss the advantages and disadvantages of both techniques. This is often substantiated by empirical studies and experiments that investigate which technique is superior to the other. However, in most cases the suggestion is made to combine inspection and testing techniques. So et al. (So et al., 2002) compared six different inspection and testing techniques, but usually, two or three techniques are compared to each other (Roper et al., 1997; Conradi et al., 1999; Andersson et al., 2003; Runeson and Andrews, 2003; Gupta and Jalote, 2007). For example, stepwise abstraction reading was compared to functional testing (i.e., equivalence partitioning and boundary value analysis) and structural testing (i.e., statement coverage) (Basili and Selby, 1987; Kamsties and Lott, 1995; Wood et al., 1997; Juristo and Vegas, 2003).

Furthermore, some empirical studies first performed inspections and then one or more testing activities and compared the overall effect when inspection and testing techniques are applied in sequence (Franz and Shih, 1994; Laitenberger, 1998; Iturbe, 1999; Berling and Thelin, 2003). Runeson et al. (Runeson et al., 2006) summarized the results of several experiments and case studies regarding the comparison of inspection and testing techniques. Moreover, Myers (Myers, 1978), Wood et al. (Wood et al., 1997), Jones (Jones, 2008), and Gack (Gack, 2010) already analyzed different combinations of inspection and testing techniques and calculated their benefit. It could be shown that in terms of cost and found defects, a mixed strategy often outperforms a strategy where only one technique is applied. Wagner (Wagner, 2006) investigated economic aspects of static and dynamic defect detection techniques and proposes using the gathered knowledge for an overall quality assurance model.

In some of the studies, defect classifications were used to analyze which kinds of defects can be found best by which quality assurance technique. Different experiments show that inspections and testing activities are able to find defects of the same defect types, meaning inspection and testing complement each other (e.g., Chaar et al., 1993; Kamsties and Lott, 1995; Laitenberger, 1998; Mantyla and Lassenius, 2009). In contrast, Runeson and Andrews (Runeson and Andrews, 2003) and Basili and Selby (Basili and Selby, 1987), for example, reported that inspection

and testing find different kinds of defects and showed which kinds of defects inspectors and testers find best.

Other combinations, which are also possible, are just briefly summarized next. For instance, inspection and testing techniques are combined with formal specifications, walkthroughs, or bug-finding tools (Wagner et al, 2005; Liu et al., 2009). Furthermore, comprehensive quality assurance processes from industrial environments are described, which comprise several inspection and testing techniques, requirements analysis, static analyses, walkthroughs, simulations, and tools (Duke, 1989; Ward, 1993; Chang et al., 1997). Chen et al. (Chen et al., 2008b) describe a view-based approach and combine this with inspection and testing techniques.

Advantages: One main motivation for applying different static and dynamic quality assurance techniques in sequence is to find more defects compared to using only a single defect detection technique. This is justified by the fact that different quality assurance techniques complement each other (Gilb and Graham, 1993). Moreover, different quality assurance techniques applied in several development stages are able to improve the quality of intermediate artifacts as well as that of the final product. More defects found before a product is delivered normally results in less rework costs.

Disadvantages: Applying different static and dynamic quality assurance techniques in sequence does not allow exploiting additional synergy effects, such as higher efficiency or effectiveness. Although it is often concluded that inspection and testing techniques should be combined, they are most often applied in isolation (i.e., applied in a compiled manner). The output of one technique is not used as input for another quality assurance technique. Thus, no additional value is gained when using different quality assurance techniques.

Integration

With respect to integration approaches, different static and dynamic analyses are integrated in order to reduce disadvantages of using them in a compiled manner. The integration of static and dynamic analyses encompasses most approaches. With respect to the order of application of static and dynamic analyses, most often static analysis is applied first, followed by dynamic analysis. However, some approaches use alternatives, for instance, using dynamic analysis first and static analysis afterwards (Jalote et al., 2006), performing static and dynamic analyses in an iterative way (Chen and MacDonald, 2008), or using dynamic analysis first, followed by static analysis and another dynamic analysis (Csallner et al., 2008). Static and dynamic analysis approaches grouped in this category focus on several vulnerability analyses (Centonze et al., 2007; Balzarotti et al., 2008; Godefroid et al., 2008; Kumar et al., 2009;

Hanna et al., 2009; Avancini and Ceccato, 2010), concurrent program analyses (Chen and MacDonald, 2008), defects in aspect-orientated programs (Massicotte et al., 2006), or on defects in general (Lucca and Penta, 2005; Csallner and Smaragdakis, 2005; Artho and Biere, 2005; Jalote et al., 2006; Joshi et al., 2007; Godefroid et al., 2008; Csallner et al. 2008; Chen et al., 2008a; Nori et al., 2009; Chebaro et al., 2010). One approach additionally supports debugging (Zhang et al., 2009). A lot of different tools and algorithms support these analyses.

Furthermore, approaches explicitly integrating inspections and testing techniques offer another way to combine static and dynamic QA techniques, for instance approaches using different scenario-based reading techniques during the inspection to derive test cases that can be used during testing (Chen et al., 2006; Winkler et al., 2005, 2010). Furthermore, another approach calls for inspecting automatically generated test code (Lanubile and Mallardo, 2007). Liu shows how executing paths from testing can guide inspectors in checking the tested paths as well as additional ones for defects (Liu, 2007). Finally, Harding (Harding, 1998) describes from a practical point of view how to use inspection data to forecast the number of remaining defects and how many defects have to be found and removed in each testing phase; i.e., the inspection results have an influence on the test exit criteria. Other defect prediction approaches that use inspection defect data are capture-recapture models (Petersson et al., 2004), detection profile methods (Briand et al., 1998), or subjective estimations (Emam et al., 2000; Biffl, 2000). While such approaches are mainly used to decide whether a re-inspection should be performed, a decision on how many tests to perform is also conceivable. However, no information on how such information can be used is currently available.

Advantages: The integration of static and dynamic quality assurance techniques can lead to additional defects being found. Furthermore, efficiency improvements are achievable. Moreover, a higher level of maturity in the defect results is gained since potential defects identified by one quality assurance technique are checked by the second one in order to confirm the findings.

With respect to the integration of static and dynamic techniques, focusing the dynamic technique by using output from the static one is sometimes observed, resulting in improved efficiency and effectiveness. Regarding the integration of inspection and testing techniques, test case derivation is improved and the quality of intermediate artifacts is enhanced.

Disadvantages: The integration of static and dynamic quality assurance techniques is often supported by tools and algorithms that are very specialized to a given context, domain, as well as other influencing factors such as programming language or defect types to be addressed.

Thus, it is usually not possible to apply them in different contexts without high adaptation effort, or not possible at all. Furthermore, a lot of the integrated approaches need tool support in order to be applicable, which currently is only given by initial tool prototypes.

With respect to software inspections, one main goal of past inspection research has been to improve the inspection itself rather than to integrate inspection and testing techniques, except for some approaches that use the inspection to support test case derivation or to predict remaining defects. However, support for focusing testing activities based on inspection results is still missing altogether.

Misc

Besides the combination of concrete quality assurance techniques, approaches for selecting, combining, and evaluating different static and dynamic QA techniques can be helpful for finding an appropriate quality assurance mix in a given environment. For example, Strooper and Wojcicki (Strooper and Wojcicki, 2007; Wojcicki and Strooper, 2007) present an approach that supports the selection of different QA techniques and suggest one possible combination based on various experiences. Another approach presents a framework for the balanced optimization of quality assurance strategies and proposes eleven steps to define a quality assurance strategy (i.e., selection of a mix of quality assurance techniques), to determine quality goals, to execute the defined strategy, and to analyze and package the results (Klaes et al., 2009). Moreover, Klaes et al. (Klaes et al., 2008a, 2008b, 2010a, 2010b) propose an approach that supports different QA management tasks, and various QA techniques (and even the combination of those) can be improved, planned, or controlled.

Advantages: Approaches mentioned in this category provide support for the selection and adaptation of static and dynamic quality assurance techniques. Moreover, support for controlling the quality assurance process is presented.

Disadvantages: In general, the approaches mentioned above focus on selecting quality assurance techniques for application in sequence (i.e., compilation approaches). Synergies between static and dynamic quality assurance techniques are neither systematically nor explicitly covered. These approaches provide a high-level view and usually do not focus on concrete quality assurance techniques that should be combined in a compiled or integrated manner.

3.2.2 Publication Years

Based on the systematic mapping study performed, the 51 identified articles describing combined approaches were ordered with respect to the year of publication. Though further approaches probably exist that could be classified, Figure 9 presents a rough idea of the development and publication of combined approaches during the last 25 years.

When analyzing the number of articles that were published in 5-year intervals, only between one and six articles were found to have been published in these timeframes until 2004. This shows that the focus regarding a combination of static and dynamic QA techniques was rather low during that time. However, between 2005 and 2009, 33 articles were published, which is twice as many as were published in the 20 years before. Consequently, this research topic has gained increased attention during the past five years and seems to be a promising research area for the future. The decrease in 2010 can be explained by the point in time when the mapping study was performed and probably does not reflect the real number of articles published both in and after 2010.

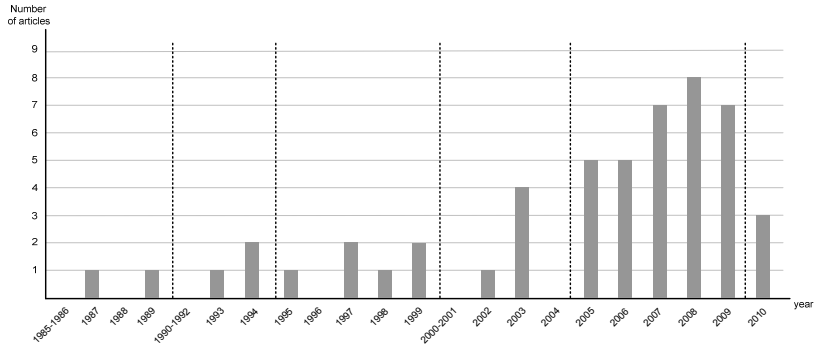


Figure 9 Number of articles published per year.

3.2.3 Evaluations

Based on the systematic mapping study performed, the 51 identified articles describing combined approaches were ordered with respect to whether they do or do not present evidence or no evidence (Figure 10).

First, articles were grouped into one of the following two categories: indirect combination or direct combination. Indirect combination means that the approach supports the selection of different static and dynamic quality assurance techniques or the article describes or analyzes different static and dynamic quality assurance techniques. Based on this, the suggestion is made to combine these techniques. For example, certain

inspection and testing techniques are described and compared with each other. Based on observations, such as inspection and testing techniques showing differences in effectiveness, the conclusion is made that it is most effective to combine them in a compiled manner. However, how this should be done and what the concrete benefit (e.g., in terms of effectiveness, efficiency, found defect types) of such a combination might be is neither described nor evaluated explicitly. Therefore, besides theoretical suggestions that more defects can be found when applying different static and dynamic quality assurance techniques, no concrete evaluation is done regarding the combined application or additional synergy effects resulting from the combination.

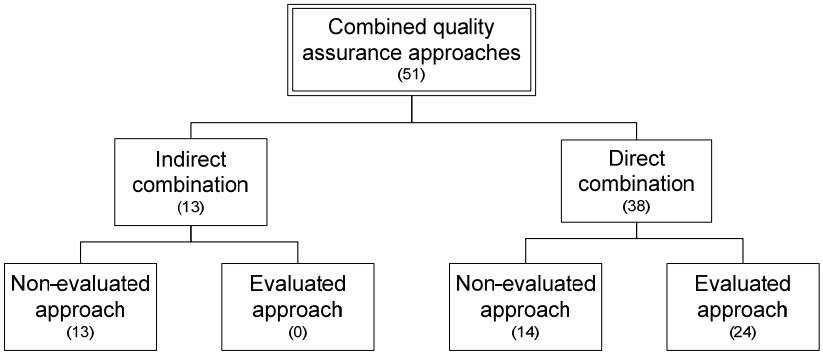


Figure 10 Number of articles that provide evidence, respectively no evidence.

In contrast, direct combination means that the combination of a static and a dynamic quality assurance technique is explicitly described. Two possible kinds of articles grouped in this category exist. First, the approach only explains how the combination could be done, either in a compiled or an integrated manner. In this case, no evaluation is presented. Second, beside the description of the combined approach, an evaluation is presented, either in a quantitative or qualitative way. Three different kinds of empirical studies were identified: experiments comprise about 60% (14 articles), case studies about 30% (8 articles), and experiences about 10% (2 articles).

Finally, Figure 11 shows the distribution of evaluated and non-evaluated combined approaches over the past 25 years. Again, the numbers of articles per year with respect to the combination categories “indirect” and “direct” are shown. The first approaches found only gave some indirect ideas for combinations or proposed a concrete combination without giving any evidence. The first evaluations were given with respect to a combination of different inspection and testing techniques in 1997 and 1998. After 2004, the number of proposed approaches increased and about half of the proposed combined approaches per year were also evaluated.

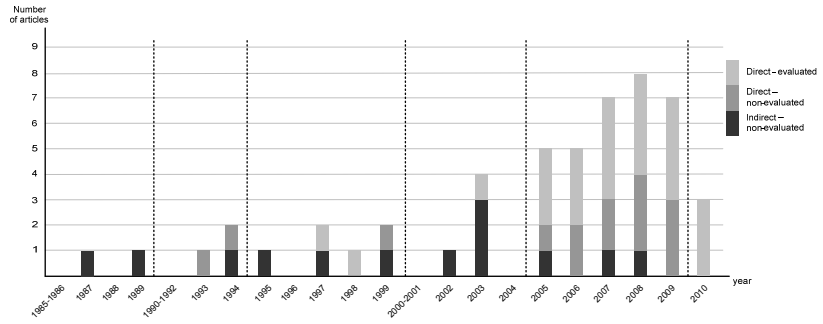


Figure 11 Numbers of evaluated and non-evaluated approaches per year.

3.2.4 Objectives

Based on the systematic mapping study performed, the 51 identified articles describing combined approaches were analyzed with respect to their objectives. However, only those articles were analyzed that were categorized as "direct combination" (see Section 3.2.3). The main reason is that the remaining ten articles, classified as "indirect combination", mainly focus on comparing inspection and testing techniques, comparing inspection and testing techniques with tools, comparing different tools, or supporting the selection and management of different quality assurance techniques. However, no concrete combined approaches are presented except for some initial suggestions. The objectives of these suggestions remain rather generic, such as general improvement of quality, or are not mentioned explicitly. Therefore, those ten articles were excluded when it came to analyzing the objectives of combined approaches.

Consequently, a total of 38 articles were taken into account. First, objectives that target the quality of the corresponding quality assurance process were considered. These objectives are either to improve the quality assurance process by applying a combined approach or to introduce a combined approach in a new environment. Figure 12 presents an overview of the identified categories and the number of articles per category. With respect to improvement, which is the most common objective, three different aspects were distinguished: improvement of coverage (13 articles), of effectiveness (22 articles), and of efficiency (11 articles). A lot of articles were classified into two or three categories, depending on which objectives should be achieved and which objectives were investigated with respect to the described approaches. Coverage was seen in two ways; first, as a measure of the proportion of a program being executed and tested by running a test suite (e.g., statement or branch coverage), and second, as coverage of requirements or use cases by executing test cases. Effectiveness is the ratio of total number of identified defects and total number of existing

defects in a quality assurance artifact. Finally, efficiency or cost effectiveness refers to the number of defects found per period of time. More than 80 percent of the corresponding approaches aim at achieving improvement objectives. In general, different approaches were found that describe how the improvements are achieved. For instance, by combining complementary static and dynamic quality assurance techniques, different kinds of defects are found and consequently, effectiveness is improved. Another idea is that the output of a static analysis directs the dynamic analysis to certain parts that can improve effectiveness and efficiency. However, the inspection results are not used to focus testing activities.

Eight more articles were classified as feasibility studies aimed at investigating whether a combined approach is able to detect defects in new environments, such as aspect-oriented software systems or web-based applications. Some articles were classified into more than one category (which explains the overall relative number of more than 100%)

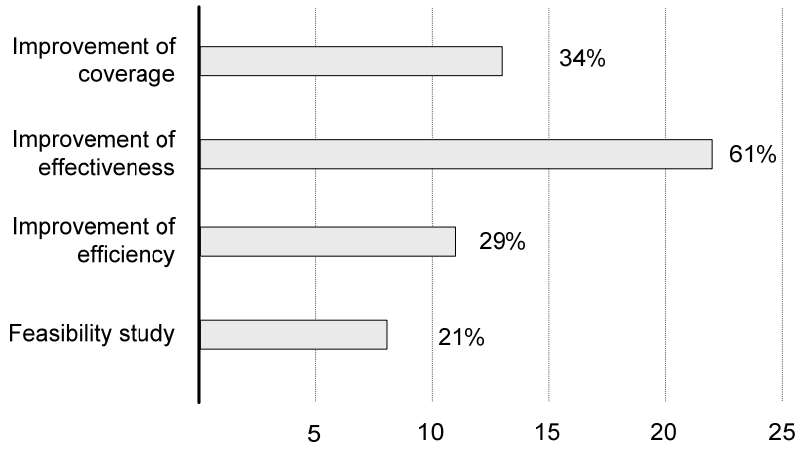


Figure 12 Numbers of articles with respect to quality assurance process objectives.

A second kind of objective of combined approaches are the defect types they address in order to achieve certain product quality objectives such as reliability, security, or safety. 23 articles do not mention any particular defect type, but the described approaches focus on finding defects in general, i.e., their objective is to improve the general reliability of the software product. 13 articles explicitly mention a concrete defect type that the combined approach focuses on, with security defects being the most common group. Finally, three more articles are classified as misc, covering not the quality of the final product, but the quality of intermediate artifacts such as design or test code. Figure 13 summarizes the articles with respect to the product quality objectives.

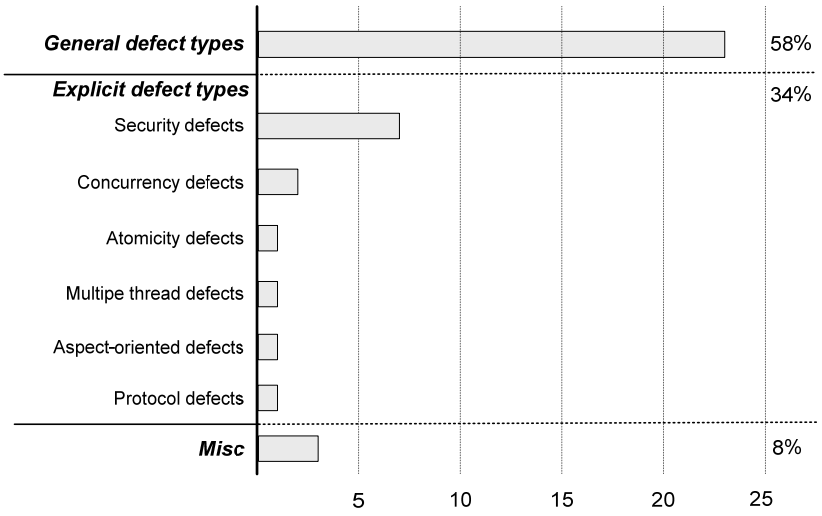


Figure 13 Numbers of articles with respect to defect types addressed by the combined approaches.

3.2.5 Summary and Conclusions

Based on the 51 articles identified in the systematic mapping study and some additional articles found independently, it can be concluded that compilation and integration are the two main approaches for the combination of static and dynamic QA techniques. Both categories comprise the combination of static and dynamic analyses and the combination of inspection and testing techniques. In addition, a mix of various static and dynamic QA techniques could also be identified for the compilation group. Finally, some other articles, categorized as misc approaches, were identified that could additionally support the combination of static and dynamic QA techniques. About 65% of the inspection and testing approaches are applied on the code level, and almost all static and dynamic approaches are applied on the code level.

With respect to the publication year, a tremendous increase of published articles started in 2005. More than 30 articles were published in the years 2005 to 2009, which is about twice as many as in the 20 years before 2005. Hence, the interest in this topic has received much more attention in the last few years, especially with respect to the integration of static and dynamic analyses. While some effort is put into the integration of static and dynamic analyses, inspection and testing techniques are currently mostly performed in an isolated manner.

Almost 50 percent of the 51 articles identified from the mapping study present evidence regarding combined approaches. The remaining articles describe just ideas on how a combination could be done or describe a combined approach concretely without giving any evidence. To some

extent, this can be explained by the fact that many combined approaches have emerged in the past five years and thus, have not been evaluated yet.

Two kinds of objectives could be extracted from the identified articles. On the one hand, there are QA process quality objectives, where improvement of effectiveness is the main goal to be achieved when applying a combined approach, followed by coverage and efficiency improvement. This could be achieved by using input from a static analysis to focus the dynamic analysis or by deriving test cases for testing during an inspection, for instance. However, inspection results are not used to focus testing activities or to prioritize certain parts of the system for testing. On the other hand, product quality objectives were identified. More than 50% of the approaches do not focus on a certain defect type and rather concentrate on improving reliability objectives. Furthermore, around one third of the approaches focus on specific defect types and thus, on specific product quality objectives. The remaining three articles focus on intermediate product quality aspects.

A common fact with respect to the results of a systematic mapping study or a systematic literature review is that it is affected by the researchers conducting the review, by the databases selected, by the search term selected, and by the timeframe chosen. Therefore, though these threats to validity are mitigated by several means (e.g., two researchers decided independently about whether to include and exclude articles, different databases were used, a comprehensive search term was used, additional articles were included), the articles found probably do not cover each existing article in this research area. However, based on the mentioned articles in this section, it can be concluded that the research area of combined static and dynamic analyses is getting increasing interest, whereas a focus on integrating inspection and testing techniques in a systematic way is still missing and not documented in the existing literature. Finally, the derived classification can be used to classify further articles, or it can be adapted if new approaches are found.

3.3 Non-Combined Approaches

Similar to Section 3.2, the results presented next are based on a systematic mapping study (Petersen et al., 2008), enhanced by some mechanisms from a systematic literature review (Kitchenham and Charters, 2007), such as considering quality criteria or using a protocol. The main goal of the mapping study was to identify approaches that improve test efficiency (Elberzhager et al., 2012b). Consequently, in this section, further approaches are considered that are able to improve the efficiency of testing. In this section, only approaches that do not combine static and dynamic quality assurance are considered, i.e., approaches other than those mentioned in the previous section (and

thus, articles found in the mapping study that either describe combined approaches or test strategy ones) are omitted here. Based on the results of this mapping study, it was decided to include valuable concepts that are easy to consider in the approach presented in this thesis.

On four electronic databases (Compendex, Inspec, ACM Digital Library, and IEEE Xplore) a comprehensive search term was applied, resulting in an initial set of 4020 articles and a total of 144 selected articles, of which 134 are considered to be relevant here.

3.3.1 Classification

Based on the systematic mapping study, three further kinds of non-combined approaches could be identified that improve test efficiency: test automation, prediction, and test input reduction.

Table 4 Number of articles per category.

Category	# articles	% articles
Test automation	71	53.0
Prediction	41	30.6
Test input reduction	22	16.4
Total:	**134**	**100**

Table 4 shows the number of articles found per category. It can be seen that the category of Test Automation contains the highest number of articles. Fifty-three percent of the articles were classified into this category, which comprises tool- and algorithm-supported approaches and tools that support different steps of a test process.

About 30% of the articles belong to the category Prediction. Such approaches support estimating the remaining defect content based on test defect data and consequently, allow determining when to stop testing. In addition, approaches that estimate parts of a system that are expected to be defect-prone were found. Such information can be used to focus testing activities.

The category Test Input Reduction comprises about 16% of all articles, and mainly includes approaches that select an optimal number of test cases from an existing test suite, i.e., the number of test cases is minimized while the number of defects found is maximized. In addition, test sequence reduction approaches and comparison studies were found.

Figure 14 presents an overview of these categories and sub-categories.

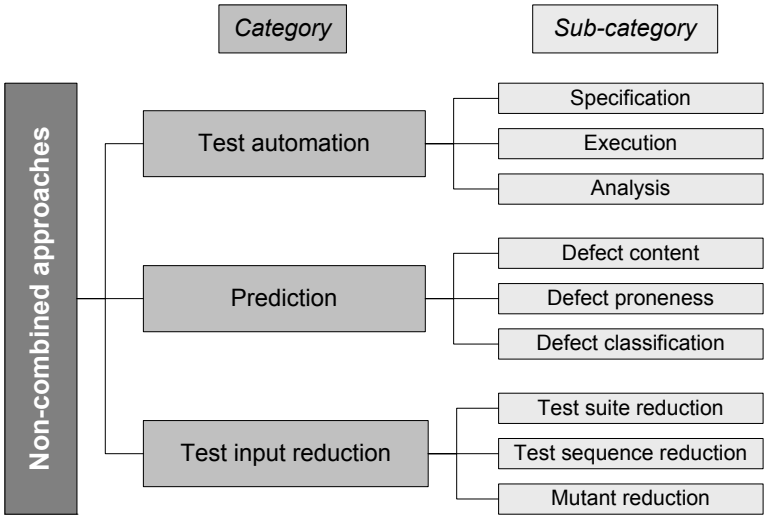


Figure 14 Classification of non-combined approaches that aim at improving efficiency.

Test Automation

Automation is the category where most articles were found. This is not surprising, as applying tools or automating certain steps in the test process can result in effort improvements due to, for example, reduced execution time of test cases, automated derivation of a set of test cases, or automated analysis of test results. The 71 articles put into the category automation were further classified with respect to the process step where they can be applied. Four steps are distinguished: planning, specification, execution, and analysis. Some approaches comprise support for more than one test process step.

For planning, which includes, for example, defining a test plan, no approaches could be found. 60 articles could be identified that support the specification phase, i.e., the definition and generation of test cases, test scenarios, test data, and test scripts. Different coverage criteria are considered. Moreover, different test bases are presented for deriving test cases, for instance, UML diagrams or textual descriptions.

With respect to the execution phase, 18 articles could be found that support running test cases or conducting regression testing. Different test levels were addressed, such as the unit, integration, or system level. Finally, five papers could be classified with respect to the analysis phase. Figure 15 gives an overview of the distribution of articles with respect to the four process steps.

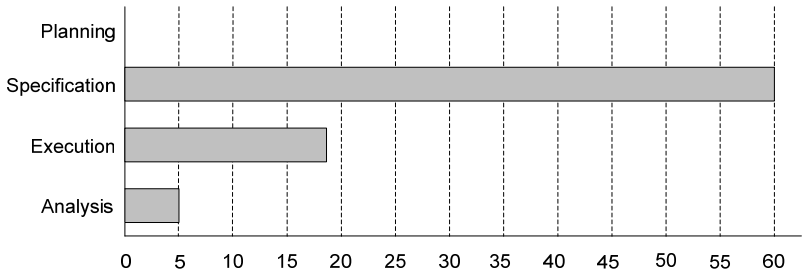


Figure 15 Distribution of articles in the category Automation.

Prediction

The category Prediction includes 41 articles (see Figure 16). Two main groups are distinguished: (1) prediction of the number of expected defects (i.e., defect content), most often based on software reliability growth models (SRGMs) in order to decide when to stop testing; (2) prediction of defect-prone parts of a system in order to focus testing on these parts. A large variety of concrete methods was found. In addition, one article focused on predicting defect types.

In more detail, 13 articles discuss the prediction of software reliability and the remaining number of expected defects. SRGMs attempt to predict software reliability using test data, which is collected during test execution. SRGMs try to correlate found defect data with known mathematical functions such as an exponential function. In the case of high correlations, the used function can be applied to predict future reliability behavior of the software under development, i.e., the number of remaining defects in the software can be predicted. This knowledge can help make a decision as to when to stop testing, which might improve the efficiency of testing activities. Various kinds of SRGMs exist, each considering different context factors and assumptions, such as experience of testers, immediate correction of defects, or unchanged code basis during testing.

Another group in this category covers approaches that predict defect-prone modules in order to focus testing activities on these parts, meaning that test effort can be saved because parts are prioritized for testing that are expected to be highly defect-prone. Twenty-six articles were classified into this category. Most of the articles use metrics such as size or complexity to predict defect-prone parts. Data from recently developed releases before and after delivery were considered, as well as historical data. Most often, the predictions are made on the code level; however, some approaches focus on the system level. Besides the results of the systematic mapping study, another overview of such approaches is given by D'Ambros et al. (D'Ambros et al., 2010), and some examples

can be found in Section 4.4.4.

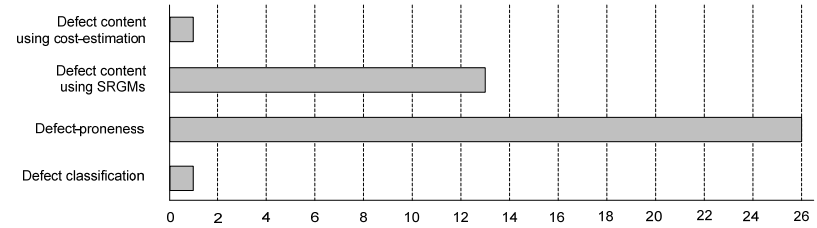


Figure 16 Distribution of articles in the category Prediction.

Test Input Reduction

Twenty-two articles belong to the category Test Input Reduction, which is sometimes also called test case or suite reduction, or test case selection. Based on an existing set of test cases (e.g., for regression testing), a reduced set of test cases should be chosen that finds the same number of defects as the complete set of test cases. Consequently, fewer test cases have to be executed, which results in reduced effort. Usually, specific coverage criteria are considered to determine the set of test cases. Furthermore, prioritization and ranking approaches are used considering certain criteria. Another approach uses finite state machine and reduction rules to reduce test sequences. Finally, one approach was found that reduces mutants during mutation testing.

In addition, two of the test suite reduction articles compare different regression test selection techniques. In these studies, the costs and benefits of five regression test selection techniques are investigated, which can support the selection of the most suitable one in a given environment.

Figure 17 gives an overview of the distribution.

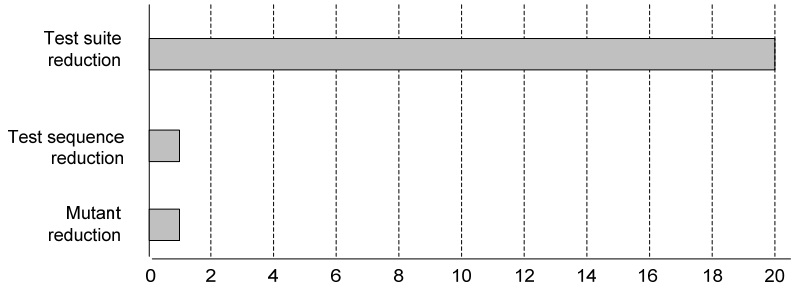


Figure 17 Distribution of articles in the category Test Input Reduction.

3.3.2 Publication Years

The 134 articles were arranged with respect to their publication years. Figure 18 shows an overview of the number of published articles per year, starting from 1991. It can be observed that until 2004, little attention was paid to non-combined approaches and methods that focus on test optimization; starting from 2005, a lot more articles about reducing testing effort were published. Therefore, it can be concluded that in recent years, testing optimization has been gaining an increased interest.

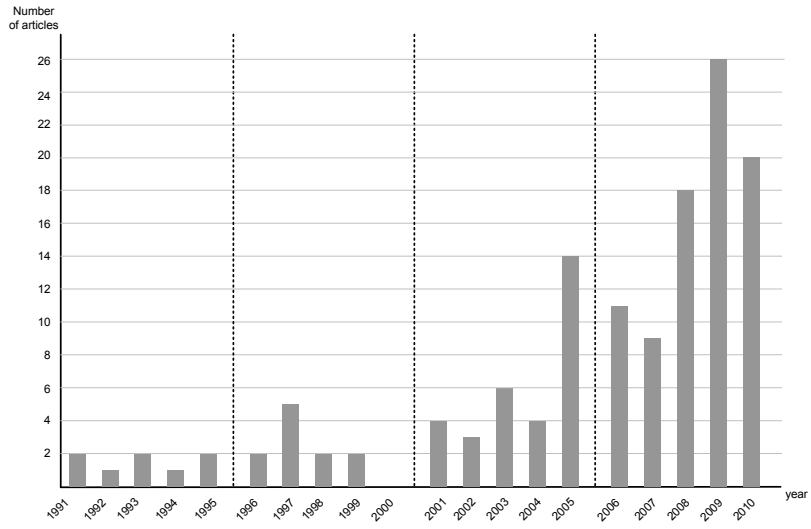


Figure 18 Number of articles published per year.

Taking a more detailed look with respect to the number of articles published per year and category, Table 5 presents an overview. Again, test automation and test prediction were of interest very early, while test input reduction received increased interest later. However, the same trend can be observed that the number of published articles and approaches increased during the past five years, which is also substantiated if we consider 5-year intervals.

Table 5 Distribution of articles by year and category.

	91	*92*	*93*	*94*	*95*	*96*	*97*	*98*	*99*	*00*	*01*	*02*	*03*	*04*	*05*	*06*	*07*	*08*	*09*	*10*	**Total**
Test automation	1	1	1		1	2	2	1			1		4	3	8	7	5	11	18	5	71
Prediction	1		1	1			1		2		2	3	2	1	5	2	1	6	5	8	41
Test input reduction					1		2	1			1				1	2	3	1	3	7	22
Total	2	1	2	1	2	2	5	2	2	0	4	3	6	4	14	11	9	18	26	20	**134**
%	6.0					8.2					23.1					62.7					**100**

3.3.3 Evaluations

The 134 identified articles were analyzed with respect to the degree of empirical evidence they present, i.e., whether the approaches are evaluated or not, and how they are evaluated. 97 articles present empirical evidence, and 37 articles do not present any evaluation results.

About half of the articles present information about the evaluation context, which was either industrial or academic. Most often, approaches were evaluated in an industrial environment. However, more than half of the 97 articles did not present clear information about the evaluation environment.

Furthermore, the evaluation method was extracted and four different kinds were distinguished: experience (i.e., impressions, opinions or subjective experiences with a certain approach), experiments, case studies, and empirical studies in general. Experiments and case studies were the two kinds that were found most often.

Figure 19 summarizes these findings.

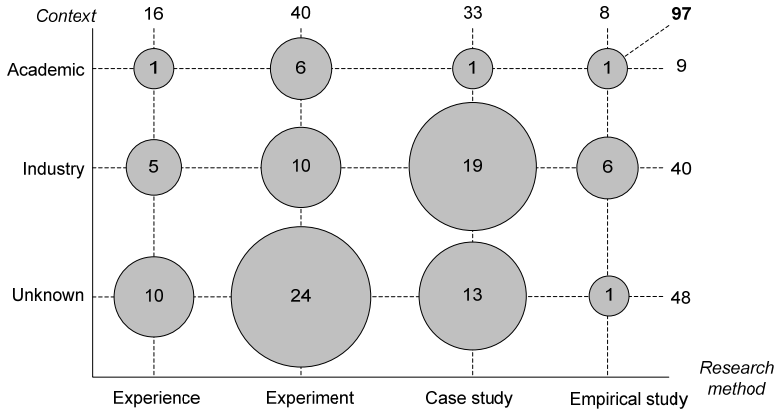


Figure 19 Evaluation scope and type of evaluation.

Compared to the general overview regarding evaluations, Table 3 presents more detailed results with respect to each of the categories. The first two columns describe the main category and the sub-categories. The next two columns present details about the evaluation context and the scope per category. For categorizing the research method, we considered the wording from the articles found, which might be designated incorrectly (Petersen et al, 2008). Furthermore, the sum of articles counted for all sub-categories may be higher than for the category itself because some articles are categorized into more than one sub-category (e.g., see category Test Automation).

Table 6 Detailed evaluation overview of non-combined approaches.

Category		Evaluation context				Evaluation method			
I	II	no	yes			experi-ence	experi-ment	case study	empiri-cal study
			industrial	academic	?				
Test automation	Preparation	24	6	4	26	8	15	10	3
Test automation	Execution	6	3	3	6	2	4	5	1
Test automation	Analysis	3	0	1	1	0	0	1	1
Automation		**28**	**9**	**4**	**30**	**9**	**18**	**13**	**3**
Prediction	Defect content	3	4	3	4	3	8	0	0
Prediction	Defect-proneness	3	23	0	0	2	4	14	3
Prediction	Defect classification	0	0	0	1	1	0	0	0
Prediction		**6**	**27**	**3**	**5**	**6**	**12**	**14**	**3**
Test input reduction	Test suite reduction	2	4	2	12	1	10	5	2
Test input reduction	Test sequence reduction	0	0	0	1	0	0	1	0
Test input reduction	Mutation reduction	1	0	0	0	0	0	0	0
Test input reduction		**3**	**4**	**2**	**13**	**1**	**10**	**6**	**2**

First of all, the category "test automation" containing the largest number of articles shows poor evaluation results. Only nine approaches were evaluated in an industrial setting, whereas 30 articles provide no information. Based on a scoring scale for relevance provided by Ivarsson and Gorschek (Ivarsson and Gorschek, 2011), the contribution can be considered as rather low. The category Prediction shows the most positive results, as 27 articles provide industrial evaluation results, most of them case studies or experiments. Most of them focus on prediction of defect proneness, i.e., on predicting areas where more defects are expected. One main conclusion is that metrics could be used to focus test efforts. However, no universal metric exists that fits best in all contexts, i.e., the best metrics have to be identified in each new context. Moreover, concrete data about particular effort savings are rarely provided. The category Test Input Reduction is similar to the Test Automation category, i.e., the context is often unclear and thus, conclusions can hardly be generalized and a lot more sound evaluations are necessary.

3.3.4 Summary and Conclusion

Based on a systematic mapping study, we extracted 134 articles that focus on improving test efficiency. Three main categories were identified: automation, prediction, and test input reduction. More than 50 percent of these articles propose Test Automation approaches. Predictions are the category containing the second highest number of articles. Prediction approaches, which, for example, determine when to stop testing or which modules or classes are defect-prone, can support better decisions regarding how much testing effort is required. The category Test Input Reduction comprises about 16 percent of all articles and includes mainly articles that present different test case selection and prioritization techniques, as well as methods for optimizing and reducing test suites.

For the performance of the systematic mapping study, the start year for including articles was set to 1991. Until 2004, less attention was paid to test optimization approaches and methods. Starting from 2005, more

articles about reducing testing effort were published. Furthermore, it can be concluded that in recent years, optimization of test efficiency received an increased interest. Articles about test optimization were published in numerous different journals, conferences, symposiums, and workshops.

About 70 percent of the approaches were evaluated, most often by means of experiments or case studies. Industrial environments were preferred to academic environments. However, more than half of the papers did not specify the evaluation scope. Moreover, though a large number of evaluations could be identified, their rigor and relevance seems to be rather poor based on our initial analysis, and many more sound evaluations are necessary to generalize the conclusions drawn. Hardly any effort reduction data were found.

In summary, it can be stated that numerous non-combined approaches exist that focus on improving test efficiency. Due to increasing time pressure in modern software development, it is expected that the topic will be of important interest in the next few years. Though not explicitly shown here, Elberzhager et al. (Elberzhager et al., 2012b) only found seven articles that consider early quality assurance activities (i.e., defect detection) before testing in order to improve testing. In addition, three more articles were found that describe test strategies.

A common issue with respect to the results of a systematic mapping study or a systematic literature review is that it is affected by the researchers conducting the review, by the databases selected, by the search term selected, and by the timeframe chosen. Therefore, though these threats to validity were mitigated by several means (e.g., two researchers decided independently about whether to include and exclude articles, different databases were used, a comprehensive search term was used), the articles found probably do not cover each existing article in this research area. However, the derived classification can be used to classify further articles or it can be adapted if new approaches are found.

3.4 Comparison

Sections 3.2 and 3.3 presented approaches that either combine static and dynamic quality assurance approaches or do not combine them, but are able to improve the efficiency of testing, among other goals. The different kinds of approaches were summarized and an evaluation was performed with respect to the strengths and weaknesses regarding specific requirements (see Section 2.4). Table 7 presents an overview of this evaluation.

Table 7 Assessing approaches with respect to determined requirements.

Requirements \ Approaches	*Combined*				*Non-Combined*		
	Static & dynamic QA (integration)	Static & dynamic QA (compilation)	Inspection & testing (integration)	Inspection & testing (compilation)	Test automation	Prediction	Test input reduction
R1: Prediction of defect-prone parts	o	-	-	-	-	+	-
R2: Prediction of defect types	-	-	-	o	-	o	-
R3: Make use of inspection results	-	-	o	o	-	-	-
R4: Make use of historical defect data and further metrics	-	-	-	-	-	+	o
R5: Make use of empirical evidence	-	-	-	+	-	+	o
R6: Store experience for later reuse	/	/	/	/	+	/	/
R7: Applicable during different lifecycle stages	o	o	+	+	o	+	o
R8: Able to integrate with different inspection and testing activities	o	o	+	+	o	+	-
R9: Adaptable to different environments	-	-	+	+	-	o	o

Obviously, none of the approaches fulfills all requirements. The prediction of defect-prone parts or defect types is only covered strongly by the prediction approaches. Most of the other approaches do not provide any mechanism or give only very limited guidance. Inspection results are rarely used, which is similar with respect to using historical data or further metrics. This is only done in the prediction approaches and, to some extent, in some other approaches. Empirical evidence is used in some approaches, especially in the prediction area and in the compilation of inspection and testing approaches.

Besides test automation approaches storing, for example, test cases for performing regression tests and their results, most of the approaches do not explicitly store experience. However, experience may be stored independent of the concrete approach.

Many of the approaches can be applied during specific phases of the development lifecycle and are adaptable to different contexts. However, tools are often specific to a certain environment, which also holds for static analyses. Integration with inspection and testing approaches is, of course, possible with such static and dynamic techniques, but also

conceivable with different approaches to improve overall quality assurance.

As prediction approaches facilitate the fulfilment of many requirements, this concept is partly integrated in the integrated approach presented in this thesis. A new integrated inspection and testing approach will be introduced in the next section, since it was demonstrated here that the existing approaches do not fulfil all stated requirements. Inspection defect data is usually not considered in the existing approaches for improving succeeding testing activities, though it cannot be completely excluded that such an approach is already being used in an ad-hoc fashion in industry. However, a defined approach was not found based on the literature review and mapping studies conducted.

3.5 Summary

This chapter analyzed two kinds of approaches that are able to improve efficiency, among other improvement goals.

On the one hand, combined approaches were considered, i.e., approaches that combine static and dynamic quality assurance activities. It could be shown that such approaches attained increased interest in recent years. Besides a general combination of different static and dynamic quality assurance techniques such as symbolic execution, testing, and runtime analysis (Godefroid et al, 2008), theorem proving, test case derivation and execution (Csallner and Smaragdakis, 2005), or model checking and model-based testing (Chen et al., 2008a), a combination of inspection and testing was often suggested by different authors. However, in most cases, inspection and testing are applied in sequence without deep integration to exploit additional synergy effects. Furthermore, if inspection and testing are integrated, they support test case derivation or prediction of remaining defect numbers based on the inspection results, but no usage of inspection data for focusing testing activities.

On the other hand, non-combined approaches were identified that are also able to improve efficiency, but do not use input from static quality assurance activities. These approaches are automation, test input reduction, and prediction; they provide some valuable input for the integrated inspection and testing approach presented in this thesis.

Based on the comparison of the identified combined and non-combined approaches with respect to the stated requirements, it could be shown that none of the approaches fulfills all relevant requirements, which substantiates the need for an alternative approach.

4 The In^{2}Test Approach

4.1 Overview

This chapter presents the integrated inspection and testing approach In^{2}Test and details the concept of assumptions. Section 4.2 gives an overview of the solution ideas, emphasizes the contributions, and reflects on the requirements with respect to the new approach. Section 4.3 describes the basic process of the approach, and shows different ways in which the approach could be applied. Section 4.4 gives more details on how the focusing is done. For this, the concept of assumptions that cover the knowledge about relationships between inspections and testing is described in detail, i.e., a model of assumptions is given, a way to derive and evaluate them is presented, and exemplary assumptions are shown. Section 4.5 describes initial tool support for the approach. Section 4.6 discusses the limitations of the approach. Finally, section 4.7 summarizes this chapter.

4.2 Solution Idea

Table 7 in Section 3.4 showed all state-of-the-art approaches identified and discussed in Chapter 3, and evaluated them with respect to the requirements as stated in Section 2.4. As can be seen, the approaches have different strengths and weaknesses, which is denoted by "+", "o", and "-". None of the approaches fulfills all requirements. Most of the approaches do not predict defect-prone parts or defect types in order to focus testing activities, with the exception of explicit prediction approaches. Moreover, inspection data is rarely used, both for combined and for non-combined approaches. Instead, some approaches use alternative data, such as historical data or certain metrics. Most of the approaches do not package their results explicitly. Finally, many approaches are adaptable with respect to lifecycle phases and different environments. A solution that fulfills all these requirements is necessary in order to provide an integrated inspection and testing approach that is able to focus testing activities. Thus, the following contributions are made in this thesis:

1. *Predicting defect-prone parts and defect types (R1, R2).* Except for explicit prediction approaches, combined and non-combined approaches do not focus on predicting defect proneness or defect types that are likely to appear. Thus, focusing quality assurance activities such as testing is often not supported. Assumptions are usually used in order to allow making

predictions, which can then be used to focus, e.g., testing activities. This thesis provides a structured model for assumptions, explains how to derive and evaluate them, and provides a set of assumptions with respect to relationships between inspection and testing. These assumptions are able to predict defect-prone parts and defect types.

2. *Using inspection results, historical data, and further metrics, and using empirical evidence (R3, R4, R5).* First, approaches that combine inspection and testing generally use inspection results to predict the number of remaining defects or to support test case derivation. However, such inspection data is usually not used systematically to make predictions (see R1 and R2). Thus, this thesis provides an approach that makes explicit use of inspection results that are available early (i.e., before testing is conducted). Second, prediction and test input reduction approaches often use historical data and certain metrics, such as size or complexity. Consequently, the approach presented in this thesis is also able to consider metrics and historical data, and to integrate this kind of input with inspection results. Finally, empirical evidence is necessary describing valid knowledge about the relationships between the techniques and methods that are applied. A lot of such evidence is known with respect to compiled inspection and testing approaches, as well as for prediction and test input reduction approaches. However, with respect to the integration of inspection and testing, such evidence is rare. Thus, this thesis provides some initial insights from the evaluations conducted.

3. *Storing experience for later reuse (R6):* Most of the approaches do not explicitly provide a mechanism that forces a user to store knowledge and data gained in order to reuse and improve the approach in subsequent applications. Thus, this thesis explicitly considers a mechanism that allows storing and reusing gathered experience and different kinds of data (e.g., defect data).

4. *Adaptation to different lifecycle phases and environments, and integration of different analytical quality assurance activities (R7, R8, R9).* First, many approaches support adaptation to different lifecycle stages, i.e., approaches can often be applied at certain levels, such as requirements, design, code, or different test levels. The concrete applicability depends on the concrete approach. Consequently, in order to provide the broadest possible applicability of the integrated approach presented in this thesis, different lifecycle stages should be supported. Second, many approaches can be applied in combination with different inspection and testing techniques in order to support

an overall quality assurance strategy. However, this is often done in a compiled manner (i.e., applying one quality assurance technique after the other without interchanging data between them or exploiting further synergy effects). For the approach presented in this thesis, no complete process changes are required, and the integrated inspection and testing approach is easy to apply with respect to different inspection and testing activities, i.e., the approach is able to use inspection and testing techniques already being applied. Third, some approaches, especially inspection and testing approaches, can be easily adapted to new environments. The integrated inspection and testing approach, which uses established inspection and testing techniques, is developed in a way that supports its application in different contexts.

Table 8 Composition of the In²Test approach.

Requirements \ Approaches	*Combined*				*Non-Combined*			Concepts that were slipped or considered in the In²Test approach
	Static & dynamic QA (integration)	Static & dynamic QA (compilation)	Inspection & testing (integration)	Inspection & testing (compilation)	Test automation	Prediction	Test input reduction	
R1: Prediction of defect-prone parts								Using assumptions for prediction
R2: Prediction of defect types								Using assumptions for prediction
R3: Make use of inspection results								Explicitly considered
R4: Make use of historical defect data and further metrics								Using certain metrics and historical data
R5: Make use of empirical evidence								Using initial input for assumptions
R6: Store experience for later reuse								Explicitly considered
R7: Applicable at different lifecycle stages								Considering different lifcycle phases
R8: Able to integrate with different inspection and testing activities								Considering light-weight approach using established inspection and testing activities
R9: Adapatable to different environments								Considering different contexts

influenced by alternative approaches
explicitely new in the In²Test approach

Table 8 shows which concepts were partly considered from established approaches (indicated by light-gray boxes), which basically includes using initial empirical knowledge to make predictions, and using metrics and historical data. The In²Test approach is the first systematic approach that explicitly integrates inspection and testing techniques. The approach uses inspection defect data to focus testing activities, which can be further supported by metrics and historical data. In order to be able to predict defect-prone parts and defect types, a model for assumptions is defined. Furthermore, experience and data are explicitly packaged. Flexibility and adaptability to different environments are provided. Finally, an initial tool prototype supports the In²Test approach.

4.3 Process

The main idea of the **in**tegrated **in**spection and **test**ing approach In^{2}Test is to use defect information from the inspection (i.e., a defect profile comprising quantitative defect data and defect type information) to focus testing activities on specific parts of the system under test and on specific defect types. On the one hand, the inspection defect profile can be used to prioritize parts of the system under test that are expected to be most defect-prone. On the other hand, the inspection defect profile can be used to prioritize those defect types that are expected to show up most often during testing activities. Consequently, testing activities are focused on such prioritized parts or defect types. The In^{2}Test approach builds upon existing inspection and testing techniques.

In order to focus testing activities, it is necessary to describe the relationship between defects found in the inspection and the remaining defect distribution in the system under test. The same is true for defect types. For that reason, assumptions are explicitly defined. An assumption could be, for example, that code classes in which many defects are found during the inspection are expected to contain additional defects, which can then be found during testing activities, i.e., an accumulation, respectively Pareto distribution, of defects is expected (of course, such an assumption needs to be context-specific). In order to be able to rely on defined assumptions, they should at least be grounded on explicitly described hypotheses. Ideally, assumptions to be applied in a concrete context are based on empirically validated hypotheses that are valid in the given environment. If such evidence is not available, assumptions have to be described explicitly and analyzed with respect to their suitability for the specific context (e.g., a post-testing analysis could show if an assumption was wrong), i.e., each assumption has to be validated in the given environment in order to be able to decide if the assumption leads to valuable prioritizations or not.

As a next step, the assumptions need to be quantified if they are not already defined in measurable terms. This means that concrete metrics need to be derived that make the assumptions measurable. For instance, the number of defects detected in a code class could be measured as defect content (i.e., absolute number of defects found) or defect density (i.e., absolute number of defects found divided by lines of code).

In order to allow an assumption to be applied, it has to be operationalized. Consequently, so-called selection rules are derived from assumptions. One selection rule for the above-mentioned assumption could be, for example, that those code classes should be selected for a testing activity that contain more than eight major defects (i.e., measured as defect content) based on the inspection defect profile. When prioritizing defect types for a testing activity, an exemplary

selection rule might be "Select those two defect types that are found most often with the inspection." In this case, it has to be ensured by the defect classification that the defect types can be found by both inspection and testing activities.

Based on such a prioritization, a precise focus on code classes only or on code classes and defect types for those code classes can be determined. Consequently, focused testing activities can be conducted. Overall, an effort reduction is expected due to testing only those parts of a system under test that are expected to be most defect-prone and checking only those defect types that are expected to appear most often, instead of testing all code classes and concentrating on all defect types.

In addition to the inspection defect profile, metrics and historical data, which are established concepts, can support the prioritization, provided they are combined with the inspection results to overcome the problems arising if they are used in isolation, and can give additional valuable hints for the prioritization. For instance, an assumption could be: "For code classes with a high complexity value in combination with high defect density based on inspection results, a high probability exists that further defects will be found during testing within such code classes". Metrics could be defined and selection rules could be derived accordingly.

4.3.1 One-Stage Approach

In addition to the general description of the approach given above, Figure 20 shows a concrete application for code inspection and testing.

First, the inspection (step 1) has to be performed. No specific inspection technique, respectively process, is prescribed, and the approach is flexible in such a way that different inspection processes can be used. However, it has to be ensured by the inspection process used that a suitable number of defects are found. The inspection defect profile (containing quantitative defect data, for instance the number of inspection defects found per class or per defect type) is obtained after the inspection. In some cases, additional information is gathered, such as metrics or historical data (e.g., defect data from different testing phases, defects found after testing), which is stored in an experience database, EDB for short. In order to be able to rely on the inspection data, the inspection results have to be monitored (Barnard and Price, 1994; Aurum et al., 2002). Thus, inspection quality monitoring (step 2) should be performed in order to analyze and determine the quality of the inspection results, which is done by comparing context-specific historical data and certain inspection metrics, such as reading rate or number of defects found per inspector. If no historical data is available, data from the literature can be used initially.

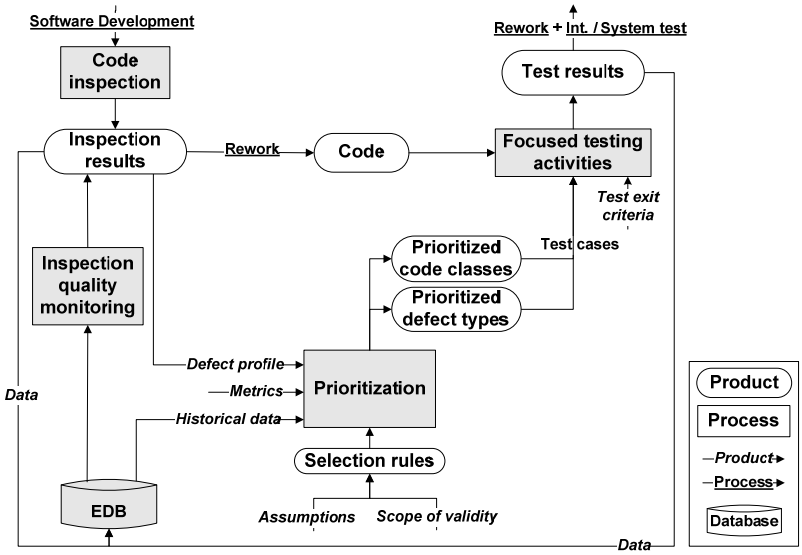


Figure 20 Overview of the integrated approach.

Next, prioritization is done (step 3). Certain assumptions, respectively selection rules, can be applied to prioritize parts of the code or defect types. Figure 21 gives a conceptual overview of the prioritization steps based on assumptions and selection rules (on the left); two simplified examples are sketched on the right.

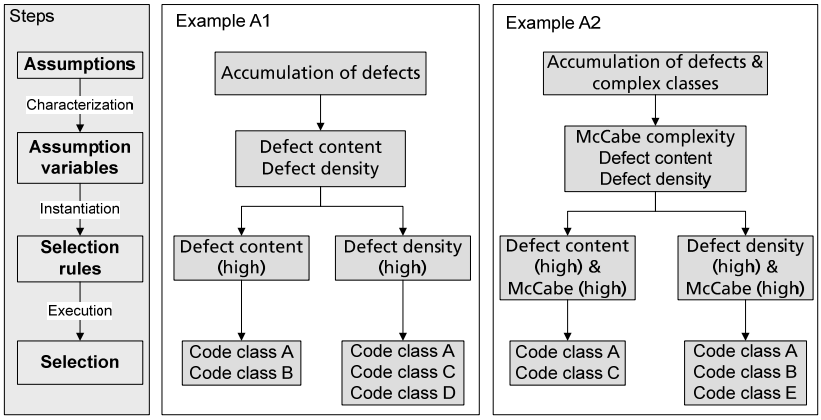


Figure 21 Conceptual overview of steps and two examples of the prioritization of code classes.

The exemplary assumption A1 claims that parts of the code where a significant number of inspection defects are found indicate remaining defects to be found by testing (i.e., an accumulation of defects, respectively a Pareto distribution of defects, is assumed). It is particularly based on the empirical observation that a high number of defects are

often contained within a small number of modules (Boehm and Basili, 2001). The number of defects can be expressed as defect content (absolute number of defects) or defect density (absolute number of defects divided by inspected lines of code). According to assumption A1, code classes that have high defect content based on the inspection results are selected for testing. The second selection rule chooses code classes for testing that have high defect density based on the inspection results. In both cases, it has to be clarified what "high defect content" and "high defect density" means in a concrete environment, i.e., a concrete metric and thresholds have to be defined. For instance, a selection rule could be stated as follows: "Focus testing activities on those parts of a system where an inspection has found more than 15 defects". An alternative example that would also fulfill the assumption is "Focus testing activities on code classes where an inspection has found more than 8 major defects". Selection rules to be chosen depend on the available and analyzed data from the concrete context.

The exemplary assumption A2 claims that parts of the code where a significant number of inspection defects are found and which are complex indicate remaining defects to be found by testing. This assumption uses the inspection defect profile and one complexity metric, expressed as McCabe complexity. Consequently, two concrete selection rules combine defect content with McCabe complexity on the one hand and defect density with McCabe complexity on the other hand, resulting in different code classes being selected. The selection rules may prioritize different code classes to be tested, which depends on the concrete context. A third example (not shown here) could be to focus only on certain defect types based on the inspection defect profile, and to prioritize them for testing.

Ideally, evidence has already been obtained in a concrete environment regarding which selection rules lead to the best selection of code classes (i.e., highest effort reduction at a comparable quality level). The approach does not define fixed values as to what high defect content means, for example. This is highly dependent on certain context factors and thus, has to be defined in the environment at hand before the approach is applied. Finally, established techniques for deriving concrete test cases for the code classes can be used, such as equivalence partitioning or boundary-value analysis.

The concrete selection rules chosen also depend on explicit context factors. For example, consider the number of available inspectors and time as two context factors. If only one inspector is available, who has to inspect certain parts of a system within a limited period of time, fewer parts can be inspected. Consequently, more effort should be spent on testing activities. Another example: Consider the experience of the inspectors as a context factor. If the inspectors' experience is low, it is expected that not many critical defects will be found. Consequently, the

inspected parts should be tested again. In contrast, if the inspectors' experience is high, it is expected that most of the defects will be found before testing, and the inspected parts can be skipped for testing (in this example, it is assumed that both inspection and testing activities are able to find the same defects). Again, it depends on the concrete environment which selection rules are chosen.

The context that has an influence on the definition or selection of selection rules, and the level of confidence (i.e., validity) of the selection rules applied in the given context are summarized as the scope of validity.

Based on the prioritization, testing of the selected code classes (step 4) can be conducted. The test focus may also have an influence on the test exit criteria (i.e., when to stop testing). Again, historical context-specific data from the EDB can give valuable hints on when to stop testing. In addition, established criteria such as branch coverage can be considered. However, test exit criteria and their improvement are not in the scope of this thesis.

Defect results from the current testing activities have to be analyzed continuously in an ideal case. If more defects are found in the selected parts, the assumptions appear to have been valid and the test activities can continue focusing on the prioritized parts. If not, the assumptions have to be adapted and another prioritization has to be performed (i.e., a re-direction of the test process is conducted). Furthermore, parts that were not prioritized can be tested in order to check if they are defect-free, which leads to stronger empirical evidence for the assumptions.

In case no continuous analysis is possible, at least a retrospective analysis of the assumptions should be performed. If the assumptions were correct, data gathered during inspections and testing should be packaged in an experience database and used for future prioritizations. Otherwise, an analysis of the assumptions and the context has to be performed in order to identify reasons for the deviations.

4.3.2 Two-Stage Approach

Figure 22 presents the application of the two-stage approach for the coding phase. The steps described for the one-stage approach remain similar, with an adaptation of the prioritization step. From the beginning, the process starts with a code inspection (step 1), leading to inspection results that are used to derive the inspection defect profile. Such a defect profile can contain, for example, the absolute number of defects found per code class, the relative number of defects found per code class, or the number of defects found per defect type. An implicit requirement

here is that a suitable number of defects has to be detected in order for the defect profile to be meaningful.

Additional metrics can be gathered (e.g., size or complexity metrics of code classes) or historical data (e.g., defect data from different testing phases, defects found after testing) from an experience database (short: EDB) can be considered. Next, step 2 consists of an inspection quality monitoring where the inspection results are checked to ensure that they can be relied on for the prioritization (Aurum et al., 2002). This can be supported by historical, context-specific inspection data and metrics derived from them (e.g., number of defects found per inspector, lines of code inspected per inspector). If historical inspection data is not available, recommendations from the literature can be used as an initial basis (Barnard and Price, 1994).

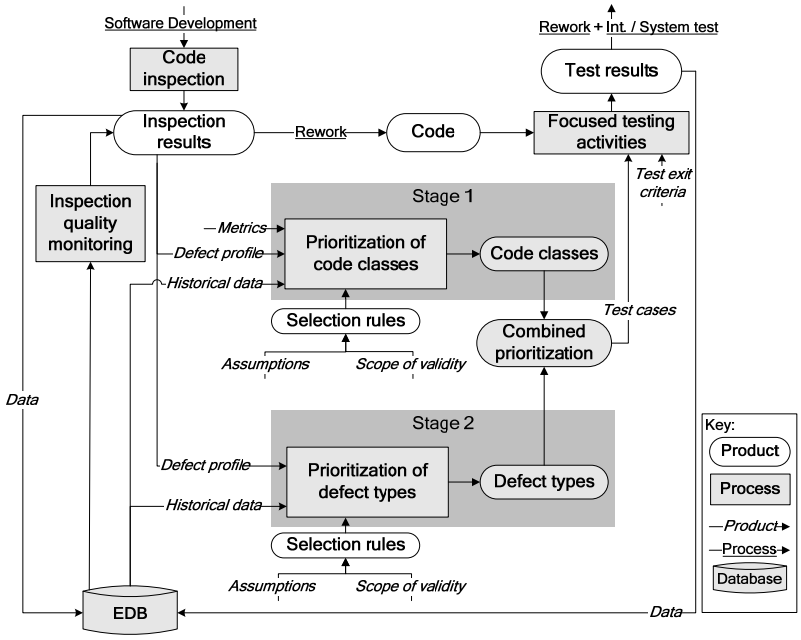


Figure 22 Integrated two-stage inspection and testing approach for focusing testing activities.

In order to be able to focus testing activities, two-stage prioritization has to be conducted, which represents step 3. For this, assumptions and refined selection rules are used to prioritize certain parts of the code and defect types. Figure 23 shows in detail how the prioritization is done and gives exemplary assumptions, selection rules, and the result of the combined prioritization.

Stage 1: First, prioritization of code classes is performed. Consider the assumption that additional defects remain in those code classes in which

a significant number of inspection defects are found and thus, testing activities should be focused on those code classes. This means that an accumulation of defects (i.e., Pareto principle) is assumed, as stated, for example, by Boehm and Basili (Boehm and Basili, 2001). However, to be able to apply the assumption 'accumulation of defects', it has to be operationalized by a definition of concrete metrics, i.e., assumption variables have to be defined. For the above-mentioned assumption, two examples of concrete metrics are defect content (i.e., absolute number of defects found per code class) or defect density (i.e., absolute number of defects found per code class divided by lines of code). Based on the general assumption and the assumption variables, two concrete selection rules can be derived. The application of the selection rules results in the selection of different code classes, i.e., based on the inspection defect profile, different code classes may be prioritized for testing activities.

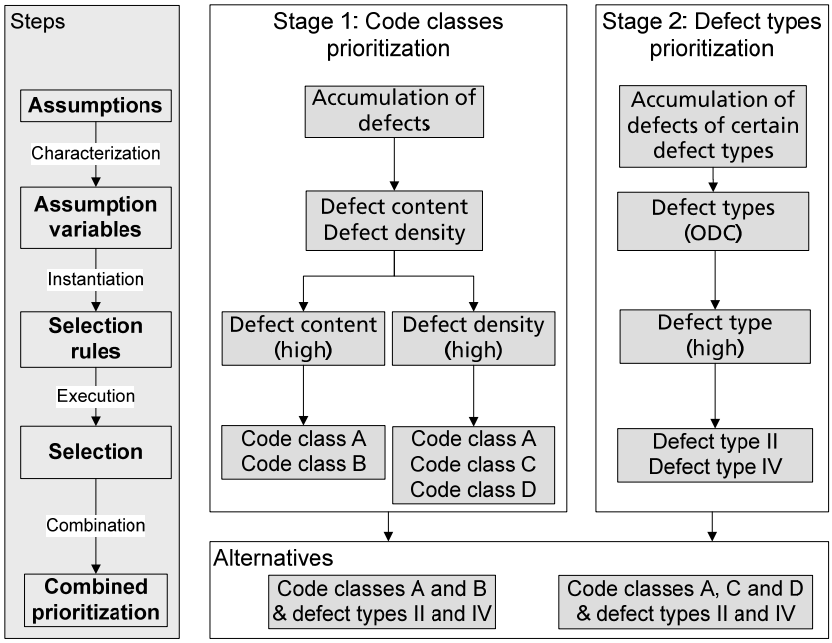


Figure 23 Conceptual overview of the steps for conducting the combined prioritization and two examples.

Stage 2: Second, prioritization of defect types is performed. The steps are the same as for stage 1. An exemplary assumption is that defect types that appeared most often during the inspection are expected to appear again in the testing activities (i.e., an accumulation of defects of certain defect types is assumed). In order to be operational, a concrete defect classification has to be selected that is able to cover defect types found by both inspection and testing activities. One concrete selection rule is instantiated, which, in this example, results in the selection of two defect types based on the inspection defect profile.

Again, the context that has an influence on the definition or selection of selection rules, and the level of confidence (i.e., validity) of the selection rules applied in the given context are summarized as the scope of validity.

Finally, the prioritization results of both stages are combined in order to focus the testing activities on (i) certain code classes and (ii) on defect types to look for within these code classes. For this, each selected set of code classes from a selection rule of stage 1 is combined with each selected set of defect types from a selection rule of stage 2. In the example, the result is two combined prioritizations. Ideally, evidence is already obtained on which selection rules lead to appropriate prioritizations of code classes and defect types in a given context. In this case, the most appropriate combined prioritization(s) can be chosen. Otherwise, different selection rules have to be applied and the applied combined prioritizations have to be analyzed in a post-testing analysis. Finally, test cases have to be derived for the selected code classes (e.g., using established techniques such as equivalence partitioning) respectively to cover selected defect types.

Step 4 comprises focused testing activities, which also include determining when to stop testing, which may be influenced by the prioritization. Finally, data from the code inspection and the testing activities have to be stored in the EDB for future analysis.

Defect Classification

Several defect classifications have been developed, such as defect classifications used in experiments for comparing inspection and testing defects (Basili and Selby, 1987; Kamsties and Lott, 1995; Juristo and Vegas, 2003) or defect classifications developed by industry companies, such as ODC by IBM (ODC, 2002). With regard to the question of whether inspections and testing are complementary QA activities, i.e., the question of whether they will find different kinds of defects or not, the results from experiments and case studies are not consistent. Laitenberger (Laitenberger, 1998) concluded that they do not complement each other. Different experiments have shown that inspections and testing activities are able to find defects of the same defect types (e.g., Chaar et al., 1993; Kamsties and Lott, 1995; Mantyla and Lassenius, 2009). In contrast, Runeson and Andrews (Runeson and Andrews, 2003) showed that inspectors and testers find different kinds of defects. Jalote and Harahopal (Jalote and Harahopal, 1998) showed during an experiment that especially interface and logic defects can be found by inspection and testing techniques, whereas maintainability and portability problems are solely found by inspections.

Some defect types might only be found by either an inspection activity or by a testing activity. Thus, a defect classification has to be chosen

carefully if defects from different QA activities are to be classified, especially if testing activities should be focused on specific defect types based on classified inspection results.

The approach assumes that both the inspection and the testing activities can find the same defect types, which might be true for only some defect types. However, future evaluations have to be performed in order to offer more detailed prioritizations with respect to defect types.

4.4 Relevance of Assumptions and Context Factors

In order to conduct focused testing activities when applying the integrated inspection and testing approach, knowledge regarding the relationships between inspections and testing is required. Such relationships are usually context-specific and not generally applicable. Therefore, it is necessary to check whether reliable evidence regarding such relationships exists in a given context (e.g., stored in an experience base (Basili et al, 1994)). If such evidence does not exist, assumptions need to be made regarding relationships between the processes to be considered. An example assumption might be that the distribution of defects found regarding certain defect types is similar for inspection and testing for the same artifact. Therefore, it might be beneficial to use the defect distribution from inspections for creating the test cases. Assumptions that describe certain relationships can initially be taken from the literature or from different contexts, but need to be analyzed with respect to their validity in the given context. Evidence regarding defined assumptions can be gathered in different ways (e.g., analytically, empirically), and has to be continuously reevaluated and updated due to the fact that context factors can change and thus, assumptions initially defined and proven to be correct can become wrong.

However, the benefits achieved depend on knowledge regarding the relationships between inspection and testing processes, especially knowledge regarding the distribution of defects in inspections and testing. If such knowledge is available, it can be used to balance inspection and testing activities or to focus testing activities based on inspection results. For instance, using the assumption stated before that both quality assurance activities mainly find defects of the same defect types, testing activities may be focused on those defect types that inspection has primarily found before. Or consider the assumption of a Pareto distribution for defects found; then testing activities may be focused on those parts where inspection has found most of the defects before.

Regarding the existing body of knowledge regarding relationships and derived explanations, which has often led to theories, Jeffery and Scott (Jeffery and Scott, 2002) presented two examples, i.e., ‘software cost

modeling and estimation' and 'software inspections'. While for the first example, valid theories could be derived and demonstrated, this could not be done for the second example. The authors state that there exists "confusion in the empirical inspection literature", which "is a result of insufficient expression of theory, a consequent lack of models, and too little attention in the experiments to the justification for the hypotheses under test" (Jeffery and Scott, 2002). Moreover, Bertolino (Bertolino, 2007) states that for testing, no universal theory exists either. Sjoberg et al. (Sjoberg et al., 2007) concluded that almost no software engineering specific theories are reported in the literature.

From the viewpoint of the author, instead of finding a theory first, in many cases it seems to be more promising to get context-specific evidence first. Later on, a valid theory might be derived.

4.4.1 Identification of Context-specific Assumptions

The field of empirical software engineering presents various concepts that guide the way from initial observations to evaluated theories (Shaw, 1990; Zelkowitz and Wallace, 1998; Perry et al., 2000; Jeffery and Scott, 2002; Endres and Rombach, 2003; Harwood, 2004; Sjoberg et al., 2007). One main objective is to improve the understanding regarding processes, products, and resources, and to build up solid knowledge in order to be able to predict future situations and make them more controllable.

There exist several models that describe how assumptions can be identified and evaluated. Jeffery and Scott (Jeffery and Scott, 2002), for instance, developed a model for scientific inquiry, starting by observing a phenomenon in the real world, understanding it, and developing a theory that explains the observed phenomenon. Such a theory has to be validated and refined by means of theory testing, replication, theory revision, and reevaluation. Jeffery and Scott use two concrete examples, i.e., 'software cost modeling and estimation' and 'software inspections', in order to demonstrate their procedure.

In contrast to the model by Jeffery and Scott, a more detailed model is proposed by Endres and Rombach (Endres and Rombach, 2003). The model starts with observations, which may be facts or impressions regarding certain relationships in a given context. When an observation reappears, one can take advantage of it. Repeatable observations are often defined as so-called laws. The authors define a law as "a statement of an order or relation of phenomena that, so far as is known, is constant under certain conditions". Exemplary laws in the field of quality assurance mentioned by the authors are that a developer is unable to test his own code or that about 80 percent of the defects come from 20 percent of the modules. Laws are explicitly derived based

on repeatable observations and lessons learned from different contexts. Because laws are based on strong empirical evidence, they can be seen as generalized observations that explain how things happen, independent of a concrete environment (though some situations may exist where a law might be wrong). Furthermore, future observations can be predicted based on laws.

A law can be explained by a theory: "A theory is a deliberate simplification of factual relationships that attempts to explain how these relationships work" (Baumwol and Blinder, 2001). Sjoberg et al. (Sjoberg et al., 2007) state that "in mature sciences, building theories is the principal method of acquiring and accumulating knowledge that may be used in a wide range of setting". Therefore, if a law is found, the next step is to find explanations for the observations, which shifts the level of understanding towards a theory. A theory itself can then be confirmed by future observations (until it may be rejected due to new insights and knowledge that falsifies the theory).

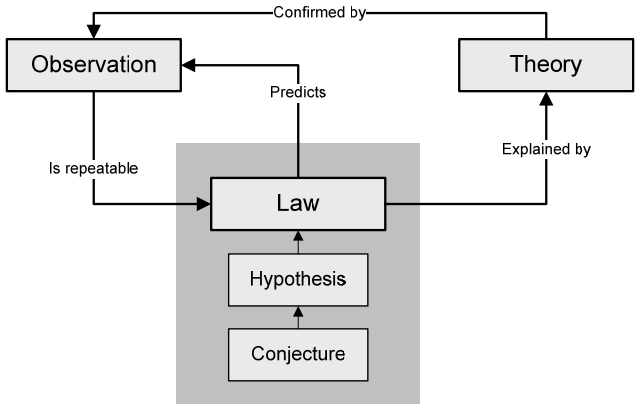


Figure 24 Concepts of empirical software and systems engineering according to Endres and Rombach (Endres and Rombach, 2003).

Figure 24 summarizes the concepts as stated by Endres and Rombach (Endres and Rombach, 2003). In addition to laws, the authors introduced two additional constructs in order to be able to describe relationships that are currently not grounded on strong empirical evidence. A hypothesis is a statement that is only tentatively accepted, for example only in a certain context. Additional evidence is needed in order for a hypothesis to become a law. A conjecture describes the lowest level in this hierarchy and is a guess or belief only.

Endres and Rombach (Endres and Rombach, 2003) describe three stringent criteria for accepting existing knowledge as a law: First, an underlying hypothesis exists that has been validated; second, the explicit kinds of studies used for the evaluations are determined (e.g., case

study, experiment); and third, replications of studies are conducted in different environments. However, it is sometimes more difficult to distinguish hypotheses and conjectures. Carver et al. (Carver et al., 2004) or Bertolino (Bertolino, 2007), for instance, utilize the term assumptions when referring to empirical studies.

Consequently, an adaptation of the model proposed by Endres and Rombach (Endres and Rombach, 2003) was performed, and a distinction into assumptions and evaluated assumptions (i.e., evidence) is made in the following. An assumption describes context-specific relationships that are observed or seem to be useful, but are not empirically grounded. In contrast, evaluated assumptions are based on empirically valid results that are accepted in the given context. In order to explain the evaluated assumptions, the results can be used to derive a theory for the given context.

Instead of starting with observations to derive assumptions, sometimes a theory is stated first, which subsequently has to be confirmed or rejected based on assumptions derived from the theory. Figure 25 summarizes these concepts.

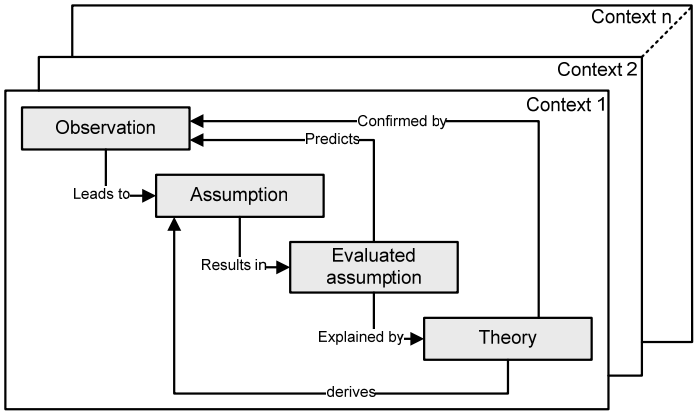


Figure 25 Concepts for empirical software engineering.

Context-specific relationships can be derived analytically or empirically.

1. Analytically: Based on a systematic analysis of a certain environment, which includes considering process and product structures (e.g., development and quality assurance processes; experiences of developers, inspectors, and testers; size and complexity of product to be developed), assumptions regarding the relationships can be derived in a logical manner. For example, consider that only parts of a system were inspected due to an insufficient amount of time available for inspections; consequently, testing should especially focus on those parts of the systems that had not been inspected.

2. Empirically: Based on (i) empirical knowledge from different environments and (ii) new experiences from a given context, assumptions regarding relationships can be derived empirically. First, accepted empirical knowledge from different contexts can be used and adapted to a given context. Examples in the area of quality assurance that are often found in the literature are that developers are unsuited to test their own code or that most of the defects are found in a small number of modules. Such empirically proven knowledge can be adapted and a corresponding assumption has to be checked in the given context. Second, when performing certain processes in a given context, new observations may be made, resulting in new or refined assumptions. This means that new empirical knowledge about certain relationships is gained. For example, when classifying defects according to a certain defect classification, new insights about which defect types are found by different quality assurance activities can be obtained.

4.4.2 Structured Description of Relationships

This section presents a structure for describing relationships between quality assurance processes. In addition, it describes how such assumptions can be made operational. This is exemplarily shown by transferring them into so-called selection rules that are used for integrating inspections and testing.

In order to be able to understand various processes, and consequently to improve them, knowledge regarding the relationships between such processes is required. Such knowledge about relationships can be considered as experience. In general, experiences are valid within a certain scope. This scope can be characterized by the context (i.e., the environment with all kinds of factors that might have an influence on the experience) and the degree of validity of the experience (simplified, this means how many times the experience has been gained or confirmed). Jacobs et al. (Jacobs et al., 2007) give an extensive overview of possible factors that characterize a context and that might have an influence on quality assurance activities. Petersen and Wohlin present a high-level overview of context factors (Petersen and Wohlin, 2009). In order to emphasize that there are many different experience items, we call one item an experience element (Feldmann et al., 1997).

In the area of inspection and testing, an exemplary assumption can be seen in Figure 26 (context and significance are initially not considered here).

Assumption 1: Parts of a system where a large number of inspection defects are found indicate more defects to be found with testing.

Figure 26 An exemplary assumption.

This assumption has already been mentioned before and describes the frequently observed Pareto distribution (Boehm and Basili, 2001) with respect to inspection and test defects, i.e., the relationship between inspections and testing is covered. Selection rules operationalize assumptions so that they can be applied. Different concrete selection rules that make the assumption operational are conceivable, for example the first three seen in Figure 27.

Selection rule 1: Focus a testing activity on those system parts where the inspection found more than 20 defects.

Selection rule 2: Focus a testing activity on those system parts where the inspection found more than 8 major defects.

Selection rule 3: Focus a testing activity on those system parts where the inspection found more than 15 defects per 1000 lines of code.

Selection rule 4: Focus a testing activity on those system parts where the inspection found more than 12 defects and which contain more than 600 lines of code.

Selection rule 5: Focus a testing activity on those system parts where the inspection found more than 12 defects and which contained more than 20 defects in the two past releases.

Figure 27 A set of different selection rules.

With respect to the validity of assumptions and derived selection rules, each one has to be evaluated in a new environment in order to identify the most suitable ones in a given context.

A selection rule consists of an action that describes what to do, and a precondition that describes what has to be true in order to perform the action. For example, an action may be to focus a testing activity such as unit testing on certain code classes. Furthermore, a precondition is made of a logical expression (simple or concatenated) and corresponding concrete thresholds, and uses inspection defect data (e.g., selection rules

1-3), and optionally metrics (e.g., selection rule 4) or historical data (e.g., selection rule 5) in addition. However, due to the fact that it is sometimes difficult to determine thresholds at the beginning, a precondition could remain vague at first (e.g., the general direction of the precondition, like "low" or "high", could be stated initially), and could get refined when more knowledge is gathered during quality assurance runs.

With respect to the validity of assumptions and derived selection rules, each has to be evaluated in a new environment in order to identify the most suitable ones in a given context, i.e., the scope of validity has to be determined.

Figure 28 summarizes the concepts of how relationships can be described in a structured manner.

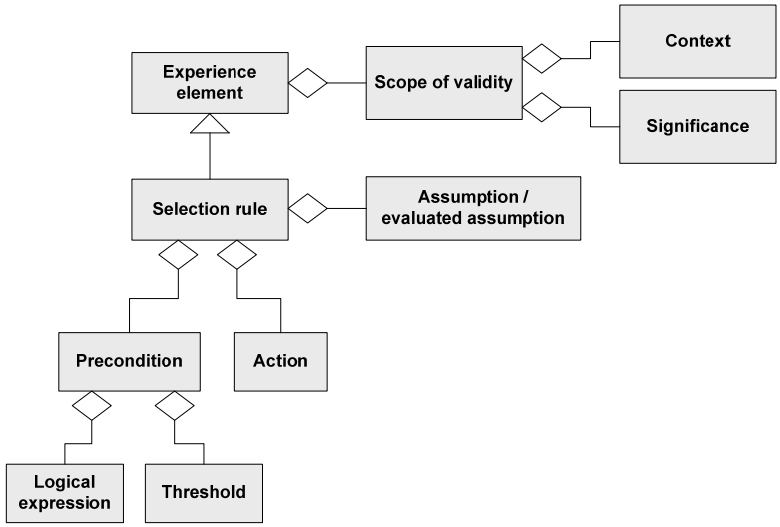


Figure 28 Structural model of relationships.

4.4.3 Guidelines for the Systematic Evaluation of Context-specific Assumptions

Assumptions that are initially stated should be refined, i.e., they should be evaluated by gaining new empirical evidence and by considering further context factors.

In general, the result of the evaluation of an assumption can be positive or negative. If the assumption was confirmed, all relevant context factors and the results should be packaged and additional evaluations should be performed in order to increase the significance of the evidence (i.e., the empirical evidence regarding the assumption). If the assumption was not confirmed, this might have different reasons; for example, the

assumption may be wrong in general or context factors were not considered or behaved differently in the given context.

Besides an initial evaluation of assumptions in order to understand certain relationships, continuous evaluations are necessary to improve the observed phenomena in the best possible way and to enable further adaptations, for example, due to subsequent context changes. For example, consider the assumption that in a certain context where software is further developed via releases, 75 percent of the defects are detected by inspections and 25 percent are found by testing. This means that, based on the inspection results, the number of remaining defects for testing can be predicted. If the context factor 'experience of inspectors' changes due to new team members who have less experience in performing an inspection, the ratio between inspection and test defects may change. Another change of a context factor might be if a complete new software product is developed instead of new releases of a mature product. This may also lead to a different ratio between inspection and test defects.

A comprehensive evaluation of assumptions, both analytically and empirically derived ones, may lead to a profound basis of empirical evidence. Ultimately, this may result in new theories.

A more detailed view on how to evaluate relationships in a certain context and how to maintain evidence is provided next.

Context Check

In order to achieve the highest possible effort reduction for a prioritized testing activity at the best possible quality, the assumptions stated should also be evaluated during inspection and testing activities. This is especially relevant with respect to context factors that might change during the development and quality assurance activities.

Consider the following example. The assumption is made that code classes in which the inspectors have found a significant number of defects contain more defects. Consequently, these code classes should be prioritized for testing. One underlying context factor is the experience of the inspectors. In the exemplary environment, the assumption is only true when low-experienced inspectors conduct the inspection. Thus, when a project manager or quality assurance engineer plans testing activities, he has to consider the context factors. If, for example, only inspectors with low experience are available, the assumption may be true and he can prioritize code classes for testing in which the inspectors have found defects. However, it may happen that suddenly inspectors are available with high experience, and the assumption initially made in this case is not true, because in past quality assurance runs, it appeared that experienced inspectors found most of the problems in the inspected

code classes and testing did not reveal many new defects. Consequently, if such a change of a context factor happens and if the project manager, respectively quality assurance engineer, knows about the influence of a context factor on the relationships, he is able to adapt the test strategy with respect to the changed context factors.

Maintenance of Evidence

In order to be able to decide which assumptions and derived selection rules are suitable in a given context for focusing testing activities based on inspection results, a retrospective analysis is necessary. For conducting such an analysis, data gathered during the quality assurance run have to be considered. This comprises at least inspection defect data and test defect data. For a detailed analysis, a representation of the number of defects found per part has to be given, e.g., the number of defects found per code class. Furthermore, if the defect type is considered in the assumptions, the number of defects found per defect type is needed. Finally, if additional metrics are used (e.g., size, complexity), these data also have to be captured, e.g., size – measured in lines of code – per code class.

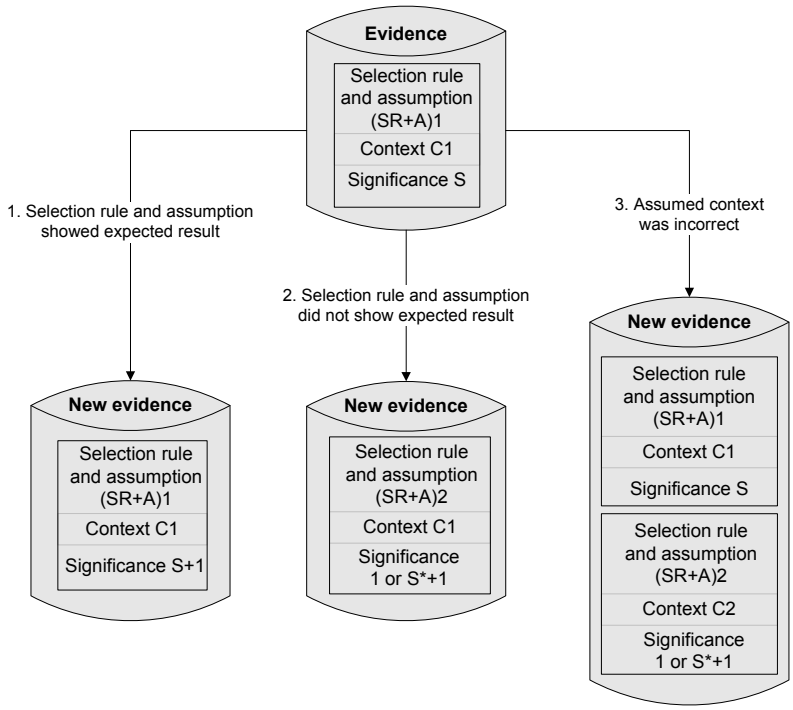


Figure 29 Maintenance of evidence.

Figure 29 gives an overview of the three possibilities when selection rules and assumptions are evaluated in order to maintain their evidence. An exemplary model for packaging project experience (Heidrich et al., 2006) is used to store a set of assumptions and selection rules, respectively their performance in the completed quality assurance run. The focusing of testing activities starts by choosing selection rules and assumptions stored in a database before they are applied in a new project. The selection rule and the assumption (SR+A)1 used are valid in a given context C1 and have a certain significance gained from application in S former projects. An analysis with respect to the gathered data can lead to three different possibilities:

1. The selection rule and the corresponding assumption were correct during the completed quality assurance run, i.e., the focusing of testing activities based on the inspection results was appropriate and led to the expected results (i.e., all defects were found). In this case, significance is increased by one.

2. The selection rule and the corresponding assumption were incorrect during the completed quality assurance run, i.e., the focusing of testing activities based on the inspection results was not appropriate and did not lead to the expected results (i.e., defects were not found). In this case, an alternative assumption and selection rule or another selection rule of the used assumption has to replace the original one used in the given context. Significance is set to one (if applied the first time) or S*+1 (if applied successfully S* times before).

3. The project context was different for the completed quality assurance run, i.e., concrete values of certain context factors were assumed (e.g., experience of inspectors is low), but after following the processes, this turned out to be wrong due to hidden or changed context (e.g., the experience of the inspectors was actually high). Consequently, the original selection rule and assumption are kept as is, and a new selection rule and assumption are used in the changed context with the two possibilities of significance as shown in the second case.

Conclusions from such an analysis should be considered in subsequent quality assurance runs.

In order to perform a maintenance analysis of assumptions, and considering experiences gained from the analyses of the assumptions and selection rules, the maintenance model seen in Figure 29 was adapted. In order to be able to judge the quality of selection rules and to

decide which assumptions are most appropriate, a four-scale evaluation scheme is introduced next to assess the selection rules. In this thesis, only the first two cases of the model are considered (i.e., assumptions and selection rules showed the expected results, respectively did not show the expected results), and the third case is omitted (i.e., the assumed context was incorrect). In addition, the focus is on system parts (and defect types are not covered explicitly in the following).

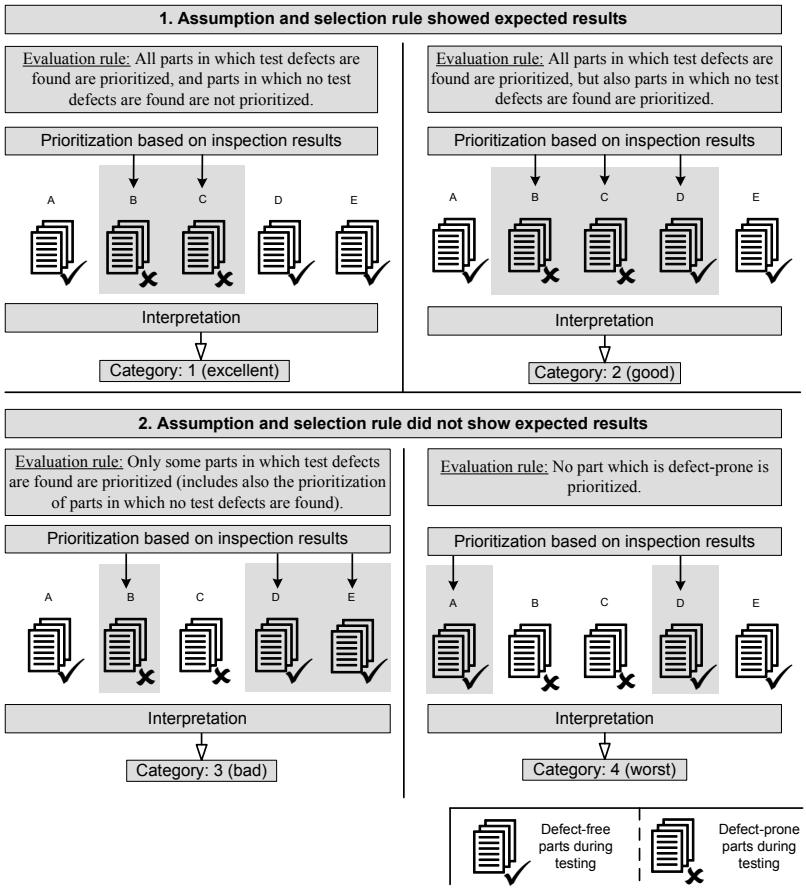


Figure 30 Four quality categories (using strong evaluation rules).

If an assumption and the derived selection rule showed the expected results, two refined possibilities for their evaluation exist, i.e., a selection rule can be classified into category one or two if all defects are found with the prioritization (i.e., the selection rule is correct). The two categories differ with regard to the selection of additional parts (e.g., code classes), which would result in lower efficiency. If an assumption and the derived selection rule did not show the expected results, again two possible evaluations exist. Categories three and four comprise

selection rules that select none or only some of the defect-prone parts (e.g., code classes), which results in reduced overall effectiveness (i.e., the selection rule is incorrect, which means that not all or none of the defects are found with the chosen selection rule). Figure 30 shows examples of each of the four categories.

Even though the main goal of the integrated approach is to reduce the effort for testing activities, no defects should remain uncovered (i.e., strong evaluation rule), respectively a certain threshold should not be exceeded (i.e., weak evaluation rule) when applying a selection rule. Thus, selection rules classified into one of the first two categories represent selection rules of the highest quality. A selection rule that prioritizes all system parts (e.g., code classes) would also have been placed into category two. However, this case is excluded because no effort reduction would be achievable. The third category contains selection rules that overlook defect-prone parts, but select some defect-prone parts. Thus, a combination or consideration of selection rules of this category could improve the prioritization of defect-prone parts and, consequently, should be further analyzed in future QA runs. Finally, selection rules of category four do not lead to any appropriate prioritization and thus are negligible.

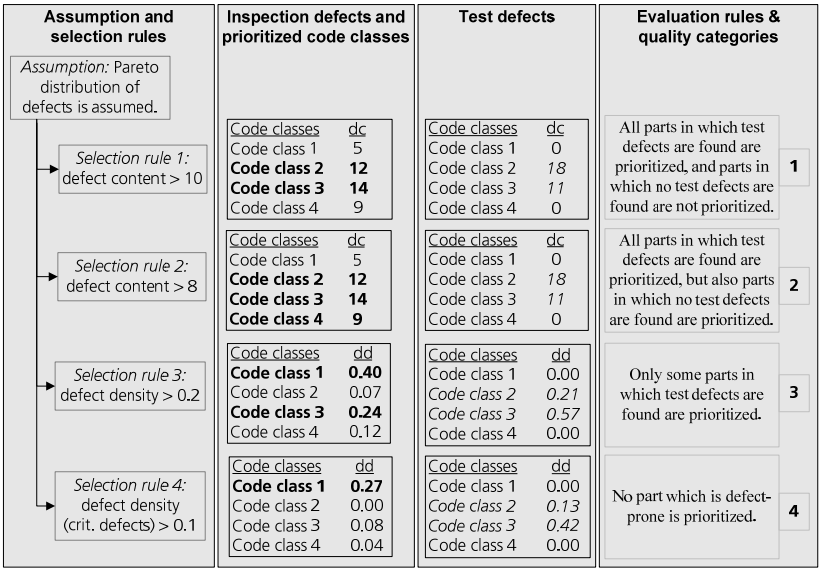


Figure 31 Exemplary analysis of selection rules for one quality assurance run.

All selection rules have to be evaluated for a given quality assurance run. Figure 31 shows a concrete example of the analysis with respect to code classes. The stated assumption and derived selection rules are applied with respect to the inspection defect data; the corresponding code classes are selected and compared with the defect data found during

testing. Afterwards, a quality category can be determined for each selection rule. Selection rules that are classified as excellent or good (i.e., 1 or 2) were correct and should be considered in detail for future quality assurance runs. Selection rules classified as bad (i.e., 3) should also be considered, as some parts are prioritized correctly. However, a combination with different selection rules might improve the prioritization. Selection rules classified as worse (i.e., 4) do not provide any appropriate prediction and are candidates for selection rules that do not fit in the given context. However, more evidence has to be gained by analyzing the selection rules during additional quality assurance runs.

The results of the selection rules can be aggregated in order to determine whether the general assumption tends to be correct or wrong. Furthermore, an analysis of assumptions and selection rules can be performed on more than one level (e.g., code level, system level).

In order to perform a trend analysis, i.e., to analyze which assumptions and selection rules are suited best across more than one quality assurance run, selection rules have to be classified according to the available data. Table 9 shows what this might look like.

Table 9 Exemplary trend analysis of selection rules.

	QA run 1	QA run 2	QA run 3	…	QA run n
Assumption 1					
Selection rule 1.1	1	3	1	…	…
Selection rule 1.2	4	3	3	…	…
Selection rule 1.3	4	4	2	…	…
Selection rule 1.4	1	1	2	…	…
Assumption 2					
Selection rule 2.1	1	2	2	…	…
Selection rule 2.2	4	3	1	…	…
Selection rule 2.3	4	4	4	…	…
Selection rule 2.4	1	1	3	…	…

For example, selection rules 1.1, 1.4, 2.1, and 2.4 show promising results with some outliers (which should be analyzed in more detail; some reasons might be that the evaluation rule was too strong or that influence factors, respectively context factors, were not considered). Selection rules 1.3 and 2.3 showed bad results. Selection rules 1.2 and 2.2 should be analyzed in more detail; the first one considered in the second and third QA run some defect-prone parts and could be combined with another selection rule; the second one showed inconsistent results, which could be explained by certain context factors that have changed (e.g., the experience of the inspectors) or were hidden (i.e., not considered). However, independent of the concrete quality categories, such a representation gives an appropriate overview that can be used for trend analyses. This view could be enhanced by using a color scheme (e.g., dark green for the best ones, red for the

worst selection rules) or aggregated (e.g., by using higher numbers for good predictions and summing them up).

It is assumed that in larger industrial environments, selection rules will be mostly rated into categories two and three, i.e., the extreme values that a selection rule finds all defect-prone parts or does not find any defect-prone part are rather untypical. Therefore, a combination (i.e., OR concatenation) of such selection rules might result in better focusing. However, in this case, one has to ensure that not all parts are tested again, i.e., one has to omit the case that the different combined selection rules select all parts again.

4.4.4 Context-specific Relationships between Inspection and Test Defects

Jeffery and Scott (Jeffery and Scott, 2002) state that a profound underlying theory in the area of software inspections is missing. This lack is even more critical when inspection and testing techniques are combined to exploit certain synergy effects, such as reduced effort or higher defect detection rates. Consequently, there is no way to avoid making assumptions regarding relationships that have to be systematically analyzed afterwards. However, there exist a number of accepted evaluated assumptions or laws, as Endres and Rombach (Endres and Rombach, 2003) call them, which can be used and adapted to the area of combined quality assurance techniques. Due to unknown or partially unknown relationships, an initial set of different assumptions are listed below that may form a starting point for evaluating them and that might lead to theories in the future. A distinction is made between analytically and empirically derived assumptions, and explanations, respectively empirical evidence, are presented to substantiate the given assumptions. Each of these assumptions has to be evaluated in different contexts in the future in order to show whether it is correct or wrong. Besides assumptions that consider only inspection results, some assumptions consider certain product metrics in addition.

Analytical Assumptions

Various assumptions can be derived analytically, i.e., they can be determined logically. Some examples are presented below.

Assumption A1: If no defined selection criterion is used to determine parts of a system that should be inspected, it is expected that a significant number of defects still remain in those parts that are not inspected. Consequently, testing should be focused especially on those uninspected parts of a system.

Sometimes, inspections of certain parts are skipped due to external reasons that are not related to quality assurance (e.g., time constraints, missing resources). This may lead to re-planning of quality assurance activities. Consequently, a testing activity should be focused on the remaining parts of the system to find additional defects.

Assumption A2: Inspection and testing activities find defects of various defect types with different degrees of effectiveness. For inspections, this includes, e.g., maintainability problems. For testing, this includes, e.g., performance problems. Consequently, inspection and testing activities should be focused on those defect types that are most convenient to find.

Among others, Gilb and Graham (Gilb and Graham, 1993) already mentioned that inspection and testing complement each other. This also means that they are able to find different kinds of defects. For example, Mantyla and Lassenius (Mantyla and Lassenius, 2009) report that code inspections find evolvability defects (e.g., defects affecting documentation or structure) that cannot be found by testing activities. One reason is that those maintainability problems do not affect functionality that is tested later. In contrast, problems that are only found when the system is running, such as performance problems, can be found better or only with testing. However, despite such defect types that are easy to assign to one quality assurance technique, it is unclear for many other defect types whether they can be found better with inspections or with testing.

Assumption A3: If inspection process conformance is high and the inspectors found a lot of defects and have a lot of experience with inspections, the inspected parts can be assumed to contain no or a negligible number of remaining defects. Consequently, testing should be focused on parts that were not inspected yet.

Furthermore, it is possible to consider context factors and inspection process conformance explicitly in an assumption. The exemplary assumption A3 takes the inspection process conformance, the number of inspection defects found, and the experience of the inspectors into account. It is assumed that during a properly performed inspection, a certain number of defects will be found. If the inspectors are very experienced, it is assumed that most of the defects have already been found, and different parts can be prioritized for testing.

Assumption A4: If the inspectors found a low number of defects and only a small amount of time was spent, the inspected parts can be assumed to contain additional remaining defects. Consequently, the inspected parts should be tested again.

The exemplary assumption A4 also considers inspection process conformance. In this case, it is explicitly checked whether the inspectors have found a low number of inspection defects due to little available time. In this case, it is assumed that at most minor defects are found, and thus, testing should be focused on these parts again.

Empirical Assumptions

As mentioned above, little empirical evidence exists in the area of combined inspection and testing techniques. Therefore, empirical evidence from related areas is taken and adapted as a starting point.

Assumption E1: Parts of a system where a large number of inspection defects are found indicate more defects to be found with testing (i.e., a Pareto distribution of defects is assumed).

A large number of different studies performed in various environments have shown that an accumulation of defects, i.e., a Pareto distribution, can be observed rather than an equal distribution of defects. One of the first studies was conducted by Endres (Endres, 1975), who showed, among other observations, that about 80 percent of the problems are found in 20 percent of the modules. Further studies were conducted by Myer (Myer, 1979) and Möller (Möller, 1985) and showed the same results. Basili and Perricone (Basili and Perricone, 1984) documented that about 60% of the defects stem from 35% of the modules. A later study by Möller and Paulish (Möller and Paulish, 1993) described the distribution of defects within three evolutionary versions of a software product and confirmed the initial results, showing that about 55% to 70% of the defects were contained in 20% of the modules. Later observations (Ohlsson et al., 1996; Fenton and Ohlsson, 2000; Ostrand and Weyuker, 2002; Denaro and Pezze, 2002) resulted in the rule of thumb that 80% of all defects can be found in 20% of the modules (Boehm and Basili, 2001; Shull et al., 2002). Some recent studies have confirmed these results (Anderson and Runeson, 2007; Turhan et al., 2009; Hamill and Goseva-Popstojanova, 2009; Ostrand et al., 2010).

Assumption E2: Parts of the system where a large number of inspection defects are found (i.e., a Pareto distribution of defects is assumed) and which are of small size indicate more defects to be found with testing.

A size metric is often used to prioritize defect-prone parts and thus, to focus a testing activity. Though this metric is often applied, a number of studies has shown inconsistent results when size is applied as the sole metric for predicting defect-prone modules. Emam et al. (Emam et al., 2002) state that if models are built to predict fault-proneness, other variables than just size should be used. A number of studies were identified in which small code modules, respectively methods, tended to be more defect-prone (Basili and Perricone, 1984; Möller and Paulish, 1993; Ostrand and Weyuker, 2002; Turhan et al., 2009). However, some studies showed the opposite (Emam et al., 2002) or inconsistent results (Fenton and Ohlsson, 2000; Anderson and Runeson, 2007). Thus, a combination of assumption E1 (i.e., Pareto distribution of defects) and size might lead (a) to a more detailed and fine-grained assumption and (b) to a better prediction of defect-prone parts than using a size metric alone.

Assumption E3: Parts of the system where a large number of inspection defects are found (i.e., a Pareto distribution of defects is assumed) and which are of high complexity indicate more defects to be found with testing.

Another metric frequently used to predict defect-prone parts and thus, to prioritize those parts for testing activities is complexity. Munson and Khoshgoftaar (Munson and Khoshgoftaar, 1992) state that "there is a clear intuitive basis for believing that complex programs have more faults in them than simple programs". However, Schröter et al. (Schröter et al., 2006) note that new metrics or combinations of existing metrics should be used to study the relationship between complexity and the presence of bugs. Thus, in order to improve the prioritization of code classes expected to be most defect-prone, the inspection results can be combined with a complexity metric, with a focus on code classes that have high complexity. Nagappan et al. (Nagappan et al., 2006) as well as Ohlsson and Alberg (Ohlsson and Alberg, 1996) proved a high correlation between a high McCabe complexity value and the number of defects. Basili et al. (Basili et al., 1996) showed the same relationship for different object-oriented complexity metrics. However, there exist also studies showing that the correlation between complexity metrics and number of defects is rather low (Fenton and Ohlsson, 2000); consequently, this assumption has to be evaluated thoroughly.

Assumption E4: Defects of the defect types that are found most often by inspections indicate more defects of these defect types to be found with testing (i.e., a Pareto distribution of defects of certain defect types is assumed).

An accumulation of defects of certain defect types can also be observed in several studies rather than an equal distribution of defect types, independent of any concrete defect classification. Several defect classifications used by Mantyla and Lassenius (Mantyla and Lassenius, 2009) to classify inspection defects show accumulations of some defect types. Results from experiments comparing inspection and testing defects that use a defect classification also show an unequal distribution of defect types (Kamsties and Lott, 1995; Laitenberger, 1998). The same observation is presented by Chaar et al. (Char et al., 1993), where ODC is used. Finally, Ohlsson et al. (Ohlsson et al., 1996) state that the majority of quality costs are often caused by very few defect types. However, one has to be aware that this is not necessarily so for each defect type.

Conclusion

In conclusion, various assumptions are possible when analyzing relationships between inspection and testing techniques. Some of them seem to be contradictory, such as assumptions A2 and E4; in this case, future evaluations might show which direction is true in certain contexts. The defined assumptions can serve as a starting point for such evaluations.

4.4.5 Application Procedure

The concepts described above (i.e., identification of assumptions, description of relationships, evaluation) can be summarized into a concrete procedure that guides a quality engineer or a project manager when using assumptions and refined selection rules during the application of the In^{2}Test approach.

First of all, two different cases have to be distinguished:

- Retrospective procedure: The inspection and testing activities are applied in a non-integrated manner, and inspection and test defect data are gathered. Afterwards, defined assumptions are analyzed in order to find the most suitable ones. This procedure is normally conducted when no information about the relationships between inspection and testing is available in a new context.

- Pro-active procedure: The In^{2}Test approach is applied and assumptions are followed during testing, i.e., focused testing is actually performed. This procedure is normally conducted when relationships between inspections and testing are known in a certain context.

The general procedure comprises four basis steps: preparation, execution, evaluation, and packaging.

Figure 32 shows the instantiated steps for the retrospective procedure. The preparation, which is a creative step, can already start before inspections are conducted, or be performed in parallel to the quality assurance steps. The execution of the inspection and testing activities leads to defect data, which are the input for the evaluation. The defined assumptions and selection rules can be assessed (i.e., maintenance of evidence can be done), and new ones can be defined, if necessary. If no concrete thresholds are determined in the selection rules, this can be done in a retrospective manner (e.g., if a selection rule calls for prioritizing code classes with high defect content, "high" can be determined after quality assurance activities are finished). In addition, it is worthwhile searching for explanations regarding why certain assumptions work well or do not work at all in the given environment, i.e., the context has to be considered during the analysis. Finally, the results should be packaged and used in subsequent quality assurance runs.

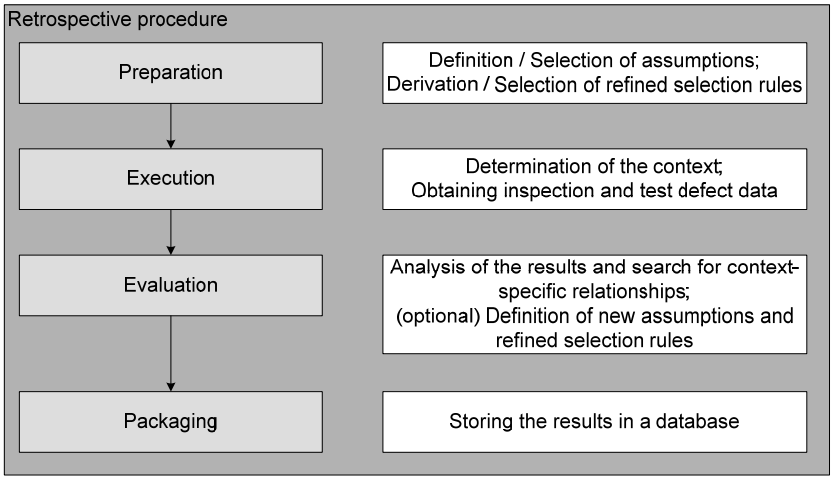


Figure 32 Retrospective procedure for assumptions.

The second case, the pro-active procedure of the In^{2}Test approach, extends especially the execution step. After assumptions are determined or selected and refined selection rules are derived that fit the context, inspection data is considered as input for the prioritization. Based on

these data, the test is focused and conducted. A continuous analysis (i.e., context check) can be applied in order to check the validity of the applied selection rules, and adjustments can be made, if necessary. During the evaluation steps, the results are checked again (i.e., maintenance of evidence is performed), and more relationships are identified, respectively established ones are checked regarding their validity. Afterwards, the results are packaged. Figure 33 summarizes these steps.

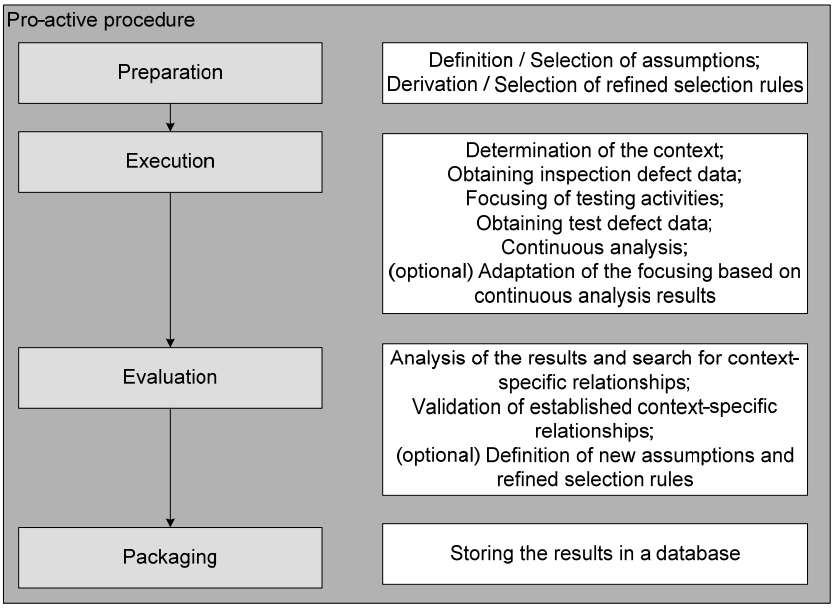


Figure 33 Pro-active procedure for assumptions.

4.5 Prototype Tool Support

In the context of the development of the *Dependability Focused Inspection Tool* (DETECT[1]) (Elberzhager et al., 2010a), a prototype module was implemented that supports the focusing of defect-prone parts and defect types for further quality assurance activities based on inspection results.

DETECT is being developed using the software development environment Eclipse. It facilitates a plug-in concept in order to allow tailoring the tool to different environments with different inspection processes. DETECT supports the inspection process, i.e., the tool assists different inspection roles during certain inspection steps. All four mandatory steps (planning, preparation, meeting, and correction) and

[1] The initial name of the tool was DEFECT, which was changed later on

one optional step (follow-up) are currently supported. The corresponding roles are assigned accordingly, i.e., each role only has access to certain functionalities. Table 10 presents an overview of the support.

Table 10 Overview of supported inspection steps, roles, and activities of the DETECT tool.

Supported inspection step	Supported inspection role	Supported activities
planning	organizer	creation of checklist(s), composition of inspection package(s)
preparation	inspector	defect detection including documentation
meeting	scribe	creation of final defect list
In² Test analysis	*QA engineer*	*focusing of subsequent QA activities*
correction	author	documentation of corrections
follow-up	organizer	evaluation of correction

Besides the general inspection support, the In²Test analysis module was developed which supports a quality assurance engineer. This module is able to analyze and illustrate inspection data gathered during defect detection in the preparation step. For example, the number of found defects per inspected code class, the defect density, or the number of defects per defect type can be displayed, i.e., different kinds of defect distributions can be analyzed. Figure 34 shows an example where the number of issues (A), the number of defects (B), and the respective defect density (C) for five different inspected artifacts are presented.

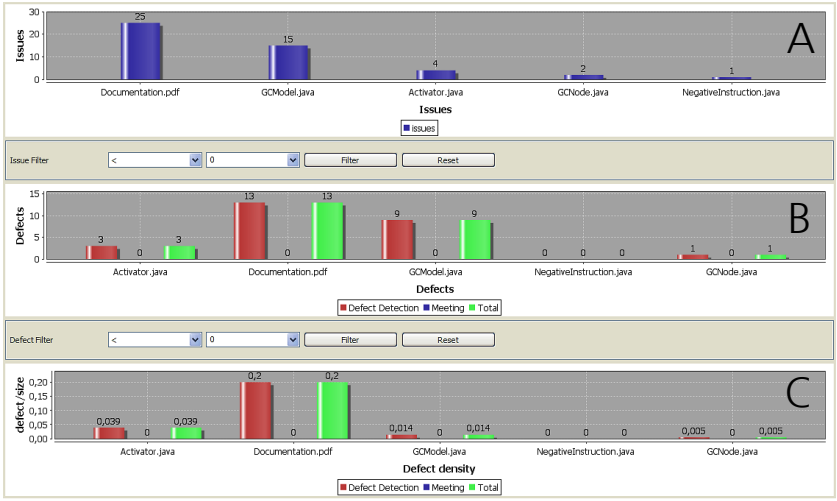


Figure 34 DETECT tool: In²Test analysis module showing inspection data.

Furthermore, the In²Test analysis module can be used to define rules and thresholds in order to prioritize certain parts of the system, and consequently, to support the focusing of subsequent quality assurance

activities, such as testing. Figure 35 shows a simplified example where two rules and exemplary thresholds are defined with respect to defect content and defect density. The tool performs an analysis with respect to the gathered inspection data, and shows only those modules that fulfill the requirements of the rules. In the given example, two artifacts fulfill the selection rule "number of defects > 5" (A), and three fulfill the rule "defect density > 0.01" (B). The displayed artifacts may be considered in subsequent testing activities. Each rule can be changed or reset in order to adapt the selection rules.

Figure 35 DETECT tool: In²Test analysis module showing different rules applied for focusing.

4.6 Limitations

With respect to the requirements stated in Section 2.4, the integrated inspection and testing approach In²Test can be assessed as follows:

R1: Prediction of defect-prone parts. The approach is explicitly able to predict parts of a system that are expected to be defect-prone and, hence, to focus testing activities on those parts. The main inputs for the prediction are inspection defect data and, optionally, additional metrics or historical data. Assumptions are used to describe the relationships between inspection and testing in order to allow a prediction based on the input information.

R2: Prediction of defect types. The approach is explicitly able to predict those defect types of which a significant number of defects are likely to appear during testing activities. The main inputs for the prediction are inspection defect data and, optionally, additional metrics or historical data. Assumptions are used to describe the relationships between inspection and testing in order to allow a prediction based on the input information.

R3: Make use of inspection results. The approach uses as main input for the prediction a defect profile, which comprises quantitative defect data and defect type information from an inspection.

R4: Make use of historical defect data and further metrics. In order to be able to improve the prediction, historical defect data and additional metrics, such as size or complexity, may be used and combined with the inspection results.

R5: Make use of empirical evidence. Knowledge about the relationships between inspections and testing, respectively between inspection and testing defects, is used to make the prediction and to continuously improve the validity of the prediction. For this, context-specific and context-independent empirical evidence can be used.

R6: Store experience for later reuse. Defect data and further experiences can be stored in a database.

R7: Applicable during different lifecycle stages. Due to the fact that inspections and testing can be applied to different lifecycle stages, this also holds for the integrated approach, which makes use of these quality assurance activities.

R8: Able to integrate with different inspection and testing activities. The In^{2}Test approach can be considered as a light-weight approach because no existing inspection or testing technique has to be replaced with any required inspection or testing technique. Instead, a defect profile can easily be derived from already applied inspection techniques, and existing testing activities can be focused based on these results.

R9: Adaptable to different environments. As inspections and testing can be applied in different environments (e.g., embedded systems domain or information systems domain), this also holds for the integrated approach.

Table 11 Assessment of requirements with respect to the In^{2}Test approach.

Requirements	In^{2}Test approach
R1: Prediction of defect-prone parts	+
R2: Prediction of defect types	+
R3: Make use of inspection results	+
R4: Make use of historical defect data and further metrics	+
R5: Make use of empirical evidence	o
R6: Store experience for later reuse	+
R7: Applicable during different lifecycle stages	o
R8: Able to integrate with different inspection and testing activities	o
R9: Adaptable to different environments	+

Table 11 summarizes the assessment results. As can be seen, not all requirements are completely fulfilled, which leads to the following limitations:

- *R5:* The approach uses a set of initial assumptions and derived selection rules, of which some are valid, but others are not. Consequently, though the approach is generally applicable, the quality of the focusing depends on the concrete assumptions made and selection rules chosen.

- *R5:* The approach does not provide a large set of empirical evidence that can be re-used in the future when applying the In^{2}Test approach in different environments. In this thesis, related empirical evidence about defect distributions was initially used. However, as already mentioned before, theories about inspections and testing that describe relationships between them are very rare. Moreover, any assumptions stated have to be re-evaluated in each new context in order to identify the ones most suitable for predictions.

- *R7:* Though the approach does not have any specific limitation regarding lifecycle stages, prediction becomes more difficult for early lifecycle phases, such as requirements. One reason is that requirements inspection results are often rather broad, which may make it difficult to focus system testing activities. In addition, considering inspection results from one development level to focus testing on another development level (e.g., using inspection results from a unit test for focusing system testing) may result in additional challenges, which are currently not incorporated explicitly in the approach.

- *R8:* Although the approach does not have any specific limitation regarding necessary inspection and testing activities, one prerequisite is a suitable number of inspection defects that is needed in order to be able to perform a prediction. Certain inspection techniques, especially more informal ones, tend to find only few defects, respectively no explicit documentation of defects is conducted, which makes it hard to apply the In^{2}Test approach.

Further limitations include:

- Although the In^{2}Test approach considers test exit criteria, the predictions are currently not used for deciding when to stop testing. No overall confidence measure is defined yet, i.e., an answer to the question of when to stop testing if the In^{2}Test approach is applied is not given yet. An answer to that question

depends on many factors, such as usage scenarios, criticality of defects, or expected number of defects. Traditional coverage criteria could be a starting point for deciding when to stop testing. However, a comprehensive confidence measure has not been defined yet.

- Although the In^{2}Test approach explicitly predicts certain parts or certain defect types for testing, no guidance is presented on how to derive or select test cases for those parts or those defect types.

- The In^{2}Test approach is currently not embedded into an overall quality assurance approach to balance specific quality assurance activities.

4.7 Summary

The integrated inspection and testing approach In^{2}Test was introduced in this chapter. The main idea is that inspection defect data is used to focus testing activities. This is supported by assumptions and derived selection rules, which cover the knowledge about relationships between inspections and testing. A differentiation was made between a one-stage and a two-stage approach. The former only focuses testing on either defect-prone parts or defect types, while the latter one focuses on defect types within the defect-prone parts.

In addition, a structural model for describing relationships was offered. It consists of assumptions and the scope of validity, and it refines the selection rules. Furthermore, concepts for identifying and evaluating assumptions were presented, and a set of initial assumptions was given.

Besides a detailed description of the approach and the assumptions, the solution idea was presented, and limitations of the approach were sketched.

5 Empirical Validation

5.1 Overview

This chapter presents the evaluation of the In²Test approach. Section 5.2 describes the evaluation procedure. A GQM plan was developed in order to ensure systematic validation. Certain measurement goals were refined into research questions, which formed the basis for corresponding hypotheses. The section closes with a summary of which hypotheses were evaluated in which case study. Sections 5.3 and 5.4 describe the results of the two case studies. In both case studies, a tool was developed, whose quality was assured with inspections and testing. The resulting defect data was used to evaluate the In²Test approach. Section 5.5 summarizes the main results of the empirical validation.

With respect to the main results of the case studies, the applicability of the approach could be shown and the design of the validation could be verified. One important prerequisite for applying the In²Test approach and for focusing testing activities is appropriate testability. With respect to effort reduction, a reduction of test execution effort of between 8% and 23% could be shown in the first study, and a reduction of between 6% and 34% could be shown in the second study when the focus was placed on certain parts of the system. Although focusing of defect types was done, no concrete numbers regarding effort reduction were obtained.

The same effectiveness was achieved with respect to the effort reductions. The highest efficiency improvement was achieved in the second study, with a total of 52%; the first study showed efficiency improvements of between 9% and 29%. However, these results are only valid for those assumptions and selection rules that selected all defect-prone parts. Finally, in the second case study, assumptions and selection rules were evaluated regarding their validity over two quality assurance runs, i.e., it was analyzed which ones provided the best focusing results in both QA runs. It could be shown that nine selection rules (out of 118) showed the best possible prioritization of code parts, and all of these selection rules took the inspection results into consideration.

5.2 Evaluation Procedure

A GQM plan (goal, questions, metrics (Basili et al., 1994b)) was used to systematically derive measurement goals, research questions, and corresponding metrics. Based on the GQM plan, hypotheses were

derived for the validation of the approach. However, only a subset of the derived hypotheses could be evaluated in this thesis.

Certain goals and the two main hypotheses were already defined in Chapter 1. The two main hypotheses are:

> H1: The effort for applying the integrated inspection and testing approach is at least 20% less compared to applying non-integrated inspection and testing processes, with the level of quality of the product under test that can be achieved being at least equal.
>
> H2: The integrated inspection and testing approach is applicable in industrial contexts.

Measurement goals 0-4 are covered by the first hypothesis; measurement goal 5 is covered by the second hypothesis.

5.2.1 GQM Plan and Hypotheses

In the following, the measurement goals, corresponding research questions, and the derived hypotheses will be described.

MG0: Analyze the integrated approach in order to compare its suitability with a non-integrated approach from the perspective of a quality assurance engineer in the context of software development.

Measurement goal 0 covers the detailed measurement goals 1-4, as can be seen in the following:

MG1: Analyze the integrated approach in order to compare its consumed effort with a non-integrated approach from the perspective of a quality assurance engineer in the context of software development.

The first goal was defined to evaluate whether the integrated inspection and testing approach leads to effort reduction for testing and, consequently, for the overall QA. For this, a comparison with a non-integrated inspection and testing approach is necessary, which does not use the inspection results as a means for focusing the test. This results in two research questions (RQ):

> RQ1.1: Does the proposed In^{2}Test approach lead to effort reduction for testing when focusing on parts of the system compared to a non-integrated approach?
>
> RQ1.2: Does the proposed In^{2}Test approach lead to effort reduction for testing when focusing on defect types compared to a non-integrated approach?

The corresponding hypotheses are stated as follows:

> H1.1: The In²Test approach leads to an effort reduction of at least 20% for testing in one QA run when focusing on parts of the system compared to a non-integrated approach.
>
> H1.2: The In²Test approach leads to an effort reduction of at least 20% for testing in one QA run when focusing on defect types compared to a non-integrated approach.

MG2: Analyze the integrated approach in order to compare its effectiveness with a non-integrated approach from the perspective of a quality assurance engineer in the context of software development.

The second goal was defined to evaluate the quality of the integrated approach. This means first to evaluate how many defects are found with the integrated approach compared to a non-integrated one. Second, to evaluate if the integrated approach uses the inspection results in a way that those defect types are selected of which most defects are found during a later testing activity. Two research questions can be derived:

> RQ1.3: Does the proposed In²Test approach find at least the same number of defects during testing compared to a non-integrated approach?
>
> RQ1.4: Does the proposed In²Test approach find at least the same number of defects of certain defect types during testing compared to a non-integrated approach?

The two corresponding hypotheses are stated as follows:

> H1.3: The In²Test approach finds at least the same number of defects during testing in one QA run when focusing on parts of the system compared to a non-integrated approach.
>
> H1.4: The In²Test approach finds at least the same number of defects of certain defect types during testing in one QA run when focusing on defect types compared to a non-integrated approach.

MG3: Analyze the integrated approach in order to compare its efficiency with a non-integrated approach from the perspective of a quality assurance engineer in the context of software development.

In order to be able to evaluate whether the integrated approach finds at least the same number of defects with reduced testing effort, assumptions and selection rules were considered and had to be evaluated with respect to their efficiency, i.e., the ratio of number of

defects per time unit. Two additional research questions were defined on which both MG1 and MG2 have an influence:

> RQ1.5: Is the proposed In^{2}Test approach more efficient compared to a non-integrated approach during testing when focusing on certain parts of the system under test, i.e., are at least the same number of defects found with reduced effort?
>
> RQ1.6: Is the proposed In^{2}Test approach more efficient compared to a non-integrated approach during testing when focusing on certain defect types, i.e., are at least the same number of defects of certain defects types re found with reduced effort?

Two hypotheses can be derived:

> H1.5: The In^{2}Test approach is at least 20% more efficient during testing in one QA run when focusing on parts of the system under test compared to a non-integrated approach.
>
> H1.6: The In^{2}Test approach is at least 20% more efficient during testing in one QA run when focusing on defect types compared to a non-integrated approach.

MG4: Analyze the integrated approach in order to evaluate the validity of the underlying assumptions from the perspective of a quality assurance engineer in the context of software development.

In order to be able to evaluate the validity of the assumptions used for focusing certain parts of the system and certain defect types for testing with the In^{2}Test approach, defect results from an environment of more than one QA run are necessary. The following research questions can be derived:

> RQ1.7: Which assumptions and derived selection rules lead to the highest efficiency for the given context for more than one QA run when applying the In^{2}Test approach for focusing parts of the system?
>
> RQ1.8: Which assumptions and derived selection rules lead to the highest efficiency for the given context for more than one QA run when applying the In^{2}Test approach for focusing defect types?

The following hypotheses are derived:

> H1.7: A set of assumptions and derived selection rules can be found that lead to the highest efficiency when applying the In^{2}Test approach in more than one QA run for focusing parts of the system.

H1.8: A set of assumptions and derived selection rules can be found that lead to the highest efficiency when applying the In²Test approach in more than one QA run for focusing defect types.

A GQM plan of the measurement goals and questions stated above, enriched by concrete metrics and covering hypothesis H1, can be found in Figure 36.

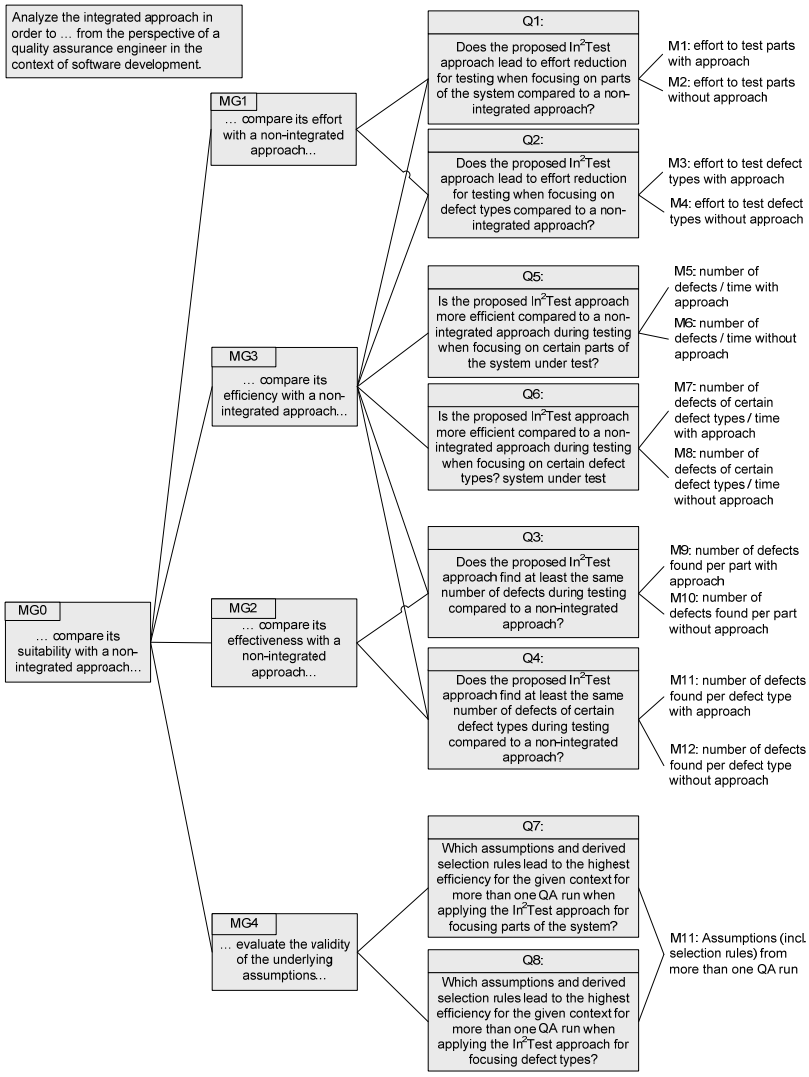


Figure 36 GQM plan, comprising measurement goals, questions, and metrics for hypothesis H1.

MG5: Analyze the integrated approach in order to evaluate its applicability in industrial contexts from the perspective of quality

assurance engineers, inspectors, testers, and developers in the context of software development.

The integrated In^{2}Test should be applicable, which can be refined into the following research questions:

RQ2.1: Do users see a benefit when using the proposed In^{2}Test approach compared to state-of-the-practice approaches?

RQ2.2: Do users easily understand the proposed In^{2}Test approach?

RQ2.3: Are users able to apply the proposed In^{2}Test approach?

RQ2.4: Do users perceive the prioritization of the proposed In^{2}Test approach as reasonable?

One hypothesis was derived from these questions:

H2.1: The proposed In^{2}Test is considered applicable by experienced practitioners.

A GQM plan of the measurement goal and derived questions stated above, enriched by concrete metrics and covering hypothesis H2, can be found in Figure 37.

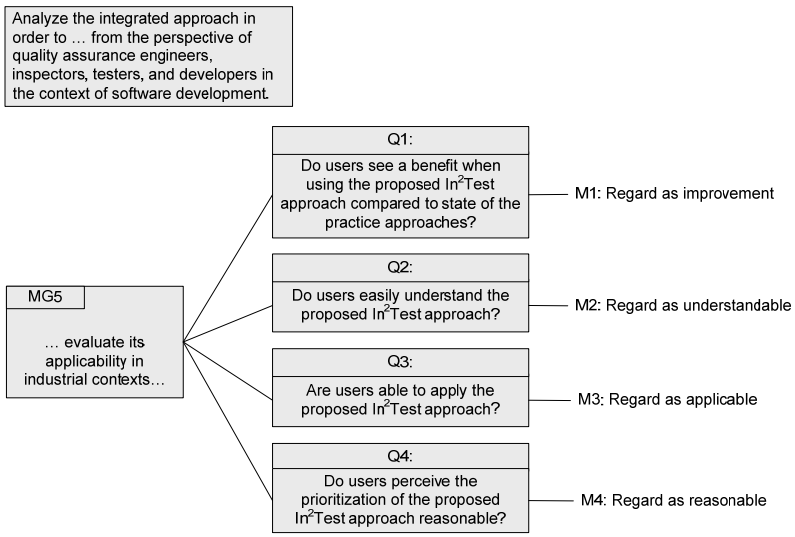


Figure 37 GQM plan, comprising measurement goals, questions, and metrics for hypothesis H2.

5.2.2 Validation Strategy

One main goal of the integrated inspection and testing approach In^{2}Test is to focus testing activities based on inspection results in order to save effort. The approach is able to focus test activities on parts of a system under test or on defect types. However, in order to rely on such prioritizations, a number of quality assurance runs have to be conducted, data have to be gathered, and assumptions have to be analyzed. Therefore, one focus of the evaluation was the application of the In^{2}Test approach in one or two quality assurance runs in order to analyze the general potential of the approach. Analyzing the validity of the underlying assumptions is rather difficult because a large amount of defect data gathered from several quality assurance runs in one or more environments is necessary. This will be part of future work. Furthermore, the approach considers test exit criteria, but their concrete influence has not been investigated in detail yet. This also belongs to future work. Finally, focusing on defect types with respect to an evaluation of effort reduction is difficult because a set of test cases has to be defined for such an analysis. However, the case studies presented in this thesis mainly use existing test cases. Therefore, further evaluation of the prioritization of defect types has to be done in the future.

Two empirical studies were conducted. In these studies, mainly measurement goals one to three were addressed, i.e., the effort, the effectiveness, and the efficiency of the In^{2}Test approach were evaluated. For this, two different tool developments were considered and corresponding quality assurance activities were performed. Besides addressing different hypotheses, the study design was also verified in the first case study. In the second case study, two quality assurance runs could be considered, which resulted in first insights regarding confidence in the underlying assumptions used by the integrated approach in a given environment. However, additional quality assurance runs would be needed in order to improve the validity of the initial focusing results.

Table 12 Research hypotheses and case studies.

Hypothesis	Case study I	Case study II
H1.1: Effort (system parts)	x	x
H1.2: Effort (defect types)	(x)	(x)
H1.3: Effectiveness (system parts)	x	x
H1.4: Effectiveness (defect types)	x	x
H1.5: Efficiency (system parts)	x	x
H1.6: Efficiency (defect types)	(x)	(x)
H1.7: Validity of assumptions (system parts)		x
H1.8: Validity of assumptions (defect types)		
H2.1: Applicability in industrial contexts		

Table 12 shows an overview of which hypotheses were evaluated in which study and which hypotheses belong to future work evaluations. Besides the two case studies conducted, two different designs for future experiments were determined (see Appendix B). The first design was applied during a pilot study with students. The study design could be verified, and the integrated approach helped the students during testing. However, due to the low number of testers, new technologies used, and time pressure, no concrete evaluation results were gained with respect to the stated hypotheses. A questionnaire was defined to allow analyzing the applicability of the approach in industrial contexts (see Appendix C). Finally, first evaluation results from an ongoing evaluation in an industrial context can be found in Appendix D.

5.3 Case Study 1: DETECT

5.3.1 Context of the Study

A Java tool called DETECT (dependability focused inspection tool) (Elberzhager et al., 2010a) was used for evaluating the integrated inspection and testing approach. The tool supports people who perform an inspection. At the time of the study, it mainly supported individual defect detection with the help of different kinds of reading support and allowed defining new checklists for use during defect detection. The different kinds of reading support include different tree structures (security goal indicator trees, short: SGIT (Peine et al., 2008), goal-indicator trees, short: GIT (Elberzhager et al., 2010b)), flow graphs (VID (Shields, 2008)), and two kinds of checklists (guided checklists (Elberzhager et al., 2009) and traditional checklists (Fagan, 1976; Laitenberger and DeBaud, 2000)). The tool provides a three-part view for the inspector: a tracking mode that documents each step on the left-hand side; the artifact to be checked in the middle; and the corresponding reading support (e.g., a checklist) on the right-hand side.

The tool was mainly developed by one developer. At the time of the study, it consisted of about 57,000 lines of code (without blank lines and comments), about 380 classes, and about 2,300 methods. The developer identified the critical code parts that should be inspected and discussed the selection of the code classes with the inspection team. In order to be able to finish the inspection within existing time constraints, it was decided to inspect only one kind of reading support, namely GITs (Elberzhager et al., 2010b). Overall, six inspectors checked 12 code classes, comprising about 7,300 lines of code. Each inspector checked four code classes, consisting of about 2,500 lines of code.

Table 13 shows the experience, respectively the knowledge, of the six inspectors regarding the inspection, the reading support to be checked, and the code structures (i.e., programming knowledge). Three values

(low, middle, high) are used for the classification. Finally, the checklist that was assigned is presented. Most got one checklist, except for inspector #4, who got two.

The testing activities were performed by the developer of the tool and one additional tester. Neither was involved in the inspection.

Table 13 Experience of inspectors and assigned checklists (o=low, +=middle, ++=high).

No.	Inspection knowledge	GIT knowledge	Programming knowledge	Defect detection focus
1	+	++	++	requirements
2	++	++	++	requirements
3	+	o	++	implementation
4	++	+	++	implementation, reliability
5	++	o	o	code documentation
6	++	++	+	code documentation

5.3.2 Design of the Study

First, a code inspection using checklists was performed by six computer scientists (step 1). Some checklist questions were taken from existing checklists that fit the given context (Burnstein, 2002), and new checklists questions were derived based on feedback from experienced developers and quality assurance engineers (see Appendix A for the checklists used). Overall, four different checklists were used, addressing requirements fulfillment, implementation, reliability, and code documentation. Each checklist consisted of three to eight questions and was assigned to those inspectors who could answer the questions effectively. Using so-called focused checklists that are adapted to the environment instead of using standard checklists improves effectiveness and is consistent with recommendations found in the literature (Gilb and Graham, 1993). Finally, the checklists were mapped to the relevant code classes by the developer of the tool so that each inspector checked four code classes. One experienced quality assurance engineer aggregated the findings from all inspectors. The developer analyzed each problem and decided whether a real defect had been found that had to be corrected or whether problems that were documented by an inspector were only due to a misunderstanding and could be removed without correction.

The next step was the quality monitoring of the resulting inspection defect profile (step 2). Reading rate, overall number of found defects, and defect distribution were considered.

Step 3 comprised the prioritization, i.e., a prediction of defect-prone parts and defect types had to be made. For this, four context-specific assumptions were determined that were to be evaluated. The intention was to generally analyze the relationships between inspection and test

defects, which is why very basic assumptions were used that are easy to apply.

Finally, selecting test cases and conducting focused testing activities were the last steps (step 4). However, in order to be able to evaluate the stated assumptions, the prioritized as well as the non-prioritized parts were tested by two testers. This enabled a detailed analysis of the assumptions regarding their appropriateness. First, a unit test of code classes was started. Test cases were derived using equivalence partitioning. Code classes that had been inspected and some additional ones identified as being most critical or important were selected for testing. However, it turned out that efficient unit testing was not possible due to bad testability of the code classes. The code structure did not suit the unit test approach (e.g., due to anonymous inner classes, anonymous threads, private fields and methods). To neutralize the problems of the code structure, mocking frameworks (i.e., a simulation of the behavior of code classes) were used. However, these frameworks turned out to be very complex for inexperienced testers.

Besides unit testing, a manual system test was conducted in order to analyze whether prioritization is possible between different levels (i.e., using defect information from the code level to guide tests for the system level). System tests were derived through typical walkthrough scenarios that followed the main functionality offered by the tool. Afterwards, the results from this testing activity were used as a baseline and compared to the prioritization when the defined assumptions were evaluated.

In the case study, the following variables were considered: The number of found defects was measured as defect content (absolute number) and defect density (number of defects per lines of code). For defect classification, ODC was used (ODC, 2002). Effort was measured in minutes; size was measured in lines of code (all lines of code were counted, including blank lines and commentary lines). Efficiency was calculated using the number of defects found per minute. Finally, three severity classes were distinguished.

5.3.3 Execution of the Study

Conducting the Inspection (Step 1)

Before the inspection was performed, a team meeting was held where the checklists were explained and an overview of the code to be inspected was presented. Afterwards, each inspector checked the assigned code classes with the assigned checklist and documented all findings and the place of occurrence in a problem list. In addition, defect type and defect severity were recorded. Each code class was checked by

at least two inspectors. Overall, 1450 minutes were spent on individual defect detection (ranging from 90 to 280 minutes consumed per inspector).

One experienced quality assurance engineer compiled the defect detection profile and the developer of the tool checked for each defect whether it had to be corrected or not. Of 236 problems found by all inspectors, 189 defects to be corrected remained. Table 14 shows the defect content (absolute number of defects) and defect density (absolute number of defects divided by lines of code) of the twelve inspected code classes. Table 15 shows a sorted list of the ODC-classified defects. 54 defects (e.g., unclear or missing comments) could not be classified according to any of the existing defect types.

Table 14 Defect content and defect density of each inspected code class.

Code class	*I*	*II*	*III*	*IV*	*V*	*VI*	*VII*	*VIII*	*IX*	*X*	*XI*	*XII*
Defect content	4	18	19	2	34	18	13	24	31	11	10	5
Defect density	.009	.021	.020	.008	.061	.057	.038	.031	.045	.026	.031	.016

Table 15 ODC-classified defects from inspection.

ODC defect types	**Sub-total**	**ODC defect types**	**Sub-total**	**Total**
algorithm / method	53	relationship	1	
checking	36	timing / serialization	0	
function / class / object	32	interface / o-o messages	0	
assignment / initialization	13	other	54	
Sub-total	134		55	**189**

Monitoring the Inspection Results (Step 2)

Because this was the first systematic quality assurance run of the DETECT tool, no historical data was available that could be used for monitoring the inspection results. Therefore, data from the literature was considered. The reading rate was about 630 lines of code per hour. The number is rather high compared to reading rates recommended in the literature (Barnard and Price, 1994), but consistent with experiences from industry (Cohen, 2006). Some reasons for the high number are that all lines of code were counted (including blank lines and comments) and that the individual checklists guided the inspectors to certain parts, whereas other parts were read faster. Finally, the overall number of found defects seemed reasonable compared to the first study and the distribution of minor, major, and crash defects was also similar to the first study.

Prioritization (Step 3)

In order to guide testing activities, a prediction of defect-prone parts and defects of those defect types that are expected to appear during testing was made, i.e., those parts and defect types were prioritized. Four assumptions were stated, including instructions for the prioritization. A short explanation of each assumption is given next. More details, empirical evidence, and explanations can be found in Section 4.4 and in (Elberzhager et al., 2010c, 2011a, 2011c).

Assumption 1: If no defined selection criterion is used to determine parts of a system that should be inspected, it is expected that a significant number of defects still remain in those parts that are not inspected (i.e., an equal distribution of defects is assumed). Consequently, testing should be focused especially on those uninspected parts of a system.

It is reasonable from a logical point of view that parts that had not been subjected to quality assurance yet would still contain defects. Therefore, parts not yet inspected should be especially considered during testing.

Assumption 2: Parts of a system where a large number of inspection defects are found indicate more defects to be found with testing (i.e., a Pareto distribution of defects is assumed). Consequently, testing should be focused especially on those inspected parts of a system that were particularly defect-prone.

In contrast, assumption 2 predicts defects in those parts of a system where the inspection already has found defects. A lot of empirical evidence exists that shows a Pareto distribution of defects, i.e., about 80 percent of the defects are often found in about 20 percent of the modules (Boehm and Basili, 2001).

Assumption 3: Inspection and testing activities find defects of various defect types with different degrees of effectiveness. For inspection, this comprises, e.g., maintainability problems. For testing, this comprises, e.g., performance problems. Consequently, inspection and testing activities should be focused on those defect types that they are most likely to find.

Inspection and testing complement each other, resulting in different defects found by those two quality assurance activities (Gilb and Graham, 1993). Therefore, it is reasonable to focus inspection and testing activities on those defect types that can be found best by using them.

Assumption 4: Defects of the defect types that are found most often by the inspection (i.e., a Pareto distribution of defects of certain defect types is assumed) indicate more defects of these defect types to be found with testing. Consequently, testing activities should be focused on those defect types that the inspection identified most often.

Ohlsson et al. (Ohlsson et al., 1996) states that the majority of quality costs are often caused by very few defect types. Consequently, focusing on certain defect types for testing based on inspections appears reasonable.

A derivation of concrete selection rules is skipped here because the assumptions are appropriate for an initial analysis in the given context. However, selection rules can be derived easily using the corresponding inspection defect profile.

Selecting Test Cases and Conducting the Testing Activities (Step 4)

To evaluate the integrated inspection and testing approach and the stated assumptions, testing activities were performed without considering the inspection defect profile for the prioritization (however, the inspection defects were corrected before testing activities started). 40, respectively 42, similar test cases were applied during system test by the two testers, covering the main functionality of the tool, i.e., different kinds of reading support, the interaction of reading support and an artifact to be inspected, the generation of a report of the findings, and the creation of a checklist were tested. In addition, some explorative testing was performed by the tester who did not develop the tool.

Table 16 Test results from system testing.

Tested functionality	Number of test cases		Number of defects found		Defect ids	
	tester 1	tester 2	tester 1	tester 2	tester 1	tester 2
reading support: GIT	3	3	1	1	id1, id8*	id1
reading support: SGIT	3	3	0	1	id9*	id1
reading support: GC	3	3	0	0	id10*	
reading support: VID	0	11	0	1	id11*	id1
reading support: CL	1	1	0	0		
interaction	15	8	5	2	id2, id3, id4, id6, id7, id12*	id2, id3
report generation	1	1	1	0	id5, id13*	
checklist creation	16	10	1	0	id4	

During the system test, seven additional defects regarding functionality were found by the two testers. Running the defined test cases took about 90, respectively 120 minutes. In addition, effort for explorative testing, test documentation, debugging, and correction was consumed, resulting in an overall test effort for both testers of about 14 hours. The distribution of defects with respect to functionality can be found in Table 16 (id1 – id7). Tester 1 found one defect (defect id 1) when testing the GIT reading support (which was inspected on the code level). However, this defect is independent of the concrete reading support. Tester 2 also found this defect when testing GITs, but also when testing SGITs or VIDs. Furthermore, most of the defects occurred when testing the interaction between reading support and the artifact view. Two more defects were found when testing checklist creation and report generation. In addition, tester 1 found six more usability problems that were equally distributed (id8* – id13*), i.e., for almost each functionality tested, one usability problem was found.

5.3.4 Results of the Study and Lessons Learned

H1.1 Effort (system parts): Overall, an effort reduction of between 8% and 31% was achieved by the integrated inspection and testing approach when focusing on system parts, i.e., testing certain functionality. Only the test case execution time was considered. The achievable effort reductions depend on different assumptions and selection rules.

H1.2 Effort (defect types): No explicit effort reduction was measured when testing was focused on defect types. However, the applied assumptions only selected a subset of defect types, which might result in effort reduction for testing.

H1.3 Effectiveness (system parts): An assumption and corresponding selection rules were found that led to finding the same defects when prioritizing only certain parts of the system for testing.

H1.4 Effectiveness (defect types): Using the inspection defect profile from the code level in order to prioritize defect types for system testing is rather difficult. An analysis showed that two defect types found during inspection are also found during testing. However, it is unclear how to use the defect type information from inspections for deriving system test cases based on the defect classification used.

H1.5 Efficiency (system parts): Our first objective was to check whether the inspection defect information could be used to predict defect-proneness within code classes in order to focus unit testing activities efficiently. Applying assumption 1, especially those code classes would have been prioritized for testing that had not been inspected. In

contrast, assumption 2 would have prioritized code classes in which many defects had been found by the inspection. Different concrete selection rules can be derived to operationalize the assumptions. For instance, when applying assumption 2, defect content or defect density could be used to express what 'large number of defects' means. Such an instantiation could have led to a concrete prioritization of code classes V and IX if a threshold had been defined to select code classes containing more than 30 defects and efficiency could have been calculated. A lot of different alternative selection rules are conceivable.

Unfortunately, the unit test activity could not be completed due to bad testability of the code and no new defects were found. Therefore, H1.5 could not be answered with respect to the unit level. Instead, the system test activity was used to analyze whether the code inspection results can provide valuable predictions for focusing system testing in an efficient manner. Assumptions 1 and 2 were applied accordingly. We were aware that this prioritization would mean a different level of granularity, because for system tests it is not possible to address certain code classes; rather, they are used to address functionalities.

Five different kinds of reading support and three additional tool functionalities were tested and revealed that most of the defects were found in parts that had not been inspected. One functional defect was found when applying the GIT reading support (which was also inspected); however, this defect occurred independent of the concrete reading support and was also found when testing with other kinds of reading support. Therefore, assumption 1 led to an appropriate prioritization, respectively prediction, of defect-proneness and would help in guiding system testing activities with reduced effort. Considering only the test execution effort, an effort reduction of between 8% and 31% depending on the concrete selection rules used is achievable, which results in an efficiency improvement of between around 9% and 44%. Omitting the GIT test cases led to the lowest reduction of test cases, while all functional defects were found. Omitting SGITs as well, which is a very similar reading support, increased the saved effort and thus, efficiency. In addition, when omitting test cases for checklists (i.e., reading support of low complexity), an effort reduction of up to 22% could be achieved with all functional defects still being found, i.e., efficiency improved by about 30%. Finally, the highest effort improvement with the same quality was shown by the fourth selection rule (i.e., focus on similar, but different reading support and further functionality). However, the absolute numbers for conducting the tests are rather low and test derivation, documentation, and further activities are not considered here due to imprecise numbers. Consequently, the figures have to be treated with caution. The results can be found in Table 17.

Consequently, assumption 1 would be appropriate for focusing testing activities efficiently, while assumption 2 (i.e., focusing on inspected parts) would not be valuable in our context in this system-level quality assurance run. One reason might be that the performance of the inspection was high and many problems were already found before testing. Another reason might be that informal quality assurance activities conducted before the systematic application of inspections and testing found some defects that had already been corrected. Finally, if we take a detailed look at the defect distribution of the inspection defects, no value is extremely prominent, which makes it harder to define a concrete threshold regarding which concrete code classes to prioritize. Therefore, one conclusion might be that due to similar defect density of all inspected code classes, testing different parts would be preferable.

Table 17 Effort savings when focusing on certain functionality during test execution.

Tested functionality	Effort (min)		Tested classes based on selection rules			
	tester 1	tester 2				
reading support: GIT	10	6				
reading support: SGIT	7	6	x			x
reading support: GC	7	6	x	x		
reading support: VID	0	30	x	x	x	
reading support: CL	3	2	x	x		
interaction	33	21	x	x	x	x
report generation	15	10	x	x	x	x
checklist creation	40	10	x	x	x	x
Effort (min)	206		190	177	159	142
Effort savings (%)	/		8%	14%	23%	31%
Efficiency	0.034		0.037	0.040	0.044	0.049
Efficiency improvement (%)	/		8.82%	17.65%	29.41%	44.12%

H1.6 Efficiency (defect types): Our second objective was to analyze the relationship between defect types identified in the inspection and during testing, and to analyze efficiency when focusing on certain defect types. Considering assumption 3, many of those inspection defects classified as 'other' were documentation problems (e.g., missing explanation of a method, unclear description). Such kinds of defects affect the maintainability of the product under test and are not detectable with testing, since they do not influence the observable functionality. Regarding testing, six additional usability problems were found by one tester (e.g., bad readability of parts of reading support). Such kinds of problems can be identified if a graphical user interface is used during testing, but are usually not found during the inspection.

In terms of maintainability and usability, it is rather easy to dedicate them to inspection respectively testing activities in order to find such problems. However, with respect to the ODC classification used for the

inspection defects, detecting a relationship to defects found during the system test is difficult due to the difference in granularity between code defect types and system defect types. A retrospective analysis of the seven functional defects found during testing revealed that most of them were classified as checking or algorithm / method defects, which exactly matches the two defect types identified most often during inspections (see Table 18). Nevertheless, it is still unclear whether it is possible to systematically select or derive system tests that can address such kinds of problems and how this could be done. Therefore, it is not possible to calculate concrete efficiency values in the given context. It may be that a defect classification, such as the ODC, is not suitable for guiding system test activities. An explorative study for identifying an appropriate defect classification would be necessary in this case. Finally, due to an uncompleted unit testing activity, no new insights regarding the relationships between inspection defect types and testing defect types could be obtained on the unit level.

Table 18 ODC-classified defects from inspection and system testing.

ODC defect types	Inspection	Testing
algorithm / method	53	2
checking	36	4
function / class / object	32	0
assignment / initialization	13	0
relationship	1	1
timing / serialization	0	0
interface / o-o messages	0	0
other	54	6
Total	**189**	**13**

Summary of the evaluation To conclude the main results, first trends have emerged that it may be possible to guide testing activities based on inspection results, i.e., to predict defect-prone parts and defect types based on inspection defect data, and to focus testing activities on certain parts and defect types. However, the quality and the efficiency of such focusing depend on the assumptions made in the given context. In the context of this case study, parts that had not been inspected contained additional defects that were found during testing. However, this can only be stated for defects found during system testing because unit testing could not be fully completed. With respect to defect types, especially maintainability defects were found during inspections, while usability problems were found during testing. An effort reduction for test case execution of up to 31% was achievable when focusing on parts of the system with a comparable quality being achieved, which is an efficiency improvement of between around 9 and 44%. Though it was possible to focus defect types for testing based on the inspection results, concrete efficiency values could not be calculated.

The applicability of the approach could be demonstrated. One important prerequisite for a suitable application of the In²Test approach on the same system level is appropriate testability, which has to be ensured by the design of further evaluations.

Finally, considering the overall effort, effectiveness, and efficiency of the inspection and testing activities, a small improvement of the efficiency of the In²Test approach compared to a non-integrated approach could be found (between 0.9% and 4%). One reason for the small improvement is the small number of test defects found. One explanation for this might be that during development, defects that are observed are corrected directly, and are then not counted during the explicit testing activities. In addition, the inspection already found a lot of defects, which were corrected after the inspection, and which then could not be found during testing. Table 19 shows an overview of the results. With respect to the test defects found, we only considered the functional defects (because the analysis of H1 also considers mostly those kinds of defects, and usability problems are rather found during a visual inspection of the graphical user interface than during functional testing). Furthermore, we considered the effort for conducting the defect detection during the inspection, respectively the effort for conducting the test execution. In our context, the inspection is superior to testing, which is also consistent with studies from different contexts (Laitenberger and DeBaud, 2000; Elberzhager, 2005). Performing the inspection only would lead to the highest efficiency value. However, in this case, not all defects would have been found. With respect to the scope of this thesis, it could be seen that the overall efficiency (i.e., reduced effort at same effectiveness) of the In²Test approach is slightly improved compared to a non-integrated approach, which is true for all four selection rules SR1-SR4.

Table 19 Comparison of different quality assurance processes.

No.	Approach	# defects found	Effort needed (minutes)	Efficiency
1	Inspection	189	1450	0.1303
2	Testing (after inspection)	7	206	0.0340
3	Non-integrated inspection and testing	196	1656	0.1184
4.1	In²Test (SR 1)	196	1640	0.1195
4.2	In²Test (SR 2)	196	1627	0.1205
4.3	In²Test (SR 3)	196	1609	0.1218
4.4	In²Test (SR 4)	196	1592	0.1231

5.3.5 Limitations of the Study

Next, a discussion of what we consider to be the most relevant threats to validity is given (Wohlin et al., 2000).

Conclusion validity: The number of testers and the number of found test defects was low. One reason might be the low experience regarding testing. Consequently, no statistically significant data could be obtained. However, the results showed a trend that the integrated inspection and testing approach is able to guide testing activities. The figures for effort saving are based only on certain parts of the test execution and are rather low. With this, an initial idea is given of what potential effort reductions might be achievable.

Construct validity: To demonstrate the integrated approach, different assumptions were derived in our context. Four assumptions were used and analyzed regarding their suitability. However, more assumptions are conceivable and may lead to better or worse predictions. In the inspection, no standard checklists were used. Finally, the selection of ODC was reasonable when focusing on the unit test level, but it might not be suitable for the system level during testing.

Internal validity: The subjects selected for the inspection and for the testing activity may have influenced the number of defects that were found. However, the effect was slightly reduced by using checklists that focus an inspector on certain aspects in the code and by using equivalence partitioning, respectively addressing the main functionality, for the testing activity. The developer of the tool also tested the tool, which resulted in a lower number of defects. However, a second tester not involved in tool development also tested the tool. Ultimately, the defects could be classified differently.

External validity: The DETECT tool inspected and tested in the case study is one example to which the integrated inspection and testing approach was applied. Few test defects were found that could be used for the analysis of the assumptions. A larger software product, as typically developed by software companies, is expected to result in more test defects to be found. Assumptions have to be evaluated anew in each new environment, meaning that the conclusions drawn with respect to the used assumptions cannot be generalized directly. Finally, the results can only be transferred to a context where a comparable number of defects are found during inspection and testing activities.

5.4 Case Study 2: JSeq

5.4.1 Context of the Study

The integrated inspection and testing approach was applied twice in the same context. The first run was primarily intended to gain experience with the approach in the new context and to obtain data for the EDB. The second run, which was performed six months after the first one,

used the gathered data to perform more meaningful inspection monitoring and a more fine-grained prioritization with respect to defect content and defect types (i.e., additional selection rules were defined). Both the one-stage approach focusing only on defect-prone parts of the system under test and the two-stage approach focusing on defect-prone parts and defect types were applied.

The artifact to be checked was a Java prototype tool, which had mainly been developed by one developer. The tool supports practitioners in performing sequence-based specifications (Prowell et al., 1999). At the point of the evaluation, it consisted of 76 classes, over 650 methods, and about 8,500 lines of code (LoC). In both runs, those code parts containing the main business logic were selected for application of the approach. In the first run, four code classes with a total of about 1,000 LoC were chosen. In the second run, four different code classes with a total of about 2,400 LoC were selected. The code classes of the first and the second run differed due to continuous development of the tool.

In both runs, four inspectors conducted the code inspection. In the first run, one inspector had very good knowledge about inspections, but only limited programming knowledge, while the remaining inspectors had very good programming experience, but only limited inspection knowledge. In the second run, one programmer who was no longer available was replaced by an experienced inspector, again with limited programming experience. Due to project restrictions, the testing activity was performed by a single developer of the tool prototype, who was not involved in the inspection.

5.4.2 Design of the Study

The case study described in this section followed a similar design as the first evaluation of the integrated inspection and testing approach.

Both runs of the case study followed the same design. First, a code inspection (step 1) was performed by four computer scientists. To prepare the code inspection, two developers and two additional persons involved in the tool prototype development gathered five relevant quality properties of the system. Afterwards, one inspection expert derived individual checklists for each collected quality property (see Appendix A). Some checklist questions were similar to those mentioned in (Burnstein, 2002). Each checklist consisted of between four and nine questions and focused on requirements fulfillment, functional aspects, extensibility, performance, and reliability. Using checklists adapted to the environment instead of standard checklists improved defect detection and is consistent with recommendations made by authors of inspection literature (Wiegers, 2002; Gilb and Graham, 1993). Based on these checklists, the inspection was performed. The inspectors were selected

systematically so that each checklist could be answered effectively by the corresponding inspector. After the inspection was finished, experienced quality assurance engineers compiled the inspection defect profile (one expert in the first run, two experts in the second run). In addition, each defect found by the inspectors was discussed in a group session until agreement on the classification of the defects was achieved.

After the inspection was finished, inspection quality monitoring (step 2) was done by those experts who put together the inspection defect profile. In the first run, inspection data from the literature, such as reading rate (i.e., inspected LoC per hour per inspector), was used to evaluate the validity of the results. In the second run, the results from the first run were treated as historical data and, consequently, used for the comparison in order to perform the monitoring step.

Step 3 comprised the prioritization, which was done based on the inspection defect profile and assumptions made to focus the testing activity.

Following the integrated inspection and testing approach, focused testing activities would be the next step. However, in order to be able to evaluate whether the prioritization of the defect types is suitable based on the inspection defect profile, one developer of the tool prototype first performed a non-integrated testing activity (step 4), i.e., an experience-based testing activity including equivalence partitioning was conducted without using any information from the prioritization step. The defect results from the experience-based testing activity were used as a baseline and compared with the prioritization (i.e., step 3) of the integrated approach in order to answer the defined research questions.

The following variables were taken into account in the case study: Effort was measured in minutes. Size was measured in lines of code, with a distinction being made between the complete length of a code class and the mean method length of a code class. Finally, McCabe complexity was used as complexity measure. An approach A is of comparable quality to another approach B if at least the same number of defects is found. Efficiency was calculated using the number of defects found per minutes. For applying the assumptions, respectively the selection rules, different quantifications of variables were used. For example, regarding the number of defects, absolute number and defect density were taken. In addition, a severity classification was used (minor, major, crash). Furthermore, ODC (ODC, 2002) was chosen for defect classification due to its high industry orientation. Experiences from industry show that ODC promises to be a suitable classification (Bridge and Miller, 1997). Moreover, the classification is able to classify both inspection and testing defects, which is a necessary prerequisite for the integrated two-stage approach. The following seven defect types, as suggested by ODC, were taken into account (for an explanation and examples of each defect

type, see (ODC, 2002): Assignment / Initialization; Checking; Algorithm / Method; Function / Class / Object; Timing / Serialization; Interface / O-O Messages; Relationship.

5.4.3 Execution of the Study

Besides answering the determined research questions in the two runs of the case study, the first run of the case study was also used to evaluate the general applicability of the integrated approach.

First Run – Conducting the Inspection (Step 1)

Before the inspection was performed, all inspectors and one tool developer met to familiarize themselves with the code and the checklists and to answer questions. Afterwards, each inspector individually checked all four code classes with a different checklist and documented the findings in a bug-tracking tool, which took a total of 435 minutes for all inspectors. Next, a group session of the inspectors took place in which the inspection issues were classified jointly according to the ODC. Based on these results, one quality assurance engineer put together the inspection defect profile, which is shown in Table 20 (defect content) and Table 21 (sorted list of ODC-classified defects).

Table 20 Inspection defect profile – Defect content.

Code class	*SBSTreeState*	*SBSTreeComperator*	*SBSTree*	*Main*	**Total**
Defect content	26	6	27	8	**67**

In total, 67 issues were found by the inspectors, of which 48 could be classified according to ODC. 19 issues could not be classified and were treated as Other defects (e.g., unclear or missing code comments).

Table 21 Inspection defect profile – ODC-classified defects.

Severity / ODC defect type	minor	major	crash	Sub-total
Algorithm / Method	11	6	1	18
Checking	4	6	3	13
Interface / O-O Messages	8	1	0	9
Function / Class / Object	2	3	3	8
Timing / Serialization	0	0	0	0
Assignment / Initialization	0	0	0	0
Relationship	0	0	0	0
Other	18	1	0	19
Total	**43**	**17**	**7**	**67**

First Run – Monitoring the Inspection Results (Step 2)

Unfortunately, no context-specific historical inspection data was available for the tool prototype in order to monitor the inspection results. Instead, data from the first study of the integrated inspection and testing approach (Elberzhager et al., 2012) was used since the environment was similar. In addition, suggestions from the literature were taken and compared with metrics, such as an inspector's reading rate. The reading rate was 550 LoC per hour, which is higher than suggestions from Barnard and Price (Barnard and Price, 1994), but consistent with experiences from industry (Cohen, 2006) and results from the first study. One reason for the high number might be that all lines of code were counted (e.g., blank lines, commentary lines). Another reason is that different checklists were used, which pointed the inspectors to different parts in the code. Consequently, some parts were read in detail while other parts were just scanned or not read at all. Furthermore, low inspection experience of some inspectors and time constraints may be further reasons. Finally, though this was just a gut feeling of the inspection experts, 67 issues found per thousand lines of code seemed to be appropriate.

First Run – Prioritization (Step 3)

One assumption was used for the prioritization of the first stage:

S(tage)1-A(ssumption)1: Parts of the code where a large number of inspection defects are found indicate more defects to be found with testing (i.e., a Pareto distribution of defects is assumed).

A large number of different studies performed in various environments has shown that an accumulation of defects, i.e., a Pareto distribution,

can be observed rather than an equal distribution of defects. A lot of studies have confirmed this observation (see Section 4.4.4), resulting in an 80/20 rule (Boehm and Basili, 2001; Shull et al., 2002). Based on the empirically validated hypothesis that an accumulation of defects can often be observed, assumption A1 was defined with respect to the integrated inspection and testing approach. The derived selection rule (SR) is:

> *S1-A1-SR1:* Prioritize those code classes for testing that have by far the highest defect content (i.e., absolute number of defects) based on the inspection defect profile.

In order to be able to evaluate the prioritization of defect types, one assumption was defined for the second stage:

> *S2-A1:* Defects of the defect types that are found most often by the inspection indicate more defects of the defect types to be found with testing (i.e., a Pareto distribution of defects of certain defect types is assumed).

An accumulation of defects of certain defect types could also be observed in several studies rather than an equal distribution of defect types, independent of a concrete defect classification (see Section 4.4.4). Thus, the derived selection rule is:

> S2-A1-SR1: Prioritize those two ODC defect types for testing that appeared most often based on the inspection defect profile.

The assumptions and selection rules given here are kept simple in order to support the evaluation. Based on the two selection rules (one for each stage), the code classes expected to be most defect-prone and the defect types expected to appear most often were prioritized. Consequently, SBSTreeState and SBSTree were selected at stage 1, and Algorithm / Method and Checking were chosen at stage 2. For the one-stage approach, only the assumption for the first stage and the derived selection rule are relevant, while for the two-stage approach, both assumptions and selection rules are relevant, resulting in one combined prioritization.

First Run – Selecting Test Cases and Conducting the Testing Activity (Step 4)

In order to be able to evaluate the proposed approach, one developer first performed the testing activity of the four already inspected code

classes without using the prioritization information. Afterwards, the defect results from this experience-based testing activity including equivalence partitioning were used as a baseline for analyzing and evaluating the prioritization of the integrated inspection and testing approach.

Second Run – Conducting the Inspection (Step 1)

Before the inspection was conducted in the second run, an overview meeting was performed again. The developer of the prototype tool explained the structure of the code classes and the relationships among them. Furthermore, the ODC defect classification was discussed in order to gain the same understanding of the defect types. Due to the experiences obtained from the first run, the inspectors were to classify each defect on their own. After the meeting, each inspector checked all four code classes with a different checklist. Each issue found was documented in a bug-tracking tool, with a short description of the problem, the place where it was found, the severity, and the ODC defect type being recorded. The inspection (i.e., the defect detection task) took about three to four hours per inspector, resulting in an overall effort of 835 minutes.

Table 22 Inspection defect profile – defect content, defect density, and severity classes.

Code class	*Sequence*	*SequenceTableModel*	*SimpleKeyedTableModel*	*SimpleOrderedKeyedT.*	**Total**
defect content (dc)	14	40	39	7	**100**
defect density (dd)	0.061	0.029	0.056	0.061	-
dc minor defects	10	31	25	5	**71**
dc major defects	3	9	14	1	**27**
dc crash defects	1	0	0	1	**2**
dd minor defects	0.043	0.023	0.036	0.044	-
dd major defects	0.013	0.007	0.020	0.009	-
dd crash defects	0.004	0.000	0.000	0.009	-

After the inspection had been performed, all issues were analyzed by two experienced QA engineers in order to eliminate duplicates and comments such as improvement suggestions and questions, and put

together the defect profile. In total, 100 defects remained. Next, the classification of each defect was checked. If the two quality assurance engineers did not agree with the classification of a defect, this was discussed with the corresponding inspector.

The final inspection defect profile can be found in Table 22 and Table 23. With respect to the inspection results, the defect content, the defect density, as well as the defect content and defect density for each severity class per code class are shown. With respect to the ODC, 54 defects could be classified, while 46 defects were treated as Other defects.

Table 23 Inspection defect profile – sorted list of ODC-classified defects.

Severity / ODC defect type	minor	major	crash	Sub-total
Function / Class / Object	10	14	0	24
Algorithm / Method	9	4	0	13
Relationship	7	0	0	7
Checking	1	2	2	5
Interface / O-O Messages	4	1	0	5
Assignment / Initialization	0	0	0	0
Timing / Serialization	0	0	0	0
Other	40	6	0	46
Total	**71**	**27**	**2**	**100**

Second Run – Monitoring the Inspection Results (Step 2)

Since the inspection results from the first run were available, the new inspection results could be compared with historical inspection results, as suggested by Aurum et al. (Aurum et al., 2002), who state that "historical data may help to determine the quality of the current inspection". Comparing the inspection data is justified by the same context and is more meaningful than using data from different environments. Table 24 summarizes some inspection metrics. The reading rate in the second run was slightly higher, namely 685 LoC per hour compared to 550 LoC per hour in the first run. The reasons given above for explaining the high reading rate in the pilot study are also applicable for the case study. The average number of defects found was a bit lower in the second run (42 defects per thousand LoC compared to 67 defects per 1000 LoC). An explanation for this is that some code parts were not commented very well and consequently, the inspectors did not inspect some parts in detail due to unclarities. The distribution of minor, major, and crash defects is similar. Finally, the rate of classified defects was lower compared to the first run. One reason was again that many of those Other defects emphasized missing or bad comments. Thus, although the performance of the inspectors was slightly lower

compared to the first run, the results seemed reasonable enough to allow them to be used in the prioritization step.

Table 24 Inspection metrics of 1st and 2nd run of the case study.

Metrics	Inspection first run (historical data)	Inspection second run (current data)
Total number of defects	67	100
Number of problems / 1000 LoC	67 issues per 1000 LoC	42 defects per 1000 LoC
Overall effort (minutes)	435	835
Average reading rate	550 LoC per hour	685 LoC per hour
ODC classified defects in %	72%	54%
Severity in % (minor / major / crash)	64% / 25% / 11%	71% / 27% / 2%

Second Run – Prioritization (Step 3)

Focusing of the testing activity for the first stage was mainly based on the inspection profile and three product metrics. Three assumptions were used, with S1-A1 being the same as in the first run. Some rationales and empirical evidence for each assumption are presented next.

S(tage)1-A(ssumption)1: Parts of the code where a large number of inspection defects are found indicate more defects to be found with testing (i.e., a Pareto distribution of defects is assumed).

As mentioned for the first run, a number of studies and experiments confirm this assumption.

S1-A2: Parts of the code where a large number of inspection defects are found (i.e., a Pareto distribution of defects is assumed) and which are of small size indicate more defects to be found with testing.

A size metric is often used to prioritize defect-prone parts and thus, to focus a testing activity. Though this metric is often applied, a number of studies have shown inconsistent results when size is applied as the sole metric for predicting defect-prone modules. Emam et al. (Emam et al., 2002) state that if models are built to predict fault-proneness, more variables than just size should be used (see Section 4.4.4). Thus, inspection results were combined with two different size metrics, namely class length and mean method length.

S1-A3: Parts of the code where a large number of inspection defects are found (i.e., a Pareto distribution of defects is assumed) and which are of high complexity indicate more defects to be found with testing.

Besides size, complexity is another popular metric often used to focus a testing activity (see Section 4.4.4). Schröter et al. (Schröter et al., 2006) note that new metrics or combinations of existing metrics should be used to study the relationship between complexity and the presence of bugs. Thus, in order to improve the prioritization of code classes expected to be most defect-prone, the inspection results were combined with one complexity metric, namely McCabe's cyclomatic complexity, with a focus on code classes that have high complexity.

Table 25 Assumption metrics and their corresponding values.

Code class	Class length (LoC)	Mean method length (LoC)	Mean McCabe complexity
Sequence	231	3.28	1.78
SequenceTableModel	1364	13.54	3.90
SimpleKeyedTableModel	701	8.11	2.91
SimpleOrderedKeyedT.	115	7	2

Overall, 32 selection rules were initially derived from the three assumptions manually, mainly based on a brainstorming session. Following the assumptions, selection rules with respect to the inspection results were derived first, resulting in a focus on the most defect-prone parts. This means that defect content, defect density, and those two metrics combined with three severity classes were determined, resulting in eight concrete selection rules. Second, common metrics identified during related work were considered and combined with the inspection results. This comprised two different size metrics and one established complexity metric. The eight defect metrics defined before were combined with each product metric, resulting in the 24 additional selection rules. Table 25 shows the values of the assumption metrics for each code class (the last two metrics were calculated using the metrics tool (Metrics, 2010)). Based on the inspection defect profile and the defined metrics, the derived selection rules were applied to prioritize code classes for the test activity.

In order to be able to perform two-staged prioritization and do a more fine-grained analysis of the defect type prioritization, two additional selection rules were derived for stage 2 compared to the first run, resulting in the following three selection rules:

S2-A1-SR1: Prioritize those two ODC defect types for testing that appeared most often based on the inspection defect profile.

S2-A1-SR2: Prioritize those three ODC defect types for testing that appeared most often based on the inspection defect profile.

S2-A1-SR3: Prioritize those three ODC defect types for testing that appeared most often based on the inspection defect profile. In addition, consider those defect types that have high severity and appeared most often both in past inspection and in past testing activities.

However, in order to avoid an exploding number of combinations of stage 1 and stage 2 assumptions and selection rules, only one assumption and the following selection rule of stage 1 was used for the combined prioritizations:

S1-A1-SR1: Prioritize those code classes for testing that have by far the highest defect content (i.e., absolute number of defects) based on the inspection defect profile.

The output of applying this single selection rule of stage 1 is combined with each output of applying the three selection rules of stage 2, resulting in three combined prioritizations.

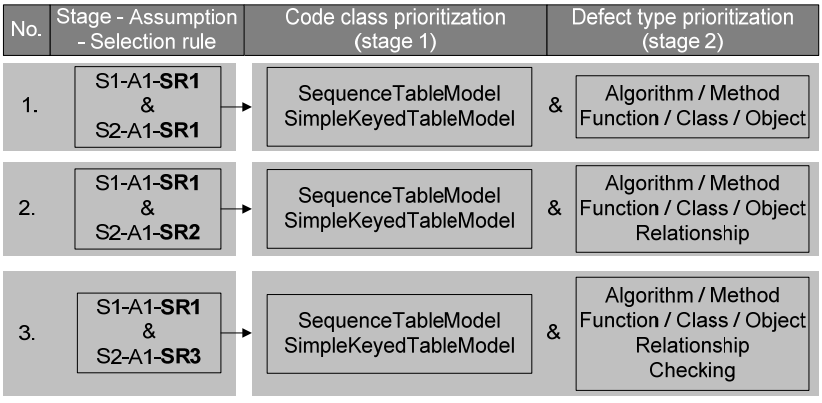


Figure 38 Combined prioritization of code classes and defect types based on applied selection rules.

Figure 38 shows the concrete prioritizations of code classes (based on Table 22) and defect types (based on Table 23 and Table 27) for each of those combined prioritizations.

Second Run – Selecting Test Cases and Conducting the Testing Activity (Step 4)

Similar to the first run, one developer of the tool prototype performed the experience-based testing activity without prioritization information. Besides the four inspected code classes, four additional, highly connected code classes were tested.

5.4.4 Results of the Study and Lessons Learned

First Run

An experience-based testing activity of all four code classes including equivalence partitioning was performed by one person. Overall, seven additional defects were found, with three defects being found in SBSTreeState and four defects in SBSTree (see Table 26). About 70% of these defects were critical ones (i.e., classified as major or crash). The prioritization of the code classes fits exactly to those code classes where the defects occurred and consequently, assumption A1 could be confirmed in the given context.

The inspection defect profile in combination with the assumption was suitable for prioritizing certain parts of the code and for focusing the testing activity, and thus, the applicability of the approach could be shown. The main results of the first run of the case study with respect to the research questions can be summarized as follows.

H1.1 Effort (system parts): It was not possible to determine any concrete test effort reduction. This was mainly due to continuous development, rapidly changing code, and an unsystematic test.

H1.2 Effort (defect types): No explicit effort reduction was measured when testing was focused on defect types.

H1.3 Effectiveness (system parts): Based on assumption S1-A1, all defects could be found with the integrated approach, just as with the non-integrated approach.

Table 26 Number of defects found (defect content) by inspection and testing per code class.

Code class / Number of defects found by	SBSTreeState	SBSTreeComperator	SBSTree	Main	Total
inspection	26	6	27	8	**67**
testing	3	0	4	0	**7**
Prioritization	**x**		**x**		

H1.4 Effectiveness (defect types): Table 27 shows a comparison of the defect types that appeared during inspection and testing. The two prioritized defect types based on the inspection defect profile were again found most often by the testing activity. No additional defect types were found. This means that in our context, the integrated two-stage inspection and testing approach was able to prioritize those defect types within the selected code classes for testing that actually appeared during the testing activity, and that no defect types were missed.

Table 27 Defect types found by inspection and testing, and prioritized defect types for selection rule of stage 2.

ODC defect type	Inspection defects	Testing defects	Prioriti-zation
Algorithm / Method	**18**	**4**	**x**
Checking	**13**	**3**	**x**
Interface / O-O Messages	9	0	
Function / Class / Object	8	0	
Timing / Serialization	0	0	
Assignment / Initialization	0	0	
Relationship	0	0	

H1.5 & H1.6 Efficiency (system parts and defect types): In order to check the feasibility of the approach in the new context, only one assumption and one derived selection rule was used for the sake of simplicity (for both system parts and defect types). The assumption and selection rule used led to appropriate predictions. However, due to a lack of clear effort numbers, concrete efficiency values could not be calculated.

Second Run

During the non-integrated testing activity in the second run, six additional defects were found (i.e., defects not already found by inspection, see Table 28). Five test defects were found in SimpleKeyedTableModel (code class three) and one test defect in TableUtil, which is invoked by that code class. Overall, 16 hours of test effort were needed to test all eight code classes with the non-integrated approach, including correction and documentation (see upper part of Table 29). Based on the test result, the integrated inspection and testing approach was applied to evaluate the determined research questions.

Table 28 Number of defects found (defect content) by inspection and testing per code class.

Code class / Number of defects found by	Sequence	SequenceTableModel	SimpleKeyedTableModel	SimpleOrderedKeyedT.	MultipleKeyedTableLinkSet	SimpleKeyedTableLinkSet	SimpleKeyedTableLink	TableUtil	Total
inspection	14	40	39	7	-	-	-	-	**100**
testing	0	0	5	0	0	0	0	1	**6**

H1.1 Effort (system parts): Overall, an effort reduction of between 6% and 34% was achieved by the integrated inspection and testing approach when focusing on code classes in the given context (considering the effective selection rules). If considering only the test execution effort (which includes test case specification effort, but not documentation and correction effort), an effort reduction of between 9% and 50% was achieved. The achievable effort reductions depend on different assumptions and the concrete selection rules. For example, fifteen hours of test effort were needed when code classes one, three, and four were tested only, i.e., code class two was not prioritized and thus, omitted, resulting in an effort reduction of one hour needed to test this code class in the non-integrated testing activity. The test documentation and the correction activity remained stable in the integrated approach.

Though not considered explicitly, traditional product metrics can be compared with this result. Selecting code classes with high complexity, low mean method length, or a large class would lead to effort reductions of between 6% (mean method length) and 9% (high complexity, large code class).

Some selection rules select a set of code classes or a single code class that was not defect-prone. In this case, the effort reduction considering test execution was between 34% and 37% (if the correction effort is

skipped in such a case, the overall effort reduction is about 62%). Finally, some selection rules were not able to select any code class, which would result in a 100% effort reduction; however, such selection rules are of no practical relevance.

H1.2 Effort (defect types): No explicit effort reduction was measured when testing was focused on defect types. However, the applied selection rules only selected a subset of defect types, which might result in effort reduction for testing.

Table 29 Effort of the non-integrated test and different effort reductions of the prioritized test.

	Test Effort	Tested code classes	Effort reduction
Test execution	11.0 h		
Test documentation	01.0 h		
Correction	04.0 h		
Non-integrated test effort	16.0 h	1-8	
Prioritized code classes:	15.0 h	1+3+4	6.25%
Test effort reduction with	14.5 h	1+3, 2+3	9.38%
quality preservation	10.5 h	3	34.38%

H1.3 Effectiveness (system parts): In the bottom part of Table 29, only those selections of code classes and the resulting test effort reduction are shown in which the defect-prone class SimpleKeyedTableModel (class three, including one calling class) is contained. Consequently, this class has to be tested with the integrated approach by all means in order to achieve comparable quality. However, 18 of the initially defined selection rules led to quality preservation, while 14 of them did not select the defect-prone class three.

H1.4 Effectiveness (defect types): Table 37 shows the defect types found by inspection and testing activities (sorted by number of inspection defects) and the prioritized defect types per selection rule of stage 2. Depending on the concrete selection rules, those defect types could be selected of which more defects are found. However, only one of three selection rules prioritized all relevant defect types.

H1.5 Efficiency (system parts): Next, a detailed analysis of each assumption and the derived selection rules with respect to efficiency (calculated as the number of defects per minute) is given in order to analyze which ones led to appropriate selections of code classes in the given context in an efficient manner. Note that during this case study, the integrated approach was only applied to prioritize code classes that were also inspected. Before the application of the selection rules, which was done by two experienced QA engineers, a clarification of what

"high", "low", "small", and "large" meant in the given context was done. In most cases, the definition was obvious or discussed until the same understanding was gained. The calculation of efficiency values can be found in Table 30. The complete test effort was taken into account for the calculation because of the more realistic view. However, if only the test execution time was considered, an efficiency improvement of up to about 100% could be achieved.

Table 30 Calculation of efficiency values.

Efficiency values	Calculation	Efficiency improvements
6 defects / 960 minutes = 0.00625	n/a	n/a
6 defects / 900 minutes ~ 0.00667	(0.00667-0.00625)*100/0.00625	6.72 %
6 defects / 870 minutes ~ 0.00690	(0.00690-0.00625)*100/0.00625	10.40 %
6 defects / 630 minutes ~ 0.00952	(0.00952-0.00625)*100/0.00625	52.32 %

S1-A1: Overall, the selection rules for assumption S1-A1 led to suitable selections of code classes for testing and an efficiency improvement could be observed for six of eight selection rules. Table 31 shows each applied selection rule for assumption S1-A1, the selected classes, and the achieved effort reduction together with the efficiency improvement. If the defect-prone class three is selected, '+' marks a suitable quality Q of the selection rule, otherwise '-' is chosen and no effort reduction is calculated (expressed as "/" in Table 31 - Table 34) because no comparable quality is achieved. For example, using the selection rule A1.01 'defect content (high)' resulted in prioritizing the code classes SequenceTableModel (40 inspection defects found) and SimpleKeyedTableModel (39 inspection defects found) and consequently, in an effort reduction of about nine percent, or in other words, an efficiency improvement of about 10 percent. The same selection of code classes was done when choosing classes containing the largest numbers of minor and major defects (A1.03 and A1.04, see Table 22 for concrete values). The two selection rules A1.02 and A1.06, which focus on crash severity, led to class selections that did not contain any defects or, more precisely, did not select the defect-prone class three, and therefore did not show an improvement of efficiency. Due to the high criticality of crash defects, already one such defect within a code class was interpreted as high defect content or high defect density for this severity class. However, due to the very low number of crash defects found, a different interpretation is conceivable.

With respect to defect density in general (A1.05), code classes one, three, and four were selected (see Table 22, where the defect density of the three mentioned classes is about twice as high as that of code class two). The resulting effort reduction was about six percent (efficiency improvement: about 6.67 percent). When prioritizing code classes with high defect density for defects classified as minor or major (A1.07 and A1.08), an effort reduction of between six and nine percent is

achievable. Based on the described results, six of eight selection rules led to an appropriate selection of code classes with improved efficiency. Consequently, based on those selection rules, assumption S1-A1 could be confirmed as efficient in our context (Table 31).

Table 31 Evaluation results of assumption S1-A1.

Assumption 1: Parts of the code where a large number of inspection defects are found indicate more defects to be found with testing.					
No.	**Selection rule:** Focus testing on those code classes in which the inspection determined..	**Prioritized classes**	**Effort reduction**	**Efficiency improve-ment**	**Q.**
A1.01	defect content (high)	2, 3	9.38%	10.4 %	+
A1.02	crash severity (defect content (high))	1, 4	/	/	-
A1.03	major severity (defect content (high))	2, 3	9.38%	10.4 %	+
A1.04	minor severity (defect content (high))	2, 3	9.38%	10.4 %	+
A1.05	defect density (high)	1, 3, 4	6.25%	6.7 %	+
A1.06	crash severity (defect density (high))	1, 4	/	/	-
A1.07	major severity (defect density (high))	1, 3	9.38%	10.4 %	+
A1.08	minor severity (defect density (high))	1, 3, 4	6.25%	6.7 %	+

Table 32 Evaluation results of assumption S1-A2 with respect to class length.

Assumption 2: Parts of the code where a large number of inspection defects are found and which are of small size indicate more defects to be found with testing.					
No.	**Selection rule:** Focus testing on those code classes in which the inspection determined..	**Prioritized classes**	**Effort reduction**	**Efficiency improve-ment**	**Q.**
A2.01	defect content (high) & class length (low)	-	/	/	-
A2.02	crash severity (defect content (high)) & class length (low)	1, 4	/	/	-
A2.03	major severity (defect content (high)) & class length (low)	-	/	/	-
A2.04	minor severity (defect content (high)) & class length (low)	-	/	/	-
A2.05	defect density (high) & class length (low)	1, 4	/	/	-
A2.06	crash severity (defect density (high)) & class length (low)	1, 4	/	/	-
A2.07	major severity (defect density (high)) & class length (low)	1	/	/	-
A2.08	minor severity (defect density (high)) & class length (low)	1, 4	/	/	-
Assumption 2: Parts of the code where a large number of inspection defects are found and which are of large size indicate more defects to be found with testing.*					
A2.01*	defect content (high) & class length (high)	2, 3	9.38%	10.4 %	+
A2.02*	crash severity (defect content (high)) & class length (high)	-	/		-
A2.03*	major severity (defect content (high)) & class length (high)	2, 3	9.38%	10.4 %	+
A2.04*	minor severity (defect content (high)) & class length (high)	2, 3	9.38%	10.4 %	+
A2.05*	defect density (high) & class length (high)	3	34.38%	52.3 %	+
A2.06*	crash severity (defect density (high)) & class length (high)	-	/		-
A2.07*	major severity (defect density (high)) & class length (high)	3	34.38%	52.3 %	+
A2.08*	minor severity (defect density (high)) & class length (high)	3	34.38%	52.3 %	+

S1-A2: With respect to the second assumption, two different size metrics were used in determining the selection rules, namely the total length of the code class and the mean method length within the code class. The selection rules, the corresponding code classes that were prioritized, and the effort reduction are shown in Table 32 and Table 33.

With respect to the class length, all eight selection rules led to inappropriate prioritizations of code classes and consequently showed no efficiency improvement. None of the eight selection rules chose the defect-prone code class three (either no code class fulfilled both criteria or classes one and four were selected). To further analyze the combination of inspection defects and class length, an alternative assumption S1-A2* was defined, which combined defect accumulation with code classes of large size (instead of small class length). As mentioned in the rationales for S1-A2, some studies showed that large-sized modules are more defect-prone than small-sized modules (e.g., Emam et al., 2002). One reason in our context could be that a lot of functionality was put into large code classes, and that therefore more problems occurred in such classes. Furthermore, the two smaller code classes are very small and thus might contain only minor (and uncritical) functionality, which led to no problems.

Table 33 Evaluation results of assumption S1-A2 with respect to mean method length.

Assumption 2: Parts of the code where a large number of inspection defects are found and which are of small size indicate more defects to be found with testing.					
No.	**Selection rule:** Focus testing on those code classes in which the inspection determined..	**Prioritized classes**	**Effort reduction**	**Efficiency improve-ment**	**Q.**
A2.09	defect content (high) & mean method length (low)	3	34.38%	52.3 %	+
A2.10	crash severity (defect content (high)) & mean method length (low)	-	/	/	-
A2.11	major severity (defect content (high)) & mean method length (low)	3	34.38%	52.3 %	+
A2.12	minor severity (defect content (high)) & mean method length (low)	3	34.38%	52.3 %	+
A2.13	defect density (high) & mean method length (low)	1, 3, 4	6.25%	6.7 %	+
A2.14	crash severity (defect density (high)) & mean method length (low)	1, 4	/	/	-
A2.15	major severity (defect density (high)) & mean method length (low)	1, 3	9,38%	10.4 %	+
A2.16	minor severity (defect density (high)) & mean method length (low)	1, 3, 4	6.25%	6.7 %	+

In the given context, the alternative assumption S1-A2* led to suitable results when class length was used as a size metric. Six of eight derived selection rules included the defect-prone class three, and thus, showed an efficiency improvement of up to 52%. When using the selection rule ‘defect density (high) & class length (high)‘ (A2.05*), only the defect-

prone code class three was prioritized (i.e., 'defect density (high)' is true for classes one, three, and four, 'class length (high)' is true for code classes two and three, resulting in a prioritization of code class three). This led to an effort reduction of 34%, respectively to an efficiency improvement of 52%. The same was true when the focus was on defect density for major and minor severity (A2.07* and A2.08*). By combining high defect content, high major and minor severity defects with high class length (A2.01*, A2.03*, A2.04*), an effort reduction of about nine percent was achieved (see Table 22, Table 24 and Table 32), resulting in an efficiency improvement of about 10%. Again, using crash severity in combination with class length led to an inappropriate prioritization. One reason might be the very low number of such defects found.

With respect to combinations with the size metric 'mean method length (low)' (a mean method length < 10 LoC was chosen), six of eight selection rules led to appropriate prioritizations of code classes (see Table 33). Furthermore, selection rules combining different defect content metrics and mean method length prioritized only the defect-prone code class three (A2.09, A2.11, A2.12), resulting in an effort reduction of about 34%. Using defect density instead, an effort reduction of between six and nine percent could be achieved. Finally, crash severity defect content and severity again led to inappropriate prioritizations (A2.10, A2.14), and to corresponding efficiency improvements.

In summary, combining two different size metrics with defect metrics led to inconsistent results. While selection rules combining class length and different defect metrics led to insufficient prioritizations when derived from S1-A2, an alternative assumption S1-A2* resulted in very promising results. With respect to the second size metric mean method length, the prioritization of code classes was very appropriate when using selection rules derived from S1-A2. Thus, different efficiency improvements could be observed.

Consequently, assumption S1-A2 can neither be confirmed nor rejected for all selection rules in our context. The quality of the prioritization depends on the size metric chosen in the corresponding selection rules. Furthermore, the results indicate the importance of analyzing assumptions and selection rules carefully in each new context in order to evaluate which ones are best suited for the prioritization of code classes, and which ones are most efficient.

S1-A3: The results for the different selection rules of assumption 3 were appropriate (see Table 34). An effort reduction of 9% was achieved when combining 'defect content (high)' with 'McCabe (high)' (A3.01), i.e., an efficiency improvement of about 10% was achieved. The same efficiency improvement, i.e., an effort reduction of 9%, was also achieved for large numbers of major and minor defects combined with a high McCabe value. A combination of high defect density and a high

McCabe value led to an effort reduction of 34% (A3.05), which is also true for a combination of high major and minor severity defect density with a high McCabe value (A3.07 and A3.08). These selection rules led to an improvement of efficiency of more than 50%. Finally, considering crash severity (A3.02 and A3.06), inappropriate prioritizations were made. However, the six selection rules prioritizing at least the defect-prone code class three confirmed S1-A3.

Table 34 Evaluation results of assumption S1-A3.

Assumption 3: Parts of the code where a large number of inspection defects are found and which are of high complexity indicate more defects to be found with testing.					
No.	**Selection rule:** Focus testing on those code classes in which the inspection determined..	**Prioritized classes**	**Effort reduction**	**Efficiency improve-ment**	**Q.**
A3.01	defect content (high) & McCabe (high)	2, 3	9.38%	10.4 %	+
A3.02	crash severity (defect content (high)) & McCabe (high)	-	/	/	-
A3.03	major severity (defect content (high)) & McCabe (high)	2, 3	9.38%	10.4 %	+
A3.04	minor severity (defect content (high)) & McCabe (high)	2, 3	9.38%	10.4 %	+
A3.05	defect density (high) & McCabe (high)	3	34.38%	52.3 %	+
A3.06	crash severity (defect density (high)) & McCabe (high)	-	/	/	-
A3.07	major severity (defect density (high)) & McCabe (high)	3	34.38%	52.3 %	+
A3.08	minor severity (defect density (high)) & McCabe (high)	3	34.38%	52.3 %	+

To recap the results with respect to H1.5, many useful selection rules were identified for our context and all three assumptions are valuable, i.e., showed an efficiency improvement, though more evaluation across a number of QA runs is necessary to identify the most beneficial selection rules and obtain more evidence in the given context in order to enable application of the integrated inspection and testing approach. For the integrated approach to be applied, evidence is needed regarding the assumptions and derived selection rules that lead to appropriate selections of code classes to be tested. Our analyses can give initial answers, with the assumptions and selections rules being able to serve as a starting point for applying and analyzing them and their efficiency in a different context.

Selection rules concentrating on inspection defects alone led to efficiency improvements of between 7% and 10%, which is equal to the best product metrics applied in our context (see Table 35 and Table 36.) Combining the inspection results with such product metrics, an improvement of up to 50% is possible. However, this is only valid for some combinations, as shown above.

Table 35 Control assumption C1 using product metrics

Assumption C1: Parts of the code which are of high size, respectively high complexity indicate more defects to be found with testing.					
No.	**Selection rule:** Focus testing on those code classes which have	**Prioritized classes**	**Effort reduction**	**Efficiency improve-ment**	**Q.**
C1.01	class length (high)	2, 3	9.38%	10.4 %	+
C1.02	mean method length (high)	2	/	/	-
C1.03	McCabe (high)	2, 3	9.38%	10.4 %	+

Table 36 Control assumption C2 using product metrics

Assumption C2: Parts of the code which are of low size, respectively low complexity indicate more defects to be found with testing.					
No.	**Selection rule:** Focus testing on those code classes which have	**Prioritized classes**	**Effort reduction**	**Efficiency improve-ment**	**Q.**
C2.01	class length (low)	1, 4	/	/	-
C2.02	mean method length (low)	1, 3, 4	6.25%	6.7 %	+
C2.03	McCabe (low)	1, 4	/	/	-

H1.6 Efficiency (defect types): Three selection rules were applied that prioritized different sets of defect types in order to check their efficiency. The first one prioritized *Function* / *Class* / *Object* and *Algorithm* / *Method* (24, respectively 13, classified defects). With testing, four more defects of the defect type *Algorithm* / *Method* were found, but no defects of the type *Function* / *Class* / *Object*. Moreover, two additional defects of different defect types were not found when applying selection rule one (SR1). The second selection rule also prioritized the defect type *Relationship*, of which one additional defect was found by testing. Thus, the second selection rule led to better prioritization. Finally, the third selection rule of stage 2 additionally prioritized *Checking* defects due to the high severity of defects of this defect type based on the inspection defect profile and due to the history, where this defect type had already appeared. Consequently, only SR3 identified all defects of prioritized defect types. Table 37 shows an overview of the prioritizations.

The integrated two-stage approach was able to prioritize those defect types that actually represent the sets with the highest number of defects during testing. The selected defect types are highly dependent on the concrete selection rules and only the most comprehensive selection rule prioritized all relevant defect types. However, all selection rules prioritized the additional defect type *Function* / *Class* / *Object* for testing, but no test defect was assigned to this defect type. Therefore, all selections resulted in slightly lower overall efficiency. Nevertheless, all three selection rules did not focus on all seven ODC defect types and, consequently, it is assumed that effort reduction (though not measured explicitly), and thus efficiency improvement, is achievable.

Table 37 Evaluation results with respect to defect types.

ODC defect type	Inspection defects	Testing defects	Prioritization		
			SR1	SR2	SR3
Function / Class / Object	**24**	0	**x**	**x**	**x**
Algorithm / Method	**13**	**4**	**x**	**x**	**x**
Relationship	**7**	**1**		**x**	**x**
Checking	**5**	**1**			**x**
Interface / O-O Messages	5	0			
Assignment / Initialization	0	0			
Timing / Serialization	0	0			

Summary of the study: With respect to the evaluated hypotheses, it is not possible to provide statistically significant results mainly due to a single tester only. This drawback came from project and effort restrictions. However, based on the test results of the single tester, it could be shown that different test effort reductions of between 6% and 34% could be achieved with the integrated approach depending on which assumptions and concrete selection rules were used to prioritize code classes in our context. This led to an efficiency improvement of between 7% and 52%. A comparison of 32 initially defined selection rules for selecting code classes was performed. 18 selection rules led to an appropriate selection (i.e., no undetected test defect), while 14 led to inappropriate selections. Changing assumption A2 with respect to the size metric class length (i.e., instead of low-sized ones, which was initially taken from identified related work, high-sized ones are used) and combining it with inspection results changed the ratio to 24 suitable ones and eight bad ones. A fine-grained consideration of minor or major defects was usually not more efficient than the more coarse-grained defect content or defect density metric. Finally, selection rules using crash defects should be omitted in our context due to only two defects being classified as such. With respect to prioritizing defect types, selection rules could be found that selected all relevant defect types. It was not possible to derive concrete effort reductions or efficiency values; however, effort savings and efficiency improvements are expected because not all defect types were prioritized.

Finally, considering the overall effort, effectiveness, and efficiency of the inspection and testing activities, an improvement in the efficiency of the In^{2}Test approach compared to a non-integrated approach could be found (between 4.1% and 28.2% depending on the selection rules applied; SR 1*-SR 3* summarize those selection rules with the same values). Again, pure inspection leads to the highest efficiency, by omitting a certain number of defects. Furthermore, inspections are again superior with respect to testing only after the inspection in the given context, which is consistent with observations from other environments (Laitenberger, 1998). Table 38 shows a summary of the quality assurance results with respect to different quality assurance approaches. We

considered the effort for conducting the defect detection during the inspection, respectively the effort for conducting the test definition and execution.

Table 38 Comparison of different quality assurance processes.

No.	Approach	# defects found	Effort needed (minutes)	Efficiency
1	Inspection	100	835	0.1198
2	Testing (after inspetion)	6	660	0.0091
3	Non-integrated inspection and testing	106	1495	0.0710
4.1	In^2Test (SR 1*)	106	1435	0.0739
4.2	In^2Test (SR 2*)	106	1405	0.0754
4.3	In^2Test (SR 3*)	106	1165	0.0910

5.4.5 Limitations of the Study

As in any empirical study, there are threats to the validity of the study results (Wohlin et al., 2000). Below, a discussion of what we consider to be the most relevant threats in our case study is presented.

Conclusion validity: Due to the low number of test effort data of the testers, it was not possible to perform statistical tests (i.e., low statistical power). However, the initial results show that the approach is able to prioritize those code classes that are defect-prone and that the two-stage approach is able to prioritize those defect types that appeared most often during testing. Furthermore, test defects found by only one single testing activity were considered, i.e., test defects that might be found in later testing activities or after delivery were not considered in the analysis of the prioritization.

Construct validity: To demonstrate the integrated approach, different assumptions were derived in our context. Nevertheless, different assumptions might have led to better or worse results. Moreover, in order to be able to apply an assumption, it has to be operationalized into concrete selection rules. This might have been done in a different way and, consequently, alternative code classes and defect types might have been prioritized. A decision on how to treat, e.g., "low" and "high" was made to allow application of the selection rules. As mentioned above, concrete values have to be chosen depending on the context because fixed values are not necessarily valid in each environment. However, a different determination might have led to different test prioritizations. A representative number of different selection rules were derived in order to compare them. Some of them were correlated. Still, additional ones may further support prioritization. In addition, no standard checklists were used, which might have affected the inspection performance.

However, some checklist questions were taken from the literature, which can be considered as standard questions. Finally, the selection of ODC may have influenced the prioritization of defect types.

Internal validity: The selection of the subjects was done systematically, but another selection might have led to different defects being found for inspection and testing. Regarding testing, the effect was slightly reduced by using equivalence partitioning; regarding inspections, the effect was slightly reduced by using checklists that focused the inspectors on certain aspects in the code. Finally, defects could be classified differently.

External validity: The prototype tool, which is rather small, can be considered as an initial example to which the integrated approach was applied. Few test defects were found (which is also a consequence of the size of the prototype tool, respectively the low number of tested code classes). Thus, only those few test defects could be classified. A larger software product, as typically developed by software companies, is expected to result in more test defects being found and classified during testing activities. Thus, the conclusions drawn have to be treated with caution. Moreover, only one classification was applied, although an industry-related one. Furthermore, assumptions and derived selection rules have to be evaluated anew in each new context, meaning that the conclusions drawn with respect to the used selection rules cannot be generalized directly. Finally, the results can only be transferred to an environment where a comparable number of defects are found in inspections.

5.4.6 Trend Analysis of Assumptions and Selection Rules

H1.7 Validity of assumptions (system parts): When applying the integrated inspection and testing approach in a new context, it has to be decided which assumptions and selection rules should be used for prioritizing parts of the system and thus, how the testing activities should be focused. The presented assumptions and selection rules can serve as a starting point for prioritizing those code classes expected to be most defect-prone. However, the selected assumptions and selection rules have to be evaluated in order to demonstrate their suitability. The steps for performing this analysis are presented below, as well as their application in the concrete context of the study for those assumptions that were used for the stage-1 prioritization.

A prerequisite for conducting this analysis is an available set of rules. Due to the fact that only one assumption and no concrete selection rules were applied in the pilot study, additional rules had to be evaluated for the pilot study. Consequently, each selection rule applied in the case study has been analyzed subsequently for the pilot study in order to be

able to analyze which assumptions and selection rules were valid in both QA runs.

The objective of the following analysis is to identify those selection rules that led to consistent predictions of defect-prone code classes with respect to both QA runs. Those selection rules appear to be stable in the given context and are promising candidates for future prioritization steps regarding code parts. All necessary information was already gathered during the QA runs, i.e., defect information per code class and product metrics. In addition, each defined assumption and all derived selection rules had to be analyzed and compared based on the gathered data.

In order to be able to analyze the quality of each assumption and the corresponding selection rules across both QA runs, each possibility for defining an assumption was exploited. With respect to the inspection defects, two different assumptions are possible:

1. Parts of the code where a large number of inspection defects is found indicate more defects to be found with testing.

2. Parts of the code where a low number of inspection defects is found indicate more defects to be found with testing.

Regarding the combination of inspection defects and size, the following four assumptions were defined:

1. Parts of the code where a large number of inspection defects is found and which are of small size indicate more defects to be found with testing.

2. Parts of the code where a large number of inspection defects is found and which are of high size indicate more defects to be found with testing.

3. Parts of the code where a low number of inspection defects is found and which are of small size indicate more defects to be found with testing.

4. Parts of the code where a low number of inspection defects is found and which are of high size indicate more defects to be found with testing.

Finally, regarding the combination of inspection defects and complexity, four assumptions were determined:

1. Parts of the code where a large number of inspection defects is found and which are of high complexity indicate more defects to be found with testing.

2. Parts of the code where a large number of inspection defects is found and which are of low complexity indicate more defects to be found with testing.

3. Parts of the code where a small number of inspection defects is found and which are of high complexity indicate more defects to be found with testing.

4. Parts of the code where a small number of inspection defects is found and which are of low complexity indicate more defects to be found with testing.

Table 39 Number of selection rules that were compared in the trend analysis.

Assumptions		Selection	Metric one	Metric two	#
I	defect content	2 x	2 x	4	= 16
		high / low	defect content / defect density	all defects / high severity defects / med. severity defects / low severity defects	
II	defect content + size	4 x	2 x	4	= 32
		high + high / high + low / low + high / low + low	defect content + class length / defect density + class length	all defects + LoC / high severity defects + LoC / med. severity defects + LoC / low severity defects + LoC	
III	defect content + size	4 x	2 x	4	= 32
		high + high / high + low / low + high / low + low	defect content + method length / defect density + method length	all defects + LoC / high severity defects + LoC / med. severity defects + LoC / low severity defects + LoC	
IV	defect content + complexity	4 x	2 x	4	= 32
		high + high / high + low / low + high / low + low	defect content + McCabe / defect density + McCabe	all defects + McCabe / high severity defects + McCabe / med. severity defecty + McCabe / low severity defects + McCabe	
V	size	2 x	1		= 2
		high / low	class length		
VI	size	2 x	1		= 2
		high / low	mean method length		
VII	complexity	2 x	1		= 2
		high / low	McCabe complexity		
				Sum:	**118**

For each of the ten assumptions, corresponding selection rules were derived based on the determined metrics (e.g., defect content, defect density, mean method length, McCabe complexity, and severity classes). This resulted in 112 selection rules to be compared. Performing the analysis in that way, each assumption and the derived selection rules were analyzed in detail and conclusions regarding which ones led to the best prioritizations of code classes could be drawn. Remember that a large number of category four selection rules were expected due to this kind of analysis. In order to be able to compare the different selection rules with selection rules using only product metrics, additional assumptions were defined, focusing on the two different size metrics (i.e., class length and mean method length) and the single complexity metric (i.e., McCabe complexity) only. Consequently, six additional selection rules were defined (e.g., small and large class length), resulting

in a total of 118 selection rules to be analyzed. Table 39 shows the calculation.

In order to judge the quality of a selection rule, four different quality categories for a selection rule were defined. Table 40 gives an overview of the four categories, as explained in detail in Section 4.4.

Table 40 Quality categories with respect to prioritization of code classes.

Category		Decription
1	excellent	All code classes in which defects are found are prioritized, and code classes in which no defects are found are not prioritized.
2	good	All code classes in which defects are found are prioritized, but code classes in which no defects are found are also prioritized.
3	bad	Only some code classes in which defects are found are prioritized (includes also the prioritization of code classes in which no defects are found).
4	worst	No code class which is defect-prone is prioritized.

Figure 39 shows how the selection rules were classified with respect to the quality categories over the two QA runs performed. The first group, called "acceptable", comprises selection rules that were either classified as one or two. Selection rules that were classified as one in both QA runs showed the best prioritizations, followed by a one in the first run and a two in the second run. Nine selection rules fit into one of those two categories. Some examples are "defect content (high)", "defect content (high) & class length (high)", and "defect content (high) & mean method length (low)". Furthermore, some combinations of size metrics with major and minor severity (defect content) showed excellent or good prioritizations.

Two more selection rules focusing either on high class length or on low mean method length only were classified into category two in both runs. These two selection rules can be treated as control selection rules because they do not consider defect results from the inspection. However, although they presented satisfactory results, they were not the most efficient ones.

The "neutral" group comprises selection rules where at least one selection rule is put into category three and the other selection rules are not categorized worse than category three. Though these twelve selection rules led only to acceptable results in one run, they prioritized at least some code classes in the other run that were defect-prone. Thus, they should be analyzed further in subsequent QA runs. Two examples are "defect density (high)" and "defect density (high) & mean method length (low)". For the second QA run, no selection rule was classified in category three since there was only one defect-prone code class (and thus, no defect-prone subset exists).

Finally, most of the selection rules (around 80 percent) were classified as "non-acceptable". Due to the high number of different concrete combinations of selection rules and the low number of QA runs, this result was expected. Selection rules that were classified as one/four or four/one showed promising results in one QA run, but performed worse in the other. Usually, McCabe complexity was used in these selection rules, which is the same as in the selection rules classified as two/four or four/two. This shows that selection rules that combine McCabe complexity with inspection results lead to inconsistent prioritizations in our context. More specifically, low McCabe complexity (and those selection rules that combine McCabe complexity and inspection defect numbers) led to suitable predictions in the first quality assurance run, while high McCabe complexity led to good predictions in the second run. One reason might be that the quality-assured parts in the first iteration were still not very complex, but did contain defects. However, this was changed in the second iteration (i.e., the software became more complex).

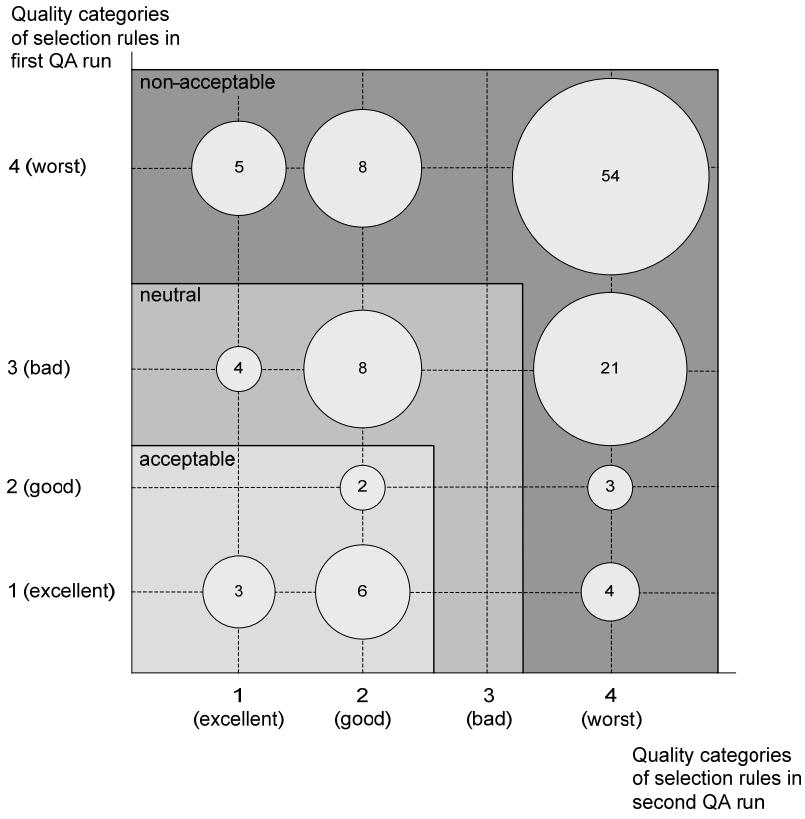


Figure 39 Quality categories of 118 selection rules over two QA runs.

The main conclusions of this analysis can be summarized as follows:

- Nine selection rules showed very promising prioritization results over the two QA runs and should be considered in subsequent QA runs. All of these nine selection rules use inspection defect results.

- Two control selection rules (class length high and mean method length low) also led to good prioritization results (twice classified as two), but not as efficiently as the aforementioned nine selection rules, i.e., the nine most promising selection rules that use inspection results only or combine inspection results with product metrics led to prioritizations that are more efficient than selection rules using only single product metrics.

- Selection rules that use McCabe complexity or combine this complexity metric with inspection results led to inconsistent results over the two QA runs.

- In general, assumptions and selection rules using inspection defect information or combining it with product metrics are a promising approach for prioritizing code classes that are expected to be defect-prone. However, the best ones have to be identified by detailed analyses in a given context.

In this analysis, initial and easy-to-gather defect and product metrics were used to define the selection rules. However, a lot of different metrics exist that may also lead to worthwhile prioritizations. Some of these are mentioned in Chapter 3. Thus, additional analyses are necessary, especially when performing further QA runs and trying to identify those selection rules that continuously lead to acceptable prioritizations. Furthermore, in order to improve confidence in the prioritization results for testing activities, not only one, but a set of the most promising selection rules can be considered. Nevertheless, different selection rules have to be analyzed over each new QA run again and the information gathered has to be taken into consideration, i.e., it has to be evaluated continuously which selection rules are the best ones (which is something that can change).

5.5 Summary

In this chapter, the integrated inspection and testing approach In²Test was evaluated during two case studies. In both case studies, a tool was developed and two quality assurance activities were performed, namely inspections and testing. The defect data was used to evaluate certain hypotheses.

An overview of the results can be found in Table 41. Overall, statistically significant results could not be obtained due to small sample sizes. However, initial positive trends could be demonstrated.

First of all, H1.1 showed positive trends. An effort reduction for executing test cases of more than 20% was achieved in both case studies when the In^{2}Test approach was applied. However, this result depends on the assumption and selection rules applied, i.e., there existed also assumptions and selection rules that did not lead to such an effort reduction. Effort reduction was achieved in our context, and we observed positive trends with respect to this hypothesis, whereas no statistically significant results could be achieved due to the small sample size. Moreover, additional effort for deriving assumptions and selection rules was not considered here. It is worth noting that the best assumptions, respectively selection rules, led to a similar effort reduction of about 30% in both case studies. With respect to H1.2, an effort reduction is also achievable when only focusing on certain defect types. However, no concrete effort numbers could be obtained. Rather, the parenthesized checkmark is based on the fact that focusing on certain defect types was possible instead of selecting all defect types (i.e., defects were not distributed equally with respect to all considered defect types, and those defect types were primarily selected, into which most of the defects were classified).

Table 41 Summary of the results of the performed case studies.

Hypothesis	Case study I[1]	Case study II[1]
H1.1: Effort (system parts)	✓	✓
H1.2: Effort (defect types)	(✓[2])	(✓[2])
H1.3: Effectiveness (system parts)	✓	✓
H1.4: Effectiveness (defect types)	✓	✓
H1.5: Efficiency (system parts)	✓	✓
H1.6: Efficiency (defect types)	(✓[2])	(✓[2])
H1.7: Validity of assumptions (system parts)		✓[3]
H1.8: Validity of assumptions (defect types)		
H2.1: Applicability in industrial context		

[1] Results are based on quantitative data, but the sample size was too small to conduct statistical analyses
[2] Results are based on quantitative data for focusing and no real effort data
[3] Results are based on two quality assurance runs

H1.3 and H1.4 indicate a potential for improved effectiveness, i.e., a similar number of defects were found when the In^{2}Test approach was applied. Again, the result is dependent on concrete assumptions and selection rules. In the first case study, H1.4 was evaluated with respect to different system levels (i.e., defect types from code inspection used for focusing defect types on the system test level). Though the same defect types were found during inspections and testing, it is unclear how to use

defect type results of the code inspection to focus on those defect types on the system test level.

Hypotheses H1.5 and H1.6 used different assumptions for focusing testing on parts of the system and on certain defect types in order to evaluate an improvement in efficiency. These two assumptions summarize the aggregated results of H1.1 together with H1.3, and of H1.2 together with H1.4 with respect to one quality assurance run. With respect to assumptions that were used to focus testing on certain parts of a system, H1.5 shows positive trends and an improvement in efficiency of between 6.7% and 52.4% was achieved. A number of different assumptions and their refined selection rules led to suitable results. With respect to H1.6, only an indirect statement could be made due to a lack of exact effort data. Hence, this hypothesis got a checkmark in parentheses.

Finally, in the second case study, two quality assurance runs could be performed, which allowed analyzing assumptions and selection rules across both quality assurance runs and initially investigating the validity of underlying assumptions with respect to focusing system parts. Again, a set of assumptions and selection rules could be found that were valid during both quality assurance runs, which strengthens the validity of these assumptions. However, additional quality assurance runs are necessary to improve the validity of the applied assumptions and selection rules.

The remaining hypotheses could not be tested during the case studies and remain to be validated during future work. However, though the approach was not applied in an industrial setting, first positive trends regarding its applicability could be gathered during the two evaluations.

Overall, hypothesis H1 can neither be confirmed nor rejected based on the two case studies because no statistical analyses were possible due to the small sample size. However, based on the analyzed sub-hypotheses, first positive results could be obtained that encourage using inspection defect data to focus testing activities in order to improve test efficiency and overall quality assurance efficiency (i.e., reduced effort at the same quality level).

6 Conclusion and Future Work

6.1 Summary and Conclusion

This thesis presented the **in**tegrated **in**spection and **test**ing approach In^{2}Test, which integrates inspection and testing in order to focus testing. It can be considered as a light-weight approach since it does not require any particular inspection or testing technique. Inspection defect data is used to predict defect-prone parts or defect types, and testing is focused on such parts or on the selected defect types. Though the use of inspection results might not be the only appropriate predictor of defect-prone parts or defect types, it can, however, give valuable support, e.g., for allocating test effort, defining the order of tests, or improving the efficiency and effectiveness of quality assurance activities and thus, for improving the overall quality of the system under test. As shown by the state-of-the-art and state-of-the-practice analyses, inspections and testing are well-established quality assurance activities. However, they are usually applied in isolation, meaning that synergy effects are often not exploited by systematic integration. In particular, inspection results are often not used for focusing testing activities. In detail, the In^{2}Test approach contributes the following components:

- The In^{2}Test approach makes explicit use of inspection defect data and is able to combine them with established metrics and historical data in order to identify potentially problematic areas and defect types, and thereby improves prioritization. A process was presented that shows the necessary steps, from the inspection to the focused test activity.

- In order to be able to conduct prioritization, knowledge about the relationships between inspections and testing is necessary. If such knowledge is not available, assumptions have to be stated. A formal model for describing assumptions was presented, and guidelines on how to derive, evaluate, and apply them were stated. Furthermore, refined selection rules that make assumptions operational were shown. Finally, a set of initial assumptions describing the relationships between inspection and testing defects were given.

- A prototype implementation of the prediction component was performed. With the DETECT tool, which can be used to conduct inspections, the inspection results can be visualized, and selection rules can be defined. Based on these rules, the

tool presents suggestions as to which parts or which defect types should be focused on during subsequent testing activities.

The In^{2}Test approach was evaluated in two case studies. The main goal of this thesis was to evaluate whether any effort improvement at a comparable level of quality could be obtained, i.e., whether any improved efficiency could be achieved. The following results were achieved in the given contexts:

- An effort reduction of between 6% and 34% was achieved when focusing testing on specific parts of the system, which depended on the selection rules used in the given contexts.

- Although no concrete effort reduction could be measured when focusing on specific defect types, some improvement is expected.

- Overall, comparable quality could be achieved in the case studies when tests were focused on specific parts and specific defect types. However, this also depended on specific assumptions and concrete selection rules.

- An efficiency improvement of between 7% and 52% was achieved when focusing testing on specific parts of the system, which depended on the selection rules used in the given contexts. With respect to overall quality assurance efficiency, this means an improvement of up to 28% of the In^{2}Test approach compared to a non-integrated approach when considering defect detection efforts.

- Although no concrete efficiency improvement could be measured when focusing on specific defect types, some improvement is expected.

- One case study demonstrated that a specific set of assumptions and selection rules that focused testing on specific parts were valid during two quality assurance runs, which improved confidence in these assumptions and selection rules in the given environment.

- New insights about the relationships between inspections and testing could be gained. For instance, in one case study, more defects were found during testing when the inspection had found a significant number of defects. Moreover, such selection rules showed better performance than established metrics such as size or complexity. In another case study, in which different development stages were addressed, focusing the test on parts

that had not been inspected resulted in additional defects. These results also indicate that assumptions and selection rules have to be validated again in each new context in order to find the most appropriate ones in a given environment.

6.2 Open Questions and Future Work

Further development and evaluation of the In^{2}Test approach will take place, especially in the context of the *Stiftung Rheinland-Pfalz für Innovation* project QKIT. In general, the following aspects may be addressed by future work:

- *Approach improvement:* The approach should be studied in terms of the extent to which it is applicable to different development phases such as requirements inspections used for prioritizing system test activities or design inspections used for prioritizing integration test activities. Moreover, results from different inspection phases may be cumulated in order to focus different testing activities.

- *Approach improvement:* Inspections are often only performed on limited parts of the product. However, the limited inspection results should also be used to prioritize those parts of the product to be tested that were not inspected (e.g., based on characteristics of the inspected parts that are similar to suggested parts of the product), i.e., a scaling mechanism for the integrated approach should be considered.

- *Approach improvement:* A procedure for deriving test cases based on prioritized defect types has to be defined.

- *Approach improvement:* Focusing on specific system parts and omitting other parts completely is a rather coarse-grained style of prioritization. Therefore, calibration of testing techniques may result in more appropriate focusing, e.g., defining how many equivalence classes should be derived for prioritized and for non-prioritized parts may lead to better prioritization.

- *Approach improvement:* Instead of omitting entire parts of the system, such as code classes or code defect types, effort could be allocated on a percentage basis, i.e., most of the test effort should be allocated to those parts of the system expected to be most defect-prone or to defect types expected to appear most often. This can be supported by creating a sorted list of the parts to be tested instead of putting all parts to be tested into a prioritized and non-prioritized set.

- *Approach improvement:* The In²Test approach currently does not offer any overall confidence measure to support the decision on when to stop testing. Confidence in the results of this thesis is supported to the extent that valid assumptions are used for the given contexts. However, an overall confidence value depends on many criteria, such as domain, usage scenarios, or criticality of defects. The In²Test approach may support finding an answer regarding this question by considering inspection and test defect data, but more criteria have to be considered in future work and a more comprehensive definition of confidence has to be given.

- *Approach improvement:* Besides inspection results and product metrics used as input for focusing testing activities, feedback from inspectors or further experts and process metrics could also provide valuable input for the prediction that could be incorporated into the existing In²Test approach. D'Ambros et al. (D'Ambros et al., 2010) state that usually a single focusing technique does not work consistently in all environments and thus, a mix of different prediction techniques might provide more valuable predictions of defect-proneness.

- *Approach improvement:* One important challenge is to explore ways on how to use the approach in highly iterative and adaptive development processes.

- *Approach embedding:* The In²Test approach may be used to balance different quality assurance activities. For this, it has to be embedded into a holistic approach for determining quality assurance activities. One goal might be to develop empirically-based guidance on how to embed different kinds of quality assurance activities and their interrelations into a development process model. Depending on the development goals and characteristics as well as on so-called defect flow models, which describe the defect slippage of the lifecycle, guidance for integrating different kinds of quality assurance activities and their interfaces could be developed. The usability of such an approach has to be ensured, meaning an answer is needed to the question of how to present which information to support a quality engineer in making the "right" decision to improve the overall quality.

- *Quality assurance strategy:* In this thesis, the In²Test approach was applied to improve efficiency at a comparable level of quality. However, exploiting the given effort and improving the number of defects found, or improving both efficiency and effectiveness, are further goals of the integrated approach that should be addressed.

- *Assumptions:* Additional assumptions should be defined and evaluated in order to better understand the relationships between inspections and testing and to obtain more evidence. This includes especially defect distributions obtained by these quality assurance activities (e.g., expressed as defect content, defect density, or number of defects per defect type) Based on such knowledge, new theories that explain the observations made could be derived. If such solid empirical knowledge and derived theories explaining the relationships were available, then it would be possible to guide and focus quality assurance activities, such as testing, in a more appropriate manner.

- *Assumptions:* The approach should be applied across several releases in order to identify the most valuable assumptions and selection rules in a given environment and to gain statistically significant results. This also includes comparing different assumptions and selection rules.

- *Selection rules:* Instead of using a single selection rule, it might be more worthwhile in real settings to combine the best possible selection rules for the most appropriate focusing and to reduce the number of overlooked defects. Guidelines describing how to combine selection rules are needed.

- *Evidence:* Assumptions that use inspection defect data for focusing testing activities should be compared to established concepts, such as using product metrics (e.g., size, complexity) for the prediction of defect-proneness and defect types.

- *Evaluations:* Additional evaluations of the integrated approach in different contexts (e.g., industrial environments, academia, or open-source projects) where inspection and test data are available may substantiate the usefulness of the integrated approach.

- *Evaluations*: An experiment design that compares groups using the In^2Test approach with groups not using the In^2Test approach may result in additional conclusions regarding the suitability and efficiency improvements of the integrated approach. Two different study designs can be found in Appendix B.

- *Evaluations:* A comparison of the integrated approach to existing approaches for predicting defect-proneness and focusing testing activities has to be conducted, for example a comparison between the In^2Test approach and approaches using different metrics with respect to efficiency improvements, for instance.

- *Alternative early data*: Instead of using inspection defect data for focusing testing activities, data from different static analyses could be used instead or in combination.

- *Sophisticated tool support:* The initial tool prototype should be extended in order to support the In^{2}Test approach more comprehensively. For example, additional product metrics could be used and combined with inspection metrics, and historical data could be considered when defining rules for the focusing activity.

References

A.F. Ackerman, L.S. Buchwald, F.H. Lewsky. Software Inspections: An Effective Verification Process. IEEE Software, vol. 6, no. 3, pp. 31-36, 1989.

A. Aggarwal, P. Jalote. Integrating Static and Dynamic Analysis for Detecting Vulnerabilities, In: Proceedings of the 30th Annual International Conference on Computer Software and Applications, pp. 343-350, 2006.

P. Andersson. The Use and Limitations of Static Analysis Tools to Improve Software Quality. CrossTalk: The Journal of Defense Software Engineering, vol. 21, no. 6, pp. 18-21, 2008.

C. Andersson, P. Runeson. A Replicated Quantitative Analysis of Fault Distributions in Complex Software Systems. IEEE Transactions on Software Engineering, vol. 33, no. 5, pp. 273-286, 2007.

C. Andersson, T. Thelin, P. Runeson, N. Dzamashvili. An Experimental Evaluation of Inspection and Testing for Detection of Design Faults. In: Proceedings of the 2003 International Symposium on Empirical Software Engineering, pp. 174-184, 2003.

C. Artho, A. Biere. Combined Static and Dynamic Analysis. Electronic Notes in Theoretical Computer Science, vol. 131, pp. 3-14, 2005.

A. Aurum, H. Petersson, C. Wohlin. State-of-the-Art: Software Inspections after 25 Years. Software Testing, Verification and Reliability, vol. 12, no. 3, pp. 133-154, 2002.

A. Avancini, M. Ceccato. Towards Security Testing with Taint Analysis and Genetic Algorithms. In: Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems, pp. 65-71, 2010.

D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In: Proceedings of the 2008 IEEE Symposium on Security and Privacy, pp. 387-401, 2008.

J. Barnard, A. Price. Managing Code Inspection Information. IEEE Software, vol. 11, no.2, pp. 59-69, 1994.

V.R. Basili, L.C. Briand, W.L. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. IEEE Transactions on Software Engineering, vol. 22, no. 10, pp. 751-761, 1996.

V.R. Basili, G. Caldiera, H.D. Rombach. The Experience Factory. Encyclopedia of Software Engineering, vol. 1, John Wiley & Sons, pp. 469-476, 1994.

V.R. Basili, G. Caldiera, H.D. Rombach. Goal Question Metric Paradigm. Encyclopedia of Software Engineering, vol. 1, John Wiley & Sons, pp. 528-532, 1994b.

V.R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Soerumgaard, M. Zelkowitz. The Empirical Investigation of Perspective-based Reading. Empirical Software Engineering, vol. 1, no. 2, pp. 133-164, 1996.

V.R. Basili, B.T. Perricone. Software Errors and Complexity: An Empirical Investigation. Communication of the ACM, vol. 27, no. 1, pp. 42-52, 1984.

V.R. Basili, R.W. Selby. Comparing the Effectiveness of Software Testing Strategies. IEEE Transactions on Software Engineering, vol. 13, no. 12, pp. 1278-1296, 1987.

W.J. Baumwol, A.S. Blinder. Macroeconomics: Principles and Policy. Orlando, Florida, Harcourt, 2001.

K. Beck. Extreme Programming. Addison-Wesley, München, 2000.

B. Beizer, Software Testing Techniques. 2nd Edition, International Thomson Computer Press, 1990.

T. Berling, P. Runeson. Evaluation of a Perspective based Review Method Applied in an Industrial Setting. IEE Proceedings, vol. 150, no. 3, pp. 177-184, 2003.

T. Berling, T. Thelin. An Industrial Case Study of the Verification and Validation Activities. In: Proceedings of the 9th International Symposium on Software Metrics, pp. 226-238, 2003.

T. Berling, T. Thelin. A Case Study of Reading Techniques in a Software Company. In: Proceedings of the 2004 International Symposium on Empirical Software Engineering, pp. 229-238, 2004.

A. Bertolino. Software Testing Research: Achievements, Challenges, Dreams. In: Proceedings of Future of Software Engineering, pp. 85-103, 2007.

A. Bertolino, E. Marchetti. A Brief Essay on Software Testing. Technical report, pp. 1-14, 2004.

S. Biffl. Using Inspection Data for Defect Estimation. IEEE Software, vol. 17, no. 6, pp. 36-43, 2000.

D.B. Bisant, J.R. Lyle. A two-Person Inspection Method to Improve Programming Productivity. IEEE Transactions on Software Engineering, pp. 1294-1304, 1989.

B. Boehm. Guidelines for Verifying and Validating Software Requirements and Design Specification. In: Proceedings of the European Conference on Applied Information Technology of the International Federation for Information Processing, pp. 711-719, 1979.

B. Boehm, V.R. Basili. Software Defect Reduction Top 10 List. IEEE Computer, vol. 34, no. 1, pp. 135-137, 2001.

L. Briand, K.E. Emam, B. Freimut. A Comparison and Integration of Capture-Recapture Models and the Detection Profile Method. In: Proceedings of the 9th Internatioal Symposium on Software Reliability Engineering, pp. 32-43, 1998.

L.C. Briand, K. El Emam, B. Freimut, O. Laitenberger. Quantitative Evaluation of Capture-Recapture Models to Control Software Inspections. In: Proceedings of the 8th International Symposium on Software Reliability Engineering, pp. 234-244, 1997.

L. Briand, B. Freimut, O. Laitenberger, G. Ruhe, B. Klein. Quality Assurance Technologies for the EURO Conversion – Industrial Experience at Allianz Life Assurance. In: Proceedings of the 2nd International Software Quality Week Europe, pp. 1-23, 1998b.

N. Bridge, C. Miller, Orthogonal Defect Classification Using Defect Data to Improve Software Development. In: International Conference on Software Quality, pp. 197-213, 1997.

British Standard 7925-2, Software Testing, Part 2: Software Component Testing, 1998.

B. Brykczynski. A Survey of Software Inspection Checklists. ACM SIGSOFT Software Engineering Notes, vol. 24, no.1, pp. 82-89, 1999.

I. Burnstein. Practical Software Testing, Springer, 2002.

J. Carver, J. van Voorhis, V. Basili. Understanding the Impact of Assumptions on Experimental Validity. In: Proceedings of the 2004

International Symposium on Empirical Software Engineering, pp. 251-260, 2004.

P. Centonze, R. Flynn, M. Pistoia. Combining Static and Dynamic Analysis for Automatic Identification of Precise Access-control Policies. In: Proceedings of the 23rd Annual Computer Security Applications Conference, pp. 292-303, 2007.

J.K. Chaar, M.J. Halling, I.S. Bhandari, R. Chillarege. In-process Evaluation for Software Inspection and Test. IEEE Transactions on Software Engineering, vol. 19, no. 11, pp. 1055-1070, 1993.

T.F. Chang, A. Danylyzsn, S. Norimatsu, J. Rivera, D. Shepard, A. Lattanze, J. Tomayko. "Continuous Verification" in Mission Critical Software Development. In: Proceedings of the 13th Hawaii International Conference on System Sciences, pp. 273-284, 1997.

R.N. Charette. Why Software Fails. IEEE Spectrum, vol. 32, no. 9, pp. 42-49, 2005.

O. Chebaro, N. Kosmatov, A. Giorgetti, J. Julliand. Combining Static Analysis and Test Generation for C Program Debugging. Tests and Proofs, vol. 6143, pp. 94-100, 2010.

J. Chen, H. Zhou, S.D. Bruda. Combining Model Checking and Testing for Software Analysis. In: Proceedings of the 2008 International Conference on Computer Science and Software Engineering, pp. 206-209, 2008a.

Y. Chen, S. Liu, W.E Wong. A Method Combining Review and Testing for Verifying Software Systems. In: Proceedings of the 2008 International Conference on BioMedical Engineering and Informatics, pp. 827-831, 2008b.

J. Chen, S. MacDonald. Towards a Better Collaboration of Static and Dynamic Analyses for Testing Concurrent Programs. In: Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis, and debugging, pp. 1-9, 2008.

T.Y. Chen, P.L. Poon, S.F. Tang, T.H. Tse, Y.T. Yu. Applying Testing to Requirements Inspection for Software Quality Assurance. Information Systems Control Journal, vol. 6, pp. 50-56, 2006.

Q. Chen, L. Wang, Z. Yang. HEAT: An Integrated Static and Dynamic Approach for Thread Escape Analysis. In: Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference, pp. 142-147, 2009.

Q. Chen, L. Wang, Z. Yang, S.D. Stoller. HAVE: Detecting Atomicity Violations via Integrated Dynamic and Static Analysis. In: Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, pp. 425-439, 2009b.

M. Ciolkowski, O. Laitenberger, S. Biffl. Software Reviews: The State of the Practice. IEEE Software, vol. 20, no. 6, pp. 46-51, 2003.

J. Cohen. Best Kept Secrets of Peer Code Review: Code Reviews at Cisco Systems. pp. 63-87, 2006.

R. Conradi, A.S. Marjara, B. Skatevik, An Empirical Study of Inspection and Testing Data at Ericsson, Norway, In: Proceedings of the 24th NASA Software Engineering Workshop, 1999.

C. Csallner, Y. Smaragdakis. Check 'n' Crash: Combining Static Checking and Testing. In: Proceedings of the 27th International Conference on Software engineering, pp. 422-431, 2005.

C. Csallner, Y. Smaragdakis, T. Xie. DSD-Crasher: A Hybrid Analysis Tool for Bug Finding. ACM Transactions on Software Engineering and Methodology, vol. 17, no. 2, pp. 1-37, 2008.

M. D'Ambros, M. Lanza, R. Robbes. An Extensive Comparison of Bug Prediction Approaches. In: Proceedings of the 7th IEEE Working Conference on Mining Software Repositories, pp. 31-41, 2010.

G. Denaro, M. Pezze. An Empirical Evaluation of Fault-Proneness Models. In: Proceedings of the 24th International Conference on Software Engineering, pp. 241-251, 2002.

C. Denger. SafeSpection - A Framework for Systematization and Customization of Software Hazard Identification by Applying Inspection Concepts. PhD Theses in Experimental Software Engineering, vol. 27, Fraunhofer IRB Verlag, 2009.

C. Denger, M. Ciolkowski, F. Lanubile. Does active guidance improve software inspections? A preliminary empirical study. In: Proceedings of the IASTED International Conference Software Engineering, pp. 408-413, 2004.

C. Denger, F. Elberzhager. Unifying Inspection Processes to Create a Framework to Support Static Quality Assurance Planning. In: Proceedings of the 33rd Euromicro Conference on Software Engineering and Advanced Applications, pp. 271-280, 2007.

E.W. Dijkstra. The Humble Programmer. Communications of the ACM, vol. 15, no. 10, pp. 859-866, 1972.

E.P. Doolan. Experience with Fagan's Inspection Method. Software – Practice and Experience, vol. 22, no. 2, pp. 173-182, 1992.

E. Duke. V&V of Flight and Mission-Critical Software. IEEE Software vol. 6, no. 3, pp. 39-45, 1989.

A. Dunsmore, M. Roper, M. Wood. Systematic Object-Oriented Inspection - An Empirical Study. In: Proceedings of the 23rd International Conference on Software Engineering, pp. 123-144, 2001.

A Dunsmore, M. Roper, M. Wood. Further Investigations into the Development and Evaluation of Reading Techniques for Object-oriented Code Inspections. In: Proceedings of the 24th International Conference on Software Engineering, pp. 47-57, 2002.

A. Dunsmore, M. Roper, M. Wood. The Development and Evaluation of Three diverse Techniques for Object-oriented Code Inspection. IEEE Transactions on Software Engineering, vol. 29, no. 8, pp. 677-686, 2003.

S.G. Eick, C.R. Loader, M.D. Long, L.G. Votta, S.V. Wiel. Estimating Software Fault Content Before Coding. In: Proceedings of the 14th International Conference on Software Engineering, pp. 59-65, 1992.

K.E. Emam, S. Benlarbi, N. Goel, W. Mela, H. Lounis, S.N. Rai. The Optimal Class Size for Object Oriented Software. IEEE Transactions on Software Engineering, vol. 28, no.5, pp. 494-509, 2002.

F. Elberzhager. Analysis of Empirical Findings on Quality Assurance Techniques. Diploma thesis, University of Kaiserslautern, 2005.

F. Elberzhager, R. Eschbach. Towards Reduction of Test Effort – Predicting Defect-prone Code Classes and Expected Defect Types based on Inspection Results. In: Proceedings of the 36th Euromicro Software Engineering and Advanced Application, Work in Progress Session, 2010.

F. Elberzhager, R. Eschbach, C. Jung, A. Klaus. DEFECT – Tool-supported Inspection Guidance. In: Proceedings of the 36th Euromicro Software Engineering and Advanced Application, Work in Progress Session, 2010a.

F. Elberzhager, R. Eschbach, J. Kloos. Indicator-based Inspections: A Risk-oriented Quality Assurance Approach for Dependable Systems. In: Proceedings of the Software Engineering 2010 edition, GI-Edition Lecture Notes in Informatics, vol. 159, pp. 105-116, 2010b.

F. Elberzhager, R. Eschbach, J. Muench. Using Inspection Results for Prioritizing Test Activities. In: Proceedings of the 21st International Symposium on Software Reliability Engineering, Supplemental Proceedings, pp. 263-272, 2010c.

F. Elberzhager, R. Eschbach, J. Muench. The Relevance of Assumptions and Context Factors for the Integration of Inspections and Testing. In: Proceedings of the 37th Euromicro Software Engineering and Advanced Application, Software Product and Process Improvement, pp. 388-391, 2011a.

F. Elberzhager, R. Eschbach, J. Muench. Using Context-specific Relationships for the Integration of Inspection and Test Processes, Project Report in the Context of the Stiftung Rheinland-Pfalz für Innovation Project Qualitäts-KIT (grant: 925), Fraunhofer IESE report no. 094.11/E, 2011c.

F. Elberzhager, R. Eschbach, J. Muench, A. Rosbach. Reducing Test Effort: A Systematic Mapping Study on Existing Approaches. Information and Software Technology, vol. 54, no. 10, pp. 1092-1106, 2012b.

F. Elberzhager, R. Eschbach, J. Muench, A. Rosbach. Inspection and Test Process Integration based on Explicit Test Prioritization Strategies. In: Proceedings of the 4th Software Quality Days, pp. 181-192, 2012.

F. Elberzhager, A. Klaus, M. Jawurek. Software Inspections using Guided Checklists to Ensure Security Goals. In: Proceedings of the International Conference on Availability, Reliability and Security, pp. 853-858, 2009.

F. Elberzhager, J. Muench. Using Early Quality Assurance Metrics to Focus Testing Activities. In: Proceedings of the DASMA Metrik Kongress, (Metrikon), pp. 29-36, 2011.

F. Elberzhager, J. Muench, D. Rombach, B. Freimut. Optimizing Cost and Quality by Integrating Inspection and Test Processes. In: Proceedings of the International Conference on Software and Systems Process, pp. 3-12, 2011d.

F. Elberzhager, J. Muench, V. Tran. A Systematic Mapping Study on the Combination of Static and Dynamic Quality Assurance Techniques. Information and Software Technology, vol. 54, no. 1, pp. 1-15, 2012a.

K.E. Emam, O. Laitenberger, T. Harbich. The Application of Subjective Estimates of Effectiveness to Controlling Software Inspections. Journal of Systems and Software, vol. 54, no. 2, pp. 119-136, 2000.

A. Endres. An Analysis of Errors and their Causes in System Programs. IEEE Transactions on Software Engineering, vol. 1, no. 2, pp. 140-149, 1975.

A. Endres, D. Rombach. A Handbook of Software and Systems Engineering," Addison-Weseley, Pearson Education Limited, 2003.

G.D. Everett, R. McLeod. Software Testing: Testing Across the Entire Software Development Life Cycle. John Wiley and Sons, New Jersey, 2007.

M.E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. IBM Systems Journal, vol. 15, no. 3, pp. 182-211, 1976.

M.E. Fagan. Advances in Software Inspections. IEEE Transactions on Software Engineering, vol. 12, no. 7, pp. 744-751, 1986.

M.E. Fagan. Inspections - Evolution and History: 1972-2001. Talk given at sd&m conference, slides, 2001, http://pioneer.chula.ac.th/~sperapho/pub/f_7_fagan.pdf, last visited: January 06, 2012.

R.L. Feldmann, J. Münch, S. Vorwieger, Experiences with Systematic Reuse: Applying the EF/QIP Approach, In: Proceedings of the European Reuse Workshop, 1997.

N.E. Fenton, N. Ohlsson. Quantitative Analysis of Faults and Failures in a Complex Software System. IEEE Transactions on Software Engineering, vol. 26, no. 8, pp. 797-814, 2000.

L.A. Franz, J.C. Shih. Estimating the Value of Inspections and Early Testing for Software Projects. Hewlett-Packard Journal, pp. 60-67, 1994.

G.A. Gack. An Economic Analysis of Software Defect Removal Methods. Based on Managing the Black Hole: The Executive's Guide to Software Project Risk, Business Expert Publishing, 2010.

V. Garousi, T. Varma. A Replicated Survey of Software Testing Practices in the Canadian Province of Alberta: What has Changed from 2004 to 2009? Journal of Systems and Software, vol. 83, no. 11, pp. 2251-2262, 2010.

A.M. Geras, M.R. Smith, J. Miller. A Survey on Software Testing Practices in Alberta. Canadian Journal of Electrical and Computer Engineering, vol. 29, no. 3, pp. 183-191, 2004.

T. Gilb, D. Graham. Software Inspections. Addison-Wesley, 1993.

P. Godefroid, P. de Halleux, A.V. Nori, S.K. Rajamani, W. Schulte, N. Tillmann, M.Y. Levin. Automating Software Testing Using Program Analysis. IEEE Software, vol. 25, no. 5, pp. 30-37, 2008.

M. Gopinathan, S.K. Rajamani. Enforcing Object Protocols by Combining Static and Runtime Analysis. In: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, pp. 245-260, 2008.

R.B. Grady. An Economic Release Decision Model: Insights into Software Project Management. In: Proceeedings of the Applications of Software Measurement Conference, pp. 225-239, 1999.

R.B. Grady, T. van Slack. Key Lessons in Achieving Widespread Inspection Use. IEEE Software, vol. 11, no. 4, pp. 46-57, 1994.

A. Gupta, P. Jalote. Test Inspected Unit or Inspect Unit Tested Code? In: Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement, pp. 51-60, 2007.

B. Hailpern, P. Santhanam. Software Debugging, Testing, and Verification. IBM Systems Journal, vol. 41 no. 1, 2002.

M. Hamill, K. Goseva-Popstojanova. Common Trends in Software Fault and Failure Data. IEEE Transactions on Software Engineering, vol. 35, no. 4, pp. 484-496, 2009.

A. Hanna, H.Z. Ling, X. Yang, M. Debbabi. A Synergy between Static and Dynamic Analysis for the Detection of Software Security Vulnerabilities. In: Proceedings of the Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009 on the Move to Meaningful Internet Systems: Part II, pp. 815-832, 2009.

J.T. Harding. Using Inspection Data to Forecast Test Defects. *Software Technology Transition*, pp. 19-24, 1998.

L. Harjumaa, I. Tervonen, A. Huttunen. Peer Reviews in Real Life – Motivators and Demotivators. In: Proceedings of the 5th International Conference on Quality Software, pp. 29-36, 2005.

M.J. Harrold. Testing: A Roadmap. In: Proceedings of the Conference on The Future of Software Engineering, pp. 61-72, 2000.

W.S. Harwood, A New Model for Inquiry: Is the Scientific Model Dead? Journal of College Science Teaching, vol. 33, no. 7, pp. 29-33, 2004.

M. Hayes. Quality First. Information Week, no. 889, p. 38, 2002.

H. Hedberg. Introducing the Next Generation of Software Inspection Tools. Product Focused Software Process Improvement; Lecture Notes in Computer Science, vol. 3009/2004, pp. 234-247, 2004.

J. Heidrich, J. Münch, W. Riddle, D. Rombach. People-Oriented Capture, Display, and Use of Process Information. New Trends in Software Process Modelling; Series on Software Engineering and Knowledge Engineering, vol. 18, pp. 121-180, 2006.

R. Hower, Software QA and Testing Resource Center, 2011, http://www.softwareqatest.com/qatfaq2.html, last visited: January 06, 2012.

W.S. Humphrey. The Software Quality Challenge. Crosstalk – The Journal of Defense Software Engineering, vol. 21, no. 6, pp. 4-9, 2008.

IEEE Standard 610.12-1990. IEEE Standard Glossary of Software Engineering Terminology, 1990.

IEEE Standard 829-1998. IEEE Standard for Software Test Documentation, 1998.

IEEE Standard 1028-1997. IEEE Standard for Software Reviews, IEEE Software Society, 1997.

Inspection repository, Fraunhofer Inspection Repository, 2011, http://inspection.iese.de, last visited: January 06, 2012.

ISO/IEC 15504 Standard (Spice). International Standard, part 1-5, 2006.

ISO/IEC 62304 Standard. Medical device software - Software life cycle processes, 2006.

F. Iturbe. Systematic Testing and Reviewing. In: Proceedings of the 11th International Conference on Software Engineering and Knowledge Engineering, pp. 295-299, 1999.

M. Ivarsson, T. Gorschek. A Method for Evaluating Rigor and Industrial Relevance of Technology Evaluations. Empirical Software Engineering, vol. 16, no. 3, pp. 365-395, 2011.

D. Jackson, M. Thomas, L.I. Millett, Editors. Software for Dependable Systems: Sufficient Evidence? Committee on Certifiably Dependable Software Systems, National Research Council, National Academy of Sciences, 2007.

J. Jacobs, J. van Mol, R. Kusters, J. Trienekens, A. Brombacher. Identification of Factors that Influence Defect Injection and Detection in

Development of Software Intensive Products. Information and Software Technology, vol. 49, no. 7, pp. 774-789, 2007.

I. Jacobson, G. Booch, J. Rumbaugh. The Unified Software Development Process. Addison-Wesley, Amsterdam, 1999.

P. Jalote, M. Haragopal. Overcoming the NAH Syndrome for Inspection Deployment. In: Proceedings of the 20th international conference on Software engineering, pp. 371-378, 1998.

P. Jalote, V. Vangala, T. Singh, P. Jain. Program Partitioning: A Framework for Combining Static and Dynamic Analysis. In: Proceedings of the 2006 international workshop on Dynamic systems analysis, pp. 11-16, 2006.

D. Janzen, H. Saiedian. Test-driven Development: Concepts, Taxonomy, and Future direction. IEEE Computer, vol. 38, no. 9, pp. 43-50, 2005.

R. Jeffery, L. Scott. Has twenty-five Years of Empirical Software Engineering made a Difference? In: Proceedings of the 9th Asia-Pacific Software Engineering Conference, pp. 539-546, 2002.

P.M. Johnson. Reengineering Inspections. Communication of the ACM, vol. 41, no. 2, pp. 49-52, 1998.

P.M. Johnson, D. Tjahjono. Does every Inspection Really Need a Meeting? Empirical Software Engineering, vol. 3, no. 1, pp. 9-35, 1998.

C. Jones. Applied Software Measurement: Assuring Productivity and Quality. McGraw-Hill, 1991.

C. Jones. Software Project Management Practices: Failure versus Success. Crosstalk – The Journal of Defense Software Engineering, vol. 17, no. 10, pp. 5-9, 2004.

C. Jones. Measuring Defect Potentials and Defect Removal Efficiency. Crosstalk – The Journal of Defense Software Engineering, vol. 21, no. 6, pp. 11-13, 2008.

P. Joshi, K. Sen, M. Shlimovich. Predictive Testing: Amplifying the Effectiveness of Software Testing. In: Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, pp. 561-564, 2007.

N. Juristo, A.M. Moreno, W. Strigel. Software Testing Practices in Industry. IEEE Software, vol. 23, no.4, pp. 19-21, 2006.

N. Juristo, A.M. Moreno, S. Vegas. Reviewing 25 Years of Testing Technique Experiments. Empirical Software Engineering, vol. 9, no. 1-2, 7-44, 2004.

N. Juristo, S. Vegas. Functional Testing, Structural Testing and Code Reading: What Fault Type do they Each Detect? Empirical Methods and Studies in Software Engineering, vol. 2765, pp. 208-232, 2003.

E. Kamsties, C.M. Lott. An Empirical Evaluation of Three Defect Detection Techniques. In: Proceedings of the 5th European Software Engineering Conference, pp. 362-383, 1995.

S.H. Kan. Metrics and Models in Software Quality Engineering. Addison-Wesley, Reading, MA, 1995.

E. Kantorowitz, A. Guttmann, L. Arzi. The Performance of the N-fold Requirements Inspection Method. Requirements Engineering Journal, vol. 2, no. 3, pp. 152-164, 1997.

D.W. Karolak. Software Engineering Risk Management, IEEE Computer Society Press, Wiley, 1996.

J. Kasurinen, O. Taipale, K. Smolander. Analysis of Problems in Testing Practices. In: Proceedings of the 16th Asia-Pacific Software Engineering Conference, pp. 309-315, 2009.

D. Kelly, T. Shepard. Task-directed Software Inspections. Journal of Systems and Software, vol. 73, no. 2, pp. 243-256, 2004.

J.C. Kelly, J.S. Sherif, J. Hops. An Analysis of Defect Densities found during Software Inspections. Journal of Systems and Software, vol. 17, no. 2, pp. 111-117, 1992.

B. Kinochita. Measuring Software Quality with Metrics – Why Static and Dynamic Analysis should Walk Hand-In-Hand. Testing Experience, no. 11, 2010.

B.A. Kitchenham, S. Charters. Guidelines for Performing Systematic Literature Reviews in Software Engineering. Technical Report EBSE-2007-01, Keele University and University of Durham, 2007.

M. Klaes, F. Elberzhager, R. v. Lengen, T. Schulz, J. Goebbels. A Framework for the Balanced Optimization of Quality Assurance Strategies Focusing on Small and Medium Sized Enterprises. In: Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications, pp. 335-342, 2009.

M. Klaes, F. Elberzhager, J. Muench, K. Hartjes, O.v.Graevemeyer. Transparent Combination of Expert and Measurement Data for Defect Prediction – An Industrial Case Study. In: Proceedings of the 32nd International Conference on Software Engineering, pp. 119-128, 2010a.

M. Klaes, F. Elberzhager, H. Nakao. Managing Software Quality through a Hybrid Defect Content and Effectiveness Model. In: Proceedings of the 2nd ACM-IEEE international symposium on Empirical software engineering and measurement, pp. 321-323, 2008a.

M. Klaes, H. Nakao, F. Elberzhager, J. Münch. Predicting Defect Content and Quality Assurance Effectiveness by Combining Expert Judgment and Defect Data - A Case Study. In: Proceedings of the 19th International Symposium on Software Reliability Engineering, pp. 17-26, 2008b.

M. Klaes, H. Nakao, F. Elberzhager, J. Münch. Support Planning and Controlling of Early Quality Assurance by Combining Expert Judgment and Defect Data--A Case Study. Empirical Software Engineering, vol. 15, no. 4, pp. 423-454, 2010b.

J.C. Knight, A.E. Myers. An Improved Inspection Technique. Communication of ACM, vol. 36, no. 11, pp. 50-69, 1993.

S. Kollanus, J. Koskinen. Survey of Software Inspection Research: 1991-2005. Computer Science and Information Systems reports, Working Papers WP-40, University of Jyväskylä, Finland, 2007.

T. Koomen, M. Pol. Test Process Improvement: A Practical Step-by-Step Guide to Structure Testing. ACM Press, 1999.

P.D. Kumar, A. Nema, R. Kumar. Hybrid Analysis of Executables to Detect Security Vulnerabilities. Proceedings of the 2nd India software engineering conference, pp. 141-148, 2009.

O. Laitenberger. Studying the Effects of Code Inspections and Structural Testing on Software Quality. In: Proceedings of the 9th International Symposium on Software Reliability Engineering, pp. 237-246, 1998.

O. Laitenberger, C. Atkinson, M. Schlich, K. El Eman. An Experimental Comparison of Reading Techniques for Defect Detection in UML Design Documents. Journal of Systems and Software, vol. 53, no. 2, p. 183-204, 2000.

O. Laitenberger, J.M. DeBaud. Perspective-based Reading of Code Documents at Robert Bosch GmbH. Information and Software Technology, vol. 39, no. 11, pp. 781-791, 1997.

O. Laitenberger, J.M. DeBaud. An Encompassing Life Cycle centric Survey of Software Inspection. Journal of Systems and Software, vol. 50, no. 1, pp. 5-31, 2000.

O. Laitenberger, K. El Emam, T.G. Harbich. An Internally Replicated Quasi-Experimental Comparison of Checklist and Perspective-Based Reading of Code Documents. IEEE Transactions on Software Engineering, vol. 27, no. 5, pp. 387-421, 2001.

F. Lanubile, T. Mallardo. Inspecting Automated Test Code: A Preliminary Study. In: Proceedings of the 8th international conference on Agile processes in software engineering and extreme programming, pp. 115-122, 2007.

E. Lee. Software Inspections: How to Diagnose Problems and Improve the Odds of Organizational Acceptance. Crosstalk, vol. 10, no. 8, pp. 10-13, 1997.

N. Leveson, C.S. Turner. An Investigation of the Therac-25 Accidents. IEEE Computer, vol. 26, no. 7, pp. 18-41, 1993.

P. Liggesmeyer. Software-Qualität: Testen, Analysieren und Verifizieren von Software. Spektrum Akademischer Verlag, Heidelberg, 2009.

S. Liu. Integrating Specification-based Review and Testing for Detecting Errors in Programs. In: Proceedings of the 9th international conference on formal methods and software engineering, pp. 136-150, 2007.

S. Liu, T. Tamai, S. Nakajima. Integration of Formal Specification, Review, and Testing for Software Component Quality Assurance. In: Proceedings of the 2009 ACM symposium on Applied Computing, pp. 415-421, 2009.

G.A.D. Lucca, M.D. Penta. Integrating Static and Dynamic Analysis to Improve the Comprehension of Existing Web Applications. In: Proceedings of the 7th IEEE International Symposium on Web Site Evolution, pp. 87-94, 2005.

M.V. Mantyla, C. Lassenius. What Types of Defects are really Discovered in Code Reviews? IEEE Transactions on Software Engineering, vol. 35, no. 3, pp. 430-448, 2009.

J. Martin, W.T. Tsai. N-fold Inspection: A Requirements Analysis Technique. Communications of ACM, vol. 33, no. 2, pp. 225-232, 1990.

P. Massicotte, L. Badri, M. Badri. Aspects-classes Integration Testing Strategy: An Incremental Approach. Rapid Integration of Software Engineering Techniques, vol. 3943, pp. 158-173, 2006.

T. McGibbon. A Business Case for Software Process Improvement. Technical Report F30602-92-C-0158. Data & Analysis Center for Software (DACS), 1996; revised version published 2007.

Metrics, Eclipse Metrics plugin, http://metrics.sourceforge.net/, 2010, last visited: January 06, 2012.

J. Miller. Estimating the Number of Defects after Inspection. Software Testing, Verification and Reliability, vol. 9, no. 4, pp. 167-189, 1999.

K.H. Möller. Fehlerverteilung als Hilfsmittel zur Qualitätsverbesserung und Fehlerprognose. VDE Fachtagung Technische Zuverlässigkeit, Berlin VDE Verlag, 1985.

K.H. Möller, D.J. Paulish. An Empirical Investigation of Software Fault Distribution. In: Proceedings of the IEEE First International Software Metrics Symposium, 1993.

J.C. Munson, T.M. Khoshgoftaar. The Detection of Fault-Prone Programs, IEEE Transactions on Software Engineering, vol. 18, no. 5., pp. 423-433, 1992.

G.J. Myers. A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections. Communications of the ACM, vol. 21, no. 9, pp. 760-768, 1978.

G.J. Myers. The Art of Software Testing. New York Wiley and Sons, 1979.

N. Nagappan, T. Ball, A. Zeller. Mining Metrics to Predict Component Failures. In: Proceedings of the International Conference on Software Engineering, pp. 452-461, 2006.

S.P. Ng, T. Murnane, K. Reed, D. Grant, T.Y. Chen. A Preliminary Survey on Software Testing Practices in Australia. In: Proceedings of the 2004 Australian Software Engineering Conference, pp. 116-125, 2004.

A.V. Nori, S.K. Rajamani, S. Tetali, A.V. Thakur. The Yogi Project: Software Property Checking via Static Analysis and Testing. In: Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, pp. 178-181, 2009.

ODC. Orthogonal Defect Classification v5.11, IBM, 2002, available: http://www.research.ibm.com/softeng/ODC/ODC.HTM, last visited January 06, 2012.

N. Ohlsson, H. Alberg. Predicting Fault-Prone Software Modules in Telephone Switches, IEEE Transactions on Software Engineering, vol. 22, no. 12, pp. 886-894, 1996.

N. Ohlsson, M. Helander, C. Wohlin. Quality Improvement by Identification of Fault-prone Modules Using Software Design Metrics. In: Proceedings of the 6th International Conference on Software Engineering, pp. 1-13, 1996.

T.J. Ostrand, E.J. Weyuker. The Distribution of Faults in a Large Industrial Software System. In: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 55-64, 2002.

T.J. Ostrand, E.J. Weyuker, R.M. Bell. We're Finding most of the Bugs, but what are we Missing? In: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation, pp. 313-322, 2010.

T. Otte, R. Moreton, H.D. Knoell. Applied Quality Assurance Methods under the Open Source Development Model. In: 32nd Annual IEEE International Computer Software and Applications Conference, pp. 1247-1252, 2008.

D.L. Parnas, D.M. Weiss. Active Design Reviews: Principles and Practices. In: Proceedings of the 8th International Conference on Software Engineering, pp. 132-136, 1985.

M.C. Paulk. The Capability Maturity Model: Guidelines for Improving the Software Process. SEI Series in Software Engineering, Addison-Wesley, 1995.

H. Peine, M. Jawurek, S. Mandel. Security Goal Indicator Trees: A Model of Software Features that Supports Efficient Security Inspection. In: Proceedings of the 11th IEEE High Assurance Systems Engineering Symposium, pp. 9-18, 2008.

D.E. Perry, A.A. Porter, L.G. Votta, Empirical Studies of Software Engineering: A Roadmap, In: Proceedings of the Future of Software Engineering, pp. 345-355, 2000.

K. Petersen, R. Feldt, S. Mujtaba, M. Mattsson. Systematic Mapping Studies in Software Engineering. In: Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering, pp. 1-10, 2008.

K. Peterson, C. Wohlin. Context in Industrial Software Engineering Research. In: Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement, pp. 401-404, 2009.

H. Petersson, T. Thelin, P. Runeson, C. Wohlin. Capture–Recapture in Software Inspections after 10 Years Research – Theory, Evaluation and Application. Journal of Systems and Software, vol. 72, no. 2, pp. 249-264, 2004.

B. Pettichord. Agile Testing Challenges. Pacific Northwest Software Quality Conference, 2004, available: http://pettichord.com/, last visited: January 06, 2012.

R. Pressman. Software Engineering: A Practitioner's Approach. 7th edition, McGraw-Hill, 2009.

A. Porter, L.G. Votta. Comparing Detection Methods for Software Requirements Specification: A Replication Using Professional Subjects. Empirical Software Engineering, vol. 3, no. 4, p. 355-379, 1998.

S.J. Prowell, C.J. Trammell, R.C. Linger, J.H. Poore. Cleanroom Software Engineering: Technology and Process, Addison-Wesley Professional, 1999.

H. Remus. Integrated software validation in the view of inspections/reviews. In: Proceedings of a symposium on Software validation: inspection-testing-verification-alternatives, pp. 57-65, 1984.

Risk Digest, Forum on Risks to the Public in Computers and Related Systems. ACM Committee on Computers and Public Policy, moderated by P.G. Neumann, 1985-2011, available: http://catless.ncl.ac.uk/risks, last visited: January 06, 2012.

D. Rombach, M. Ciolkowski, R. Jeffery, O. Laitenberger, F. McGarry, F. Shull. Impact of Research on Practice in the field of Inspections, Reviews and Walkthroughs: Learning from Successful Industrial Uses. ACM SIGSOFT Software Engineering Notes, vol. 33, no. 6, pp. 26-35, 2008.

M. Roper, M. Wood, J. Miller. An Empirical Evaluation of Defect Detection Techniques. Information and Software Technology, vol. 39, no. 11, pp. 763-775, 1997.

W. Royce. Managing the Development of Large Software Systems. In: Proceedings of the IEEE WESCON 26, pp. 1–9, 1970; reprinted in: Proceedings of the 9th International Conference on Software Engineering, pp. 328-338, 1987.

P. Runeson. A Survey of Unit Testing Practices. IEEE Software, vol. 23, no. 4, pp. 22-29, 2006.

P. Runeson, C. Andersson, T. Thelin, A. Andrews, T. Berling. What Do We Know about Defect Detection Methods? IEEE Software, vol. 23, no. 3, pp. 82-90, 2006.

P. Runeson, A. Andrews. Detection or Isolation of Defects? An Experimental Comparison of Unit Testing and Code Inspection. In: Proceedings of the 14th International Symposium on Software Reliability Engineering, pp. 3-13, 2003.

A. Schröter, T. Zimmermann, R. Premraj, A. Zeller. If your Bug Database could Talk. In: Proceedings of the 5th International Symposium on Empirical Software Engineering, pp. 18-20, 2006.

M. Shaw, Prospects for an Engineering Discipline of Software, IEEE Software, vol. 7, no.6, pp. 15-24, 1990.

Shields consortium, D2.1: Formalism definitions and representation schemata, deliverable of the SHIELDS research project within the European Community's Seventh Framework Programme, 2008, available: http://www.shields-project.eu, last visited: January 06, 2012.

F. Shull, V. Basili, B. Boehm, A.W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, M. Zelkowitz. What we have Learned about Fighting Defects, In: Proceedings of the 8th IEEE Symposium on Software Metrics, pp. 249-258, 2002.

D.I.K. Sjoberg, T. Dyba, M. Jorgensen. The Future of Empirical Methods in Software Engineering Research. Future of Software Engineering, pp. 358-378, 2007.

T. Spillner, P. Liggesmeyer. Software Qualitätssicherung in der Praxis – Ergebnisse einer Umfrage. Informatik Spektrum, vol. 17, no. 6, pp. 368-372, 1994.

T. Spillner, T. Linz. Basiswissen Softwaretest, Aus- und Weiterbildung zum Certified Tester, Heidelberg, dpunkt Verlag, 2003.

T. Spillner, K. Vosseberg, M. Winter, P. Haberl. Wie wird in der Praxis getestet? Online-Umfrage in Deutschland, Schweiz und Österreich. OBJEKTspektrum Ausgabe Testing/2011, 2011.

T. Spillner, K. Vosseberg, M. Winter. Umfrage 2011 - Softwaretest in der Praxis vom 01.05. - 31.05. Aktuelle Ergebnisse, available: http://www.softwaretest-umfrage.de/, last visited: January 06, 2012.

S.S. So, S.D. Cha, T.J. Shimeall, Y.R. Kwon. An Empirical Evaluation of Six Methods to Detect Faults in Software. Software Testing, Verification and Reliability, vol. 12, no. 3, pp. 155-171, 2002.

P. Strooper, M.A. Wojcicki. Selecting V&V Technology Combinations: How to Pick a Winner? In: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems, pp. 87-96, 2007.

D. Talby, A. Keren, O. Hazzan, Y. Dubinsky. Agile Software Testing in a Large-Scale Project. IEEE Software, vol. 23, no. 4, pp. 30-37, 2006.

G. Tassey. The Economic Impacts of Inadequate Infrastructure for Software Testing. National Institute of Standards and Technology, Research Triangle Institute – Health, Social, and Economic Research, Planning Report 02-3, Triangle Park, N.C., 2002.

T. Thelin, P. Runeson, B. Regnell. Usage-based Reading – an Experiment to Guide Reviewers with Use Cases. Information and Software Technology, vol. 43, no. 15, pp. 925-938, 2001.

T. Thelin, P. Runeson, C. Wohlin. An Experimental Comparison of Usage-based Reading and Checklist-based Reading. IEEE Transactions on Software Engineering, vol. 29, no. 8, pp. 687-704, 2003.

T. Thelin, P. Runeson, C. Wohlin, T. Olsson, C. Andersson. Evaluation of Usage-based Reading – Conclusions after Three Experiments. Empirical Software Engineering, vol. 9, no. 1-2, pp. 77-110, 2004.

TMap, Test Management Approach, 2011, available: http://www.tmap.net/en/tmap-next, last visited January 06, 2012.

G. Travassos, F. Shull, M. Fredericks, V.R. Basili. Detecting Defects in Object Oriented Designs: Using Reading Techniques to Improve Software Quality. In: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 47-56, 1999.

L.L. Tripp, W.F. Struck, B.K. Pflug. The Application of Multiple Team Inspections on a Safet-critical Software Standard. In: Proceedings of the 4th Software Engineering Standards Application Workshop, pp. 106-111, 1991.

B. Turhan, G. Kocak, A. Bener. Data Mining Source Code for Locating Software Bugs: A Case Study in Telecommunication Industry. Expert Systems with Applications, vol. 36, no. 6, pp. 9986-9990, 2009.

V. Venkatesh, M.G. Morris, G.B. Davis, F.D. Davis. User Acceptance of Information Technology: Toward a Unified View. MIS Quarterly, vol. 27, no. 3, pp. 425-478, 2003

L.G. Votta. Does every Inspection Need a Meeting? In: Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering, pp. 107-117, 1993.

S. Wagner. A Literature Survey of the Quality Economics of Defect Detection Techniques. In: Proceedings of the 2006 ACM / IEEE International Symposium on Empirical Software Engineering, pp. 194-203, 2006.

S. Wagner, J. Jürjens, C. Koller, P. Trischberger. Comparing Bug Finding Tools with Reviews and Test. In: Proceedings of the 17th International Conference on Testing of Communicating Systems, pp. 40-55, 2005.

N. Ward. Integrated Formal Verification and Validation of Safety Critical Software. In: Proceedings of the Aerospace Software Engineering for Advanced Systems Architectures Conference, pp. 10-13, 1993.

E.F. Weller. Lessons from three years of inspection data. IEEE Software, vol. 10, no. 5, pp. 38-45, 1993.

K.E Wiegers. Peer Reviews in Software. Addison-Wesley, 2002.

S.V. Wiel, L. Votta. Assessing Software Designs Using Capture-Recapture Methods. IEEE Transactions on Software Engineering, vol. 19, no. 11, pp. 1045-1054, 1993.

D. Winkler, S. Biffl, K. Faderl. Investigating the Temporal Behavior of Defect Detection in Software Inspection and Inspection-based Testing. Product-Focused Software Process Improvement, vol. 6156, pp. 17-31, 2010.

D. Winkler, B. Riedl, S. Biffl. Improvement of Design Specifications with Inspection and Testing. In: Proceedings of the 31st Euromicro Conference on Software Engineering and Advanced Applications, pp. 222-231, 2005.

M. Wood, M. Roper, A. Brooks, J. Miller. Comparing and Combining Software Defect Detection Techniques – A Replicated Empirical Study. In: Proceedings of the 6th European Software Engineering Conference, pp. 262-277, 1997.

C. Wohlin, P. Runeson. Defect Content Estimations from Review Data. In: Proceedings of the 20th International Conference on Software Engineering, pp. 400-409, 1998.

C. Wohlin, P. Runeson, J. Brantestam. An Experimental Evaluation of Capture-Recapture in Software Inspection. Software Testing, Verification and Reliability, vol. 5, no. 4, pp. 213-232, 1995.

C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, A. Wesslen. Experimentation in software engineering an introduction, Kluwer, 2000.

M.A. Wojcicki, P. Strooper. An Iterative Empirical Strategy for the Systematic Selection of a Combination of Verification and Validation Technologies. In: Proceedings of the 5th International Workshop on Software Quality, pp. 9-14, 2007.

M.V. Zelkowitz, D.R. Wallace, Experimental Models for Validating Technology, IEEE Computer, vol. 31, no. 5, pp. 23-31, 1998.

S. Zhang, Y. Lin, Z. Gu, J. Zhao. Effective Identification of Failure-inducing Changes: A Hybrid Approach. In: Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pp. 77-83, 2008.

D.M. Zimmerman, J.R. Kiniry. A Verification-centric Software Development Process for Java. In: Proceedings of the 9th International Conference on Quality Software, pp. 76-85, 2009.

Appendix A Checklists used during Evaluation

Appendix A shows the different checklists that were used during the two case studies.

A.1 DETECT Evaluation: Checklists

1. Requirements checklist

- Does the navigation in the GIT model work properly?
- Does the depth-first search work correctly in the GIT model?
- Does the recovery mechanism work correctly?
- Do the logical functions for the GIT work correctly?
- Does the history view reflect the structure of the GIT?
- Are all comments saved and shown properly?
- Does the parser of the history file work correctly?
- Is the report shown correctly?

2. Functional checklist

- Are all calculations done correctly?
- Are all loops, branches and logical operators complete, correct, and nested correctly?
- Are all break conditions correctly implemented?
- Are data structures used correctly?
- Does each variable have its correct type? Is each variable instantiated correctly?
- Are values correctly committed?

- Are there any methods that were not called? Do redundant variables exist?

3. Extensibility checklist

- Is the code documented completely and correctly? Do the comments fit to the source code?
- Are the methods easy to understand?
- Were appropriate method names chosen?
- Is the code structured in a consistent way?
- Please check side effects.

4. Reliability checklist

- Does the code have appropriate exception handling?
- Is debugging and error information gathered?
- Is xml-code parsed sufficiently?

A.2 JSeq Evaluation: Checklists

1. Requirements checklist

- Is a mechanism for organizing requirements implemented and implemented correctly?
- Is a mechanism for organizing interfaces, stimuli, and responses implemented and implemented correctly?
- Is a correct enumeration implemented (e.g., enumeration of sequences of a determined length, extensibility with respect to length of sequences, etc.)?
- Is a mechanism for organizing states implemented and implemented correctly?
- Is a mechanism for allocating states implemented and implemented correctly?
- Please check all change functions with respect to enumerations (e.g., renaming, deleting, sorting stimuli).

- Please check if changes within the stimulus list are conducted correctly.
- Please check if changes within the response list are conducted correctly.
- Please check all further enumeration functionalities.

2. Functional checklist

- Are all calculations done correctly?
- Are all loops, branches and logical operators complete, correct, and nested correctly?
- Are all break conditions correctly implemented?
- Are data structures used correctly?
- Does each variable have its correct type? Is each variable instantiated correctly?
- Are values correctly committed?
- Are there methods that were not called? Do redundant variables exist?

3. Extensibility checklist

- Is the code documented completely and correctly? Do the comments fit to the source code?
- Are the methods easy to understand?
- Were appropriate method names chosen?
- Is the code structured in a consistent way?
- Please check side effects.

4. Performance checklist

- Please check if more efficient methods or algorithms could improve the performance.

- Are values calculated efficiently (e.g., is each calculated value used, are the same values calculated more than once and could they be reused)?
- Are loops used efficiently?
- Are methods called in an unnecessary manner?

5. Reliability checklist

- Does the code have appropriate exception handling?
- Is debugging and error information gathered?
- Is data stored periodically?
- Is external data used correctly in the software?
- Can a saved file be loaded correctly?

Appendix B Experimental Designs

B.1 Design 1

This design is similar to a typical industry setting and could be applied during concrete development and quality assurance activities.

First, a code inspection using a checklist is performed by two independent groups. Each group consists of one developer and one tester. Each group checks a set of those code classes the developer was responsible for. The checklist covers different quality aspects, such as commentaries, structure, and functional aspects. Each group uses the same checklist, which consists of various questions. Using a so-called focused checklist that is adapted to the environment instead of using a standard checklist improves effectiveness and is consistent with recommendations found in the literature (Gilb and Graham, 1993). Due to the availability of the developer, each finding can be discussed immediately and it can be decided if a real defect was discovered, which is documented and corrected afterwards. After the inspection has been completed, an experienced quality assurance engineer aggregates the findings for each group into a defect profile.

The second step comprises the quality monitoring of the derived defect profile. Reading rate, overall number of found defects, and defect distribution across the inspected code parts are considered and checked for each defect profile from the two groups.

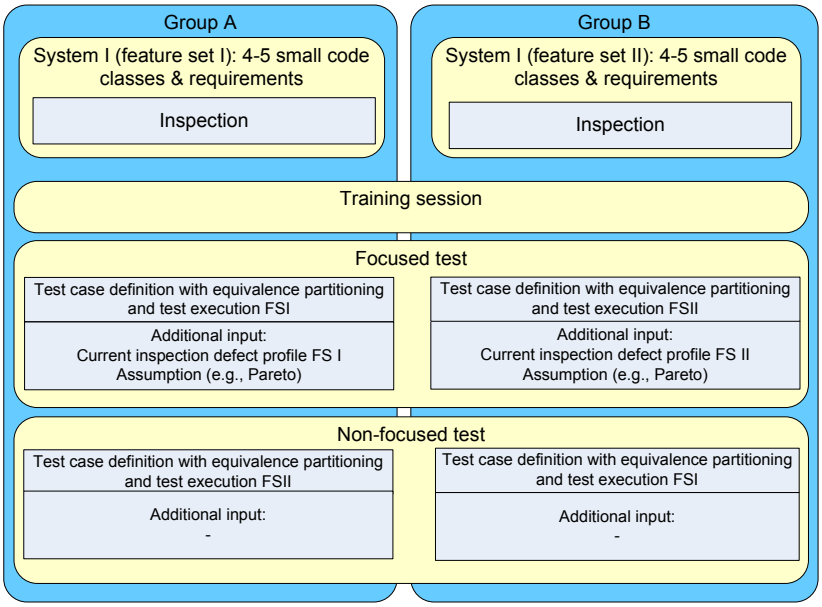
A crossover design is selected to allow a comparison of a non-focused test with a focused test using the In^{2}Test approach. Testing techniques are taught in a training session. First, the focused test is conducted.

In step 3, the prioritization of parts of the system is done, i.e., the test strategy for the focused test is derived. One assumption, already used in the case studies performed before, is made, namely a Pareto distribution. Code classes in which a significant number of defects were found during the inspection are selected due to the assumption that more defects are expected in such parts. Those code classes are then tested more intensively using mainly equivalence partitioning and, to some extent, boundary-value analyses (step 4), and non-prioritized code classes are rarely tested or even not tested at all. Focusing could be done by omitting and selecting complete code classes, or by selecting more or fewer test cases (per equivalence classes) per code class. It could also

include different efforts for explorative testing. Defects found during testing are documented and corrected afterwards.

In order to be able to compare the focused test with a non-focused test in terms of effectiveness and efficiency, each group also performs a non-focused test of the code classes of another group without using their inspection results. No prioritization strategy is derived, but a standardized testing where each code class is treated equally is done on the uncorrected code from the prioritized testing activity (i.e., corrected code from after the inspection is done), again using equivalence partitioning and boundary value analyses. The time needed and the defects found are documented, and compared with the results from the prioritized test.

In the experiment, the following variables are considered: number of found defects is measured as defect content (absolute number) and defect density (relative number); effort is measured in minutes, and number of test cases; and size is measured in lines of code. Efficiency is calculated using the number of defects found per time period.



The figure above shows an overview of the design[2]. Finally, several variations and extensions are possible with respect to the given design, such as groups with more subjects or more groups in order to achieve higher validity in the results. Furthermore, more than one quality assurance run could be performed in order to adapt the initial

[2] The design was presented during a poster session at the ISERN 2011 meeting.

assumption and to find those assumptions that fit best in the given context.

B.2 Design 2

This design is to be used in a lecture with a number of students and the constraint of limited time. Therefore, some material has to be prepared for the experiment.

Instead of an inspection being conducted by the students themselves to derive a defect profile, this has to be prepared in advance, i.e., inspection results are predefined with respect to a certain system under inspection. In addition, historical inspection and test defect data are prepared and are used as input for the students to conduct the prioritization.

Two groups of students are determined, each of which gets four to five small code classes. This forms the basis for the testing activity, where a test case definition using equivalence partitioning is used. The code classes also have to be prepared with respect to seeded defects. A training session that teaches the students basic testing should be held.

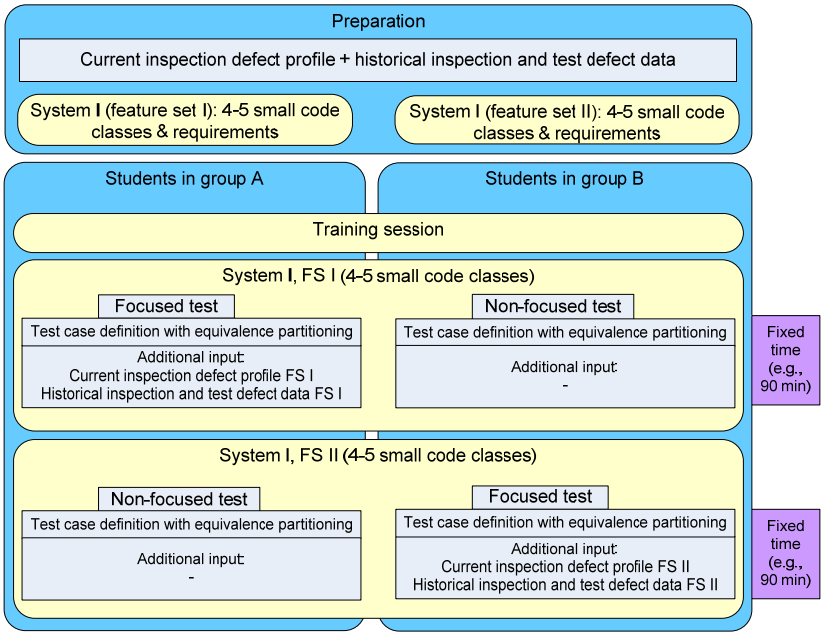
Since there are two groups, a crossover design is possible. The first group starts with a focused test of those code classes for which they also got a defect profile (i.e., current inspection data) and the historical defect data. Based on the historical data, each student has to derive an assumption that seems valid from his point of view. The historical defect data implies a Pareto distribution; however, we did not want to state an instruction such as "Focus on the three top defect-prone code classes", as this would decrease the significance of the experiment. Consequently, we push the students into a certain direction, but also give them some degree of freedom for the prioritization. Based on the derived assumption, each student in the focused group should derive test cases for the prioritized code classes. The non-focused group derives test cases for the same code classes without any prioritization. The time is restricted to, for instance, 90 minutes, which is the normal time of one lecture session.

Afterwards, students from the focused group do a non-focused test on the second set of code classes, and vice versa.

A comparison of the groups in both runs can be done in order to analyze whether the focused group was more efficient than the non-focused group. In the experiment, the following variables are considered: number of found defects during testing is measured as defect content (absolute

number), and effort is measured in minutes. Efficiency is calculated using the number of defects found per time period.

Certain options for adaptations exist, for instance, performing the test cases after defining them. However, this depends on the available resources and on time restrictions. Finally, the figure below gives an overview of the design[3].



[3] The design was presented during a poster session at the ISERN 2011 meeting.

Appendix C Questionnaire

A questionnaire was developed to allow getting feedback from practitioners. Such feedback can be gathered based on a presentation about the approach, i.e., no time-consuming evaluation is needed for gaining such feedback. The questionnaire is based on the Unified Theory of Acceptance and Use of Technology (UTAUT) model (Venkatesh et al, 2003), and has been adapted with respect to the In²Test approach. Furthermore, some general questions are included at the beginning in order to be able to classify the participants of the questionnaire.

The specific question parts A to E (see part "Evaluation of the Use and Acceptance of the In²Test approach" in the questionnaire) can be mapped to hypothesis 2.1, respectively to the research questions 2.1 to 2.4 as follows:

- RQ 2.1 (improvement): Questions A, C, (E)
- RQ 2.2 (understandability): Questions B
- RQ 2.3 (applicability): Questions B, C, D
- RQ 2.4 (reasonability): Questions A

Questionnaire about the In²Test Approach

Please answer the following questions. Answer as spontaneously as you can. This will take you about 5-10 minutes. Of course, your answers will remain completely anonymous. Thank you very much for taking the time to fill out the questionnaire.

General Questions

Company:		**Start Time:**
Domain:		**End Time:**
Profession / role:		

Dealing with Quality Assurance

	Question	**A lot**	**Rather a lot**	**Neither a lot nor little**	**Rather little**	**Little**
1.	How do you rate your knowledge about	○	○	○	○	○

	software inspections / reviews?					
2.	How do you rate your knowledge about software testing?	○	○	○	○	○
3.	How do you rate your experience with software inspections / reviews?	○	○	○	○	○
4.	How do you rate your experience with software testing?	○	○	○	○	○

	Question	**Answer**
5.	For how many **years** have you been working with software inspections / reviews?	
5.1	Which **inspection techniques** have you applied (e.g., informal reviews, formal inspections, walkthroughs)?	
5.2	Which kinds of **documents** have you inspected (e.g., requirements, code)?	
6.	For how many **years** have you been working with software testing?	
6.1	Which **testing techniques** have you applied (e.g., experience-based, requirements-based, boundary-value analysis)?	
6.2	On which **level** have you done testing (e.g., unit, components, system)?	

	Question	**Daily**	**Weekly**	**Monthly**	**Rarely**	**Never**
7.	How often have you personally performed or attended software inspections / reviews during the past 12 months?	○	○	○	○	○
8.	How often have you personally performed or attended software testing during the past 12 months?	○	○	○	○	○

Motivation

	Question	**Agree**				**Disagree**		
9.	I am interested / motivated in getting to know new approaches for software development.	○	○	○	○	○	○	○
10.	I am interested / motivated in getting to know new approaches for software quality assurance.	○	○	○	○	○	○	○
11.	I would like to know more about quality assurance techniques.	○	○	○	○	○	○	○
12.	The topic was too new for me to comprehend it.	○	○	○	○	○	○	○

I would like to make the following comment(s) / improvement suggestion(s):

I had a problem with … <please explain>:

Evaluation of the Use and Acceptance of the In²Test Approach

The following questions are based on the Unified Theory of Acceptance and Use of Technology (UTAUT).

(A) Performance expectancy	**Agree**				**Disagree**		
I would find the In²Test approach useful in my work.	○	○	○	○	○	○	○
Using the In²Test approach enables me to accomplish tasks more quickly.	○	○	○	○	○	○	○
Using the In²Test approach increases my productivity.	○	○	○	○	○	○	○
If I use the In²Test approach, I will increase my chances of getting a raise (e.g., by decreasing effort, by finding more defects).	○	○	○	○	○	○	○

(B) Effort expectancy	**Agree**				**Disagree**		
My interaction with the In²Test approach would be clear and understandable.	○	○	○	○	○	○	○
It would be easy for me to become skillful at using the In²Test approach.	○	○	○	○	○	○	○
I would find the In²Test approach easy to use.	○	○	○	○	○	○	○
Learning to apply the In²Test approach is easy for me.	○	○	○	○	○	○	○

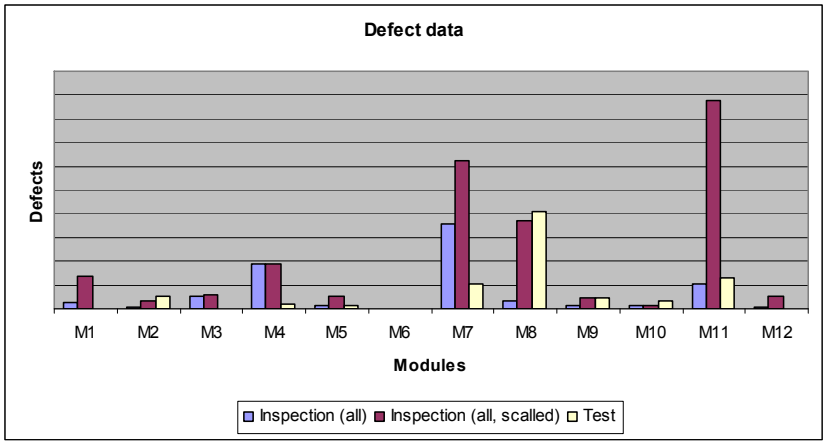
(C) Attitude toward using technology	**Agree**				**Disagree**		
Using the In²Test approach is a good idea.	○	○	○	○	○	○	○
The In²Test approach makes work more interesting.	○	○	○	○	○	○	○
Working with the In²Test approach is fun.	○	○	○	○	○	○	○
I like working with the In²Test approach.	○	○	○	○	○	○	○

(D) Facilitating conditions	**Agree**				**Disagree**		
I have the resources necessary to use the In²Test approach.	○	○	○	○	○	○	○
I have the knowledge necessary to use the In²Test approach.	○	○	○	○	○	○	○
The In²Test approach is compatible with other approaches I use.	○	○	○	○	○	○	○

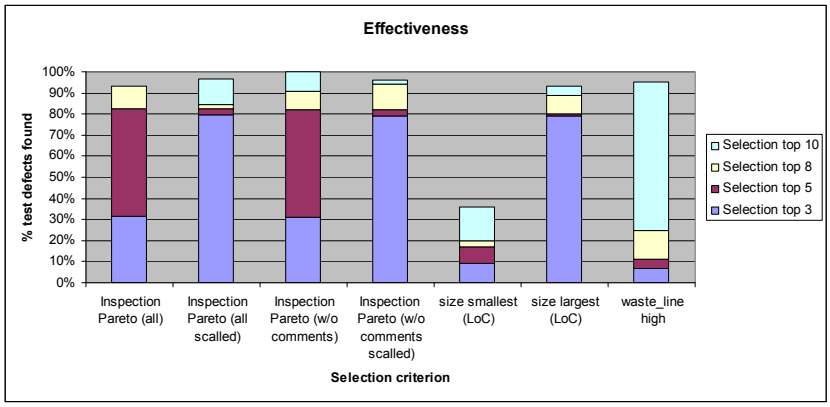
(E) Behavioral intention to use the system	**Agree**				**Disagree**		
I intend to use the In²Test approach in the next 6 months.	○	○	○	○	○	○	○
I predict I would use the In²Test approach in the next 6 months.	○	○	○	○	○	○	○
I plan to use the In²Test approach in the next 6 months.	○	○	○	○	○	○	○

Appendix D Initial Industrial Evaluation Results

In order to continue the evaluation of the integrated approach, we are currently analyzing defect data from an industry partner from the automotive domain. We analyzed inspection and test defect data from twelve different modules (see next figure), and calculated effectiveness values when omitting a certain number of modules due to currently seven different criteria (i.e., we are doing a retrospective analysis).



For example, look at the first bar in the figure below where we considered all inspection and test defects that were documented. If the top three modules were selected for testing, which contained the most inspection defects, about 30% of all testing defects had been found. Furthermore, if the top-5 modules were selected based on the inspection results, more than 80% of the test defects have been found. Focusing on two thirds of all modules would lead to more than 90% of all test defects found. In the last case, a selection of the top-10 modules would lead to no additional benefit. Besides taking all inspection defects into account, we performed the same analysis with respect to three additional inspection metrics and three product metrics. The two scaled bars consider the fact that for some code modules, only parts were inspected, and we upscaled the inspection defect data accordingly. The two bars "inspection w/o comments" discarded those inspection issues that were classified as comment. Next, we considered size in lines of code, considering the smallest ones first and then the largest ones first. Finally, we analyzed waste_line, which is a measure that expresses how much has changed between released versions of a module.



Based on the results displayed in the figure above, we could draw some initial conclusions for the given context.

1. General observations: First of all, three metrics provide appropriate results when only the top-3 modules are considered (two inspection metrics, one size metric). With respect to the top-5 selection criterion, all four inspection metrics provide suitable results, i.e., more than 80% of the defects were found. With respect to the top-10 selection, all except one metric lead to suitable results.

2. Specific observations: With respect to the inspection metrics, only two of four led to appropriate results when selecting the top-3 defect-prone modules based on the inspection results (i.e., about 80% of the test defects were found). This changes when the top-5 modules are considered. With respect to the top-8 modules, three metrics out of four led to defect numbers for testing of more than 90%. One inspection metric even found all defects when the top-10 modules were considered, i.e., two modules could completely be omitted during testing and all test defects would have been found. This means that an effort reduction would be possible without any reduction in quality. With respect to size, focusing on the largest modules first also led to appropriate results in this context, but these were not as good as when the inspection results were considered. However, the difference is small.

3. Other observations: Though the size metric focusing on the smallest modules first is also mentioned in the literature as a good predictor of defect-proneness, it showed bad results in our context (even for the top-10 smallest modules).

We are currently focusing on the comparison of the superior size metric and the inspection metrics with respect to defect data of additional modules, i.e., the analysis in the given context is still continuing.

Lebenslauf

Name	Frank Elberzhager	
Wohnort	Wilhelmstr. 16 67655 Kaiserslautern	
Geburtsdatum	29. März 1980	
Geburtsort	Wipperfürth	
Familienstand	Verheiratet	
Staatsangehörigkeit	Deutsch	
Schulbildung	1986-1989	Städtische Gemeinschaftsgrundschule Hückeswagen
	1989-1996	Städtische Realschule Hückeswagen
	1996-1999	Engelbert von Berg Gymnasium Wipperfürth Abschluss: Abitur
Zivildienst	1999-2000	Ev. Altenzentrum Hückeswagen
Studium	2000-2005	Technische Universität Kaiserslautern Abschluss: Dipl.-Inform.
Berufstätigkeit	2005-heute	Wissenschaftlicher Mitarbeiter am Fraunhofer-Institut für Experimentelles Software Engineering, Kaiserslautern

Kaiserslautern, den 9. Januar 2012

PhD Theses in Experimental Software Engineering

Volume 1 **Oliver Laitenberger** (2000), *Cost-Effective Detection of Software Defects Through Perspective-based Inspections*

Volume 2 **Christian Bunse** (2000), *Pattern-Based Refinement and Translation of Object-Oriented Models to Code*

Volume 3 **Andreas Birk** (2000), *A Knowledge Management Infrastructure for Systematic Improvement in Software Engineering*

Volume 4 **Carsten Tautz** (2000), *Customizing Software Engineering Experience Management Systems to Organizational Needs*

Volume 5 **Erik Kamsties** (2001), *Surfacing Ambiguity in Natural Language Requirements*

Volume 6 **Christiane Differding** (2001), *Adaptive Measurement Plans for Software Development*

Volume 7 **Isabella Wieczorek** (2001), *Improved Software Cost Estimation A Robust and Interpretable Modeling Method and a Comprehensive Empirical Investigation*

Volume 8 **Dietmar Pfahl** (2001), *An Integrated Approach to Simulation-Based Learning in Support of Strategic and Project Management in Software Organisations*

Volume 9 **Antje von Knethen** (2001), *Change-Oriented Requirements Traceability Support for Evolution of Embedded Systems*

Volume 10 **Jürgen Münch** (2001), *Muster-basierte Erstellung von Software-Projektplänen*

Volume 11 **Dirk Muthig** (2002), *A Light-weight Approach Facilitating an Evolutionary Transition Towards Software Product Lines*

Volume 12 **Klaus Schmid** (2003), *Planning Software Reuse – A Disciplined Scoping Approach for Software Product Lines*

Volume 13 **Jörg Zettel** (2003), *Anpassbare Methodenassistenz in CASE-Werkzeugen*

Volume 14 **Ulrike Becker-Kornstaedt** (2004), *Prospect: a Method for Systematic Elicitation of Software Processes*

Volume 15 **Joachim Bayer** (2004), *View-Based Software Documentation*

Volume 16 **Markus Nick** (2005), *Experience Maintenance through Closed-Loop Feedback*

Volume 17 **Jean-François Girard** (2005), *ADORE-AR: Software Architecture Reconstruction with Partitioning and Clustering*

Volume 18 **Ramin Tavakoli Kolagari** (2006), *Requirements Engineering für Software-Produktlinien eingebetteter, technischer Systeme*

Volume 19 **Dirk Hamann** (2006), *Towards an Integrated Approach for Software Process Improvement: Combining Software Process Assessment and Software Process Modeling*

Volume 20 **Bernd Freimut** (2006), *MAGIC: A Hybrid Modeling Approach for Optimizing Inspection Cost-Effectiveness*

Volume 21 **Mark Müller** (2006), *Analyzing Software Quality Assurance Strategies through Simulation. Development and Empirical Validation of a Simulation Model in an Industrial Software Product Line Organization*

Volume 22 **Holger Diekmann** (2008), *Software Resource Consumption Engineering for Mass Produced Embedded System Families*

Volume 23 **Adam Trendowicz** (2008), *Software Effort Estimation with Well-Founded Causal Models*

Volume 24 **Jens Heidrich** (2008), *Goal-oriented Quantitative Software Project Control*

Volume 25 **Alexis Ocampo** (2008), *The REMIS Approach to Rationale-based Support for Process Model Evolution*

Volume 26 **Marcus Trapp** (2008), *Generating User Interfaces for Ambient Intelligence Systems; Introducing Client Types as Adaptation Factor*

Volume 27 **Christian Denger** (2009), *SafeSpection – A Framework for Systematization and Customization of Software Hazard Identification by Applying Inspection Concepts*

Volume 28 **Andreas Jedlitschka** (2009), *An Empirical Model of Software Managers' Information Needs for Software Engineering Technology Selection*
A Framework to Support Experimentally-based Software Engineering Technology Selection

Volume 29 **Eric Ras** (2009), *Learning Spaces: Automatic Context-Aware Enrichment of Software Engineering Experience*

Volume 30 **Isabel John** (2009), *Pattern-based Documentation Analysis for Software Product Lines*

Volume 31 **Martín Soto** (2009), *The DeltaProcess Approach to Systematic Software Process Change Management*

Volume 32 **Ove Armbrust** (2010), *The SCOPE Approach for Scoping Software Processes*

Volume 33 **Thorsten Keuler** (2010), *An Aspect-Oriented Approach for Improving Architecture Design Efficiency*

Volume 34 **Jörg Dörr** (2010), *Elicitation of a Complete Set of Non-Functional Requirements*

Volume 35 **Jens Knodel** (2010), *Sustainable Structures in Software Implementations by Live Compliance Checking*

Volume 36 **Thomas Patzke** (2011), *Sustainable Evolution of Product Line Infrastructure Code*

Volume 37 **Ansgar Lamersdorf** (2011), *Model-based Decision Support of Task Allocation in Global Software Development*

Volume 38 **Ralf Carbon** (2011), *Architecture-Centric Software Producibility Analysis*

Volume 39 **Florian Schmidt** (2012), *Funktionale Absicherung kamerabasierter Aktiver Fahrerassistenzsysteme durch Hardware-in the-Loop-Tests*

Volume 40 **Frank Elberzhager** (2012), *A Systematic Integration of Inspection and Testing Processes for Focusing Testing Activities*

Software Engineering has become one of the major foci of Computer Science research in Kaiserslautern, Germany. Both the University of Kaiserslautern's Computer Science Department and the Fraunhofer Institute for Experimental Software Engineering (IESE) conduct research that subscribes to the development of complex software applications based on engineering principles. This requires system and process models for managing complexity, methods and techniques for ensuring product and process quality, and scalable formal methods for modeling and simulating system behavior. To understand the potential and limitations of these technologies, experiments need to be conducted for quantitative and qualitative evaluation and improvement. This line of software engineering research, which is based on the experimental scientific paradigm, is referred to as 'Experimental Software Engineering'.
In this series, we publish PhD theses from the Fraunhofer Institute for Experimental Software Engineering (IESE) and from the Software Engineering Research Groups of the Computer Science Department at the University of Kaiserslautern. PhD theses that originate elsewhere can be included, if accepted by the Editorial Board.

ISBN 978-3-8396-0445-8

Fraunhofer IESE

Technische Universität Kaiserslautern
AG Software Engineering

PhD Theses in Experimental Software Engineering