Optimizing Cost and Quality by Integrating Inspection and Test Processes

Frank Elberzhager[1], Jürgen Münch[1,2], Dieter Rombach[1,2]
[1]Fraunhofer IESE, [2]University of Kaiserslautern
Kaiserslautern, Germany
{first.last}@iese.fraunhofer.de

Bernd Freimut
Robert Bosch GmbH, Stuttgart, Germany
bernd.freimut@de.bosch.com

ABSTRACT
Inspections and testing are two of the most commonly performed software quality assurance processes today. Typically, these processes are applied in isolation, which, however, fails to exploit the benefits of systematically combining and integrating them. Expected benefits of such process integration are higher defect detection rates or reduced quality assurance effort. Moreover, when conducting testing without any prior information regarding the system's quality, it is often unclear which parts or which defect types should be prioritized. Existing approaches do not explicitly use information from inspections in a systematical way to focus testing processes. In this article, we present an integrated two-stage approach that routes inspection data to test processes in order to prioritize code classes and defect types. While an initial version of the approach focused on prioritizing code classes, this article focuses on the prioritization of defect types for testing. Results from a case study where the approach was applied on the code level show that those defect types could be prioritized before the testing that afterwards actually showed up most often during the test process. In addition, an overview of related work and an outlook on future research directions are given.

Categories and Subject Descriptors
D.3.3 [**Software Engineering**]: Testing and Debugging – *code inspections and walkthroughs*

General Terms
Verification

Keywords
Inspection, testing, testing focus, two-stage prioritization, quality assurance strategy, defect types, case study, ODC

1. INTRODUCTION
The software industry has to cope with increasing cost pressure, calls for shorter times to market, and a growing demand for high quality. Simultaneously, software systems are growing in their complexity. In order to achieve ambitious time, cost, and quality goals under these constraints, the development and maintenance approach needs to be optimized. One important strategy here is a process for the systematic optimization and integration of quality assurance (QA) [45].

Today's QA processes, such as inspections or testing [1][2], are often applied in isolation. Benefits such as reduced effort or higher defect detection rates that might result from a systematic combination are often not exploited. Especially testing is often a very time-consuming QA activity, sometimes requiring more than 50% of the overall development effort [3][4]. Consequently, one goal is often to reduce both testing effort and overall QA effort.

Several approaches exist that focus or combine QA activities in order to exploit synergy effects and reduce testing costs. With respect to focusing testing activities, many approaches have been developed; for instance, prioritization of system parts (e.g., code classes) that are expected to be most defect-prone [5], focusing on certain defect types that are expected to appear most often [47], or a selection of test cases that are most important by some measure [8]. To be able to perform such prioritizations, different product metrics, a defect classification, or statistics are often used. Nevertheless, none of these techniques uses defect data from earlier QA activities in order to focus testing activities.

With respect to a combination of different QA activities, Strooper and Wojcicki [9] propose a procedure for choosing and combining different QA activities in order to apply them in an efficient manner. Tools combining static and dynamic analysis in order to, for example, support test case generation can reduce the test effort [11]. Furthermore, combining inspection and testing techniques is often suggested to reduce the overall QA effort and to be more effective [12][13]. However, even if a high number of studies have been performed comparing various inspection and testing techniques, resulting in the suggestion to apply them in a combined way, inspection and testing techniques are most often applied in isolation without exploiting synergy effects, i.e., inspection data is often not systematically used for focusing testing activities.

This article presents an integrated inspection and testing approach that uses inspection defect data to focus testing activities. Elberzhager et al. [14] presented a one-stage prioritization approach in which inspection defect information is used to prioritize those code classes that are expected to be most defect-prone and thus, to focus testing activities on those parts. In contrast, this article focuses on a two-stage prioritization approach. First, code classes expected to be most defect-prone are selected, i.e., code classes are prioritized. Second, defect types expected to appear most often are chosen. On the one hand, prioritization of certain defect types can support the planning of testing processes by deriving and prioritizing test cases targeting

ICSE'11, May 21-28, 2011, Waikiki, Honolulu, HI, USA

the chosen defect types. On the other hand, it can support the controlling of testing processes by comparing expected defect types with the distribution of found defect types.

The prioritization of code classes and defect types is put together to obtain the combined prioritization. The proposed approach uses explicitly defined assumptions and so-called selection rules derived from them to support the prioritization of code classes and defect types. A case study was performed to evaluate the defect type prioritization. Two runs showed that the prioritization of defect types led to additional defects being found by a testing activity.

The remainder of the article is structured as follows. Section 2 presents an overview of related work. Section 3 shows the integrated two-stage inspection and testing approach. Section 4 describes the case study. Finally, Section 5 concludes the article and presents future work.

2. RELATED WORK

The integrated two-stage inspection and testing approach *combines* a static and a dynamic QA activity in an integrated way in order to *focus* testing activities. Thus, an overview of related work regarding those two aspects is presented. In addition, background information on defect classifications is given and the relation to the proposed approach is discussed briefly.

2.1 Combination of Static and Dynamic Quality Assurance Techniques

The combination of different static and dynamic QA techniques can be done in a compiled or in an integrated way. In this regard, compilation means that different static and dynamic QA techniques are applied for a common goal but in isolation, without using input from one analysis for the second one. In contrast, integration means that the output of one QA technique is used as input for the second QA technique.

One example of compilation is to apply both static and dynamic analyses to improve a certain quality property, but without using results from each other. For example, Aggarwal and Jalote [42] propose an approach that combines a static and a dynamic analysis technique in a compiled manner in order to detect certain vulnerabilities. With respect to inspection (i.e., static QA) and testing (i.e., dynamic QA) techniques, many studies and experiments have been performed to compare them [13][24][29], often followed by the suggestion to apply both. Wood et al. [43] and Gack [44] analyzed different combinations of inspection and testing techniques and calculated their benefit. It could be shown that a mixed strategy often outperforms a strategy where only one technique is applied in terms of cost and found defects. However, although it is often concluded that inspection and testing techniques should be combined, they are most often applied in isolation in this case without exploiting any synergy effects between them. Finally, it is often suggested to perform continuous verification and validation, which includes the combination of a number of different static and dynamic QA techniques and tools [28]. However, despite the fact that an isolated application of different QA techniques improves the overall quality of the product, again, often no synergy effects are explicitly exploited.

With respect to integration**,** different static and dynamic analyses are integrated in order to reduce drawbacks of using them in isolation. For instance, static analysis results can be used as input for a dynamic analysis (e.g., results from a static analysis can be used to focus a dynamic analysis onto specific code parts [11]). Furthermore, approaches explicitly combining inspections and test-case generation, such as inspection-based testing (e.g., UBT-i, [30]), offer another way to integrate static and dynamic QA techniques. Finally, Harding [31] 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. However, any support for focusing testing activities is still missing.

Besides the combination of concrete QA techniques, approaches for selecting, combining, and evaluating different static and dynamic QA techniques can be helpful for finding an appropriate QA mix in a given environment (e.g., [9]).

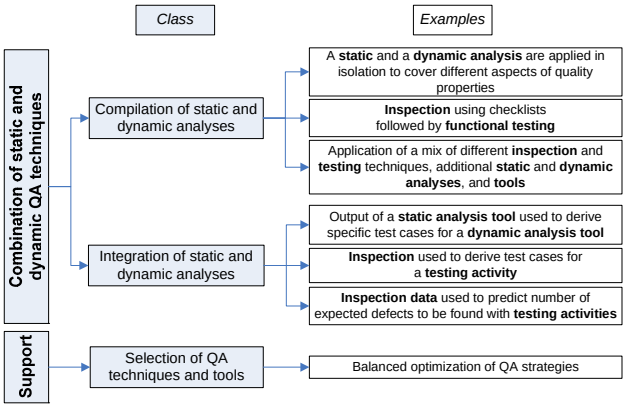

Figure 1: Overview of the combination of static and dynamic QA techniques and selection approaches

Figure 1 gives an overview of these approaches. In summary, there are many approaches that combine static and dynamic QA techniques in a compiled way. However, synergy effects are often not exploited. Furthermore, the integration of static and dynamic QA techniques mainly focuses on tool-supported analyses. With respect to software inspections, one main goal of past inspection research has been to improve the inspection itself rather than to integrate inspection techniques with testing, except for some approaches that use the inspection to support test case derivation. However, although inspection and testing techniques are sometimes integrated in an informal way in industry, the authors could not find any systematic approach in the existing literature that integrates them in order to exploit additional synergy effects.

2.2 Focusing Testing Activities

One popular approach for focusing testing activities is to use one or several product metrics, such as size or complexity metrics, to predict those parts of a system that are expected to be most defect-prone. Consequently, testing activities should focus on those parts. Several studies have shown that size has an influence on defect-proneness [15][16][17]. However, the results in those studies were not consistent, i.e., some studies showed that small code classes [15][17], respectively large code classes [16][17], tend to be more defect-prone. Consequently, Emam et al. [16] suggested not to use only size when predicting defect-proneness. Another metric frequently used to predict defect-prone code

classes and thus, to prioritize those parts for testing activities, is McCabe complexity [49]. Nagappan et al. [18] and Fenton and Ohlsson [19], for example, performed studies to investigate the correlation between McCabe complexity and the number of defects. Schröter et al. [20] concluded that new or combined metrics are necessary for appropriate predictions of defect-prone parts. Moreover, comprehensive evaluations of fault prediction models were performed and showed that the best model depends on certain criteria [46]. Besides gathering metrics from a current system under test (SUT), historical data is another source for prioritizing defect-prone parts. A number of studies has shown the relationship between different indicators, such as 'high number of performed changes' or 'historical defect content', and defect-proneness [21][22][23]. However, it often remains unclear which kind of historical data are best for prioritizing parts to be tested. Finally, risk-based approaches can support prioritization of those parts of a SUT that are expected to be most critical [27].

Another possibility for focusing testing activities is to select certain defect types to, for instance, derive test cases covering those defect types. Orthogonal Defect Classification (ODC) [7] provides so-called triggers, which represent information on the environment or condition that had to exist for defects to surface. Mapping the triggers to defect types could lead to information on where and what defect types to focus testing activities on and thus, which test cases to derive. Schultz [47] proposes an approach for using ODC to measure effectiveness and to plan and design tests to improve the testing focus. However, though defect classifications are sometimes used to focus testing activities, classified defect data from QA activities performed earlier, such as inspections, are often not used for focusing testing activities.

Output from static analysis tools (e.g., [10]) can also be used to prioritize certain parts of the SUT or certain defect types for focusing testing activities. Another approach is test case prioritization. Elbaum et al. [8][26] describe different test case prioritization techniques for finding a cost-efficient and effective set of test cases for regression testing.

Table 1: Overview of what can be prioritized for testing and which bases exist for the prioritization

Prioritization object	Prioritization basis
System parts, e.g., code classes	Product metrics or historical data or tool output or risk
Defect types	Defect classification and classified defects or tool output
Test cases	Control techniques, statement-level techniques, function-level techniques

Table 1 gives an overview of what can be prioritized for testing (prioritization object) and based on which information or approach the prioritization can be done (prioritization basis). Prioritization of system parts, defect types, or test cases is possible. However, regarding the basis of the prioritization, defect information from early QA activities, especially inspections, is rarely used or not used systematically to focus testing activities.

2.3 Defect Classifications

Several defect classifications have been developed, such as defect classifications used in experiments for comparing inspection and testing defects [24] or defect classifications developed by industry companies, such as ODC [7]. 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 [13] concludes that they do not complement each other. Different experiments show that inspections and testing activities are able to find defects of the same defect types (e.g., [37][38][39]). In contrast, Runeson and Andrews [25] showed that inspectors and testers find different kinds of defects. Some defect types might only be found by 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.

3. APPROACH

The main idea of the integrated two-stage inspection and testing approach 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 SUT and on specific defect types. The prioritization is performed in a two-stage manner. First, the inspection defect profile is used to prioritize parts of the SUT that are expected to be most defect-prone [14]. Second, the inspection defect profile is used to prioritize those defect types that are expected to appear most often during testing activities. 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, for the sake of simplicity, this assumption is made and the classification used has to ensure this assumption.

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 SUT. The same counts 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. In order to be able to rely on defined assumptions, they should at least be grounded on explicitly described hypotheses. However, 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.

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 based on the inspection defect profile. Both prioritizations are then combined, resulting in a precise focus on code classes and defect types for those code classes. Consequently, focused testing activities can be conducted. Overall, an effort reduction is expected due to testing only those parts of an SUT 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.

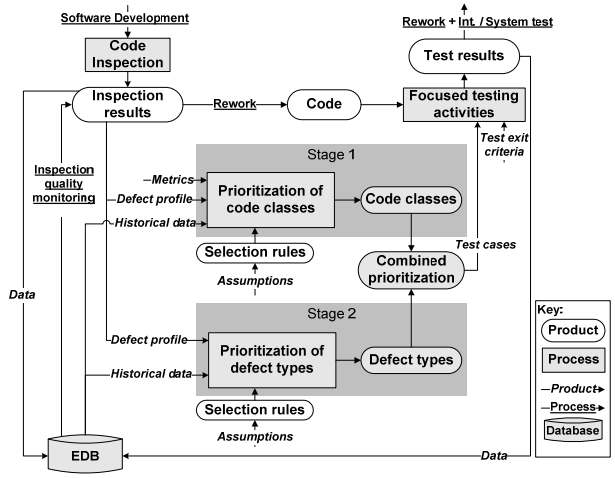

Figure 2: Integrated two-stage inspection and testing process for focusing testing activities

While these concepts give an overview of the integrated approach, Figure 2 presents its application for the coding phase. First, a code inspection (step 1) has to be performed, resulting in 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 [1]. 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 [36].

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 3 shows in detail how the prioritization is done and gives exemplary assumptions, selection rules, and the result of the combined prioritization.

Stage 1: First, a 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 [32]. 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.

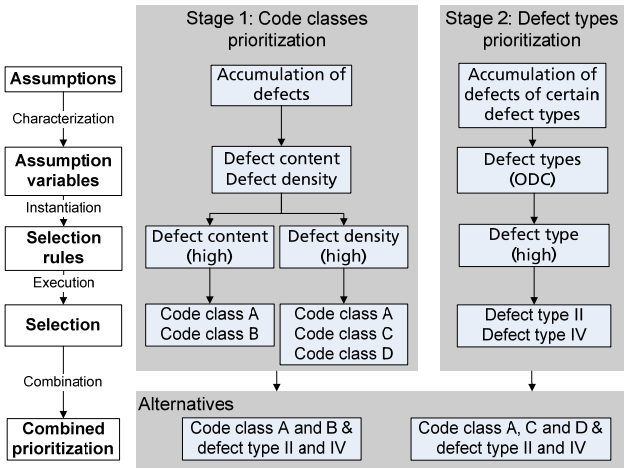

Figure 3: Application of the prioritization steps

Stage 2: Second, a 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 both by 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.

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 (which is not in the focus of this article).

Step 4 comprises focused testing activities, which also includes determining when to stop testing. However, even if prioritization may have an influence on this, test exit criteria are not in the

focus of this article. Finally, data from the code inspection and the testing activities have to be stored in the EDB for future analysis.

4. CASE STUDY

4.1 Goal of the Study

The sole prioritization of code classes was already evaluated in [14]. Thus, the main goal of the study presented here is to evaluate if the integrated two-stage 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. In order to check the achievement of this goal, the integrated approach was compared to a non-integrated approach, i.e., first, an experienced-based testing activity was performed without using the prioritization information and afterwards, the test result of the non-integrated approach was used as a baseline, which was compared to the prioritization. The goal results in the following research question:

Research Question (RQ) 1: Assuming that the defects detected during inspection are grouped with respect to their defect type: Does the integrated two-stage approach lead to a prioritization of those defect types for testing that represent the sets with the highest numbers of test defects actually found?

4.2 Context of the Study

The integrated two-stage inspection and testing approach was applied twice in the same context. The first run was primarily intended to gain experience with the approach and 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 types (i.e., additional selection rules for stage 2 were defined).

A Java tool prototype was developed to support practitioners in performing a sequence-based specification (SBS). Currently, the source code comprises about 8,500 lines of code (LoC), structured into 76 classes and over 650 methods. 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.

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.

For focusing testing activities on code classes that are expected to be most defect-prone, results already described in detail in [14] are reused. Only one assumption and one derived selection rule is taken into account for stage 1 in this case study.

4.3 Design of the Study

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 all collected quality properties. Some checklist questions were similar to those mentioned in [33]. 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 in [34][35]. 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 two-stage 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 experienced-based testing activity including equivalence partitioning was conducted without using any information from the prioritization step. The defect results from the experienced-based testing activity were used as a baseline and compared with the prioritization (i.e., step 3) of the integrated two-stage approach in order to answer RQ1.

The following variables were taken into account in the case study: Effort was measured in minutes, and size of code classes was measured in LoC. A severity classification was used (minor, major, crash). Furthermore, ODC [7] was chosen for defect classification due to its high industry orientation. Experiences from industry show that ODC promises to be a suitable classification [48]. 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, are taken into account (for an explanation and examples of each defect type, see [7]): *Assignment / Initialization*; *Checking*; *Algorithm / Method*; *Function / Class / Object*; *Timing / Serialization*; *Interface / O-O Messages*; *Relationship*.

4.4 Execution of the Study

The integrated two-stage inspection and testing approach was applied twice to answer RQ1. It builds upon some results of the

one-stage evaluation to focus the testing activity on code classes, as described in [14]. Thus, the prioritization of code classes is summarized briefly (stage 1), while the prioritization of defect types is described in more detail (stage 2).

4.4.1 First Run

Step 1: Conducting the inspection – Before the inspection was performed, all inspectors and one tool developer met to get familiar 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 was performed 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 2 (defect content) and Table 3 (sorted list of ODC-classified defects). 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 2: Inspection defect profile – Defect content

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

Table 3: 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**

Step 2: Monitoring the inspection results – Unfortunately, no context-specific historical inspection data were available for the tool prototype in order to monitor the inspection results. Instead, 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 [36], but consistent with experiences from industry [50]. 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. Finally, though this was just a gut feeling of the inspection experts, 67 issues found per thousand LoC seemed to be appropriate.

Step 3: Two-stage prioritization – One assumption was considered at the first stage:

> S(tage)1-A(assumption)1: Parts of the code where a large number of inspection defects are found (i.e., an accumulation of defects is observed) indicate more defects to be found with testing.

This assumption is grounded on the empirically validated hypothesis that an accumulation of defects can often be observed, rather than an equal distribution of defects. Elberzhager et al. [14] mention a number of studies and experiments that confirm this assumption. 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.

For the second stage, one assumption was also used:

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

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 a concrete defect classification. Several defect classifications used by Mantyla and Lassenius [37] 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 [13][38]. The same observation is presented in [39], where ODC is used. Finally, Ohlsson et al. [40] state that the majority of quality costs are often caused by very few defect types. 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 assumption 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, resulting in one combined prioritization.

Step 4: Conducting the testing activity – 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 experienced-based testing activity including equivalence partitioning were used as a baseline for analyzing and evaluating the prioritization of the integrated two-stage approach.

4.4.2 Second Run

Step 1: Conducting the inspection – 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, recording a short description of the problem, the place where it was found, the severity, and the ODC defect type. The inspection

(i.e., the defect detection task) took about three to four hours per inspector, resulting in an overall effort of 835 minutes.

Afterwards, two experienced quality assurance engineers examined all issues and eliminated duplicates, questions, and improvement suggestions. In total, 100 defects remained. Next, the classification of each defect was checked. When 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 4 (defect content) and Table 5 (sorted list of ODC-classified defects). With respect to the ODC, 54 defects could be classified, while 46 defects were treated as *Other* defects.

Table 4: Inspection defect profile – Defect content

Code class	*Sequence*	*SequenceTable Model*	*SimpleKeyedTa bleModel*	*SimpleOrdered KeyedT.*	Sum
Defect content	**14**	**40**	**39**	**7**	**100**

Table 5: Inspection defect profile – 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**

Step 2: Monitoring the inspection results – Since the inspection results from the first run were available, the current inspection results could be compared with historical inspection results, as suggested by Aurum et al. [1]. Comparing the inspection data is justified by the same context and is more meaningful than using data from different environments. 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. 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.

Step 3: Two-stage prioritization – For stage 1 prioritization, the same assumption and the same derived selection rule were used that were already applied in the first run (i.e., prioritization of the code classes with the highest defect content). For a more fine-grained analysis of the defect type prioritization, two additional selection rules were derived for stage 2:

S2-A1-SR1: <*See first run*>

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 that appeared most often both in past inspection and in past testing activities.

The output of applying the 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 4 shows the concrete prioritizations of code classes (based on Table 4) and defect types (based on Table 5 and Table 6) for each of those combined prioritizations.

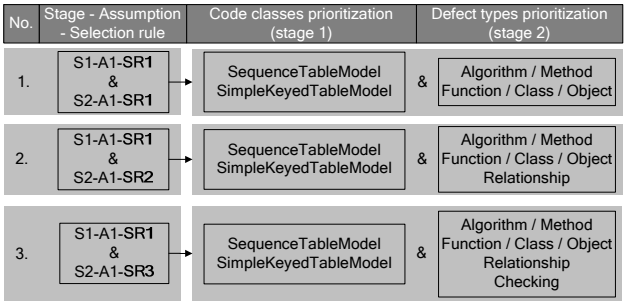

Figure 4: Combined prioritization of code classes and defect types based on applied selection rules

Step 4: Conducting the testing activity – Similar to the first run, one developer of the tool prototype performed the experienced-based testing activity without prioritization information. Besides the four inspected code classes, four additional, highly connected code classes were tested.

4.5 Results of the Study and Lessons Learned

4.5.1 First Run

Seven additional defects were found during the non-integrated testing activity (i.e., defects not already found by the inspection), with three defects being found in *SBSTreeState* and four defects being found in *SBSTree*. This exactly matches the stage 1 prioritization (see Table 2).

Table 6: Defect types found by inspection and testing, and prioritized defect types for selection rule of stage 2 (1st run)

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	

Moreover, Table 6 shows the defect types that appeared during inspection and testing activities in order to compare them. 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 (**RQ1**), and that no defect types were missed.

4.5.2 Second Run

In the second run, six additional defects were found during the non-integrated testing activity (i.e., defects not already found by inspection). Five test defects were found in *SimpleKeyedTableModel* and one test defect in *TableUtil*, which is invoked by that code class. The selection rule for the stage 1 prioritization did not prioritize *SimpleKeyedTableModel* only, but this code class was at least selected among others. This means that with the stage 1 prioritization, no test defect is missed, though the prioritization is not the most efficient one due to prioritized code classes that were not defect-prone.

Table 7 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. Three selection rules were applied that prioritized different sets of defect types. 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 SR1. The second selection rule also prioritized the defect type *Relationship*, from 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 already appeared. Consequently, only SR3 identified all defects of prioritized defect types.

Table 7: Defect types found by inspection and testing, and prioritized defect types per selection rule of stage 2 (2nd run)

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			

Thus, 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 an effort reduction (though not measured explicitly) is achievable.

4.5.3 Lessons Learned

The main result with respect to **RQ1** is that, in our context, the integrated two-stage inspection and testing approach was able to prioritize defect types for the testing activity based on the inspection defect profile. In the case study, the defect types identified with the inspection and testing activity were similar with respect to ODC, whereas defects classified as *Other* could not contribute to the prioritization. However, the quality of the prioritization depends on the concrete assumptions and selection rules, which have to be evaluated carefully in a given environment. In the first run, all defect types could be prioritized based on the assumptions and selection rules used, while in the second run, only the most comprehensive selection rule three prioritized all three defect types that actually appeared during testing. Defect data gathered in the first run of the case study could be reused as historical data in the second run and improved the selection of defect types (see selection rule S2-A1-SR3). One main goal of the integrated approach is to reduce effort. An effort reduction of between 6 and 34% has been shown for the stage 1 prioritization [14], depending on different assumptions and derived selection rules. Even if no concrete effort reduction data was gathered when applying the integrated two-stage approach, it is assumed that effort reduction still increases due to the prioritization of certain defect types. Beside effort reduction, an alternative goal of the integrated approach might be to improve effectiveness (i.e., to find more defects) without increasing effort. Finally, the approach could be used to control testing activities by comparing expected test results with actual test results.

What is important from an industry point of view is the inspection quality monitoring and the requirement of having found a suitable number of inspection defects to enable appropriate prioritizations.

4.6 Threats to Validity

Next, a discussion of what we consider to be the most relevant threats to validity is given [41].

Conclusion validity: No statistically significant results could be obtained due to the low number of testers and found test defects. However, the initial results showed that the two-stage approach is able to prioritize those defect types that appeared most often during testing.

Construct validity: To demonstrate the integrated two-stage 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 defect types might have been prioritized. In the inspection, no standard checklists were used. However, some checklist questions were taken from the literature, and these can be considered as standard questions. Finally, the selection of ODC may have influenced the prioritization of defect types.

Internal validity: The subjects selected for the inspection and 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 for the testing activity. Finally, defects could be classified differently.

External validity: The prototype tool, which is rather small, can be considered as an initial example for which the integrated two-stage 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. Larger software, as

typically developed by software companies, is expected to result in more test defects to be found and classified during testing activities. Thus, 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. SUMMARY AND OUTLOOK

Facing the challenge of reducing quality assurance effort, an integrated two-stage inspection and testing process was presented that exploits synergy effects between static and dynamic quality assurance processes in order to focus testing activities. In particular, defect results from an inspection are used in a two-stage manner: first, to prioritize parts of the system that are expected to be most defect-prone (stage 1) and second, to prioritize defect types that are expected to appear most often (stage 2) during testing activities. A case study was conducted where the approach was applied on the code level. The results of two runs show that code classes could be prioritized where further defects were found with a testing activity, and defect types could be prioritized that showed up most often during the testing activity. This means that the integrated approach was able to select those parts that are actually defect-prone as well as corresponding defect types for testing based on the inspection results, where the prioritization highly depends on the defined assumptions and derived selection rules. Hence, these must be evaluated in each new environment and adapted, if necessary.

From our point of view, using inspection results is not the only appropriate predictor of defect-prone parts or defect types, but it can give valuable support for, e.g., 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. To the best knowledge of the authors, this is the first approach that combines inspection and testing processes in a systematic and integrated way to focus testing activities and thus, to exploit synergy effects such as increased efficiency or effectiveness.

With respect to future work, further development of the two-stage process is planned. Especially a procedure for deriving test cases based on prioritized defect types has to be defined. Furthermore, instructions for how to control testing activities based on the prioritizations have to be determined. With respect to the application of the process, one further idea is to use results from requirements or design inspections for the prioritization. Furthermore, instead of omitting entire code classes or 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. Moreover, a systematic procedure should be defined that describes how to determine a set of more comprehensive assumptions and corresponding selection rules in order to prioritize code classes and defect types. Finally, additional evaluations of the integrated approach are planned in different environments, including an analysis of which defect types are found with inspection and testing techniques in order to improve the prioritization.

6. ACKNOWLEDGMENTS

This work has been funded by the Stiftung Rheinland-Pfalz für Innovation project “Qualitäts-KIT” (grant: 925). We would like to thank Sonnhild Namingha for proofreading.

7. REFERENCES

[1] Aurum, A., Petersson, H., Wohlin, C., State-of-the-art: software inspections after 25 years, *Software Testing, Verification and Reliability*, *12* (3), 133-154, 2002

[2] Juristo, N., Moreno, A.M., Vegas, S., Reviewing 25 years of testing technique experiments, *Empirical Software Engineering*, *9* (1-2), 7-44, 2004

[3] Harrold, M.J., Testing: A Roadmap, *International Conference on Software Engineering*, *The Future of Software Engineering*, 61-72, 2000

[4] Health, Social, and Economic Research, The economic impacts of inadequate infrastructure for software testing, *National Institute of Standards and Technology*, 2002

[5] Basili, V.R., Briand, L.C., Melo, W.L., A validation of object-oriented design metrics as quality indicators, *IEEE Transactions on Software Engineering*, 751-761, 1996

[6] Illes-Seifert, T., Paech, B., Exploring the relationship of a file’s history and its fault-proneness: An empirical method and its application to open source programs, *Information and Software Technology*, *52* (5), 539-558, 2009

[7] Orthogonal Defect Classification, IBM, available: http://www.research.ibm.com/softeng/ODC/ODC.HTM

[8] Elbaum, S., Rothermel, G., Kanduri, S., and Malishevsky, A. G., Selecting a cost-effective test case prioritization technique, *Software Quality Control*, *12* (3), 185-210, 2004

[9] Strooper, P., Wojcicki, M.A., Selecting V&V technology combinations: how to pick a winner? *12th IEEE International Conference on Engineering Complex Computer Systems*, 87-96, 2007

[10] FindBugs, http://findbugs.sourceforge.net/

[11] Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand., J., Combining static analysis and test generation for C program debugging, *International Conference on Tests and Proofs*, 94-100, 2010

[12] Endres, A., Rombach, D., *A Handbook of Software and Systems Engineering*, Addison Wesley, 2003

[13] Laitenberger, O., Studying the effects of code inspections and structural testing on software quality, *9th International Symposium on Software Reliability Eng.*, 237-246, 1998

[14] Elberzhager, F., Eschbach, R., Muench, J., Using inspection results for prioritizing inspection data, *21st International Symposium on Software Reliability Engineering*, Supplemental Proceedings, 263-272, 2010, available: http://inspection.iese.fhg.de/?p=documents

[15] Ostrand, T.J., Weyuker, E.J., The distribution of faults in a large industrial software system, *International Symposium on Software Testing and Analysis*, 55-64, 2002

[16] Emam, K.E., Benlarbi, S., Goel, N., Mela, W., Lounis, H., Rai, S.N., The optimal class size for object oriented software, *IEEE Trans. Softw. Eng., 28* (5), 494-509, 2002

[17] Anderson, C., Runeson, P., A replicated quantitative analysis of fault distributions in complex software systems, *IEEE Trans. Softw. Eng., 33* (5), 273-286, 2007

[18] Nagappan, N., Ball, T., Zeller, A., Mining metrics to predict component failures, *International Conference on Software Engineering*, 452-461, 2006

[19] Fenton, N.E., Ohlsson, N., Quantitative analysis of faults and failures in a complex software System, *IEEE Trans. Softw. Eng., 26* (8), 797-814, 2000

[20] Schröter, A., Zimmermann, T., Premraj, R., Zeller, A., If your bug database could talk, *5th International Symposium on Empirical Software Engineering*, 18-20, 2006

[21] Illes-Seifert, T., Paech, B., Exploring the relationship of a file's history and its fault-proneness: An empirical method and its application to open source programs, *Information and Software Technology*, *52* (5), 539-558, 2009

[22] Nagappan, N., Ball, T., Murphy, B., Using historical in-process and product metrics for early estimation of software failures, *17th International Symposium on Software Reliability Engineering*, 62-74, 2006

[23] Ostrand, T.J, Weyuker, E.J, Bell, R.M, We're finding most of the bugs, but what are we missing? *3rd International Conference on Software Testing, Verification and Validation*, 313-322, 2010

[24] Basili, V.R., Selby, R.W., Comparing the effectiveness of software testing strategies, *IEEE Transactions on Software Engineering, 13* (12), 1278-1296, 1987

[25] Runeson, P., Andrews, A., Detection or isolation of defects? An experimental comparison of unit testing and code inspection, *14th International Symposium on Software Reliability Engineering*, 3-13, 2003

[26] Elbaum, S., Malishevsky, A.G., Rothermel, G., Test case prioritization: a family of empirical studies, *IEEE Transactions on Software Eng.*, 28 (2), 159-182, 2002

[27] Karolak, D.W., *Software Engineering Risk Management*, IEEE Computer Society Press, Wiley, 1996

[28] Liu, S., Tamai, T., Nakajima, S., Integration of formal specification, review, and testing for software component quality assurance. *ACM Symposium on Applied Computing*, 415-421, 2009

[29] Runeson, P., Andersson, C., Thelin, T., Andrews, A., Berling, T., What do we know about defect detection methods? *IEEE Software, 23* (3), 82-90, 2006

[30] Winkler, D., Biffl, S., Fader, K., Investigating the temporal behavior of defect detection in software inspection and inspection-based testing, *11th International Conference on Product Focused Software Development and Process Improvement*, 17-31, 2010

[31] Harding, J.T., Using inspection data to forecast test defects, *Software Technology Transition*, 19-24, 1998

[32] Boehm, B., Basili, V.R., Software defect reduction top 10 list, *IEEE Computer, 34* (1), 135-137, 2001

[33] Burnstein, I., *Practical Software Testing*, Springer, 2002

[34] Wiegers, K.E., *Peer Reviews in Software*, A.-Wesley, 2002

[35] Gilb, T., Graham, D., *Software Inspections*, A.-Wesley, 1993

[36] Barnard, J., Price, A., Managing code inspection information, *IEEE Software, 11* (2), 59-69, 1994

[37] Mantyla, M.V., Lassenius, C., What types of defects are really discovered in code reviews? *IEEE Transactions on Software Engineering, 35* (3), 430-448, 2009

[38] Kamsties, E., Lott, C.M., An empirical evaluation of three defect detection techniques, *5th European Software Engineering Conference*, 362-383, 1995

[39] Chaar, J.K., Halling, M.J., Bhandari, I.S., Chillarege, R., In-process evaluation for software inspection and test, *IEEE Transactions on Software Eng., 19* (11), 1055-1070, 1993

[40] Ohlsson, N., Helander, M., Wohlin, C., Quality improvement by identification of fault-prone modules using software design metrics, *6th International Conference on Software Engineering*, 1-13, 1996

[41] Wohlin, C., Runeson, P., Host, M., Ohlsson, M.C., Regnell, B., Wesslen, A., *Experimentation in software engineering an introduction*, Kluwer, 2000

[42] Aggarwal, A., Jalote, P., Integrating static and dynamic analysis for detecting vulnerabilities, *30th Annual International Conference on Computer Software and Applications*, 343-350, 2006

[43] Wood, M., Roper, M., Brooks, A., Miller, J., Comparing and combining software defect detection techniques - a replicated empirical study, *6th European Software Engineering Conference*, 262-277, 1997

[44] Gack, G.A., 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

[45] Freimut, B., Denger, C., Ketterer, M., An industrial case study of implementing and validating defect classification for process improvement and quality management *11th IEEE International Software Metrics Symp.*, 19-28, 2005

[46] Arisholm, E., Briand, L.C., Johannesson, E.B., A systematic and comprehensive investigation of methods to build and evaluate fault prediction models, *Journal of Systems and Software, 83* (1), 2-17, 2009

[47] Schultz, C., Orthogonal defect classification-based test planning and development, *International Conference on Applications of Software Measurement*, 15-19, 1999

[48] Bridge, N., Miller, C., Orthogonal defect classification using defect data to improve software development, *Int. Conference on Software Quality*, 197-213, 1997

[49] McCabe, T.J., A complexity measure, *IEEE Transactions on Software Engineering, 2* (4), 308-320, 1976

[50] Cohen, J., *Best Kept Secrets of Peer Code Review:* Code reviews at Cisco Systems, 63-87, 2006