The Role of the Tester's Knowledge in 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 while performing exploratory software testing (ET) in industrial settings. We video recorded 12 testing sessions in four industrial organizations, having our subjects think aloud while performing their usual functional testing work. Using applied grounded theory, we analyzed how the subjects performed tests and what type of knowledge they utilized. We discuss how testers recognize failures based on their personal knowledge without detailed test case descriptions. The knowledge is classified under the categories of domain knowledge, system knowledge, 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 for test design, to guide them in selecting objects for test and designing tests. Interestingly, a large number of failures, windfall failures, were found outside the actual focus areas of testing as a result of exploratory investigation. We conclude that the way exploratory testers apply their knowledge for test design and failure recognition differs clearly from the test-case-based paradigm and is one of the explanatory factors of the effectiveness of the exploratory testing approach.

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], [2], [3], [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 when performed as a manual activity, is a more or less mechanical task. During execution, the predefined test cases are run and their outputs 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], [7], [8].

Although test automation has been the focus of a great deal 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], [10], [11], [12]. In many software development contexts, the manual testing effort of professional testers and application domain experts is crucial to ensuring that products fulfill the needs of the users or please the markets. In this context, exploratory software testing has been proposed as an effective testing approach.

The exploratory testing (ET) approach 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], [16], [17]. There are a growing number of practitioner reports and studies on the benefits of exploratory testing [13], [18], [19], [20], [21]. In these reports, ET is commonly described in the context of system-level testing of interactive systems through the GUI and from the end user's point of view.

In studies of manual testing and, in particular, in the ET context, the experience and especially the application domain knowledge of testers have been recognized as important aspects affecting the tester's behavior and 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 recognize 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 behavior of the tested system, can be used to create new, better tests during exploratory testing.

In this paper, we present a study in which we examined how testers provoke and recognize software failures in

• *The authors are with the Department of Computer Science and Engineering, School of Science, Aalto University, PO Box 15400, FI-00076 Aalto, Finland. E-mail: {juha.itkonen, mika.mantyla, casper.lassenius}@aalto.fi.*

Manuscript received 11 May 2011; revised 25 Feb. 2012; accepted 4 Sept. 2012; published online 10 Sept. 2012.
Recommended for acceptance by L. Williams.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2011-05-0143.
Digital Object Identifier no. 10.1109/TSE.2012.55.

real-world exploratory testing work. We provide a detailed analysis of the actual failure incidents during functional software testing in four software development organizations. We focus on the types of knowledge that testers apply when recognizing failures and how they apply this knowledge to recognize failures (as a test oracle) and to design tests to provoke failures. In addition, we analyze the types of failures that are recognized based on the different knowledge types.

We use the term *failure* as defined by the IEEE: "the inability of a system or component to perform its required functions within specified performance requirements" [25] in a broad sense, including any failure of the system to satisfy the user's or customer's needs or expectations. The term *fault* is used in this paper where the distinction between the fault and failure is essential, i.e., when referring to the actual "incorrect step, process, or data definition in a computer program" [25].

Recently, researchers in the empirical software engineering 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 the superiority of methods or technologies [12], [26], [27]. 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 exploratory testing.

In the next section, 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 and discussed in Section 4. A short summary and a discussion of the limitations of this study are presented in Section 5, followed by the conclusions and recommendations for future work in the last section.

2 Related Work

The context of this study is the exploratory software testing approach, and our research focuses on the role of knowledge in testing and failure recognition and the types of recognized failures. Thus, we begin the review of related work with the exploratory testing literature. Then we present the role of experience and knowledge in software testing in general, and continue with motivating the role of knowledge as a test oracle. Finally, we conclude this section by covering research on existing software failure classifications.

2.1 Exploratory Software Testing

Exploratory testing is an experience-based testing approach that differs fundamentally from the highly document-driven test-case-based testing 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].

ET is an approach, not a specific testing technique, such as boundary value analysis or pairwise testing. Many test design techniques can also be applied in an exploratory way since the difference is that in ET test design, execution, result interpretation, and learning are not temporally disconnected as separate phases, but happen more or less in parallel. The goal is not to transfer the knowledge and experience between people in a form of documented test cases, but to use knowledgeable individuals to perform test design, execution, and interpretation simultaneously. Thus, instead of focusing only on predesigned test cases that are run in a repetitive way, reproducing tests and confirming existing information, ET leverages the knowledge of testers to create varying and diverse tests on the fly in order to reveal as many yet unknown problems in the software as possible.

The term "exploratory testing" was introduced by Kaner et al. [3]. The ET approach has been acknowledged in software testing books since the 1970s [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], [28], [29]. Bach described exploratory testing in more detail [13], and Tinkham and Kaner [15] covered the need for questioning skills and the heuristic nature of ET.

Practitioner reports on exploratory testing claim it is both effective in detecting failures (or bugs) and cost efficient [13], [18], [19], [20], [30]. These reports, however, are personal experience reports without any scientifically documented empirical basis. Practitioner literature also describes how to manage ET using session-based test management [19], [30], which is an approach in which testing is managed as time-boxed (few hours) sessions, higher level guidance and goals, regular debriefings, and logging instead of test suites and test cases. Another example is the tour-based approach [17] that utilizes a tour metaphor and uses different types of tours through the tested system to guide exploratory testing activities.

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. [31] studied defect detection effectiveness in the executable specification context, comparing systematic testing and experience-based ad hoc simulation, and Itkonen et al. [32] performed a student experiment comparing the failure detection effectiveness of ET and TCBT. In an experiment, do Nascimento and Machado [33] 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 [31], [32], [33] and to require less effort than the testing approaches it was compared to [31], [33]. 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. [34]

describe empirically observed ET practices. Pichler and Ramler [35] 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 [36] 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. [26] 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. [37] 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 [38] and the effect of personality traits on exploratory testing performance, showing that extrovert personalities might be more likely to excel in exploratory testing [39].

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 enhancement of 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, other than our previous work [34].

2.2 Experience and Knowledge in Software Testing

Experience and knowledge have been studied in the contexts of software engineering [40] and software design [41], [42]. Sandberg [43] 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 [44] and medical [20], [45] 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], [46]. 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, attending 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 into the exploratory testing approach.

In a survey on the effect of experience and individual differences in software testing, Merkel and Kanij [46] 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 to be 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 fault and failure recognition has been identified in software testing, spreadsheet error finding, and usability inspection contexts. Multiple studies report the importance of domain knowledge in testing [12], [16], [23], [46]. In the context of usability inspections, Følstad [47] 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 [47]. Galletta et al. [48] studied error-finding performance in the spreadsheet context. They compared fault identification performance of domain area (accounting) experts versus novices, and system (spreadsheet software) experts versus novices. They found that both types of expertise increased fault identification performance, but the performance of those with both types of expertise far exceeded the performance of other groups.

2.3 Knowledge as a Test Oracle

One way of applying knowledge in software testing is to use it as a test oracle. A *test oracle* is a concept referring to a method used to distinguish between a correct and an incorrect result during software testing [2], [14], [49], [50]. Failure 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 [49], [50], [51], [52].

The challenge of finding a reliable oracle is referred to as "the oracle problem." The TCBT paradigm aims to solve 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, requirements, specifications, and thus test cases, are seldom perfect in terms of comprehensiveness and accuracy. 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 [26], [27]). In industrial practice, the recognition of failures is left as a human decision [50], [51]. Thus, the oracle problem is highly relevant in manual testing, and is typically solved using the personal knowledge of testers and varying types of documentation.

The challenge of using a human oracle is that humans are fallible, i.e., testers do not always recognize a failure, even when a test case reveals it. In an experiment by Basili and Selby [53], subjects recognized only 70 percent of observable failures. On the other hand, human testers are able to detect incorrect results using experience-based partial oracles, even when they cannot know the exact correct result [54].

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.

Existing research on the role of knowledge and experience in software testing and failure and fault recognition in general raises the hypothesis that knowledge has an important effect on failure and fault recognition performance and that domain knowledge is more important than testing experience. To our knowledge, the use of personal knowledge as a test oracle has not been studied in the context of manual testing with the intent of understanding, describing, or improving the way humans recognize failures. 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 failure recognition when performing ET remain an unstudied area, and in this paper, we present a detailed analysis of how and what types of personal knowledge testers apply to recognize failures.

2.4 Failure Type Classifications

In this study, we focused on *failure* incidents that exploratory testers recognized by analyzing the externally visible symptoms of the failures. Commonly cited classifications usually classify *faults* based on technical, often source-code level, characteristics, e.g., [56], and are not suitable for this study. For a summary of existing functional fault classifications, see [57]. In this study, our target was to study the symptoms that a tester, or end user, perceives when the failure occurs. Some failure classifications classify failures with respect to the symptoms [58], [59]. However, in [58], all user-observable failures are classified in a single class, and in [59], the classification only focuses on technical crashes, deadlocks, and error messages and does not describe the failure symptoms of functional behavior.

Some failure classifications that characterize the symptoms, however, can be found. One such classification was used in a study of medical device failures [60]. In this study, Wallace and Kuhn analyzed software-related failures of medical devices that led to recalls by the manufacturers and presented a 13-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 [61]. The usability problem classification, even though similar to the tester's failure classification, includes usability-specific classes that are not applicable in the context of externally visible failure symptoms that were our focus. One generic and simple classification dimension, which has been used in testing technique experiments [53], [62] and is visible as part of the usability problem [61] and fault classifications [63], is the omission versus commission dichotomy, which simply separates failures based on whether something is missing or wrong. We choose to take this simple dichotomy as the basis for our symptom analysis and propose a more detailed preliminary symptom-based failure classification.

We also analyzed the difficulty of provoking the failures. One characteristic of failure that is directly related to the provocation difficulty is the number of interacting conditions that together cause the failure. This characteristic can be analyzed by using the failure-triggering fault interaction (FTFI) number [64]. This original term by Kuhn et al. is rather misleading considering the common meaning of the term "fault." We define the concept, using the terminology applied in this paper, as *failure-triggering condition interaction number (FTCI)*. The FTCI number refers to the number of interacting variables or conditions that together cause a certain failure to occur. In one-way faults, only one condition triggers the failure, while two-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 study in which the testing practices of eight software development professionals in four software development organizations were studied. Our research methodology consists of empirical participant observations [65] as the primary data collection method and qualitative data analysis using applied grounded theory. We also present quantitative summaries of the recognized 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 failure recognition in exploratory software testing. We aimed at describing how testers recognize failures as they occur in exploratory testing and at drawing hypotheses grounded on our observational data. We studied the types of knowledge involved and characterized the observed failure types. The high-level research objective was to *understand how failures are recognized 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 provoking and recognizing failures when performing exploratory testing?*
- RQ2: *How do testers apply their knowledge when performing exploratory testing?*
- RQ3: *What types of failures do testers recognize using knowledge in exploratory testing?*

As the analysis for RQ1 and RQ2 was qualitative and requires lengthy descriptions, we present the results regarding the first two research questions together in Section 4.1. The quantitative analysis for RQ3 is presented separately in Section 4.2.

3.1 Data Collection Methods

We collected the data using participant observation [65] 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" [66]. Because participant observation can mean many things, depending on the source (see, e.g., [66], [67], [68]), we describe our approach using the six dimensions of fieldwork variations presented by Patton [66].

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. Verbal communication occurred between the observer and subject as the subject was thinking aloud, and the observer asked occasional clarifications.
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 as a researcher* were fully disclosed to the subjects.
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 functional testing in 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 environments. 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, e.g., in studying programmer behavior [69] and pair and side-by-side programming [70], [71], [72]. 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 and all defect reports were recorded to support analysis.

Since much of the behavior 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 [66], meaning that we asked the subjects to think aloud, i.e., to 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 [73], but instead to use the subject's verbalizations in the analysis as part of the video recordings. We conducted short (about 30 min) interviews after each observation. The topics covered were the subject's background and a discussion of how typical, for the subject, the observed session was overall, and in terms of the revealed failures and issues. In the interviews, we used a general interview guide approach [66].

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 the 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,

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

and failures in the software could cause major financial losses or severely harm the core business processes of the customer organizations. The products of companies A and B were software systems used for designing physical structures, which means that software failures could indirectly cause life-critical consequences. In addition, one of the tested products in company 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 of planning and managing the testing activities, and the actual testing tasks were carried out by people in varying (nontester) roles in the organization.

The subjects of this study were selected using purposeful sampling [66] 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. This means that their testing contributions were respected in their organizations, but does not indicate superior performance with any objective metrics. 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 failures 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 [74] (GT) is a suitable research method for qualitative analysis of data in this type of research. Grounded theory is a method for generating theory from data instead of using data to confirm existing theories, i.e., the research is theory-generating. We had no strong existing theories about failure recognition 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 [74], [75]. We applied the GT methodology in the context of video data analysis, as we describe in the next section.

3.3.1 Qualitative 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 [72].

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 observation sessions were coded in full length. This coding was performed by applying the codes directly in the video recordings.

Second, based on our research questions, we selected all video excerpts that were coded to include the recognition of a failure. In the second coding round, we aimed at coding the selected excerpts from the perspectives of failure recognition and knowledge used when provoking and recognizing failures. We soon realized that directly coding the video material was not feasible since the concepts of failure recognition and how the tester applied the knowledge were not short isolated passages. Instead, it seemed that the concepts most often spanned the whole test execution 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 behavior of the tester, the general approach to testing, the context, and the observed symptoms of the recognized failure itself.

Fourth, we applied open coding to the transcripts of the test execution episodes. This led to over 50 codes,

representing concepts that emerged from the data, which we classified in categories. During this coding and further analysis, we alternated between open and axial coding. The purpose of axial coding is to reassemble data that were fractured during open coding. It is a process of relating categories to subcategories along the lines of their properties and dimensions [74]. 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 analyzed again against those new concepts in a cyclic manner to confirm the findings. The detailed coding of the transcribed episodes was performed using the ATLAS.ti software package.

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 [53], [61], [62] as a basis and employed 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 analyzed all failure incidents using the failure-triggering condition interaction (FTCI) number (see Section 2.4). For this purpose, we performed one more coding round in which we focused purely on this aspect and coded all the failure incidents using the FTCI classification. The FTCI number for each observed failure was determined by observing the failure occurrence on the video recording. The FTCI number could be analyzed based on how the tester interacted with the system, investigated the failure, and commented on the triggering conditions. For example, in the case of the failure related to the collapsing hierarchical view, described in Section 4.1.1, the FTCI number was 2: Triggering the failure required the tester to first open the hierarchy levels in the view, and second, to toggle the sorting order. If the number of interacting conditions that were triggering a failure could not be determined based on the data, the incident was coded as "unclear."

4 Results and Discussion

Altogether, our data contained the recognition of 88 failures in 12 observation sessions. These were all distinct failures in terms of the observable symptoms from the testers' viewpoint, i.e., data did not include duplicate recognitions of the same failures. We did not have the longitudinal follow-up data on how the reported defects were processed and resolved. In the following, we present the results in two sections. First, we present the results of our analysis on the role of knowledge in ET and a categorization of knowledge types in Section 4.1. Second, we present a preliminary failure type classification and analyze the failure provocation difficulty in Section 4.2. We present the discussion together with the results in separate sections.

4.1 The Role of Knowledge in Recognizing Failures

Our analysis revealed three types of knowledge that testers utilized to recognize failures in the observed sessions: domain knowledge, system knowledge, and generic software engineering knowledge. The knowledge types are summarized in Table 2. The knowledge types are applied together when recognizing a certain failure. In the categorization, each of the failure incidents was categorized in exactly one category indicating the dominant knowledge type. This means that, e.g., failures in the domain knowledge category oftentimes additionally required some level of system knowledge to be recognized.

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 test design, i.e., the tester chose what and how to test based on his or her knowledge. (see Table 2). Using knowledge for test design 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 example was simulating an experience-based usage scenario when testing.

4.1.1 Domain Knowledge

Application domain knowledge was applied to recognize failures in 20 of 88 failure incidents. A failure belongs to the domain knowledge category when the *recognition 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.

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

TABLE 2
Categories of Knowledge Used for Recognizing Failures 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 (design) • Using the system to prepare tests and data (design)
		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 (design)
	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 (design) • Observing the overall response of the system to changes in configuration, state, or data (oracle) • Recognizing failures outside the actual testing scope, e.g., in other features (oracle) • Comparing to similar features (oracle)
		Knowledge of past failures • Recognizing familiar symptoms (oracle) • Testing for frequently occurring failure types (design)
	Individual features and functional perspective	Knowledge of features and views of the system • Visual inspection of GUI or a report (oracle) • Comparing to earlier behavior (oracle) • Recognizing failures outside the actual testing scope, e.g., in other features (oracle)
		Knowledge of the detailed technical aspects • Systematic searching for errors in logs (design) • 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) • Recognizing failures outside the actual testing scope, e.g., in other features (oracle)
	Usability perspective	Practical knowledge of usability of software systems • Recognizing failures outside the actual testing scope, e.g., in other features (oracle)
	Direct failure perspective	Practical knowledge to recognize crashes and error messages (oracle)

usually experienced users of the system themselves, which allowed them to recognize problems that restricted their usage procedures or were in conflict with their practical knowledge of the users' activities. In these cases, the testers also needed a deep system knowledge of the features, in most cases, based on personal experience as a user of the system; however, the actual failures were identified by reflecting on the system's behavior with domain knowledge of realistic 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 criterion is changed. The tester commented:

> "this [behavior] 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 for test design 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 for test design. This way, the testers recognized 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 data 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 were 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 would 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 recognize this problem immediately, even though it was not obvious when using simple and small models as test data.

Conceptual knowledge was applied as an oracle 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 forward 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 failure was obvious, but the practical domain knowledge about the customer's working context made the tester look for such failures in the software.

The tester applied the knowledge of real customer cases to aid in test design 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 the specific risk.

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 identified 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 to filter 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 behavior, 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.

Discussion. The use of domain knowledge shows the power of the exploratory approach to testing and describes how the testers are able to use their practical knowledge in testing in ways that would be difficult to document as test cases. The failures identified by domain knowledge are related to how the features are used in practice and how they support the users' actual tasks. In our data, the importance of customers' perspective was strongly emphasized. One could argue that such failures are due to poor specifications and requirements engineering practices. However, in many software development contexts, requirements are not specified in such a form that these problems could be resolved in requirements engineering. The problems might manifest themselves only when an experienced user of the system tries to use it for realistic tasks, and it may not be possible to recognize without simulating the real usage context. In addition, the above quotes indicate that testers rely even more on their personal knowledge than the documented specifications. This was also indicated by the testers' preference for asking developers or other

people instead of trying to find the information in documentation when they were uncertain of the intended or correct behavior of the system.

4.1.2 System Knowledge

System knowledge was utilized in 41 of 88 failure incidents. A failure belongs to the system knowledge category when *the recognition requires specific knowledge of the features or technical details 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).

Interacting features and system perspective. The testers' knowledge and understanding of the system and its features 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 recognize failures based on that understanding. The focus is not necessarily on the accuracy 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 react correctly 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 recognizing inconsistent behavior 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 behavior and how things affect each other and thus could recognize when 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 recognized unwanted changes after log out, application shutdown, or other 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 in areas not directly in the testing focus, e.g., unintentional changes. Finally, system knowledge was applied as a consistency heuristic when testers compared a new feature to similar features and recognized 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 to focus testing on revealing certain types of failures and thus to guide test design. 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 failures previously found in the same system, and he utilized that knowledge to discover other similar situations.

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 knowledge of what features and functions are available and how data are presented for users in the views of the system. Testers recognized visible failures and omissions based on this 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 noticing images and icons missing 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 example in this category 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 behavior of the system. If a current function clearly deviated from the earlier behavior without a good reason, the testers interpreted it as a failure. For example, the tester had a clear idea of 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 became immediately clear to the tester that this was a failure.

This knowledge was applied in a rather straightforward manner: First, a simple visual inspection of the system's user interface, or a report, was a common way. Second, it was applied as part of other testing tasks whenever a tester encountered obvious problems. Third, the tester compared features to the earlier behavior. It was clear that on many occasions, a tester could recognize 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 runtime errors generate messages in these logs and applied the knowledge by systematically checking logs for such messages. Testers used command shell tools and application-specific script languages to access the internal data and to 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.

Discussion. In addition to domain knowledge, system knowledge emerged as important in exploratory testing. An intimate knowledge of the workings of system was helpful for finding problems, as was knowledge of past failures. System knowledge was used to recognize "strange" behavior, such as inconsistencies or unintentional side effects, or to test for commonly occurring problems, such as buffer overflows. In addition to the system-wide perspective, the detailed knowledge of individual features and the functional perspective also provided important clues for testing. This included both looking for consistency in behavior, as well as inspecting logs and the internal state of the system. While it might be possible to describe the expected results in test cases for most of the problems identified using this perspective, in many cases, it would require predicting and documenting a vast amount of potential failures and things that could go wrong. Indeed, one of the strengths of ET is the human capability to recognize a wide variety of problems when they occur without the need of predicting them beforehand and the ability to drill down and investigate a problem in greater detail as testing progresses.

4.1.3 Generic Software Engineering Knowledge

The generic software engineering knowledge category included 27 of 88 failure incidents. A failure belongs to the generic knowledge category if *it is such a generic software failure that recognition does not require specific understanding of the tested system or its application domain*. These failures 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, evaluation of results or outputs, or as part of other testing activities when unexpected failures occurred.

TABLE 3
Failure Examples in the Generic Knowledge Category

Generic correctness	• *Typos*, e.g., excessive control characters or badly formatted text. • *Functional failures* 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 placed over the page margins, truncated diagrams, paging problems, missing content, extra pages.
Usability	• *Usability problems* that make using a feature difficult or restrict 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.

The generic knowledge was applied without the testers paying much attention to describing the findings or how they recognized the failures. They seemed to think that the failures in this category were rather obvious. This means that there was not much to analyze from the knowledge viewpoint in these cases. For example, when the tested software crashed or a layout of a view was obviously flawed, it was rather straightforward to see how the failure was recognized.

4.1.4 Discussion on the Role of Knowledge in ET

In this section, we present an overall discussion of our findings regarding the role of knowledge in exploratory testing. First, we discuss the identified knowledge types. Second, we discuss the ways knowledge was applied during testing, and, third, we give a more detailed discussion of the interesting windfall failure phenomenon that we identified.

Knowledge types. Testers were able to recognize 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 recognizing failures. This supports the practitioners' reasoning for using exploratory testing. The analysis of the knowledge types emphasizes the power of ET. In particular, domain knowledge was applied to reveal failures that could probably not be related to any explicit specification. Specifications cannot be precise and comprehensive enough to describe all types of relevant failures that occur in practice. Testing in an exploratory context does not mean checking against a specification but rather testing against implicit expectations. The expectations became explicit when faced with a concrete context and system behavior during testing.

In earlier research, it was recognized that the ET approach is motivated in development organizations by stating that

TABLE 4
Focused and Holistic Knowledge Types

	Domain knowledge	System knowledge	Use
Focused	Appliction domain perspective	Individual features and functional perspective	Oracle
Holistic	User's perspective	Interacting features and system perspective	Design Oracle

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 support this finding by describing in detail the types of knowledge testers utilize in recognizing failures.

Our findings resemble the main concepts that Sandberg [43] identified when studying the work of engine optimizers: separate qualities, interacting qualities, and the customers' perspective. It seems that in both the domain knowledge and system knowledge categories, the knowledge divides into focused knowledge of separate features or details and holistic knowledge of interactions or the usage context (see Table 4). It is an interesting question whether testers with more experience and expertise would be more capable of applying this holistic type of knowledge and whether less experienced testers would rely on focused knowledge on individual details. Sandberg did not identify this type of change in the conception of work when novices gain more expertise [43].

Based on our knowledge analysis, we state the following hypothesis:

- H_1: Exploratory testing is more efficient than test-case-based testing in recognizing functional failures since the testers are able to use different types of personal knowledge in designing tests and recognizing failures on the fly, rather than restricting their focus to predefined tests.

Application of the knowledge. Based upon our analysis, we differentiated two main approaches to applying personal knowledge to testing. First, the most common way of applying knowledge was using knowledge as a *test oracle* (see Table 2). Applying knowledge as an oracle clearly differs 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 is similar to earlier findings on comparison techniques applied in manual testing [34]. A comparison to most prior work on software test oracles (see Section 2.3) is difficult because researchers have not studied personal knowledge as a test oracle. Weyuker [54], however, described her personal experiences of how testers can apply knowledge as a partial oracle that is very similar to our findings. Weyuker uses the term "partial oracle" to describe how testers are able to recognize false test results without knowing the exact correct outcome. This aspect is visible also in our results in the form of the testers' ability to recognize failures, even though they do not have exactly specified expected results for their tests.

Our work contributes to the body of software engineering knowledge by using empirical results to broaden 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 to applying the tester's knowledge for test design in ET, instead of just for oracle purposes (see Table 2). In these cases, knowledge was applied to real-time test case selection, design, and execution, including using the knowledge as a test oracle. These findings are concrete examples of how the exploratory testing approach is applied in practice.

When considering holistic versus focused knowledge (see Table 4), it seems that focused knowledge was more often applied as a pure oracle, whereas holistic knowledge was applied both for test design and oracle purposes. Based on this finding, we propose that effective ET requires holistic knowledge of the tested system, as well as the users and usage scenarios and context.

Windfall failures. One of the most significant findings related to the way testers applied their knowledge was the recognition of *windfall failures*. We define a windfall failure as *a failure that occurred when testers exercised other features or areas of the system than the actual target of the testing session.* In our qualitative coding process, concepts related to these windfall 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 recognized failures. Testers also occasionally performed ad hoc exploratory testing activities for many features outside the actual focus of the testing session. Many times, testers recognized failures during these activities.

A more detailed analysis revealed that 20 percent (18 of the total 88) of the observed failures were categorized as windfall failures. Most of the windfall failures were unexpected error messages, extraneous, or incorrect functions. The windfall failures were recognized based on generic (61 percent) or system knowledge (39 percent).

Testers recognized 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 windfall failures were related not only to test preparation activities, but also to ad hoc exploration and following hunches, both of which are core activities of ET. The relatively high number of windfall failures (20 percent of all) and their important role emerged clearly from our data. This finding supports the claims that the ET approach enables the tester to explore the system functionality in a versatile and creative way. These findings indicate 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 windfall failures were

TABLE 5
Failure Types in Knowledge Categories (Percentages of Observed Failures in Each 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 %

recognized based on system and generic knowledge, which means that windfall failures were not related to the tester's experience with the domain, but rather to the varied usage of the system. The windfall failures also seem to be related to realistic usage situations, which makes them relevant to the product.

Based on these findings, we state the following hypothesis:

- H_2: A major factor explaining the efficiency of exploratory testing is the large number of failures (windfall failures) incidentally found outside the actual target features of the testing activities.

4.2 Failure Symptom Classification and Provocation Difficulty

The second part of our analysis focused on the observable symptoms and provocation difficulty of the failures that the testers recognized. Based on this analysis, we present a preliminary classification of visible failure symptoms in Section 4.2.1 and an analysis of failure provocation difficulty in Section 4.2.2.

4.2.1 Symptom Classification

Based on our analysis of failure types, we created a preliminary classification of visible failure symptoms, which we consider a valuable additional contribution from this research. We divided the failures into omission and commission failures, meaning that the failure manifests itself either as an incorrect behavior or as a missing feature or capability in the system. The percentage distribution of the failure incidents to failure types in each knowledge category is presented in Table 5.

Commission failures. We further classified the commission failures into five subclasses.

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.

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 messages 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.

Extraneous functionality includes failures recognized by extra and unnecessary system behavior 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 behavior that the tester was expecting.

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 recognized 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.

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.

Omission failures. The omission failure class consists of four subclasses.

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.

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.

Lack of feedback includes failures in which the tester does not obtain 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. Another group of lacking feedback was 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 proceed correctly.

Lack of capability includes failures in which the tester determines that some aspect of a function or part of a feature is missing. Examples include completely missing parts of calculation results, partially incomplete features, or missing documentation. In addition, missing special case handling was an example of this class, in which failures were recognized as missing handling logic for extreme or exceptional cases.

Discussion. We analyzed the relationship between the failure type and knowledge categories using cross tabulation, shown in Table 5. A few interesting relationships should be noted. First, the commission: Presentation and layout as well as the error message classes seem to mostly require only generic knowledge to be recognized. This relation is plausible since these kinds of failures are not system specific, and are recognizable without deep domain

TABLE 6
FTCI Number Distribution of Failures

FTCI number	Domain	System	Generic	Total
Always	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 %
# of failures	20	41	27	88

or system knowledge. 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 related to the internal logic of the tested system, which matches with the subcategory knowledge of the 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.

Our failure symptom classification (see Table 5) focuses purely on the observable symptoms from the users' viewpoint and is directly related to the testers' oracle decisions. Compared to the existing failure classification by Wallace and Kuhn [60], we find that their classification includes a few classes—namely "data," "quality," and "timing"—which refer to relevant symptom types that did not emerge in our analysis. Bondavalli and Simoncini [58] also identified the timing failure class. Our classification is a preliminary contribution to classifying software failures based on visible symptoms. The classification increases the knowledge of different symptoms that can be used to better recognize failures as they occur. An improved classification could also serve as a checklist for exploratory testing and even motivate development of testing techniques targeted to specific types of symptoms.

4.2.2 Failure Recognition Difficulty

In order to understand the provocation difficulty of the failures and how knowledge types relate to recognizing failures that are complicated or straightforward to provoke, we analyzed all failure incidents using the FTCI number (see Section 2.4). In this analysis, all observed failure incidents were classified based on the FTCI number. The FTCI number essentially refers to the complexity of the provocation of a failure in terms of the number of inputs or other conditions that interact together in order to provoke the actual failure. This is one aspect of detection difficulty, but does not take into account, e.g., how obvious or unusual the provocation conditions are.

The results are presented in Table 6. We identified the number of affecting failure parameters or conditions for each failure and categorized the failure incidents as *one-way*, *two-way*, *three-way*, or *always* failures. The always class includes failures that do not require any tester's actions or inputs to provoke the failure, e.g., GUI problems that are recognizable without performing any functions. In addition, we used an *unclear* category when the number of affecting conditions could not be determined based on the data.

We can see that in our data that 45 percent of the failure incidents were one-way failures, 26 percent were two-way, and only 7 percent were three-way failures. Eight percent of the failures were in the always class, and 14 percent were unclear.

When comparing the FTCI distribution and the knowledge categories, we found that in our observation data, failures related to domain knowledge were more straightforward to provoke (80 percent always or one-way), and failures related to system knowledge or generic software development knowledge were more complicated to provoke (46 percent and 48 percent always or one-way, respectively) in terms of the number of interactions. However, the provocation difficulty distribution of generic knowledge category included 26 percent unclear cases, making such data rather unreliable.

In addition, 33 percent of the windfall failures were one-way, 28 percent two-way, and 6 percent three-way; 33 percent were unclear. This distribution is not significantly different from the ones in other categories.

Discussion. The FTCI analysis in Table 6 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 6) that has been reported in other studies in which empirical failure data have been analyzed [60], [64]. Our results show that a high proportion of recognized failures in the ET sessions were straightforward to find (53 percent), in terms of the FTCI numbers, or required only generic software engineering knowledge (30 percent). While we do not know to what extent this distribution is a consequence of exploratory testing as an approach and to what extent it reflects properties of the tested systems, we assume that this finding mostly reflects ET as an approach and speculate about its meaning in the following.

First, our finding contradicts the general claim that ET would require a high level of *testing* experience to be effective. Even though our subjects were regarded as high-performing testers in their organizations, it seems that the testing performance was not based on complicated test design, but more on the tester's knowledge. Our findings from an ET context raise an initial hypothesis.

- H_3: a large fraction of the failures in software applications and systems do not require complicated test designs or descriptions to be provoked and recognized.

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 percent) of one-way or always failures in the domain knowledge category (see Table 6), we state a hypothesis:

- H_4: The majority of failures recognized by the ET approach and related to domain knowledge are straightforward to provoke, whereas failures related to system knowledge or generic software development knowledge are more complicated to provoke in terms of the number of interacting conditions.

This finding can be explained either by the relatively straightforward nature of domain-related failures or by the low level of testing skills of testers with domain knowledge. However, we did not recognize such differences between individuals in our data. Also, the domain knowledge might be applied at a superficial level, which might explain the fact that such a small number of two- and three-way failures was recognized.

This finding supports having domain experts conduct ET, since a large number of domain-specific failures can be recognized without complicated test designs. Such domain experts are people who are not in testing roles in development organizations, but who have strong domain knowledge or close customer contact. Our findings are partly similar to the results of Følstad [47], 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 the practical contribution of domain experts in finding and reporting software failures that was reported in a case study of three software product organizations [76].

Our findings do not mean that testing skills and expertise would not be beneficial for ET. Indeed, ET would probably be even more effective when the practitioners have a higher degree of testing expertise. However, these findings are important for two reasons. First, the results show that ET can produce results efficiently even without a high level of testing expertise, given that the testers possess the required knowledge for revealing and recognizing the failures. On the other hand, the results show that there are a large number of straightforward or otherwise obvious failures to be found, which suggests that, in practice, different types of testing are needed with different levels of test design and approaches to knowledge utilization.

5 Summary

In this section, we present a brief summary of the results under each research question, followed by answers to each question. In addition, we present the limitations and threats to the validity of our study.

5.1 Knowledge Types in Exploratory Testing

Our first research question was: *What types of knowledge do testers utilize for provoking and recognizing failures 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* (see Table 2). Furthermore, we divided the knowledge types into *focused* and *holistic*. Focused knowledge refers, e.g., to the understanding of individual features, and theoretical concepts of the application domain. Holistic knowledge deals, e.g., with interacting features, and larger usage scenarios.

5.2 Application of Knowledge

The second research question was: *How do testers apply their knowledge 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. Second, knowledge was used for on-the-fly *test design*. It seems that focused knowledge was used more as a test oracle, whereas holistic knowledge was applied both for test design and as an oracle.

Another important finding was that many failures are recognized in areas not directly under study as a consequence of exploratory testing activities. We refer to these failures as *windfall failures* as they represent unexpected positive outcomes of ET.

5.3 Failure Types Recognized in ET Sessions

Finally, the third research question was: *What types of failures do testers recognize using knowledge in exploratory testing?* We analyzed the types of identified failures from two different viewpoints: the failure symptoms, and provocation difficulty.

We constructed a preliminary failure symptom classification based on our data (see Table 5). The classification describes typical failure symptoms for the identified knowledge categories. These results can be used for developing checklists and targeted exploratory testing techniques for provoking and recognizing failures that are manifested as specific types of symptoms.

The FTCI analysis showed that the majority of the identified failures were relatively straightforward to reveal. In addition, the failures that were found using domain knowledge were often even more straightforward to reveal than the other failures. This contradicts the assumption that ET requires a high level of testing expertise in order to be effective.

5.4 Validity Threats

We analyzed 12 exploratory testing sessions in four software development organizations. The external validity of our results is limited by the context in which our observations were made. The results would mainly be generalizable to functional testing in similar contexts with exploratory software testing approaches.

In hypotheses generating qualitative studies, theoretical saturation is important for internal validity in terms of completeness and accuracy of the theory. We can state, based on our data, that the identified knowledge categories seem to be saturated in the context of our subject organizations since all categories were observed in multiple cases. A few of the detailed knowledge types (knowledge of problems in customer cases and practical knowledge of the subject matter and tools) were observed only in one case. Practical limitations restricted the number and selection of development organizations and subjects that we could include in the analysis; thus, we are not able to claim full theoretical saturation.

Due to the low number of subjects and the fact that we used purposeful sampling, the presented quantitative

results are not a representative sample of the software industry and must not be used to draw statistical conclusions.

Threats to the 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 focused 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 test sessions were selected for observation based on availability.

In addition, coding and main analysis of the data were performed by the first author alone, which may have introduced the risk of researcher bias. The actual faults and the properties of the tested software systems naturally had an effect on the results of this study, but the similarity of the failure type distributions compared to earlier published failure data [60], [64] supports the assumption that the tested systems were not anomalous in terms of the fault distributions.

Since this study focused mainly on the knowledge in the failure recognition part of testing, we feel it is appropriate to remind the reader that 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 the knowledge that is required to design effective tests in order to provoke failures, but we presented examples of such application of knowledge in test design, where testers applied their knowledge for targeted attacks to certain risks.

Based on our data, we were not able to analyze whether the revealed failures in the observed sessions were a representative sample of the actual failure distribution, in terms of failure types and provocation difficulty, in the tested products. Unfortunately, longitudinal follow-up data on how the reported defects were later processed and resolved were not available to us, which is an important topic to be studied in future research since ET seems to produce relatively high numbers of findings, but the question of how important and beneficial these findings are remains unanswered.

6 Conclusions and Future Work

In this paper, we reported the results of an empirical observation study of the role of testers' knowledge in provoking and recognizing failures in the context of exploratory software testing in industry. Our results show that testers are capable of recognizing different types of software failures 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. 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 than how the test-case-based paradigm understands the software testing activity. Our results show that the ways of applying knowledge in exploratory testing involve evaluating the overall behavior of the system, comparing the features with other features, and applying knowledge of earlier versions. Knowledge is sometimes applied for test design to design targeted attacks to known risks or customer problems.

When the knowledge is applied to test design, it is similar to traditional test design. However, in the exploratory approach, testers are able to learn how the tested software works during the testing and combine that new knowledge with some experience-based knowledge of past failures, or real use scenarios to design new tests based on their knowledge. This test design happens in parallel with the testing, not as a separate test design and documentation round. 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 emerge as a result of exploratory testing activities of parts of the software that are not in the actual focus of the current ET activity. This further emphasizes the diverse and creative opportunities of the exploratory testing approach.

Based on the combined analysis of the knowledge and failure data, we state four testable hypotheses.

- H_1: Exploratory testing is more efficient than test-case-based testing in recognizing functional failures since the testers are able to use different types of personal knowledge in designing tests and recognizing failures on the fly, rather than restricting their focus to predefined tests.
- H_2: A major factor explaining the efficiency of exploratory testing is the large number of failures (windfall failures) incidentally found outside the actual target features of the testing activities.
- H_3: A large fraction of the failures in software applications and systems do not require complicated test designs or descriptions to be provoked and recognized.
- H_4: The majority of failures recognized by the ET approach and related to domain knowledge are straightforward to provoke, whereas failures related to system knowledge or generic software development knowledge are more complicated to provoke in terms of the number of interacting conditions.

In conclusion, our results on the one hand suggest that the exploratory testing approach could be effective even with low levels of testing experience. On the other hand, it seems 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.

6.1 Future Work

More research is needed to better understand various aspects of the ET approach and the role of experience and knowledge in it. In this study, we focused on knowledge in failure recognition and stated multiple testable hypotheses that should be tested in other studies on ET. The exploratory testing strategies and techniques should be studied in greater depth in future research. We plan to continue studies on how testers apply heuristics and knowledge to design tests and to interpret and investigate results and failures.

ACKNOWLEDGMENTS

The authors would like to thank the 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.* John Wiley & Sons, 1979.
[2] B. Beizer, *Software Testing Techniques.* Van Nostrand Reinhold, 1990.
[3] C. Kaner, J. Falk, and H.Q. Nguyen, *Testing Computer Software.* John Wiley & Sons, Inc., 1999.
[4] L. Copeland, *A Practitioner's Guide to Software Test Design.* Artech House Publishers, 2004.
[5] J.B. Goodenough and S.L. Gerhart, "Toward a Theory of Test Data Selection," *IEEE Trans. Software Eng.,* vol. 1, no. 2, pp. 156-173, Mar. 1975.
[6] C. Andersson and P. Runeson, "Verification and Validation in Industry—A Qualitative Survey on the State of Practice," *Proc. Int'l Symp. Empirical Software Eng.,* pp. 37-47, 2002.
[7] S. Ng, T. Murnane, K. Reed, D. Grant, and T. Chen, "A Preliminary Survey on Software Testing Practices in Australia," *Proc. Australian Software Eng. Conf.,* pp. 116-125. 2004,
[8] E. Engström and P. Runeson, "A Qualitative Survey of Regression Testing Practices," *Proc. 11th Int'l Conf. Product-Focused Software Process Improvement,* 2010.
[9] M. Fewster and D. Graham, *Software Test Automation.* Addison-Wesley, 1999.
[10] C. Kaner, J. Bach, and B. Pettichord, *Lessons Learned in Software Testing.* John Wiley & Sons, Inc., 2002.
[11] S. Berner, R. Weber, and R.K. Keller, "Observations and Lessons Learned from Automated Testing," *Proc. Int'l Conf. Software Eng.,* pp. 571-579, 2005.
[12] A. Beer and R. Ramler, "The Role of Experience in Software Testing Practice," *Proc. Euromicro Conf. Software Eng. and Advanced Applications,* pp. 258-265, 2008.
[13] J. Bach, "Exploratory Testing," *The Testing Practitioner,* second ed., E. van Veenendaal, ed., UTN Publishers, pp. 253-265, 2004.
[14] A. Abran, J.W. Moore, P. Bourque, R. Dupuis, and L.L. Tripp, *Guide to the Software Engineering Body of Knowledge.* IEEE CS, 2004.
[15] A. Tinkham and C. Kaner, "Exploring Exploratory Testing," *Proc. Software Testing Analysis & Rev. Conf. East,* p. 9, 2003.
[16] J. Itkonen and K. Rautiainen, "Exploratory Testing: A Multiple Case Study," *Proc. Int'l Symp. Empirical Software Eng.,* pp. 84-93, 2005.
[17] J.A. Whittaker, *Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design.* Addison-Wesley Professional, 2009.
[18] J. Våga and S. Amland, "Managing High-Speed Web Testing," *Software Quality and Software Testing in Internet Times,* D. Meyerhoff, B. Laibarra, R. van der Pouw Kraan, and A. Wallet, eds., pp. 23-30, Springer-Verlag, 2002.
[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," *Proc. 10th Int'l Conf. 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 J.,* 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," *Proc. 19th Int'l Symp. Software Testing and Analysis,* pp. 231-240, 2010.
[24] P.N. Robillard, "The Role of Knowledge in Software Development," *Comm. ACM,* vol. 42, no. 1, pp. 87-92, 1999.
[25] "IEEE Standard Glossary of Software Engineering Terminology," IEEE Standard 610.12-1990, p. 83, 1990.
[26] 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," *Proc. Int'l Conf. Software Eng.,* pp. 602-611, 2007.
[27] 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, nos. 5/6, pp. 559-580, 2009.
[28] J.A. Whittaker, *How to Break Software: A Practical Guide to Testing.* Addison Wesley, 2003.
[29] L. Crispin and J. Gregory, *Agile Testing: A Practical Guide for Testers and Agile Teams.* Addison-Wesley, 2009.
[30] J. Bach, "Session-Based Test Management," *Software Testing and Quality Eng.,* vol. 2, no. 6, 2000.
[31] F. Houdek, T. Schwinn, and D. Ernst, "Defect Detection for Executable Specifications—an Experiment," *Int'l J. Software Eng. and Knowledge Eng.,* vol. 12, no. 6, p. 637, Dec. 2002.
[32] J. Itkonen, M.V. Mäntylä, and C. Lassenius, "Defect Detection Efficiency: Test Case Based vs. Exploratory Testing," *Proc. Int'l Symp. Empirical Software Eng. and Measurement,* pp. 61-70, 2007.
[33] L.H.O. do Nascimento and P.D.L. Machado, "An Experimental Evaluation of Approaches to Feature Testing in the Mobile Phone Applications Domain," *Proc. Workshop Domain Specific Approaches to Software Test Automation,* pp. 27-33, 2007.
[34] J. Itkonen, M.V. Mäntylä, and C. Lassenius, "How Do Testers Do It? An Exploratory Study on Manual Testing Practices," *Proc. Third Int'l Symp. Empirical Software Eng. and Measurement,* pp. 494-497, 2009.
[35] J. Pichler and R. Ramler, "How to Test the Intangible Properties of Graphical User Interfaces?" *Proc. First Int'l Conf. Software Testing, Verification, and Validation,* pp. 494-497, 2008.
[36] J. Itkonen, K. Rautiainen, and C. Lassenius, "Toward an Understanding of Quality Assurance in Agile Software Development," *Int'l J. Agile Manufacturing,* vol. 8, no. 2, pp. 39-49, 2005.
[37] J. Kasurinen, O. Taipale, and K. Smolander, "Test Case Selection and Prioritization: Risk-Based or Design-Based?" *Proc. Int'l Symp. Empirical Software Eng. and Measurement,* article 10, 2010.
[38] A. Tinkham and C. Kaner, "Learning Styles and Exploratory Testing," *Proc. Pacific Northwest Software Quality Conf.,* 2003.
[39] L. Shoaib, A. Nadeem, and A. Akbar, "An Empirical Evaluation of the Influence of Human Personality on Exploratory Software Testing," *Proc. 13th IEEE Int'l Multitopic Conf.,* 2009.
[40] R.T. Turley and J.M. Bieman, "Competencies of Exceptional and Nonexceptional Software Engineers," *J. Systems and Software,* vol. 28, no. 1, pp. 19-38, Jan. 1995.
[41] B. Adelson and E. Soloway, "The Role of Domain Experience in Software Design," *IEEE Trans. Software Eng.,* vol. 11, no. 11, pp. 1351-1360, Nov. 1985.
[42] S. Sonnentag, "Expertise in Professional Software Design: A Process Study." *J. Applied Psychology,* vol. 83, no. 5, pp. 703-715, 1998.

[43] J. Sandberg, "Understanding Human Competence at Work: An Interpretative Approach," *Academy of Management J.*, vol. 43, no. 1, pp. 9-25, 2000.
[44] 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.
[45] C. Engelke and D. Olivier, "Putting Human Factors Engineering into Practice," *Medical Device & Diagnostic Industry*, vol. 24, no. 7, July 2002.
[46] R. Merkel and T. Kanij, "Does the Individual Matter in Software Testing?" Swinburne Univ. of Technology, Centre for Software Analysis and Testing, Technical Report 2010-001, May 2010.
[47] A. Følstad, "Work-Domain Experts as Evaluators: Usability Inspection of Domain-Specific Work-Support Systems," *Int'l J. Human-Computer Interaction*, vol. 22, no. 3, pp. 217-245, 2007.
[48] 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.
[49] W. Howden, "Theoretical and Empirical Studies of Program Testing," *IEEE Trans. Software Eng.*, vol. 4, no. 4, pp. 293-298, July 1978.
[50] L. Baresi and M. Young, "Test Oracles," Technical Report CISTR-01-02, Dept. of Computer and Information Science, Univ. of Oregon, Eugene, Aug. 2001.
[51] J.A. Whittaker, "What Is Software Testing? and Why Is It So Hard?" *IEEE Software*, vol. 17, no. 1, pp. 70-79, Jan./Feb. 2000.
[52] A. Memon, I. Banerjee, and A. Nagarajan, "What Test Oracle Should I Use for Effective GUI Testing?" *Proc. 18th Int'l Conf. Automated Software Eng.*, pp. 164-173, 2003.
[53] V.R. Basili and R.W. Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Trans. Software Eng.*, vol. 13, no. 12, pp. 1278-1296, Dec. 1987.
[54] E.J. Weyuker, "On Testing Non-Testable Programs," *The Computer J.*, vol. 25, no. 4, pp. 465-470, 1982.
[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 Trans. Software Eng.*, vol. 18, no. 11, pp. 943-956, Nov. 1992.
[57] M.V. Mäntylä and C. Lassenius, "What Types of Defects Are Really Discovered in Code Reviews?" *IEEE Trans. Software Eng.*, vol. 35, no. 3, pp. 430-448, May/June 2009.
[58] A. Bondavalli and L. Simoncini, "Failure Classification with Respect to Detection," *Proc. Second IEEE Workshop Future Trends of Distributed Computing Systems*, pp. 47-53, 1990.
[59] D. Cotroneo, S. Orlando, and S. Russo, "Failure Classification and Analysis of the Java Virtual Machine," *Proc. 26th IEEE Int'l Conf. Distributed Computing Systems*, pp. 17-26, 2006.
[60] D.R. Wallace and D.R. Kuhn, "Failure Modes in Medical Device Software: An Analysis of 15 Years of Recall Data," *Int'l J. Reliability, Quality and Safety Eng.*, vol. 8, no. 4, pp. 351-371, 2001.
[61] E.T. Hvannberg and L. Law, "Classification of Usability Problems (CUP) Scheme," *Proc. Ninth Int'l Conf. Human-Computer Interaction*, pp. 655-662, 2003.
[62] N. Juristo and S. Vegas, "Functional Testing, Structural Testing and Code Reading: What Fault Type Do They Each Detect?" *Empirical Methods and Studies in Software Eng.*, Springer, 2003.
[63] B. Freimut, "Developing and Using Defect Classification Schemes," Research Report IESE-Report 072.01/E, Fraunhofer IESE, Kaiserslautern, 2001.
[64] D. Kuhn, D. Wallace, and A. Gallo, "Software Fault Interactions and Implications for Software Testing," *IEEE Trans. Software Eng.*, vol. 30, no. 6, pp. 418-421, June 2004.
[65] C. Seaman, "Qualitative Methods in Empirical Studies of Software Engineering," *IEEE Trans. Software Eng.*, vol. 25, no. 4, pp. 557-572, July/Aug. 1999.
[66] M.Q. Patton, *Qualitative Research and Evaluation Methods*, third ed. Sage, 2002.
[67] C. Seaman and V. Basili, "Communication and Organization: An Empirical Study of Discussion in Inspection Meetings," *IEEE Trans. Software Eng.*, vol. 24, no. 7, pp. 559-572, July 1998.
[68] T.C. Lethbridge, S.E. Sim, and J. Singer, "Studying Software Engineers: Data Collection Techniques for Software Field Studies," *Empirical Software Eng.*, vol. 10, no. 3, pp. 311-341, 2005.
[69] H. Wu, Y. Guo, and C.B. Seaman, "Analyzing Video Data: A Study of Programming Behavior under Two Software Engineering Paradigms," *Proc. Int'l Symp. Empirical Software Eng. and Measurement*, pp. 456-459, 2009.
[70] A. Höfer, "Video Analysis of Pair Programming," *Proc. Int'l Workshop Scrutinizing Agile Practices or "Shoot-Out at the Agile Corral,"* pp. 37-41, 2008.
[71] L. Prechelt, U. Stärk, and S. Salinger, "7 Types of Cooperation Episodes in Side-by-Side Programming," Technical Report B-08-17, Freie Universität Berlin, Institut für Informatik, Germany, Dec. 2008.
[72] 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.
[73] J. Hughes and S. Parkes, "Trends in the Use of Verbal Protocol Analysis in Software Eng. Research," *Behaviour & Information Technology*, vol. 22, no. 2, p. 127, 2003.
[74] A.L. Strauss and J.M. Corbin, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Sage Publications, 1998.
[75] J.C. van Niekerk and J.D. Roode, "Glaserian and Straussian Grounded Theory: Similar or Completely Different?" *Proc. Ann. Research Conf. South African Inst. of Computer Scientists and Information Technologists*, pp. 96-103, 2009.
[76] M.V. Mäntylä, J. Itkonen, and J. Iivonen, "Who Tested My Software? Testing as an Organizationally Cross-Cutting Activity," *Software Quality J.*, vol. 20, no. 1, pp. 145-172, 2012.

Juha Itkonen received the DSc degree from Aalto University in 2012. He works as a postdoctoral researcher in the Department of Computer Science and Engineering, School of Science, Aalto University, Finland. His research focuses on experience-based and exploratory software testing and human issues in software engineering, including quality assurance in agile context. He conducts empirical research relying both on qualitative and quantitative methods and prefers research in industrial context. He is a member of the IEEE.

Mika V. Mäntylä received the DSc degree in software engineering from the Helsinki University of Technology, Finland, in 2009. He is a postdoctoral researcher at Aalto University, Finland. In 2010, he was a visiting scholar at the Simula Research Laboratory, Oslo, Norway. In 2011-2012, he was a postdoctoral researcher at Lund University, Sweden. His previous studies have appeared in journals such as the *IEEE Transaction on Software Engineering*, *Empirical Software Engineering Journal*, and *Information and Software Technology*. His research interests include empirical software engineering, software testing, human cognition, defect databases, and software evolution. He is a member of the IEEE.

Casper Lassenius received the PhD degree in computer science from the Helsinki University of Technology, Finland. He is a professor (pro tem) in the Department of Computer Science and Engineering, School of Science, Aalto University. His research interests include software product development, agile methodologies, and global software development. He is a member of the IEEE and ACM.