Publication V

Juha Itkonen, Mika V. Mäntylä, and Casper Lassenius. The role of knowledge in failure detection during exploratory software testing. IEEE Transactions on Software Engineering, May 2011, 17 pages, submitted for publication.

The Role of Knowledge in Failure Detection During Exploratory Software Testing

Juha Itkonen, *Member, IEEE,* Mika V. Mäntylä, *Member, IEEE,* and Casper Lassenius, *Member, IEEE*

Abstract—We present a field study on how testers use knowledge to detect failures while performing exploratory software testing in industrial settings. We video recorded 12 testing sessions in four industrial organizations, having our subjects think aloud while performing their usual exploratory testing work. Using applied grounded theory, we studied how the subjects detected failures, and what type of knowledge they utilized. The results show how testers recognize failures based on their personal knowledge without detailed test case descriptions. The knowledge is classified under categories of domain knowledge such as users needs and goals, system knowledge such as interactions of many features and the workings of the system as a whole, and general software engineering knowledge. We found that testers applied their knowledge either as a test oracle to determine whether a result was correct or not, or as a test wizard, guiding them in selecting objects for test and designing tests. Additionally, we studied the knowledge in relation to defect detection difficulty and visible failure symptoms. We conclude that exploratory testing can be an efficient method for utilizing domain experts in testing as a high proportion of failures detected based on domain knowledge were straightforward to reveal.

Index Terms—Software testing, exploratory testing, validation, test execution, test design, human factors, methods for SQA and V&V

1 Introduction

Software testing is traditionally considered a process of executing test cases, which are carefully designed using test case design techniques [1]–[4]. Test case design techniques aim at ensuring systematic coverage, detection of typical error types, and reduction of redundant testing [1], [2], [5]. The test-case based testing paradigm (TCBT) assumes that actual test execution, even if performed as a manual activity, is a more or less mechanical task. During execution, the pre-defined test cases are run, and their output compared to the documented expected results. However, studies on industrial practice report that real-world testing seldom is based on rigorous, systematic, and thoroughly documented test cases [6]–[8].

Although test automation has been the focus of a lot of research, manual testing is still widely utilized and appreciated in the software industry, and is unlikely to be replaced by automated testing in the foreseeable future [6], [9]–[12]. In many software development contexts, the manual testing effort of professional testers and application domain experts is crucial for assuring that products fulfil the needs of the users or please the markets. In this context exploratory software testing (ET) has been proposed as an effective testing approach.

Exploratory testing differs significantly from traditional software testing in that it is not based on predesigned test cases. Instead, it is a creative, experience-based approach in which test design, execution, and learning are parallel activities, and the results of executed tests are immediately applied for designing further tests [13]. Exploratory testing is a recognized testing approach [14], but has commonly been referred to as ad-hoc testing or error-guessing [1], [2], [14]. Practitioners, however, recognize that exploratory aspects are fundamental to most manual testing activities [10], [15]–[17]. There are a growing number of practitioner reports and studies on the benefits of exploratory testing [13], [18]–[21]. In these reports, ET is commonly described in the context of system-level testing of interactive systems through GUI and from the end user's point of view.

In the studies of manual testing and in particular ET context, the experience and especially application domain knowledge of the testers have been recognized as important aspects that affect the tester's behaviour and the results [12], [16], [22], [23].

In this paper, we use the term *knowledge* to refer to the tester's *personal* knowledge in a rather wide meaning. Using the terminology of Robillard [24], we include both topic, i.e., meaning of words, and episodic, i.e., experience with knowledge, types of knowledge, and, to some extent, tacit knowledge.

Knowledge can be applied to different exploratory testing tasks and purposes. First, knowledge can be used as information to guide exploratory test design. Second, knowledge can be used to identify failures, i.e., as an oracle to distinguish between a correct, expected outcome and an incorrect, defective, outcome [14]. Third, knowledge, together with the observed actual behaviour of the tested system, can be used to create new, better tests during exploratory testing.

The oracle problem is a hard challenge in test automation (see, e.g., [14], [25]), and the problem of availability or existence of an oracle sometimes fundamentally restricts testing [26]. In TCBT, the oracle problem is sometimes bypassed, since each test case is supposed to

Authors are with the Department of Computer Science and Engineering, Aalto University School of Science, P.O. Box 19210, 00076 Aalto, Finland. E-mail: {juha.itkonen, mika.mantyla, casper.lassenius}@aalto.fi

include the expected outcome. In practice, even in TCBT, it is often not possible to pre-define the expected results in such detail and comprehensiveness that determining the correct or defective behaviour would be a matter of simple comparison for the tester. Instead, testers are typically forced to struggle with the oracle problem continuously during testing activities, and it seems that the experience-based approach is an important complementary part of testing even in contexts in which rigorous and systematically documented testing is required.

In this paper, we present a study in which we examined how testers detect software failures in real-world exploratory testing work. This paper provides a detailed analysis of the actual failure incidents during industrial software testing work in four software development organizations. We focus on the types of knowledge that testers apply when detecting failures and how they apply this knowledge. In addition, we analyse the types of failures that are detected based on the different knowledge types.

Recently, researchers in the empirical SE research community have called for research on "testing as it is carried out in real-life circumstances" and studies that focus on how testing unfolds in industrial practice instead of trying to demonstrate superiority of methods or technologies [12], [27], [28]. With this paper, we contribute to that line of research by reporting on how software development professionals recognize software failures in practice, and the role of personal knowledge in failure detection.

In the next section, the related work is reviewed. The research goals and questions as well as a presentation of the research methods are covered in Section 3. Results are presented in Section 4. A discussion with answers to research questions and limitations of this study are presented in Section 5, followed by the conclusions in the last section.

2 Related work

The context of this study is the exploratory software testing approach, and our research question focuses on how testers identify failures when performing exploratory testing, as well as on what type of defects they detect. Thus, we begin the review of related work with the exploratory testing literature followed by the role of experience and knowledge in software testing in general, and, finally, we cover relevant literature on the test oracle problem and failure classifications.

2.1 Exploratory Software Testing

Exploratory testing (ET) is an experience-based testing approach that differs fundamentally from the highly document-driven TCBT approach. Exploratory testing can be defined as: "simultaneous learning, test design, and test execution; that is, the tests are not defined in advance in an established test plan, but are dynamically designed, executed, and modified" [14].

The term "exploratory testing" was introduced by Kaner et al. [3]. The ET approach has been acknowledged in software testing books since the 1970's [1], but mostly referred to as an ad-hoc approach or error-guessing without any concrete description of how to perform it. Only a few scientific papers on exploratory testing have been published, but it has been well covered in some specialized books, e.g. [10], [17], and discussed briefly in generic testing books, e.g. [4], [29]–[31]. James Bach described exploratory testing in more detail [13]. Tinkham and Kaner [15] covered the need of questioning skills and the heuristic nature of ET.

Practitioner reports on exploratory testing claim ET is both effective in detecting defects and cost-efficient [13], [18]–[20], [32]. These reports, however, are only personal experience reports without any scientific basis. Practitioner literature describes how to manage ET using session-based test management [19], [32], and the tour-based approach [17].

Some scientific case studies and experiments studying the effectiveness and efficiency of ET have been published during the last decade. Itkonen and Rautiainen [16] studied the perceived benefits of the ET approach in three organizations. Houdek et al. [33] studied defect detection effectiveness in the executable specification context, comparing systematic testing and experience-based ad-hoc simulation, and Itkonen et al. [34] performed a student experiment comparing the defect detection effectiveness of ET and TCBT. In an experiment, do Nascimento and Machado [35] compared the exploratory and model-based testing approaches to feature testing in the mobile phone applications domain. In the results of these studies, exploratory testing was found to be as effective as TCBT [33]–[35], and to require less effort than the testing approaches it was compared to [33], [35]. While few in number, these studies support the hypothesis that ET could be an effective and efficient testing approach in certain contexts.

Studies on how exploratory testing is applied in practice in software development organizations also exist. Itkonen and Rautiainen [16] reported how three organizations applied exploratory testing, including related motivations, benefits, and challenges. In another study Itkonen et al. [36] describe empirically observed ET practices. Pichler and Ramler [37] applied ET for testing a highly interactive GUI editor and developed software tools to support exploratory GUI testing. Researchers have also determined that the ET approach seems to be a good match with agile development processes [38] and, e.g., Tuomikoski and Tervonen [21] describe positive experiences of using team exploratory testing sessions as part of the agile Scrum development process. Martin et al. [27] give a detailed description of a "systems integration testing" approach that is highly exploratory in their ethnography of testing in a small agile company, while Kasurinen et al. [39] observed exploratory testing as part of a more generic risk-based approach to testing. These few empirical studies give some examples of how

ET is applied in industrial practice.

Other studied aspects of ET include the effect of individual characteristics on ET, including the effect of learning style on testing [40], and the effect of personality traits on exploratory testing performance, showing that extrovert personalities might be more likely to excel in exploratory testing [41].

In conclusion, exploratory testing has been promoted in the practitioner literature, and scientific studies of ET are emerging. Studies comparing ET with other testing approaches support the effectiveness and efficiency of the ET approach. Empirical studies of industrial practice propose the applicability of the ET approach in the context of system-level functional testing from the user's viewpoint, highly interactive GUI testing, and enhancing the testing practices of agile development. To our knowledge there are no empirical studies on the details of the actual exploratory testing practices and activities, in addition to our previous work [36].

2.2 Experience and knowledge in software testing

Experience and knowledge have been studied in the contexts of software engineering [42] and software design [43], [44]. Sandberg [45] studied human competence at work in a different engineering field. Practitioners have reported the benefits of experience-based testing approaches, e.g., in the financial [46] and medical [20], [47] domains, but these reports are not based on scientific research. Scientific research on the role of experience and knowledge in the software testing context is still rare, but some studies have been published that focus specifically on the subject [12], [22], [23], [48]. In these studies, the term *experience* was defined as "practical knowledge that is developed in direct observation or participation in activities" [12], or as the amount of professional experience [22]. Next, we describe these studies in more detail.

Beer and Ramler [12] studied the role of experience in the development of test cases, regression testing, and test automation. They found in their case studies that domain knowledge, in addition to testing knowledge, is crucial in testing. They describe the typical knowledge development path of senior testers, which started with strong domain knowledge. Testing experience was later gained through working in testing, seminars, and working with external consultants. They conclude that "test design is to a considerable extent based on experience and experience-based testing is an important supplementary approach to requirements-based testing" [12]. In all three cases, test cases were designed before the actual testing, which means that their research did not give any insight about the exploratory testing approach.

In a survey on the effect of experience and individual differences in software testing, Merkel and Kanij [48] found that testing practitioners consider both testing and domain experience important factors affecting performance. Testing-specific training or certification was not considered important, but the respondents considered the individual traits of testers highly influential on tester performance. Kettunen et al. [23] also report domain knowledge as the most emphasized area of testers' expertise and highlight the role of technical knowledge, particularly in the agile development context.

Poon et al. [22] experimentally compared the types and amounts of mistakes inexperienced and experienced testers make in test case identification. They found large variations among individual subjects, especially in the case of inexperienced ones. Experienced subjects identified more test categories and made fewer mistakes; in particular, the number of missing categories in complex cases was considerably lower for experienced subjects. However, experienced testers made more of certain kinds of mistakes. In addition, the contribution of experience to performance decreases when the complexity of the tested functionality increases. Using checklists reduced the number of missing categories and all types of mistakes. Poon et al. [22] conclude that software development experience cannot replace the need for a systematic methodology and suggest involving testers with varying experience levels in industrial settings.

The effect of domain knowledge on defect identification has been identified in the software testing, spreadsheet error finding, and usability inspection contexts. Multiple studies on software testing report findings on the importance of domain knowledge in testing [12], [16], [23], [48]. In the context of usability testing, Følstad [49] studied work-domain experts as usability evaluators and found that the findings of work-domain experts were classified as more severe, and given higher priority by developers [49]. Galletta et al. [50] studied error-finding performance in the spreadsheet context. They compared error-finding performance of domain area (accounting) experts versus novices, and spreadsheet (software) experts versus novices. They found that both types of expertise increased error-finding performance, but the performance of those with both types of expertise far exceeded the performance of other groups. Spreadsheet expertise increased the speed of revealing spreadsheet-related defects.

Existing research on the role of knowledge and experience in software testing and defect identification in general raises the hypothesis that experience has an important effect on defect identification performance and that domain knowledge is more important than testing experience. The existing studies in the software testing context are based on interviews in case studies and surveys, and the results do not provide insight into how testers actually work and apply their knowledge. The types of knowledge used and how they are applied in defect detection when performing ET remains an unstudied area.

2.3 The oracle problem

A *test oracle* is a concept referring to a method used to distinguish between a correct and incorrect result during

software testing [2], [14], [51], [52]. Defect recognition is one of the most crucial activities in testing, and the existence of a test oracle is recognized as a fundamental requirement in all kinds of testing [25], [51]–[53]. The challenge of finding a reliable oracle is referred to as "the oracle problem". The oracle problem is particularly investigated in the context of test automation striving for automated test oracles [14], [52]. However, the same problem is associated with all software testing. TCBT paradigm aims at solving the oracle problem by predefining the expected result in detail: "In real testing, the outcome is predicted and documented before the test is run" [2].

In practice, however, requirements, specifications, and thus test cases are seldom perfect in terms of comprehensiveness and accuracy. Weyuker [26] identified different types of non-testable programs for which an oracle does not exist or the correct result would be too difficult to determine. Nevertheless, in practice, testers are able to detect incorrect results using partial oracles even if they cannot know the exact correct result [26]. From the empirical research on real-world testing activities, it seems that a human oracle is in many cases the way test results are evaluated in practice (see, e.g., the ethnographic descriptions in [27], [28]). In industrial practice, even in many test automation approaches the oracle problem is left as a human decision [52], [53]. The oracle problem is highly relevant in manual testing, and typically solved using the personal knowledge of testers and varying types of documentation. The challenge of using human oracle is oracle mistakes, i.e., testers do not always recognize a defect even if a test case reveals it. In an experiment by Basili and Selby [54], subjects recognized only 70% of observable failures.

In the context of exploratory testing, consistency heuristics have been proposed to work as experience-based oracles [10], [55]. These are a set of rules for checking for the consistency of functionality against various targets, such as the history of the product, comparable products, and users' expectations. The core idea in ET is that the tester can and should use any available sources of information in the testing [10], [13], which suggests that the oracle in ET can be any knowledge, documentation, model, or software available to the tester.

To our knowledge, the oracle problem has not been studied in the context of manual testing with the intent of understanding, describing, or improving the way humans recognize defects. The literature on exploratory testing has presented hypotheses that an experience-based approach to defect detection would be effective. Earlier research on exploratory and experience-based testing indicates that in practice, testers use an experience-based approach in detecting defects and determining the correct result of tests [36].

2.4 Failure type classifications

In this study, we focused to analysing *failure* incidents that exploratory testers detected. Understanding failure types from the perspective of testing focuses on the externally visible symptoms of the failures. Few published failure type classifications exist, and commonly cited classifications usually classify the actual faults based on technical, often source code level characteristics. Such technical *fault* classifications as, e.g., orthogonal defect classification [56], were not suitable for this study. In this study, our target was to study the types of failures based on how a tester, or end user, perceives the symptoms of the failure when it occurs. Some failure classifications classify failures with respect to detection [57], [58], but for example in the classification of Bondavalli and Simoncini [57], all user observable failures are classified in a single class. Cotroneo et al. classified Java virtual machine failures, but their classification only focuses on technical crashes, deadlocks, and error messages and does not describe the failure symptoms of functional behaviour [58].

Some failure classifications that characterize the symptoms, however, can be found. One such classification was used in a study of medical device failures [59]. In this study, Wallace and Kuhn analysed software-related failures of medical devices that led to recalls by the manufacturers and presented a thirteen-class classification of the failure symptoms. The classification is somewhat specific to the medical device domain and not descriptive of the actual symptoms. The classes are also rather generic and briefly described (e.g., "System: the total system").

Classifying failure incidents from the tester's viewpoint is a problem similar to classifying usability problems from the user's viewpoint. In both cases, a classification should capture the externally visible symptoms of something that is missing or wrong with the software system. An example of failure classification in the usability testing context is the Classification of Usability Problems (CUP) scheme [60]. The usability problem classification, even though similar to the tester's failure classification, includes such usability-specific classes as "incongruent mental model" and "test participant overlooked something" that are not applicable in our context. One generic and simple classification dimension that has been used in testing technique experiments [54], [61] and is visible as part of usability problem [60] and defect classifications [62] is the omission versus commission dichotomy, which simply separates failures based on whether something is missing or wrong.

We also wanted to analyse the difficulty of detecting the failures. One characteristic of failure that is directly related to the detection difficulty is the amount of interacting conditions that together cause the failure. This characteristic can be analysed by using the failure-triggering fault interaction (FTFI) number [63]. The FTFI number refers to the number of interacting variables or conditions that together cause a certain failure to occur. In 1-way faults, only one condition triggers the failure, while 2-way faults would need two interacting conditions to occur together to trigger a failure.

3 RESEARCH GOALS AND METHODS

The research presented in this paper is a field observation study in which the testing practices of eight software development professionals in four software development organizations were studied. Our research methodology consists of participant observations and qualitative data analysis [64]. We also present quantitative summaries of the detected failures in the observed sessions. Next, we describe the research objectives and questions as well as the methodology employed.

Our research approach was both exploratory and descriptive. In the area of exploratory testing, existing research does not provide a basis for strong theoretical hypotheses. Instead, the objective of this study was to understand the phenomenon of defect identification in exploratory software testing. We aimed at describing how testers detect failures as they occur in exploratory testing and at drawing hypotheses grounded on our observational data. We studied the types of knowledge involved as well as characterized the observed failure types. The high-level research objective was to *understand how defects are detected by testers performing exploratory software testing, and the role of knowledge in it*. The research questions were:

- RQ1: *What types of knowledge do testers utilize for detecting defects when performing exploratory testing?*
- RQ2: *How do testers apply their knowledge for detecting defects when performing exploratory testing?*
- RQ3: *What types of failures do testers identify using knowledge in exploratory testing?*

3.1 Data collection methods

We collected the data using participant observation [64] because we wanted to understand the actual testing tasks the subjects performed in their natural working environments. In direct observation, "the inquirer has the opportunity to see things that may routinely escape awareness among the people in the setting" [65]. Because participant observation can mean many things depending on the source (see, e.g., [65]–[67]), we describe our approach using the six dimensions of fieldwork variations presented by Patton [65]: 1) The *role of the observer* was onlooker. The observer sat beside the subject for the entire testing session and did not participate in the actual testing activities. Even though the observer tried to be as inconspicuous as possible, the subjects communicated directly with the observer during the sessions. 2) *The perspective of the observer* was outsider dominant. The observer was not part of the organization nor involved in the product development. The observer was familiar with the organization and the tested software products through existing long research cooperation. 3) *The observation was conducted* by a single researcher. 4) *The observer and his role* were fully disclosed to the subjects. The observer was clearly present in the testing situation, and the observed subject was strongly conscious of his presence. 5) *The duration of the observation* was one or two 1–2.5 hour observation sessions per subject. 6) *The focus of the observations* was on individual test execution tasks of single testers in the context of exploratory testing sessions. Any activities outside the observed testing sessions were excluded from the study.

The context of the observations was professionals performing their actual testing tasks in their normal working environment. Most of the sessions took place in front of the subject's personal workstation. A few sessions were observed in small teamwork rooms at the request of one of the companies. A single subject at a time was observed performing individual testing tasks. The total number of observed sessions was 12.

We used comprehensive video and audio recording of the sessions, augmented with field notes. Video-based field observations have been used as a research method in software engineering, e.g., in studying programmer behavior [68], and pair and side-by-side programming [69]–[71]. We performed the recording with two cameras. One recorded the subject's computer screen, as well as the audio of the discussion. The other camera was used to film an overall view of the tester in the working environment to capture all activities that were not observable on the computer screen, e.g., reading paper documentation and taking notes with pen and paper. The field notes were recorded in written format using a laptop computer. In addition, the test documentation that was used during the observed sessions as well as all defect reports were recorded to support analysis.

Since much of the behaviour that we were interested in happens inside a tester's head, we had to include some way of understanding what the observed tester was doing and thinking during the testing. For this purpose, we used the think-aloud method [65], meaning that we asked the subjects to think aloud, i.e., describe what they were thinking while testing. To keep the testing session as natural as possible, the researcher did not enforce continuous verbalization, but only briefly encouraged the subject to verbalize every now and then. The goal of the think-aloud method was not to perform direct verbal protocol analysis [72], but instead to use the subject's verbalisations in the analysis as part of the video recordings. We conducted short (about 30 min) interviews after each observation. The covered topics were the subject's background, and a discussion of how typical, for the subject, the observed session was overall and in terms of the detected defects and issues. In the interviews, we used a general interview guide approach [65].

3.2 Development Organizations and Subjects

The study was carried out in four organizations in three medium-sized software product companies. The companies were selected based on use of the ET approach and accessibility through existing research collaboration. A summary of the characteristics of the organizations and subjects involved in this study is presented in Table 1.

Company *A* is a worldwide market leader of software systems in its engineering domain. Companies *B* and *C* have strong market positions in Scandinavia and the Baltic countries. All companies are profitable and growing, and have been in business for more than 10 years, two of them over 20 years.

The products of all of the companies were relatively mature, with more than 10 years of development each. In all cases, the customers of the studied companies were engineering organizations. All tested systems were applications or systems with rich graphical user interfaces targeted for professional use, meaning that the end users were domain experts, mainly engineers in different fields. All products were highly business-critical for their customers, and failures in the software could cause major financial losses or severely harm the core business processes of the customer organizations. The products of company *A* and *B* are software systems that are used for designing physical structures, which means that software failures could indirectly cause life-critical consequences. In addition, one of the tested products in case A is directly life-critical.

Only company *C* was using a separate testing organization. The other organizations did not have any separate, independent testing organizations. In these cases, a few people took the managerial responsibility for planning and managing the testing activities, and the actual testing tasks were carried out by people in varying (non-tester) roles in the organization.

The subjects of this study were selected using purposeful sampling [65] among software development professionals who had functional testing as one of their duties. We selected eight subjects with different roles and backgrounds, two from each organization. The subjects were high-performing testers, according to the subjective evaluation of their test managers. Three of the observed professionals had testing and quality assurance as their primary role, whereas the rest were application or domain experts (see Table 1). In all organizations, the managers highly appreciated the testing contribution of these application domain experts and considered it crucial for revealing high priority defects from the viewpoint of end users. The subjects had an average of 5.3 years of domain experience and 7 years of software engineering experience.

3.3 Data analysis

Grounded theory [73] (GT) is a suitable research method for qualitative analysis of data in this type of research. In GT, the analysis is grounded in the data instead of existing theories, and the research is theory generating. We had no strong existing theories about defect identification or the role of knowledge in exploratory software testing, or even about manual testers' test execution activities in general. The overall approach to data analysis was "Straussian" grounded theory [73], [74]. We applied the GT methodology in the context of video data analysis as we describe in the next subsection.

3.3.1 Qualitative video data analysis

The research data consisted of video-recorded testing sessions, and the unit of analysis was a single failure incident. We used the Noldus Observer XT software, which is an effective software package specifically designed for coding and analysing video data. Using rich video data as primary documents creates certain challenges in applying a GT approach [71].

We performed an applied GT analysis in four phases. First, we performed open coding focusing on the activities of the testers. In this phase, all full-length observations were coded directly in the video recordings.

Second, based on our research questions, we selected all video excerpts that were coded to include the detection of a defect. In the second coding round, we aimed at coding the selected excerpts from the perspectives of defect detection and knowledge used when detecting defects. We soon realized that directly coding the video material was not feasible, since the concepts of defect detection and how the tester applied the knowledge were not short isolated passages. Instead, it seemed that the concepts most often spanned the whole defect detection episode, which might last anywhere from a few minutes to half an hour.

Third, because of the aforementioned challenge, we transcribed all the selected episodes to text. In the transcriptions, we transcribed not only the think-aloud protocol, but also described with sufficient detail the behaviour of the tester, the general approach to testing, the context, and the observed symptoms of the detected failure itself.

Fourth, we applied open coding to the transcripts of the defect detection episodes. This led to over 50 codes, representing concepts that emerged from the data, that we classified in categories. During this coding and further analysis, we alternated between open and axial coding. As new concepts emerged, they were compared and grouped with similar concepts, and categories were formed around groups of concepts describing similar findings. When the categories and classes emerged in the analysis, the transcriptions were analysed again against those new concepts in a cyclic manner to confirm the findings. The detailed coding of the transcribed episodes was performed using ATLAS.ti software.

3.3.2 Failure type analyses

We discovered that there are only a few failure type classifications available in the literature (see Section 2.4). However, we took the generic omission versus commission classification that has been used in failure and fault classifications earlier [54], [60], [61] as a basis and used our qualitative analysis approach (see Section 3.3.1) and open coding to create a finer failure classification under the two main classes.

In addition, we analysed all failure incidents using the Failure-Triggering Fault Interaction (FTFI) number [63]. For this purpose, we performed one more coding round

Table 1
Development organizations and subjects

	Company A	Company B, Unit 1	Company B, Unit 2	Company C
# of employees	>100	>400, Unit: >200	>400, Unit: <100	>100
Customers	Hundreds	Thousands	About 100	Hundreds
Product	3D modelling for structural engineering	3D modelling for structural engineering and construction information	Network management for energy distribution networks and civil engineering	Data management and simulation for free energy markets
Customization	No (COTS)	No (COTS)	Yes	Yes
End Users	Engineers	Engineers, architects	Company staff, engineers	Company staff
Independent testing org.	No	No	No	Yes
Subjects (#: role)	2: Quality manager and customer service	2: Senior software specialist and technical customer support	2: Software developer and customer service consultant	2: Test manager and software tester
# of sessions	4	4	2	2

in which we focused purely on this aspect and coded all the failure incidents using the FTFI classification.

4 Results

Our data contained the detection of 91 failures in 12 observation sessions. The results are organized in four subsections. First, the role of knowledge in detecting failures is presented in the form of a categorization of knowledge types. Second, we analysed the failure types and present a preliminary failure symptom type classification for exploratory software testing. Third, we analysed the difficulty of detecting the defects. Finally, our analysis revealed that failures were commonly encountered by the testers as a side effect of testing activities. This phenomenon is discussed in the last subsection.

4.1 The role of knowledge in detecting defects

Our analysis revealed three types of knowledge that testers utilized to detect defects in the observed sessions: *domain knowledge (20 failures), system knowledge (41 failures), and generic software engineering knowledge (27 failures).* The knowledge types are summarized in Table 2.

In many cases, knowledge was applied straightforwardly as a test oracle. However, in some cases, knowledge was used as a basis for a more comprehensive tactic to guide testing, i.e., the tester chooses what and how to test based on his or her knowledge. We call this use of knowledge a *test wizard* (see Table 2). Using knowledge as a test wizard was shown, for instance, when a tester designed a targeted attack to investigate certain risks that he or she identified based on earlier experience. Another typical wizard was simulating an experience-based usage scenario when testing.

4.1.1 Domain knowledge

Application domain knowledge was applied to detect failures in 20 of 91 failure incidents. *A failure belongs to the domain knowledge category if the detection requires knowledge of the application domain rules, customers' (including users) processes or needs, or the operational usage context*. The use of domain knowledge was mainly indicated through the verbal comments of the observed tester. An incident was categorized as domain knowledge when the tester reasoned about the failure by referring to concepts and rules of the domain or the circumstances of the real use of the system, e.g., the authentic activities of the users, the operational environment, the users' processes and goals, or the effects of using realistic data. It is very difficult to draw a clear line between domain knowledge and system knowledge, since almost all testing requires some amount of system knowledge and domain knowledge to be performed. Domain knowledge was divided into two main perspectives: the *users' perspective* and the *application domain perspective.*

4.1.1.1 The users' perspective: The users' perspective includes knowledge of the practical procedures of real use and real users of the system together with a good understanding of the real operational context of the system. This also included knowledge of the higher level needs and goals of the users, i.e., for what purposes the system serves as part of the users' own work. This knowledge category was further divided into three subtypes: *episodic knowledge of usage procedures and context*, *conceptual knowledge of the information content and presentation in the usage context*, and *knowledge of problems in customer cases*.

Episodic knowledge of usage procedures and context covers the tester's practical knowledge of how the users perform their tasks using the system. This kind of knowledge is difficult for the practitioners to articulate or describe, but they can perform the actual work activities using the system based on this knowledge. In these cases, testers were usually experienced users of the system themselves, which allowed them to identify problems that restricted their usage procedures or were in conflict with their practical knowledge of the users' activities. In most incidents in this class, the tester also had a deep system knowledge of the features used; however, the actual failures were identified based on reflecting on the system's behaviour with domain knowledge of realistic

Table 2
Categories of knowledge used for detecting defects in software

Knowledge category and perspective		Knowledge type and how it was applied
Domain knowledge	Users' perspective	Episodic knowledge of usage procedures and context • Simulating realistic usage tasks (wizard) • Using the system to prepare tests and data (wizard)
		Conceptual knowledge of the information content and presentation in usage context • Evaluating test results and outputs (oracle)
		Knowledge of problems in customer cases • Testing for specific risks (wizard)
	Application domain perspective	Conceptual knowledge of the subject matter • Evaluating test results and outputs (oracle)
		Practical knowledge of the subject matter and tools • Creating reference results for tests (oracle)
System knowledge	Interacting features and system perspective	Knowledge of system's working mechanisms, logic, and interactions • Simulating realistic usage tasks (wizard) • Observing overall response of the system to changes in configuration, state, or data (oracle) • As a side effect of other testing tasks (oracle) • Comparing to similar features (oracle)
		Knowledge of past failures • Recognizing familiar symptoms (oracle) • Testing for frequently occurring failure types (wizard)
	Individual features and functional perspective	Knowledge of features and views of the system • Visual inspection of GUI or a report (oracle) • Comparing to earlier behaviour (oracle) • As a side effect of other testing tasks (oracle)
		Knowledge of the detailed technical aspects • Systematic searching for errors in logs (wizard) • Investigating error messages or suspicious results and log messages (oracle)
Generic knowledge	Generic correctness perspective	Knowledge of software user interfaces and presentation • Visual inspection of GUI or a report (oracle) • Evaluating test results and outputs (oracle) • As a side effect of other testing tasks (oracle)
	Usability perspective	Practical knowledge of usability of software systems • As a side effect of other testing tasks (oracle)
	Direct failure perspective	Practical knowledge to recognize crashes and error messages (oracle)

usage tasks and context.

For example, the tester realized that in a hierarchical view of data objects, the view is collapsed every time the sorting order or criteria is changed. The tester commented:

> "this [behaviour] is unacceptable, because collapsing the hierarchy heavily distracts the user's attention. And users rarely use this feature to edit complicated data hierarchy, but if mistakes are made, fixing them is costly ..."

In another case, a tester found that a copy function for an entity did not copy the longest attribute, which typically was only slightly modified for new copies. This severely reduced the utility of the function. These cases illustrate how seemingly minor usability glitches are revealed as real problems when they are understood in a realistic usage context.

Episodic knowledge was applied by using the tested features for activities that simulated realistic usage tasks. This can be described as using knowledge as a wizard, because the tester's knowledge of the real usage guided the testing and the tester's strategy of selecting test scenarios, instead of simply acting as an oracle. Another usage context was using the system to prepare tests and data as part of the testing activity, which essentially meant using the system for realistic tasks, and was also an approach to use domain knowledge as a test wizard. This way, the testers detected that features were inadequate for real use even though they were implemented and technically correct to some extent.

The conceptual knowledge of the information content and presentation in the usage context refers to testers' domain-specific knowledge of the information content of the system. The knowledge included understanding the presentation of results or outputs in the realistic context of the users' needs and goals. When testers relied on the knowledge of the information content or presentation, they recognized that the system presented inadequate or

deficient data or presented it in a form that is not useful for the users, i.e., users would need more data, different data, or the data must be presented in a different way, to be fully useful to the users. Testers often referred to what the users would actually do with the outputs of the system and indicated the importance of particular data for the users. Testers judged problems in data presentation as unacceptable in real use, even though the features might seem technically correct.

For example, the most essential and critical data for the users was buried in the middle of a lengthy report that the system produced, rather than being highlighted at the beginning. The tester reflected on his experience of how the report will be used in practice and realized that the form of the report did not support the users' needs.

Testers recognized limitations and inadequacies related to realistic data and usage context that would make the tested functions useless or severely restrict their benefits. For example, in an engineering software, the tester recognized that in a certain view mode the software showed all the dimension texts for a construction model. The tester realized that with a realistic (large) model, the entire view would be filled by dimension readings that obscure the actual model. In this case, the tester understood the real usage context of the feature and could detect this problem immediately, even though it was not obvious when using simple and small models as test data.

Conceptual knowledge was applied when evaluating test results and the outputs of the system. The testers evaluated the results against the needs and goals of the end users based on their knowledge of the real usage context of the system.

The knowledge of problems in customer cases was applied when testers performed exploratory tests based on their knowledge of specific real cases of how the customers, in practice, use or are forced to use the system. This knowledge was related to the identified risks based on past problems of real customers.

As an example, one tester explored the backward compatibility of a new feature. The tester saved a complex data model using the previous version of the software and then opened the model using the new version. The goal was to test whether the version under test could import and use the previous-version data model. The tester commented:

> "Generally, we recommend that customers use the same version of the software within one project, but in practice customers have certain situations where they must upgrade the software in the middle of the project and continue working with the same models even though it is not officially supported. And we always say to customers that you can do it and the models can be converted to new versions."

In this case, the defect was obvious, but the practical domain knowledge about the customer's working context made the tester look for such defects in the software.

The tester applied the knowledge of real customer cases as a wizard by constructing tests to evaluate a specific risk, i.e., evaluating how a new or changed feature works in problematic situations that the tester knows real customers have faced before—and will face in the future. This could be described as using the knowledge to address a specific risk.

4.1.1.2 Application domain perspective: The application domain perspective represents knowledge of the subject matter in the application domain area and includes incidents in which the tester applies knowledge of a more theoretical nature. This knowledge is related directly to the concepts, theories, rules, and technical details of the application domain and not to the usage context. We divide this perspective into two types: *conceptual knowledge of the subject matter*, and *practical knowledge of the subject matter and tools*.

Conceptual knowledge of the subject matter. Testers detected implementation errors that would be hard to recognize without a deep understanding of the domain-specific details behind the features. For example, a tester found out that a view-filtering feature in an engineering software used the wrong part number value for filtering the visible parts in the view. The tester reasoned based on his personal domain knowledge that the filtering should be performed on the other part number. The tester commented:

> "In principle the filtering works technically correctly, but not as desired."

The tester explained that his long experience helped him recognize and understand the behaviour, and that it would be difficult for a novice to find the issue.

In another case, a tester recognized that the engineering software produced control files for the wrong types of parts. The control files used to instruct automatic machines manufacturing steel parts were also produced for concrete parts. This failure seemed to be obvious for the tester based on her application domain knowledge. The tester commented:

> "At these moments one thinks that perhaps the developers should know something about the application domain, or someone has been specifying this feature with blinders on."

Practical knowledge of the subject matter and tools. Testers utilized their domain knowledge to perform equivalent operations using other tools to verify the results produced by the application under test. For example, they performed reference calculations that they compared to the system's output.

4.1.2 System knowledge

System knowledge was utilized in 41 of 91 failure incidents. *A failure belongs to system knowledge category if the detection requires specific knowledge of the features or technical details of the tested system*. An incident was categorized as system knowledge when the detection of the failure clearly required knowledge of the tested system, but not a specific understanding of the application domain and usage contexts. We found two main perspectives of applied system knowledge: *interacting features and system perspective*, and *individual features and functional perspective* (see Table 2).

4.1.2.1 Interacting features and system perspective: The testers' tacit knowledge and intuitive understanding of the system, its features, and overall working logic can be further divided into *knowledge of the system's working mechanisms, logic, and interactions*, and *knowledge of past failures*.

Knowledge of the system's working mechanisms, logic, and interactions. Testers know how the features work together and the fundamental working logic of the system. The tester understands how the system is supposed to react to certain kinds of changes in input data or configuration and can detect defects based on that understanding. The focus is not necessarily on the accurateness of the details, but rather on the general picture of how the system is supposed to react, if the system reacts at all, and if the reaction is correct. For example, a tester was testing a system that simulated real-life situations based on an engineering model. In this case, the tester recognized the system's failure to correctly react to changes in the simulation parameters and properties of the model. The system either did not react at all or reacted only partially.

Another example of common failures was situations in which system indicators incorrectly showed data or calculations as being up-to-date. Testers identified failures by making operations that revealed inconsistencies between the status indicators and the actual status of the system. For instance, an application indicates that a report item is up-to-date, but after the view is refreshed, the contents of the item are completely different.

A distinct type of applying system knowledge was identifying inconsistent behaviour by comparing features within the same system. Inconsistencies occurred in the way different types of data were processed, in the functioning of similar features, and in the order of applied actions. An example of the inconsistency in functioning of similar features was a case, where a cross-reference linking feature was not present in a new feature, and the tester knew that such linking is always used in similar features of the product.

The knowledge of the main working logic of a system also enables testers to recognize unintentional and false changes in the system state that their testing activities should not have caused. Testers had a deep understanding of the system's behaviour and how things affect each other and thus could recognize if something in the system changed without reason. For example, testers recognized that in a graphical view of an engineering model, some properties had suddenly changed without the tester taking any explicit actions to perform such changes. Another typical example was that testers detect unwanted changes after log out, application shut down, or otherwise resetting actions in the system.

Knowledge of the working mechanisms, logic, and interactions was applied by observing the overall response of the system to changes in the configuration, state, or data or by simulating realistic usage scenarios. This knowledge was also applied to recognize failures that occurred as a side effect of other testing activities, e.g., unintentional changes. Finally, system knowledge was applied as a consistency heuristic when testers compared a new feature to similar features and detected failures based on the inconsistency between the new feature and similar features of the same system.

Knowledge of past failures. Knowledge of past system failures was used either as an oracle to help recognize the symptoms of a failure, or as a wizard to focus testing on revealing certain types of defects. For example, a problem with an empty date field that was erroneously printed in a report as a default value of "1.1.1970" alerted the tester, because she had seen this same problem before in other parts of the system. In another example, a tester tested input fields with maximum length inputs and searched for symptoms of buffer overflow. He quickly discovered that the end of the input string appeared in a field in another dialogue. The tester had experience with similar buffer overflow defects previously in the same system, and he utilized that knowledge to discover other similar situations.

4.1.2.2 Individual features and functional perspective: The individual features and functional perspective describes the testers' knowledge of isolated features and how system knowledge is applied to evaluate individual features, views, or reports of the system locally, without comparison or consideration of the system's workings as a whole. This perspective is divided into *knowledge of the features and views of the system*, and *knowledge of detailed technical aspects*.

Knowledge of the features and views of the system. Testers had intuitive knowledge of what features and functions there are and how data is presented for users in the views of the system. Testers recognized visible defects and omissions based on this intuitive understanding of what elements should be in the application user interface and how things usually look in the application. Examples of failures revealed using this knowledge type are missing images and icons from the GUI dialogues and recognizing that a result was missing in an application that shows a large number of values after calculations.

Another common type intuitively recognized was cases in which a feature or function did not work at all, or a certain capability was missing. For example, a tester tried to input data into a table in the tested application and found that he could not edit the table directly.

Testers recognized failures based on their understanding of the earlier behaviour of the system. If a current function clearly deviated from the earlier behaviour, without a good reason, the testers interpreted it as a failure. For example, the tester had a clear idea how fitting a work area of a graphical view to the objects on the area should work. When the area fitting function left the area too big, it was immediately clear to the tester that this was a failure.

This intuitive knowledge was applied in a rather straightforward manner: First, simple visual inspection of the system's user interface, or a report, was common way. Second, it was applied as part of other testing tasks,

Table 3
Failure type examples in the generic knowledge category

Generic correctness	• *Typos*, e.g., excessive control characters or badly formatted text. • *Functional defects* that are easily identifiable on GUI level, e.g., multiplied menu items in a context menu. • *Layout problems* in reports and printouts, e.g., tables or diagrams are placed over the page margins, truncated diagrams, paging problems, missing content, extra pages.
Usability	• *Usability problems* that make using a feature difficult or restricts the user unnecessarily. E.g, GUI element collapses to too small a size and cannot be resized, faulty or misleading error messages, lack of user interface feedback.
Direct failures	• *Explicit error message dialogs* that pop up on top of the application window and are clearly technical failures rather than informative messages for the user. • *Crashes*, e.g., desktop application crashes and server side errors in web applications.

Table 4
Failure type versus knowledge category

Failure type	Domain	System	Generic	All
Commission				
Presentation and layout	15 %	17 %	**37 %**	24 %
Error message	5 %	12 %	**26 %**	14 %
Extraneous functionality	10 %	2 %	7 %	5 %
Inconsistent state	5 %	**17 %**	0 %	9 %
Incorrect results	**45 %**	20 %	4 %	21 %
Commission total	80 %	68 %	74 %	74 %
Omission				
Presentation and layout	5 %	2 %	4 %	3 %
Missing function	0 %	5 %	0 %	2 %
Lack of feedback	0 %	7 %	11 %	7 %
Lack of capability	**15 %**	**17 %**	11 %	14 %
Omission total	20 %	32 %	26 %	26 %

whenever a tester encountered obvious problems. Third, the tester compared features to the earlier behaviour. It was clear that on many occasions, a tester could identify a failure based on an observed change in a feature without being able to exactly specify the correct function.

Knowledge of detailed technical aspects includes testers utilizing their knowledge to interpret error and log messages and to use the command shell or equivalent tools to check results and investigate the internal status of the system. Testers applied this knowledge as an oracle when recognizing error messages or abnormal log entries in the application command shell or log window. They knew that some types of run time errors generate messages in these logs and applied the knowledge as a wizard by systematically checking logs for such messages. Testers used command shell tools and application-specific script languages to access the internal data and verify test results. Technical tools and technical knowledge were also applied to investigate further observed symptoms of recognized failures and to better understand what was happening in the system.

4.1.3 Generic software engineering knowledge

The generic software engineering knowledge category included 27 of 91 failure incidents. *A failure belongs to generic knowledge category if it is such a generic software defect that detection does not require specific understanding of the tested system or its application domain*. These defects would be obvious to recognize for most software testers or software engineering professionals. In practice, the failures in this category were also obvious to the researcher who observed the testers.

The knowledge perspectives for this category were: *generic correctness*, *usability*, and *direct failures*. The subcategories are further characterized in Table 2, and examples of failure types are listed in Table 3. Generic software engineering knowledge was applied mainly by visual inspection, evaluating results or outputs, or as part of other testing activities when unexpected failures occurred.

4.2 Classification of visible failure symptoms

Our second analysis of the data focused on the observable symptoms of the failures that the testers recognized. Based on this analysis, we present a preliminary classification of visible failure symptoms in Table 4. We divided the failures into omission and commission failures, meaning that the failure manifests itself either as an incorrect behaviour, or as a missing feature or capability in the system. The distribution of the failure incidents is also presented in Table 4.

4.2.1 Commission failures

We further classified the commission failures into five subclasses.

4.2.1.1 Presentation and layout: includes symptoms visible in the outputs or result presentations of the system. The failures were observed, e.g., on screen or in report printouts either in files or on paper. The symptoms of this class can be visually observed by evaluating the views or outputs of the system without interaction.

Concrete examples of this failure symptom class were data elements that shrink or are cut-off in a way that restricts or prevents usage; positioning failures of report elements, e.g., content overlapping other content or page margins; failures in paginating text, tables, and figures on pages; incorrect layout after changing sizes or contents of GUI elements; and incorrect presentation of the data or results.

4.2.1.2 Error message: includes failures recognized based on explicit error messages. The error messages can either be immediately visible to the end user or hidden. Immediately visible are obvious, shown to the user as pop-up windows or another clear mechanism. Hidden messages are logged in the system log window or file, and the tester must look there specifically for the

error messages to recognize the failure. Sometimes other symptoms alerted the tester to look for the hidden error messages, but in most cases, the tester had a frequent habit of checking log windows for suspicious messages.

4.2.1.3 Extraneous functionality: includes failures recognized by extra and unnecessary system behaviour that is harmful or distractive for the users' goals. The failures typically occurred when one function was being tested and the system performed some, usually small, harmful side effect in addition to the behaviour that the tester was expecting.

Examples of such behaviour include unexpected collapsing of hierarchy items and closing data entries that are being edited; unexpectedly resetting user input; producing faulty, extra output files; duplicate or non-functional user interface widgets or options, e.g., menu items.

4.2.1.4 Inconsistent state: includes failures that manifest as inconsistencies of the system's internal state or logic. The tester identified these failures by recognizing an inconsistent state of the data, status indicators, or other systems properties. In this failure class, the symptoms are related to interaction between different functions or different parts of the system. Typically these failures are not detected as a result of a certain single function; instead, testers recognize the inconsistency after some set of operations. Such failures typically require extensive investigation to isolate the actual cause.

Examples include: changes that the tester made were not correctly reflected in other parts of the system; the status indicators for the data items were not consistent with the actual status of the items; the tester could identify that something in the system had changed without a reason or after, for example, shut down and restart.

4.2.1.5 Incorrect results : include failures that show directly as an incorrect result of a function or functions that the tester executes. In this class, the symptoms are directly connected to a specific function and its results, compared to the inconsistent status class in which the symptoms are related to interaction of different functions or parts of the system.

Examples of failures in this class include: incorrect results of a calculation operation; filtering of graphical view objects that resulted to a wrong view; incorrect contents in reports and outputs; application crash or server side exception.

4.2.2 Omission failures

The omission failure class consists of four subclasses.

4.2.2.1 Data presentation and layout: includes failures that the tester recognizes as missing elements such as headings, backgrounds, icons and images in the outputs or result presentations of the system.

4.2.2.2 Missing function: includes failures that appear as functions that do not perform the intended operation at all and do not trigger explicit error messages. Another type of failure in this class was that the function cannot even be found in the system.

4.2.2.3 Lack of feedback: includes failures in which the tester does not get the expected or assumed feedback from the system. These failures were related to immediate feedback issues such as selection feedback and cursor feedback in certain events. These problems decreased the usability of the system because the tester did not know what was happening or whether the requested event occurred in the system. Another group of lacking feedback were situations in which the user would need a more informative error messages or other instructions to understand what is wrong with the input or how to correctly proceed.

4.2.2.4 Lack of capability: includes failures in which the tester determines that some aspects of a function or part of a feature is missing.

Examples include completely missing parts of calculation results; partially incomplete features, such as impossibility of editing certain data; inability to resize views; functions operating only on a partial set of data; functions that work only for one of several supported data formats; insufficient datasets for the actual purpose of the processed data; missing documentation; lack of consistently applied features, e.g., cross-referencing hyperlinks in one feature or view.

Missing special case handling was a distinct subclass of this omission class in which failures were recognized as missing handling logic for extreme or exceptional cases. Examples of such failures were missing checks for valid filenames and other inputs, inability to handle large input values, and missing checks for the length of input data.

4.2.3 Symptom types versus knowledge categories

We analysed the relationship between the failure type versus knowledge categories using cross-tabulation presented in Table 4.

There are a few interesting cells in this table. First, the *commission: presentation and layout* as well as the *error message* classes seem to mostly require only generic knowledge to be recognized. This is quite intuitive, and is directly related to the very explicit and visible nature of these failure types. Second, the *inconsistent status* failure type seems to be typical for the *system knowledge* category. This relationship can be explained by noting that inconsistent status failures are typically defects in the internal logic of the tested system, which matches with the subcategory *knowledge of system's working mechanisms, logic, and interactions*. Third, *incorrect results* relates heavily to *domain knowledge*. This is explained by the need for *conceptual knowledge of the subject matter* to recognize the incorrect results. Finally, in the omission failure classes, the *missing capability* failure class was related to both *domain* and *system knowledge*. The missing capabilities were omissions that restricted the usefulness of the features in real use or were system-specific omissions in the details of specific features.

4.3 Failure recognition difficulty

To understand the difficulty of detecting the defects, we analysed all failure incidents using the *failure-triggering fault interaction* (FTFI) number [63]. The FTFI number refers to the number of interacting variables or conditions that together cause a certain failure. In this analysis, all failure incidents were classified based on the fault's FTFI number. The results are presented in Table 5. We identified the number of affecting failure parameters or conditions for each failure and categorized the failure incidents as *1-way*, *2-way*, *3-way*, or *directly visible* failures. In addition, we used an *unclear* category if the number of affecting conditions could not be determined based on the data.

Table 5
FTFI number distribution of failures

FTFI number	Domain	System	Generic	Total
0 Dir. visible	5 %	12 %	4 %	8 %
1-way	75 %	34 %	44 %	45 %
2-way	15 %	37 %	19 %	26 %
3-way	5 %	5 %	7 %	7 %
Unclear	0 %	12 %	26 %	14 %
Total	20	41	27	91

We can see that in our data, 45% of the failure incidents were 1-way failures, 26% were 2-way, and only 7% were 3-way failures. 8% of the failures were in the directly visible class, and 14% were unclear.

When comparing the FTFI distribution and the knowledge categories, we found that in our observation data failures related to domain knowledge were more straightforward to detect (80% directly visible or 1-way), and failures related to system knowledge or generic software development knowledge were more complicated to detect (46% and 48% directly visible or 1-way, respectively) in terms of the number of interactions.

4.4 Side effect failures

A clear and unexpected finding that emerged from our data was the frequency of *side effect failures.* We define a side effect failure as *a failure that occurred when testers used other features or areas of the system than the actual target of the testing session*. In our qualitative coding process, concepts related to these side effect failures emerged frequently, which motivated us to make a selective coding round focusing on this phenomenon. We found that testers often needed to, for example, set up some data or applicable situations for their actual testing activities, or use other features of the system during the testing for various reasons such as further investigation of found failures. Testers also occasionally performed ad hoc exploratory testing activities for different features of the system. Many times, testers recognized failures during these activities.

A more detailed analysis revealed that 20% (18 of the total 91) of the observed failures were categorized as side effects. Most of the side effect defects were unexpected error messages, extraneous, or incorrect functions. The side effect failures were recognized based on generic (61%) or system knowledge (39%), and were most often 1-way (33%) or 2-way failures (28%).

5 Discussion

In this section, we present the discussion of the main findings. We answer the research questions in relationship to the existing literature and evaluate the limitations of our study.

5.1 Knowledge types used for defect detection

Our first research question was: *What types of knowledge do testers utilize for detecting defects when performing exploratory testing?* We identified and categorized the knowledge utilized in a hierarchical taxonomy consisting of three main categories: *domain knowledge*, *system knowledge*, and *generic software engineering knowledge*.

We want to highlight a few specific findings. First, in the domain knowledge category, the customers' perspective was strongly emphasized. The knowledge of the real usage context and procedures including the users' needs and goals formed one strong perspective, and detailed subject matter knowledge in the application domain the other perspective. System knowledge was further divided into knowledge of system-level feature interactions, and detailed knowledge of individual features and the functional perspective. These findings resemble the main concepts that Sandberg [45] identified when studying the work of engine optimizers: separate qualities, interacting qualities, and the customers' perspective.

Second, testers were able to identify a large number of failures in the observed sessions without any explicit descriptions of the expected results of the tests that they executed. The results show that testers utilize three different types of knowledge for detecting failures. This supports the practitioners' reasoning for using exploratory testing. In our earlier research, we found that the ET approach is motivated in development organizations by stating that the testing requires such a deep understanding of the application domain of the system that it could only be tested by people with deep domain knowledge [16]. The results of this study deepen our insights into this, and show the crucial role of system knowledge. Based on our knowledge analysis we state the following hypothesis. *In exploratory testing, testers are able to utilize their personal knowledge of the application domain, the users' needs, and the tested system for defect detection.*

5.2 Application of knowledge

The second research question was: *How do testers apply their knowledge for detecting defects when performing exploratory testing?* Based upon our analysis, we differentiated two main approaches. First, the most common way

of applying knowledge was using knowledge as a *test oracle* (see Table 2). Applying knowledge as an oracle differs clearly from the traditional test-case based paradigm in which the expected result is specified prior to test execution. The testers took a wider, system perspective or compared features with other functionality, which matches our earlier findings [36]. A comparison to most prior work on software test oracles (see Section 2.3) is difficult because they have not studied knowledge as test oracle. Weyuker [26], however, described her personal experiences of how testers can apply knowledge as a partial oracle that is very similar to our findings. Our work brings a novel contribution to the area of software test oracles because it uses empirical results to widen the perspective of test oracles to consider the use of the tester's personal knowledge as an oracle. We also provided a detailed description of the knowledge types and how the knowledge was applied as an oracle.

Second, our analysis revealed several more comprehensive approaches for applying tester knowledge in ET. We call these approaches applying the knowledge as a *test wizard* (see Table 2), instead of just for oracle purposes. One could consider applying knowledge as a test wizard to be real-time test case selection, design, and execution, including using the knowledge as a test oracle. These findings are examples of how the exploratory testing approach is applied in practice.

One of the most significant findings related to the way testers applied their knowledge was the recognition of *side-effect failures*. Testers detected failures as part of all their activities besides testing the actual target feature, in test data preparation, ad hoc exploring, and other situations. It is important to note that side-effect failures were related to test preparation activities, but also to ad hoc exploration and following hunches that are core activities of ET. The relatively high number of side effect failures, 20% of all detected failures, and the important role of side-effect findings emerged clearly from our data. This finding supports the claims that the ET approach enables the tester to explore the functionality in a versatile and creative way. The analysis of the side effect failures indicated that the testers were able to cover a wider range of essential features during their testing sessions even though they were primarily focusing on a certain feature or area of the application. In addition, all side-effect failures were detected based on system and generic knowledge, which means that side-effect failures were not related to the tester's experience with the domain, but rather to the varied usage of the system. The side-effect failures also seem to be related to realistic situations, which makes them relevant in terms of the effect on the users. Based on these findings we state a hypothesis: *In exploratory testing, testers frequently recognize relevant failures in a wider set of features than the actual target features of the testing activity.*

5.3 Failure types detected in ET sessions

Finally, the third research question was: *What types of failures do testers identify using knowledge in exploratory testing?* We analysed the types of identified failures from two different viewpoints: the failure symptoms, and the detection difficulty. The incidents were classified based on the visible symptoms forming a preliminary symptom classification (see Section 4.2.). That classifies failures according to the symptoms visible to the tester, and thus also the end user, if not fixed. Compared to the existing failure classification by Wallace and Kuhn [59] we find that their classification includes a few classes, namely "data", "quality", and "timing" that refer to relevant symptom types that did not emerge in our analysis. Bondavalli and Simoncini [57] also identified the timing failure class. Compared to the usability failure classification in CUP [60], we found that our classification is more focused and detailed in terms of the visible failure symptoms, whereas the CUP classification includes classes "missing", "extraneous", and "wrong", but also in the same classification includes "better way" and "incongruent mental model" that are related to the subjects' opinions and understanding of the system, not to the failure symptoms directly. Our failure symptom classification (see Table 4) focuses purely on the observable symptoms from the users' viewpoint and is directly related to the testers' oracle decisions.

The classification presented in this paper is a preliminary contribution to classifying software failures based on visible symptoms. The observational dataset that the classification is based on does not exhaustively cover different types of target systems or all possible types of software failures. However, this focused failure-symptom-based classification provides a new empirically based contribution to types of software failures that has not been covered in previous research. The classification can be used to increase the knowledge of testers and developers about different symptoms so they can better recognize failures as they occur. An improved classification could also serve as a checklist for exploratory testing and even motivate developing testing techniques, either exploratory or test-case based, targeted to specific types of symptoms. In particular, the identified test wizard approaches in combination with certain failure types would be a good basis for new ET techniques.

Our FTFI analysis in Table 5 reflects how complex a failure is to trigger in terms of the number of interacting variables or features. Our analysis showed a similar n-way distribution (see Table 5) that has been reported in other studies in which empirical failure data has been analysed [63]. For example, in medical device failure analysis "Only three of 109 failures indicated that more than two conditions were required to cause the failure" [59]. Our results show that a high proportion of detected failures in the ET sessions were straightforward to find (53%), in terms of the FTFI numbers, or required only

generic software engineering knowledge (30%). This finding is important from multiple viewpoints.

First, it contradicts the general claim that ET would require a high level of testing experience to be effective. Our findings from an ET context raise an initial hypothesis that *a large number of the failures in software applications and systems can be detected without detailed test design or descriptions*.

Second, it raises a question of if, and how much, the trivial failures and easily spotted problems actually mask more complicated domain or system-specific issues from surfacing during testing.

Third, based on the large proportion (80%) of 1-way or directly visible failures in the domain knowledge category (see Table 5) we state a hypothesis that *the majority of failures related to domain knowledge are straightforward to recognize, and failures related to system knowledge or generic software development knowledge are more complicated to recognize, in terms of the number of interactions.* This finding supports having domain experts conducting ET, since a large number of domain-specific failures can be detected without complicated test designs. Such domain experts are people, who are not in testing roles in development organizations, but have strong domain knowledge or close customer contact. Our findings are partly similar with the results of Følstad [49], who found that domain experts were capable of revealing severe and relevant findings in usability inspections. The importance of this finding is emphasized when considering our other study in software product organizations in which the high practical contribution of domain experts in detecting defects was shown [75].

Since this study focused only on the role of knowledge in the failure recognition part of testing, the effect of knowledge in the test design activity and in the exploratory testing approach as a whole still remains open. We cannot draw conclusions, based on this analysis, about how much and what kind of knowledge is required to be able to select good or correct tests, but we showed examples of such application of knowledge as a test wizard, in which testers applied their knowledge for targeted attacks to certain risks.

5.4 Validity threats

We have analysed 12 exploratory testing sessions in four software development organizations. The external validity of our results is limited to similar contexts with exploratory software testing approaches. In hypotheses generating qualitative studies, the external generalizability comes through theoretical saturation of the data. In this research, practical limitations restricted the number of development organizations and subjects that we could include in the analysis. Due to the low number of subjects and the fact that we used purposeful sampling, the presented quantitative amounts are descriptive in nature and cannot be used to draw statistical conclusions.

Threats to construct validity of this study are the effects of the observer's presence, the amount and quality of the verbal protocols, and the selection of the observed testers and individual testing sessions. The observer's presence affected the subject's way of working. Based on our interviews, we assume that in comparison to their normal working conditions, the subjects worked more intensively and were more focussed on their testing tasks, which they performed without any interruptions. Many subjects also commented that they seemed to find a larger number of failures and issues than usual. The amount of verbal think-aloud protocol varied between the subjects, because of their individual characteristics. The selection of subjects was based on interviews with their managers, and the criterion was to select good testers in the context of the organization in question. The individual observed test sessions were selected based on availability.

In addition, coding and main analysis of the data was performed by the first author alone, which may have introduced the risk of researcher bias. The actual defects and the properties of the tested software systems naturally had an effect on the results of this study, but the similarity of the defect type distributions compared to earlier published failure data [59], [63] supports the assumption that the tested systems were not anomalous in terms of the defect distributions.

5.5 Future work

More research is needed to better understand all aspects of the ET approach and the role of experience and knowledge in it. In this study, we stated multiple testable hypotheses that should be tested in other studies on ET. The exploratory testing strategies and techniques should be studied more in-depth in future research. We plan to continue analysing our data in more detail along the research path we initiated in one of our earlier studies [36].

More empirical data is needed on the actual failure types and role of side-effect failures in different development contexts and application domains.

6 Conclusions

In this article, we have reported the results of an empirical observation study of the role of testers' knowledge in detecting failures in the context of exploratory software testing in industry. Our results show that testers are capable of recognizing different types of software defects based on their personal knowledge without detailed test case descriptions. Testers apply knowledge of the system under test and its application domain, including users' needs and goals for revealing defects. The knowledge covers not only individual features, but also, and even more importantly, interactions of many features and the workings of the system as a whole.

Personal knowledge is applied for testing in a distinctly different fashion compared to how the test-case based paradigm describes the software testing activity. Our results show that the ways of applying knowledge

in exploratory testing involve evaluating the overall behaviour of the system, comparing the features with other features, and knowledge of earlier versions. Knowledge is sometimes applied as a wizard to design targeted attacks to known risks or customer problems. The knowledge is also applied to generate the expected results as part of the testing activity. In addition, a significant share of findings in exploratory testing seems to originate as side effects of the actual testing activity, which further emphasizes the diverse and creative opportunities of the exploratory testing approach.

Based on combined analysis of the knowledge and failure data, we state four testable hypotheses. We suggest that the exploratory testing approach could be effective even when less experienced testers are used. On the other hand, our hypothesis is that the exploratory testing approach is an effective way of involving the knowledge of domain experts, who do not have testing expertise, in testing activities.

This research adds to the body of knowledge on empirical understanding of software failure types in industrial software systems. We present a preliminary failure symptom classification based on the externally visible symptoms, i.e., those seen and experienced by testers or users. This classification can be used to guide testers and to create focused failure-driven exploratory testing techniques. The classification increases understanding of software failures from the viewpoint of the effects the failures have on end users. The classification is preliminary and needs to be further improved and extended with more failure data from different contexts.

ACKNOWLEDGMENTS

The authors would like to thank the study subjects and participating companies for providing access to the field settings. This research was partly funded by Tekes (Finnish Funding Agency for Technology and Innovation) and SoSE (Graduate School on Software Systems and Engineering).

REFERENCES

[1] G. J. Myers, *The Art of Software Testing*. New York: John Wiley & Sons, 1979.

[2] B. Beizer, *Software Testing Techniques*. New York: Van Nostrand Reinhold, 1990.

[3] C. Kaner, J. Falk, and H. Q. Nguyen, *Testing Computer Software*. New York: John Wiley & Sons, Inc., 1999.

[4] L. Copeland, *A Practitioner's Guide to Software Test Design*. Boston: Artech House Publishers, 2004.

[5] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," *IEEE Transactions on Software Engineering*, vol. 1, no. 2, pp. 156–173, 1975, journal Article.

[6] C. Andersson and P. Runeson, "Verification and validation in industry - a qualitative survey on the state of practice," in *Proceedings of International Symposium on Empirical Software Engineering*, 2002, pp. 37–47.

[7] S. Ng, T. Murnane, K. Reed, D. Grant, and T. Chen, "A preliminary survey on software testing practices in australia," in *Proceedings of Australian Software Engineering Conference*, 2004, pp. 116–125.

[8] E. Engström and P. Runeson, "A qualitative survey of regression testing practices," in *Proceedings of 11th international conference on product-focused software process improvement*, 2010.

[9] M. Fewster and D. Graham, *Software Test Automation*. Harlow, England: Addison-Wesley, 1999.

[10] C. Kaner, J. Bach, and B. Pettichord, *Lessons Learned in Software Testing*. New York: John Wiley & Sons, Inc., 2002.

[11] S. Berner, R. Weber, and R. K. Keller, "Observations and lessons learned from automated testing," in *Proceedings of International Conference on Software Engineering*, 2005, pp. 571–579.

[12] A. Beer and R. Ramler, "The role of experience in software testing practice," in *Proceedings of Euromicro Conference on Software Engineering and Advanced Applications*, 2008, pp. 258–265.

[13] J. Bach, "Exploratory testing," in *The Testing Practitioner*, 2nd ed., E. van Veenendaal, Ed. Den Bosch: UTN Publishers, 2004, pp. 253–265.

[14] A. Abran, J. W. Moore, P. Bourque, R. Dupuis, and L. L. Tripp, *Guide to the Software Engineering Body of Knowledge 2004 Version*. Los Alamitos, CA, USA: IEEE Computer Society, 2004.

[15] A. Tinkham and C. Kaner, "Exploring exploratory testing," in *Software Testing Analysis & Review Conference (STAR) East*, Orlando, FL, United States, 2003, p. 9.

[16] J. Itkonen and K. Rautiainen, "Exploratory testing: a multiple case study," in *Proceedings of International Symposium on Empirical Software Engineering*, 2005, pp. 84–93.

[17] J. A. Whittaker, *Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design*. Upper Saddle River, NY: Addison-Wesley Professional, 2009.

[18] J. Våga and S. Amland, "Managing high-speed web testing," in *Software Quality and Software Testing in Internet Times*, D. Meyerhoff, B. Laibarra, R. van der Pouw Kraan, and A. Wallet, Eds. Berlin: Springer-Verlag, 2002, pp. 23–30.

[19] J. Lyndsay and N. van Eeden, "Adventures in Session-Based testing," http://www.workroom-productions.com/papers/AiSBTv1.2.pdf, May 2003.

[20] B. Wood and D. James, "Applying Session-Based testing to medical software," *Medical Device & Diagnostic Industry*, vol. 25, no. 5, p. 90, May 2003.

[21] J. Tuomikoski and I. Tervonen, "Absorbing software testing into the scrum method," in *Proceedings of 10th International Conference on Product-Focused Software Process Improvement*, 2009.

[22] P. Poon, T. H. Tse, S. Tang, and F. Kuo, "Contributions of tester experience and a checklist guideline to the identification of categories and choices for software testing," *Software Quality Journal*, vol. 19, no. 1, pp. 141–163, 2011.

[23] V. Kettunen, J. Kasurinen, O. Taipale, and K. Smolander, "A study on agility and testing processes in software organizations," in *Proceedings of 19th international symposium on software testing and analysis*, 2010, pp. 231–240.

[24] P. N. Robillard, "The role of knowledge in software development," *Communications of the ACM*, vol. 42, no. 1, pp. 87–92, 1999.

[25] A. Memon, I. Banerjee, and A. Nagarajan, "What test oracle should i use for effective GUI testing?" in *Proceedings of 18th International Conference on Automated Software Engineering*, 2003, pp. 164–173.

[26] E. J. Weyuker, "On testing Non-Testable programs," *The Computer Journal*, vol. 25, no. 4, pp. 465 –470, 1982.

[27] D. Martin, J. Rooksby, M. Rouncefield, and I. Sommerville, "'Good' organisational reasons for 'Bad' software testing: An ethnographic study of testing in a small software company," in *Proceedings of International Conference on Software Engineering*, 2007, pp. 602–611.

[28] J. Rooksby, M. Rouncefield, and I. Sommerville, "Testing in the wild: The social and organisational dimensions of real world practice," *Computer Supported Cooperative Work*, vol. 18, no. 5–6, pp. 559–580, 2009.

[29] R. D. Craig and S. P. Jaskiel, *Systematic Software Testing*. Boston: Artech House Publishers, 2002.

[30] J. A. Whittaker, *How to Break Software A Practical Guide to Testing*. Boston: Addison Wesley, 2003.

[31] L. Crispin and J. Gregory, *Agile testing: a practical guide for testers and agile teams*. Boston: Addison-Wesley, 2009.

[32] J. Bach, "Session-Based test management," *Software Testing and Quality Engineering*, vol. 2, no. 6, 2000.

[33] F. Houdek, T. Schwinn, and D. Ernst, "Defect detection for executable specifications — an experiment," *International Journal of Software Engineering and Knowledge Engineering*, vol. 12, no. 6, p. 637, Dec. 2002.

[34] J. Itkonen, M. V. Mäntylä, and C. Lassenius, "Defect detection efficiency: Test case based vs. exploratory testing," in *Proceedings of International Symposium on Empirical Software Engineering and Measurement*, 2007, pp. 61–70.

[35] L. H. O. do Nascimento and P. D. L. Machado, "An experimental evaluation of approaches to feature testing in the mobile phone applications domain," in *Workshop on Domain specific approaches to software test automation*, 2007, pp. 27–33.

[36] J. Itkonen, M. V. Mäntylä, and C. Lassenius, "How do testers do it? an exploratory study on manual testing practices," in *Proceedings of 3rd International Symposium on Empirical Software Engineering and Measurement*, 2009, pp. 494–497.

[37] J. Pichler and R. Ramler, "How to test the intangible properties of graphical user interfaces?" in *Proceedings of 1st International Conference on Software Testing, Verification, and Validation*, 2008, pp. 494–497.

[38] J. Itkonen, K. Rautiainen, and C. Lassenius, "Toward an understanding of quality assurance in agile software development," *International Journal of Agile Manufacturing*, vol. 8, no. 2, pp. 39–49, 2005.

[39] J. Kasurinen, O. Taipale, and K. Smolander, "Test case selection and prioritization: risk-based or design-based?" in *Proceedings of International Symposium on Empirical Software Engineering and Measurement*, 2010, p. 10.

[40] A. Tinkham and C. Kaner, "Learning styles and exploratory testing," in *Pacific Northwest Software Quality Conference (PNSQC)*, 2003.

[41] L. Shoaib, A. Nadeem, and A. Akbar, "An empirical evaluation of the influence of human personality on exploratory software testing," in *Proceedings of IEEE 13th International Multitopic Conference*, 2009.

[42] R. T. Turley and J. M. Bieman, "Competencies of exceptional and nonexceptional software engineers," *Journal of Systems and Software*, vol. 28, no. 1, pp. 19–38, Jan. 1995.

[43] B. Adelson and E. Soloway, "The role of domain experience in software design," *Software Engineering, IEEE Transactions on*, vol. SE-11, no. 11, pp. 1351–1360, 1985.

[44] S. Sonnentag, "Expertise in professional software design: A process study." *Journal of Applied Psychology*, vol. 83, no. 5, pp. 703–715, 1998.

[45] J. Sandberg, "Understanding human competence at work: an interpretative approach," *Academy of Management Journal*, vol. 43, no. 1, pp. 9–25, 2000.

[46] M. Kharlamov, A. Polovinkin, E. Kondrateva, and A. Lobachev, "Beyond brute force: Testing financial software," *IT Professional*, vol. 10, no. 3, pp. 14–18, 2008.

[47] C. Engelke and D. Olivier, "Putting human factors engineering into practice," *Medical Device & Diagnostic Industry*, vol. 24, no. 7, Jul. 2002.

[48] R. Merkel and T. Kanij, "Does the individual matter in software testing?" Swinburne University of Technology, Centre for Software Analysis and Testing, Technical Report 2010-001, May 2010.

[49] A. Følstad, "Work-Domain experts as evaluators: Usability inspection of Domain-Specific Work-Support systems," *International Journal of Human-Computer Interaction*, vol. 22, no. 3, p. 217, 2007.

[50] D. F. Galletta, D. Abraham, M. E. Louadi, W. Lekse, Y. A. Pollalis, and J. L. Sampler, "An empirical study of spreadsheet error-finding performance," *Accounting, Management and Information Technologies*, vol. 3, no. 2, pp. 79–95, 1993.

[51] W. Howden, "Theoretical and empirical studies of program testing," *IEEE Transactions on Software Engineering*, vol. 4, no. 4, pp. 293–298, 1978.

[52] L. Baresi and M. Young, "Test oracles," University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A., Tech. Rep. Technical Report CISTR- 01-02, Aug. 2001.

[53] J. A. Whittaker, "What is software testing? and why is it so hard?" *IEEE Software*, vol. 17, no. 1, pp. 70–79, 2000.

[54] V. R. Basili and R. W. Selby, "Comparing the effectiveness of software testing strategies," *IEEE Transactions on Software Engineering*, vol. 13, no. 12, pp. 1278–1296, 1987.

[55] M. Bolton, "Testing without a map," *Better Software*, Jan. 2005.

[56] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, and M. Wong, "Orthogonal defect classification-a concept for in-process measurements," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 943–956, 1992.

[57] A. Bondavalli and L. Simoncini, "Failure classification with respect to detection," in *Proceedings of 2nd IEEE Workshop on Future Trends of Distributed Computing Systems*, 1990, pp. 47–53.

[58] D. Cotroneo, S. Orlando, and S. Russo, "Failure classification and analysis of the java virtual machine," in *Proceedings of 26th IEEE International Conference on Distributed Computing Systems*, 2006, pp. 17–26.

[59] D. R. Wallace and D. R. Kuhn, "Failure modes in medical device software: An analysis of 15 years of recall data," *International Journal of Reliability, Quality and Safety Engineering*, vol. 8, no. 4, pp. 351–371, 2001.

[60] E. T. Hvannberg and L. Law, "Classification of usability problems (CUP) scheme," in *Proceedings of the 9th International Conference on Human-Computer Interaction*, 2003, pp. 655—662.

[61] N. Juristo and S. Vegas, "Functional testing, structural testing and code reading: What fault type do they each detect?" in *Empirical methods and studies in software engineering: experiences from ESERNET*. Springer, 2003.

[62] B. Freimut, "Developing and using defect classification schemes," Fraunhofer IESE, Kaiserslautern, Research Report IESE-Report, 072.01/E, 2001.

[63] D. Kuhn, D. Wallace, and A. Gallo, "Software fault interactions and implications for software testing," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 418–421, 2004.

[64] C. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 557–572, 1999.

[65] M. Q. Patton, *Qualitative Research and Evaluation Methods*, 3rd ed. Thousand Oaks: Sage, 2002.

[66] C. Seaman and V. Basili, "Communication and organization: an empirical study of discussion in inspection meetings," *IEEE Transactions on Software Engineering*, vol. 24, no. 7, pp. 559–572, 1998.

[67] T. C. Lethbridge, S. E. Sim, and J. Singer, "Studying software engineers: Data collection techniques for software field studies," *Empirical Software Engineering*, vol. 10, no. 3, pp. 311–341, 2005.

[68] H. Wu, Y. Guo, and C. B. Seaman, "Analyzing video data: A study of programming behavior under two software engineering paradigms," in *Proceedings of International Symposium on Empirical Software Engineering and Measurement*, 2009, pp. 456–459.

[69] A. Höfer, "Video analysis of pair programming," in *Proceedings of International workshop on scrutinizing agile practices or shoot-out at the agile corral*, 2008, pp. 37–41.

[70] L. Prechelt, U. Stärk, and S. Salinger, "7 types of cooperation episodes in Side-by-Side programming," Freie Universität Berlin, Institut für Informatik, Berlin, Germany, Technical Report B-08-17, Dec. 2008.

[71] S. Salinger, L. Plonka, and L. Prechelt, "A coding scheme development methodology using grounded theory for qualitative analysis of pair programming," *Human Technology*, vol. 4, no. 1, pp. 9–25, May 2008.

[72] J. Hughes and S. Parkes, "Trends in the use of verbal protocol analysis in software engineering research," *Behaviour & Information Technology*, vol. 22, no. 2, p. 127, 2003.

[73] A. L. Strauss and J. M. Corbin, *Basics of qualitative research: techniques and procedures for developing grounded theory*. SAGE, 1998.

[74] J. C. van Niekerk and J. D. Roode, "Glaserian and straussian grounded theory: similar or completely different?" in *Proceedings of Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, 2009, pp. 96–103.

[75] M. V. Mäntylä, J. Itkonen, and J. Iivonen, "Who tested my software? testing as an organizationally cross-cutting activity," *Software Quality Journal*, 2011, forthcoming, Submitted Oct. 2010.