FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Sampo Niittyviita

The Cost Efficiency of Exploratory Testing: ISTQB Certified Testing Compared with RST

Bachelor's Thesis
Degree Programme in Computer Science and Engineering
November 2016

Niittyviita S. (2016) The cost efficiency of exploratory testing: ISTQB certified testing compared with Rapid Software Testing. University of Oulu, Degree Programme in Computer Science and Engineering. Bachelor's Thesis, 33 p.

ABSTRACT

The research of software testing and the practices of software testing in the industry are separated by gaps in some areas. One such gap regards Exploratory Testing (ET). ET is probably the most widely used software testing approach in the industry, yet it is lacking research and many of the manuals of software engineering either ignore or look down on it. In addition, ET has the absence of widespread methodology and teaching. Rapid Software Testing (RST) is a complete testing methodology attempting to integrate exploratory testing into the whole spectrum of testing. It has a different basis for testing than the traditional way of testing, which dominates the textbooks and the certifications. International Software Testing Qualifications Board (ISTQB) has created the most successful scheme for certifying testers, having issued more than 470,000 testing certifications worldwide.

For this thesis, experiments were performed using RST as a basis. The results were documented as separate bugs and they were reconstructed into ISTQB test cases, which would be required to detect the discovered bugs. These reconstructed test cases were analysed for determining factors affecting test case documentation effectiveness on finding bugs or particular tests benefitting from testing automation. Additionally, comparison was made in terms of time spent between the actual test process (RST testing) and the reconstruction of the test cases (ISTQB test design) to give indication of the required amount of time for each method.

The testing phase of the experiments took 5 hours, while the reconstruction of the test cases took 4.5 hours. 33 % of the reconstructed test cases were identified to benefit considerably from test automation. In addition, three types of factors were identified, which have the potential to increase the amount of required test case documentation exponentially; external factors: test speed; platform factors: built-in browser functions and precision factors: level of detail for test cases.

Keywords: software testing, certificate, exploratory testing, Rapid Software Testing

Niittyviita S. (2016) Tutkivan testauksen kustannustehokkuus: ISTQB-sertifioitu testaus suhteessa RST-testaukseen. Oulun yliopisto, tietotekniikan tutkinto-ohjelma. Kandidaatintyö, 33 s.

TIIVISTELMÄ

Ohjelmistotestauksen tutkimuksen ja käytännön ohjelmistotestauksen välillä on aukkoja joillakin toiminnan osa-alueilla. Yksi tällainen aukko koskee tutkivaa testausta. Tutkiva testaus on yksi käytetyimmistä lähestymistavoista ohjelmistotestauksen toimialan parissa. Tästä huolimatta, tutkivasta testauksesta on hyvin vähän tutkimusta ja monet ohjelmistotestauksen käsikirjat joko välttävät aiheen tyystin tai vähättelevät sitä. Lisäksi, tutkivasta testauksesta puuttuu laajalle levinnyt metodologia sekä koulutus. Rapid Software Testing (RST) on testausmetodologia pyrkimyksenään integroida tutkiva testaus osaksi testauksen koko kirjoa. Sillä on erilaiset lähtökohdat testaukselle kuin perinteisellä testauksella, joka vallitsee kirjallisuutta sekä testauksen sertifikaatteja. International Software Testing Qualifications Board (ISTQB) on luonut menestyksekkäimmän järjestelmän ohjelmistotestaajien sertifioinnille. Sertifikaatteja on myönnetty yli 470,000 kappaletta.

Tämän työn empiirisinä kokeina suoritettiin ohjelmistotestausta käyttäen RST-metodologiaa lähtökohtana. Lopputulokset dokumentoitiin ohjelmistovirheinä, jotka sitten rekonstruoitiin riittäviksi testitapauksiksi löytämään kyseiset ohjelmistovirheet. Nämä rekonstruoidut testitapaukset analysoitiin niiden tekijöiden määrittämiseksi, jotka vaikuttaisivat testitapausten dokumentaation tehokkuuteen ohjelmistovirheiden löytämiseksi. Lisäksi, testausautomaatiosta selkeästi hyötyvät testitapaukset eriteltiin myös. Käytetty aika testaukseen ja rekonstruoimiseen kirjattiin vertailua varten.

Kokeiden suoritus kesti kokonaisuudessaan 9,5 tuntia, josta 5 tuntia kului testaukseen ja loput 4,5 tuntia löydettyjen ohjelmistovirheiden rekonstruoimiseen ISTQB-testitapauksiksi. 33% rekonstruoiduista testitapauksista tunnistettiin hyötyvän huomattavasti testausautomaatiosta. Lisäksi tunnistettiin kolme erityyppistä tekijää, joilla on potentiaalia lisätä testausdokumentaation määrää eksponentiaalisesti; ulkoiset tekijät: nopeus; alustan tekijät: selaimen toiminnot sekä tarkkuuden tekijät: testitapausten yksityiskohtaisuus.

Avainsanat: ohjelmistotestaus, sertifikaatti, tutkiva testaus, Rapid Software Testing

TABLE OF CONTENTS

FOREWORD

For accomplishing in the writing of this thesis, I would like to thank my supervisor Prof. Juha Röning and my advisor Christian Wieser. For ideas and their support, I would like to thank Jaakko Sakaranaho and Antti Niittyviita.

Oulu, 20.11.2016
Sampo Niittyviita

ABBREVIATIONS

ET	Exploratory testing
ISTQB	International Software Testing Qualifications Board
RST	Rapid software testing

1. INTRODUCTION

This thesis investigates the question of how RST and the core of ISTQB testing approaches compare in testing in terms of cost efficiency. As experiments, software testing with the RST approach is conducted. The goal of the experiments is to compare the cost efficiency of ISTQB test case testing compared to Rapid Software Testing. These experiments attempt to point out the complexity of the test cases required to find relevant bugs; the numerous potential variables of the test cases to be taken into account. The time spent on testing and writing test cases combined with the test case complexity are used as metrics to point out potential differences in effectiveness. High complexity and details are expected to lead into increased time consumption, which would convert into higher software testing costs.

Since software developers are humans, and all humans err, there are plenty of bugs in software. Bug detection and correction takes around 30-50% of the total time in software engineering projects [1 p.492]. If either of them is misconducted expenses can be expected. Research done by NIST estimated the annual cost of inadequate software testing to range from 22.2 to 59.5 billion dollars [2]. Another research estimated software testing being a major cause in the product failure costs, which exceed 10% of corporations' turnover [3]. The global cost of debugging software was 312 billion dollars per year, according to a research conducted in Cambridge University in 2013 [4]. Their study merely counted wages of the programmers. If these calculations are correct, the real costs of software bugs are even higher for the developers, for example, diminishing of the customer base through bugs, which leads to bad user interface experience. If bugs like these are not found before production, the costs can be high for the developer.

The later bugs are found in the software development process, the more expensive the software production costs become. There have been several studies on the defect resolution costs: the order of magnitude for the cost is tenfold in the acceptance phase and hundredfold in the production phase, compared to the inspection phase of a software project [5 p.417-418]. Since software testing is expensive and insufficient testing is even more expensive, there is demand for better testing approaches.

Testing is the set of activities conducted to facilitate discovery or evaluation of properties of one or more test items. A program can be assumed to contain errors and testing should be done as an attempt to find those errors. Testing can never find all of the errors in a program. In software testing a human being is used as a tool to perform complex tasks. Software testing can be divided into two categories: black-box testing and white-box testing. Software testing can also be divided into manual and automated testing.

In black-box testing the tested program is viewed in relation to the output of the program, and the internal behaviour of the program is ignored. Instead the software is tested from the viewpoint of a basic user of the software: the tested software is a black box, whereas white-box testing focuses at the program logic [6 p.16-17].

2. STATE OF THE ART

2.1. Software Testing Research

Surveyors of testing research and many others claim that the current state of software testing practices is insufficient [7, 8]. There are the following problems hindering software testing: gaps between research and industry; research is insufficient and methodologies, techniques and tools are in need of improvement [9, 10, 11, 12, 13, 14]. The state of software testing in industry, according to a 2002 study by NIST (National Institute of Standards and Technology) is described as follows: "Companies indicated that in the current environment, software testing is still more of an art than a science, and testing methods and resource are allocated based on the expert judgment of senior staff." [2 p.117] This seems to apply even in this day, although some people believe otherwise, such as Paul C. Jorgensen, the author of Software Testing: A Craftsman's Approach, who has believed that the art has transformed into a craft, as stated in the first edition of his book in 1995 [5 p.XXV].

Four goals can be be identified for software testing research [12]:

1. Universal test theory,
2. Test-based modeling,
3. 100% automated testing and
4. Efficacy-maximized test engineering.

There is one aspect of software testing research that has received very little emphasis: the human part of software development. It has not received much interest within the scopes of software testing research. A recent study states that only 5% of software engineering or development papers list a human related topic, which indicates that there are more than ten times more technology and process related topics [15]. With the low amount of studies listing human related topics, secondary studies lack them as well [16]. This could be identified as one of the biggest flaws of software testing. To fill this void, it has been suggested to combine quantitative and qualitative research methodologies, more utilization of different fields of science and even a proposal of a new concept: Behavioral Software Engineering [17]. A similar occurrence has already affected economics; behavioral economics has emerged, and it has gained much recognition. For example, behavioral economics has been embraced by the U.S. government since 15.9.2015 [18].

2.2. Schools of software testing

There are different schools, different approaches, on how to develop software effectively, such as Spiral or Agile development [19]. The trends in software development change over time, and the best method for software development might never be discovered. The problem with finding the best method for software development lies in the key component of software development: the human. Software

development works around using the human as a resource as effectively as possible to perform the various tasks related to software development. The case is the same for software testing, as software testing uses the human resource as well and develops hand in hand with software development. The nature of both of these subjects is in a similar state as many social sciences: lack of consilience; there exists a multitude of theories, which can be conflicting and none of them clearly stands out as the best theory. An example of such a field of social science is psychology. There are many different schools of psychology, such as: Structuralism, Functionalism, Psychoanalysis, Behaviorism, Gestalt Psychology, and Humanistic Psychology [20 p. 261, 322, 369, 437, 491, 560]. As there are different schools of psychology, and different schools of software development, there can be grounds to classify testing to schools of software testing [21].

Schools of software testing is an idea introduced by James Bach and Cem Kaner and it was first formalized by Bret Pettichord[22, 21]. A school of software testing is defined by its intellectual affinity, social interaction and common goals; not by any common doctrine or specific technique. In essence, the umbrella term schools of software testing is a classification of the different standpoints related to software testing. This classification can be used as a tool to get an idea of the differences in software testing and their underlying reasons, even if the descriptions were flawed in some ways. In an illustration of the schools of software testing by Bret Pettichord, who has written the book Lessons Learned in Software Testing: A Context-Driven Approach alongside with James Bach and Cem Kaner. the schools are divided into five different schools: analytic, standard, quality, context-driven, and agile (Table 1) [21]. In relevance to the topic of this thesis, the core of International Software Testing Qualifications Board could be used as an example of the standard school, while Rapid Software Testing labels itself as a context-driven approach. This classification could be seen as a manifestation of a gap between the research and the industry.

Table 1. Schools of Software Testing

School	View of testing
Standard	Testing is a way to measure progress with emphasis on cost and repeatable standards. Prevalent in tester certifications (e.g. ISTQB) and standards (e.g. IEEE).
Quality	Emphasis on the test process, policing developers and acting as the gatekeeper.
Context-Driven	Emphasis on people and seeking bugs that stakeholders care about
Analytic	Testing is a rigorous and technical task. Common in academia.
Agile	Testing is used to prove that development is complete. Emphasis on automated testing.

2.3. ISTQB and RST

Although software testing has seen a growing interest, testing remains to be the soft spot of software development [6]. Software testing is an area of software development, which has a very minor and trivial part in curricula of software engineering. As a manifestation of this, internationally, the most prominent universities provide only an overview on the subject of software testing [23] This is the case, even though a great deal of resources, up to 50%, are allocated to testing in software development [1, 6]. As the teaching of testing is inadequate, additional teaching about software testing is sought. Employers often send their software testers to software testing courses or require them to acquire software testing certifications. The most prominent certification is by International Software Testing Qualifications Board (ISTQB), it has issued over 470,000 certifications worldwide [24]. The purpose of ISTQB certification in terms of testing skills is to provide a standard framework for testing and professionally qualifying the testers. [25 p.67]. They hold a traditional approach to software testing: emphasising manual testing and vast documentation and relying on strict following of test "roadmaps" to gain confirmation on the tested software (Figure 1) [26].

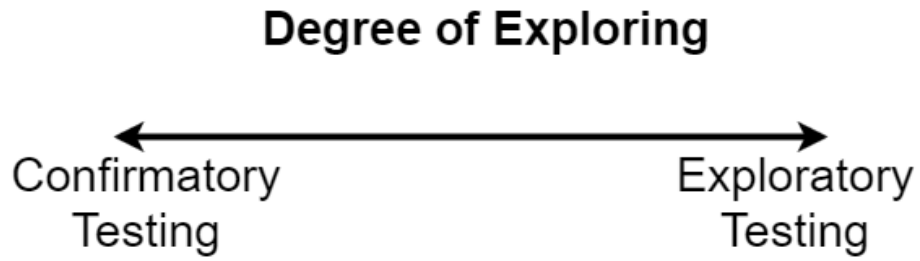

Figure 1. Degree of Exploring; software testing can be represented in degree of exploring.

On the other side of the testing spectrum (Figure 1), there is a software testing approach, which emphasises the human side of testing: exploratory testing (ET). An individual tester may fall anywhere within the degree of exploring spectrum depending on how much emphasis is put on confirmation and exploration. There is variation between individuals, especially in the side of ET [27]. ET is a testing approach, which is practiced in many different variations: two different testers may use entirely different paths to achieve the same goal for testing, given the same testing charter [27]. This could be attributed to various factors, including experience, specific skills, knowledge and personality [27]. The variations are furthermore reinforced by the lack of a uniform methodology for ET, and since there is an insignificantly small amount of research about it [27]. Kaner has identified nine different styles of exploration:

1. "hunches" (past experiences, recent changes),
2. models (architecture diagrams, bubble diagrams, state tables, failure models, etc.),
3. examples (use cases, feature walkthroughs, etc.),

4. invariances (tests that change things that should have no impact on the application),
5. interference (finding ways to interrupt or divert the program's path),
6. error handling (checking that errors are handled correctly),
7. troubleshooting (bug analysis, test variance when checking that a bug was fixed),
8. group insights (brainstorming, group discussions of related components, paired testing) and
9. specifications (active reading, comparing against the user manual, using heuristics) [27].

For the core of ET, the following activities could be considered to apply: valuing the tester's knowledge and skills, focusing in finding defects usually at the expense of vast documentation, confirmation and strict test "roadmaps" [28, 29]. Although ET is probably the most widely used practice[1] in software testing [30 p.89], it is often overlooked [5 p 373-374] and ignored completely [6] or only mentioned in passing [30 p.89]. While ET is more of a software testing approach, rather than a methodology, there have been attempts to formalise ET; James Bach, one of the advocates of ET, has created a software testing methodology called Rapid Software Testing (RST) [31]. RST is not only about ET; it is only a part of it, yet RST could be considered a framework for ET or formalised ET.

[1] Guide to the software engineering body of knowledge (SWEBOK) classifies Exploratory testing and ad hoc testing as separate entities [30 p.89], yet the Context-Driven School does not make any distinction between them [29].

3. THE STRUCTURE OF ISTQB CERTIFIED TESTING

Syllabi of ISTQB consists of different levels: foundation, advanced, and expert levels. In addition, there are extensions for agile testing and model-based testing [32]. Each one of them have individual certifications. There are also new certifications in development for ISTQB, which have not yet been published; automotive tester, usability tester, performance tester, and organizational and advanced level for agile tester extension [32]. These certifications under development will likely broaden the certification in some levels. The foundation level is the first level of the ISTQB syllabi on which the other levels are built upon. For the purpose of this thesis, which is on manual black-box testing, foundation level establishes the platform on how manual testing should be performed, as is stated in the foundation level syllabus [25]. The foundation level contains information about the basics of testing, the structure of testing, test techniques, test design techniques, tools for testing, test management and different levels and types of testing in software life cycle [25]. Other than test case testing itself, which is mostly black-box testing, white-box testing and code reviews are addressed as well. The different test levels, e.g. acceptance testing and integration testing, and the relation of testing with different software development models have their own sub-chapters in the syllabus. In essence, the foundation level is about manual testing, mostly focusing on different parts of testing with test cases.

Test case is the rudimentary piece of ISTQB testing. In order to perform testing properly, test cases are to be made. Only if there are heavy time constraints, should they be left aside [25]. The testing is performed via making these test cases, which are used as “road maps”. The testing is then performed precisely according to their instructions [25]. The test cases are designed with several test design techniques: specification based techniques (black-box techniques), structure-based (white-box techniques) or experience-based techniques. These types of techniques furthermore consist of several individual techniques, e.g. equivalence partitioning or boundary value analysis. With these techniques, the test cases are to be designed and written into test cases with test procedure specifications. Test design is one part of the whole test process in ISTQB testing.

Fundamental test process in ISTQB testing consists of activities: test planning and control, test analysis and design, test implementation and execution, evaluating exit criteria and reporting and test closure activities [25]. The activities are shown and described in Figure 2. The activities are sequential, yet they can overlap or take place concurrently. For grasping the concept of ISTQB testing in more detail, the definition of testing according to ISTQB gives an overview of it.

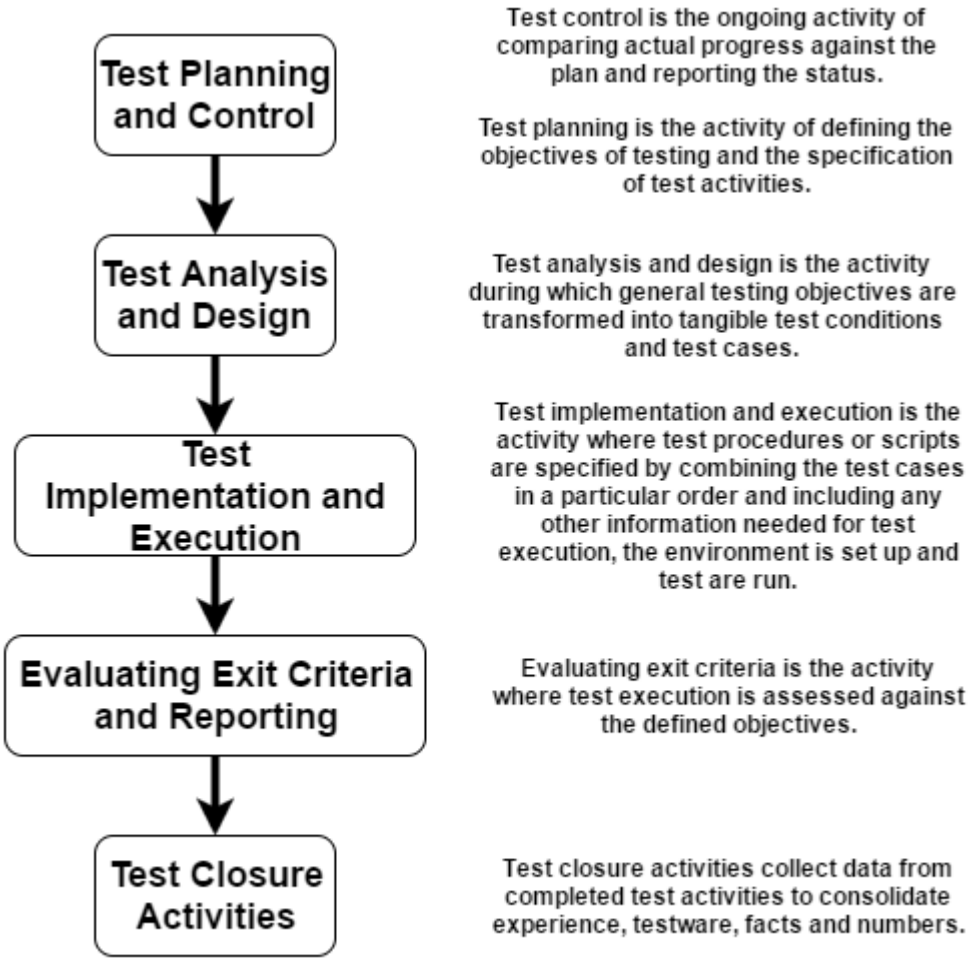

Figure 2. Fundamental test process of ISTQB testing.

3.1. The Definition of Testing According to ISTQB

ISTQB defines testing as: "The process consisting of all life cycle activities, both static and dynamic, concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects." [33 p.12]. Furthermore, the definition can be broken into two parts: testing as a process, and the objectives for testing. This division is explained in foundation level syllabus of ISTQB.

3.1.1. Testing as a Process

Testing is a process consisting of a series of activities. These activities take place throughout the software development cycle. The thought process of designing tests in the early stages of software development cycle can help to prevent defects from being

introduced into code. The test basis includes documents such as requirements and design specifications.

Testing can be either static or dynamic; in dynamic testing, software code is executed to demonstrate the results of running tests, contrary to static testing, where testing is done without executing any code. This type of testing includes reviewing of documents and source code, and static analysis.

The testing process also consists of planning, preparation, and evaluation. The planning activities take place before and after test execution. The plans are made to manage: what is wanted to perform, to control the test activities, to report on testing progress and the status of the software under test, and to be able to finalize or close testing when a phase completes. The preparation is done to choose what testing should be done, by selecting test conditions and designing test cases. The evaluation is done to check results of the executed tests, and to evaluate the software under test and the completion criteria, which helps to decide, whether the product has passed the tests.

As an addition to the actual testing of the code, there are other related material to be tested as well. This material includes requirements, design specifications, and test related documents such as operation, user and training material. [33 p.13-15]

3.1.2. Objectives for Testing

The second part of the definition covers the objectives of testing. The reasons for testing are either: determining that specified software requirements are satisfied, to demonstrate the software is fit for its purpose, or to detect defects.

Some of the testing is done to determine whether the product satisfies specified requirements. For example, the design is reviewed if it meets requirements and then some code may be executed to check that it meets the design. This is done to help stakeholders judge the quality of the product and decide whether it is ready for use. However, there is a possibility that the specified requirement has been specified wrongly or incompletely. In order to discover these faults, the software product must be demonstrated to meet the purpose of the software. Software testing is a means of detecting faults or defects that in operational use will cause failures. [33 p.13-15]

4. THE STRUCTURE OF RAPID SOFTWARE TESTING

Rapid Software Testing (RST) is a complete testing methodology developed by James Bach [31]. The methodology is based on the principles of the book Lessons Learned in Software Testing: a Context-Driven Approach, by Cem Kaner, James Bach, and Bret Pettichord. According to the creator of RST, the goal of RST is to eliminate unnecessary work, assuring everything gets done, and trying to speed the software project through testing. RST attempts to produce good results more quickly and less expensively over traditional testing [31]. RST puts a lot of responsibility for the person performing the testing and on his/her knowledge regarding testing and the software at hand. The testing in RST has a cyclic structure, where testing is constantly re-optimized with heuristic methods. The structure is captured in RST framework (Figure 3) [31]. RST attempts to formalise different aspects of testing into a teachable structure.

Different people define testing slightly differently, the definition for testing according to RST is as follows: “Testing is acquiring the competence, motivation, and credibility to create the conditions necessary to evaluate a product by learning about it through exploration and experimentation, which includes to some degree: questioning, study, modelling, observation and inference, including operating a product to check specific output so that you help your clients to make informed decisions about risk. And perhaps help make the product better, too.” [31]

The Rapid Software Testing methodology includes test strategies, different testing skills, test planning and heuristics, while relying heavily on practical examples. RST highlights the skills of the tester as the core element in software testing. The methodology is about providing the tester with necessary tools and skills to act as an individual tester within an organization. Differentiation from more traditional ways of testing include the following: RST does not use test procedure specifications as its basis, instead they are separated from testing as a different entity. RST classifies that type of testing as “checking” [31]. The differentiation factor between checking and testing is stated to be learning. RST claims that checking does not necessarily make the tester learn about the product, rather the checking provides information about the software whether the software product can work, yet it does not provide information whether it will work.

Most of the testing in RST is performed in a cyclic structure, where testing story is developed to provide information about the status of testing (Figure 3) [31]. To develop this testing story, the software product is studied with any useful or necessary means from which experiments are designed by synthesizing ideas and mechanisms to systematically question the status of the product. These are then formed into test procedures to be performed as experiments. The experiments translate into explicit and tacit test results, which are interpreted by the tester to make sense of the test process, progress and results to render them into a testing story. This process is enhanced by the skills, heuristics of the tester, while the tester solicits the help of others involved in the project.

Outside the cyclic structure lies the planning of the testing [31]. The planning used in RST is labeled as context-driven planning, it is performed to determine test strategy, test logistics and test products. Test strategy is used to determine the coverage of the

product to be tested. Test logistics is used to determine how and when to apply resources to execute the test strategy, including how to coordinate with other people on the project and how the tasks are assigned between other participants. Test products are the materials and results, e.g. bug reports, test report and test scripts, which are produced and visible to the clients of testing.

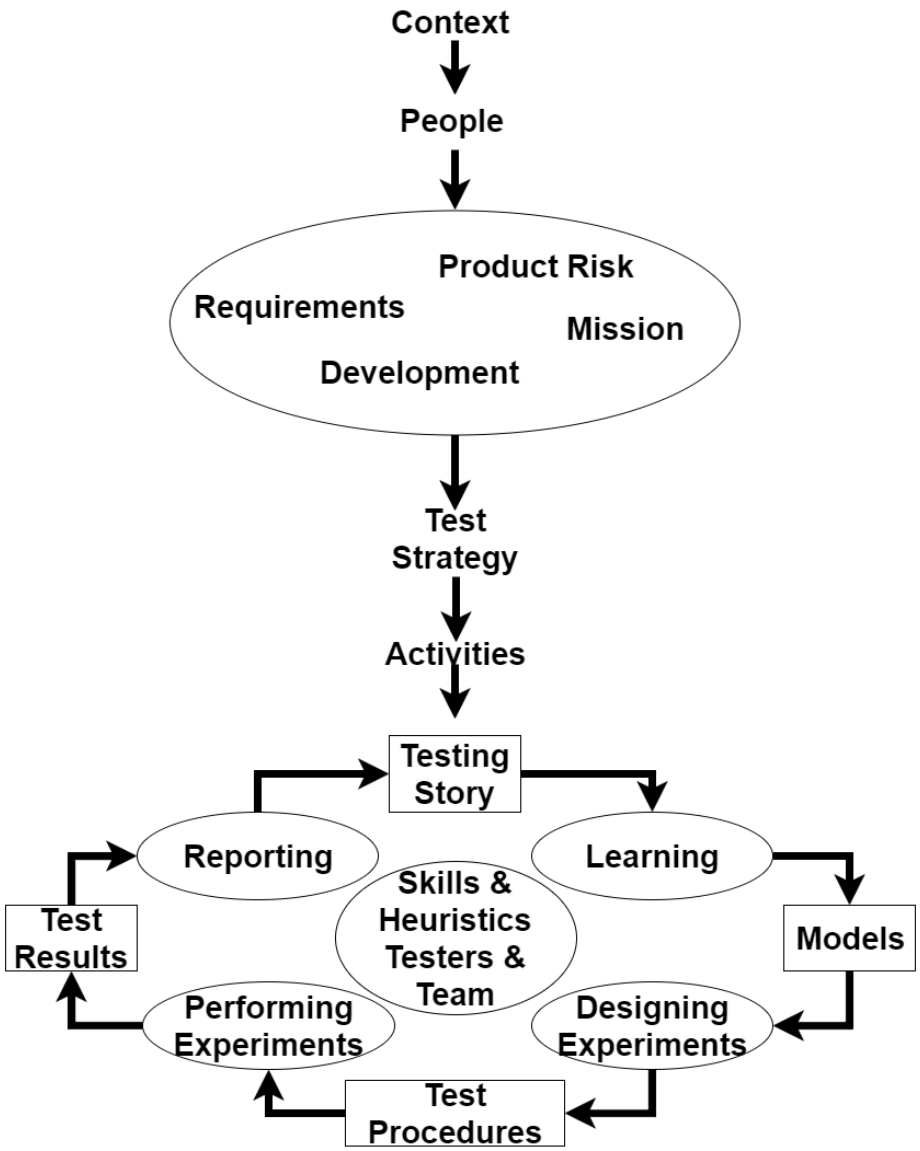

Figure 3. A Rapid Software Testing Framework.

5. DIFFERENCES IN ISTQB CERTIFIED TESTING METHODS AND RAPID SOFTWARE TESTING

5.1. Position towards Test Cases

The most noticeable difference in ISTQB and RST lies in the test cases: the testing process and the test case design are clearly separated in ISTQB, whereas in RST, they are usually combined together. ISTQB names pesticide paradox as one of its problems. About the portrait of the problem and its resolution, ISTQB states: “If the same tests are repeated over and over again, eventually the same set of test cases will no longer find any new bugs. To overcome this 'pesticide paradox', the test cases need to be regularly reviewed and revised, and new and different tests need to be written to exercise different parts of the software or system to potentially find more defects.” [25 p.14] The Optimal time, for abandoning and creating new test cases, is nearly impossible to determine. In RST, pesticide paradox is not necessarily a noteworthy problem, since RST does not encourage creating written test cases, rather exploratory testing is often preferred [31].

As for the definitions of RST and ISTQB testing, the concepts of testing are very different. While ISTQB puts a great deal of focus on checking the specifications and whether they achieve their goal, RST values exploration and experimentation highly in testing. Although ISTQB acknowledges that ET is capable of finding bugs, which may not be detected by conventional means, it provides little to none insight into how to actually perform it. Issues other than those related to Agile testing, exploratory testing is discussed at the conceptual level in [25 p.43] and in relation to test management in [34, 35]. ET is claimed to be most useful in situations with inadequate specifications and severe time pressure, or in order to complement more formal testing [25 p.43]. Furthermore, due to these points, in ISTQB Agile Tester Extension, exploratory testing is claimed to be important in Agile projects [36 p.36-38]. Only in the Agile extension, there is more descriptive look on the practice of ET.

The key differences between RST and ISTQB testing are listed in Table 2. The table is based on the ISTQB foundation level syllabus and RST course materials [25, 31].

Table 2. Key differences of ISTQB and RST

	ISTQB	RST
Structure	More linear structure.	More cyclic structure.
Definition of bug	Bug is an error in the program code, or in a document.	A bug is anything about the product that threatens its value (not necessarily an error).
Formality	More formal.	Less formal.
Exploratory testing (ET)	ET is a complementary testing method, useful, when there are heavy time constraints.	ET is a valuable testing method. Encourages to apply ET in testing.
Test procedure specification	Substantial part of software testing.	Not necessarily required.
Documentation	Detailed documentation of test cases and their results even if they do not discover defects.	Detailed procedural documentation is expensive and largely unnecessary. Concise documentation minimizes waste.
Automation	Test automation is a part of software testing.	Test automation is a part of software testing.

5.2. Exploratory Testing

The definition of exploratory testing can be construed by referring to the two inventors of ET [37]: James Bach and Cem Kaner. James Bach phrases exploratory testing as "simultaneous learning, test design and test execution" [29]. Whereas Cem Kaner defines exploratory testing as "a style of software testing that emphasizes the personal freedom and responsibility of the individual tester to continually optimize the value of her work by treating test-related learning, test design, test execution, and test result interpretation as mutually supportive activities that run in parallel throughout the project**"** [38].

The concept of ET was first introduced by Cem Kaner in 1984, and since then the concept has evolved [38]. The concept of ET has been acknowledged even before that, yet the methods of ET were not usually described or presented any further besides a mention [39]. ET was considered as 'ad-hoc' or error guessing method. It is to be noted that many still consider ET and ad-hoc testing as separate entities [30 p.89], whereas James Bach for instance considers them as the same [29].

ET is distinguished as "Perhaps the most widely practiced technique...", according to Guide to the Software Engineering body of knowledge (SWEBOK) [30 p.89] Exploratory testing is practiced in many different ways [27]. This can be attributed to lack of consensus, research and literature; the optimal way to perform ET has not been discovered. Quite often exploratory testing is labeled as unreliable and insufficient

method for testing. The most cited literature regarding software testing usually ignore [6] or deprecate exploratory testing [5 p.369-374]. As such, this is no wonder, as the descriptions of the process involved in ET have previously been rather vague [40, 29].

Effectiveness of ET is considered to be dependent on testing skills and knowledge [30 p.89; 25 p.43; 28]. Research on ET, however, suggests that ET could be effective even with low levels of testing experience [41]. Exploratory testing is often considered as a complementary testing approach suitable for certain situations [28, 25].

There are rather few studies on exploratory testing, at least few studies, which mention exploratory testing [26].

Replicated studies have been made to determine the efficiency of ET compared to scripted testing (test case based testing) [42, 39]. These studies suggest that exploratory testing is significantly more effective than scripted testing. The time spent to find the bugs are within the same boundaries, yet, when accounting the time to document the test cases into the total time spent, ET turns out to be much more effective.

Prof. Juha Itkonen in his licentiate titled “Do Test Cases Really Matter? An Experiment Comparing Test Case Based and Exploratory Testing.” [43], demonstrates that predesigned and documented test cases find a significantly lower number of defects to ET. For this comparison, bugs found outside of the test cases were excluded, as 17.7% of defects found by scripted testing were actually found by exploring. Additionally, this work suggests that scripted testing produces a significantly higher number of false defect reports, i.e. incomprehensible, duplicate or non-existent defects.

6. EXPERIMENTS

6.1. Goal of the Experiments

The goal of the experiments was to compare the cost efficiency of ISTQB test case testing compared to Rapid Software Testing. These experiments attempted to point out the complexity of the test cases required to find relevant bugs; the numerous potential variables of the test cases to be taken into account. The time spent on testing and writing test cases combined with the test case complexity were used as metrics to point out potential differences in effectiveness. Higher complexity and details were expected to lead into increased time consumption, which would convert into higher software testing costs.

6.2. Experiment Setup

The software project, which testing was used as the experiments, had one day of testing per week throughout its development. Because of this, the software was already familiar to the tester, but the internal structure from the technical perspective, nor the specifications of the software were not familiar to the tester. No emphasis was put on the specifications; if contradictions to specifications were encountered; they were neglected as long as they did not pose any threat to the value of the software. The scope of the experiments was set as one day of testing with a few exceptions: the verification of the bugs found during the previous day of testing were removed from the experiment and the time spent on writing the daily test report were cut out. The time spent on testing was five hours, when the verification, the writing of the report, and the work not related to testing were discounted. The tester, performing the testing, had two months of prior testing experience.

The software was a web service used to buy a service accessible via browser. The service product had many different options to tailor the service for the customer's specific needs. The service could be ordered to any desired location in Finland. Only certain pre-defined parts of the software were set as the target of the tests; one type of the many services and additionally accounts setting for the account used to log in to the service. Google Chrome, Mozilla Firefox, Internet Explorer, and Opera were used to access the software. The target audience for the software is mainly business users.

The testing was performed in a test environment already set in place before the testing started. There were no testing tools in use. A brief description (a few sentences) of the new implemented features of the software and the scope of the tests for the day were used as the basis for the tests. No other planning was done prior to starting of the tests. For time management, a rough timetable was made during the testing, for the allocation of time for different features included in the tests.

6.3. Experiment Process

The experiments were performed using the Rapid Software Testing methodology with an exploratory testing approach. In Figure 4 the experiment's structure can be seen. The cyclic structure of the performed tests is also described in the Chapter 5: The Structure of Rapid Software Testing and Figure 3.

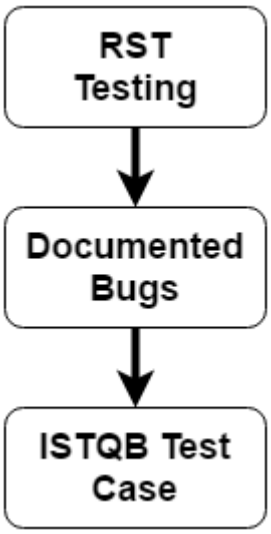

Figure 4. The order of the experiment's different phases (phases 1 to 3).

For the RST testing phase (phase 1 in Figure 4), the testing had a narrowed down list of features, which were to be tested. Whilst testing knowledge is gained about these features, previous testing experience is combined with this knowledge to constantly produce new test cases by making up new questions about the features at hand, and tested with different variations that can be found. When a bug is detected, time is spent to track down the cause of the bug, the exact steps to repeat it. Right away after the bug is detected and investigated thoroughly, a description of the bug and exact steps to reproduce the bug are written down as documented bugs (phase 2 in Figure 4). As the testing was exploratory testing, no test procedure was documented.

The results of the testing were documented as bugs with descriptions, images and test procedure specifications: their causalities solved. The aim, for this documentation, was to be as simple and clear as possible: only the bare minimum. The documented bugs were then formed into textbook test cases (phase 3 in Figure 4) according to International Software Testing Qualifications Board: "Test case is a set of input values, execution preconditions, expected results and execution postconditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement." (IEEE 610) [44]. The definitions of the terms used in the reconstructed test cases are included in Table 3. No distinction was made between postconditions and expected results. The Documented bugs were included as the actual results for each reconstructed test case. The documented actual results are used for comparing with the expected results to discover bugs.

The specifications of the software were not regarded by any means. Only the relevant test steps were produced to design the corresponding test cases. By

reconstructing these test cases, it has not been taken into account that for some of these bugs, there can be missing or flawed specifications. This makes it technically impossible for some of the test cases to be created through specifications. Due to this and the nature of the experiments, some of the reconstructed test cases may not be possible to be designed at all based on the specifications per se.

Table 3. Test Case Terminology

Term	Definition
Execution Precondition	Environmental and state conditions that must be fulfilled before the component or system can be executed with a particular test or test procedure.
Input Value	An instance of an input.
Expected Result	The behavior predicted by the specification, or another source, of the component or system under specified conditions.
Execution Postcondition	Environmental and state conditions that must be fulfilled after the execution of a test or test procedure.
Actual Result	The behavior produced/observed when a component or system is tested.

6.4. Results

The time spent on testing was five hours (phase 1 in Figure 4), including documentation. In total fifteen bugs were encountered, these bugs and their corresponding features are described and their quality estimated in Table 4. The qualities of the bugs were labeled either as;

- trivial: no importance for the use of the software or potentially not a bug;
- minor: little importance for the use of the software;
- major: great importance for the use of the software or
- critical: prevents the use of the software.

In total one bug was labeled as critical, two as major, nine as minor and three as trivial. In addition to these fifteen bugs, a few typos were found, which were not included in the results however. It is to be noted that a higher number of bugs were discovered during the testing than the average day in the test project, an estimate of the average number of bugs is around eight bugs per day for the project. The RST definition for a bug were used: “A bug is anything about the product that threatens its value.” [31]. The 15 bugs were reconstructed into ISTQB test cases. They were listed into Table 5 (phase 3 in Figure 4). The creation of these particular test cases took 4.5 hours from existing bug descriptions and images of the bugs. In theory, these could be the test cases that would fail during ISTQB testing. The test cases show roughly the amount of documentation needed to be written for finding fifteen bugs, excluding passing test

cases. Although some of the test cases could be easily combined into one, they were separated for making a clearer distinction between the expected results. As such they can be examined as separate entities. This amount of documentation excludes the documentation, in respect to finding the same amount of bugs, i.e. the passing test cases, which provide confirmatory information. Taking the level of detail of the reconstructed test cases into account, there should be many times more passing test cases compared to failing test cases.

Some of the reconstructed test cases hold a high level of complexity, while some of them are trivial and straightforward. When test cases are to be followed in specific detail, as is supposed in ISTQB, bugs 3, 4 and 5 demonstrates a glimpse in the potential test case complexity of a single widget that should be followed for them to be comprehensive. The single, simple function that is performed in all of the three test cases, contains much detail in their corresponding expected results. Whereas, for straightforward test cases; the reconstructed test cases labeled from 9 to 14 are such bugs. They are related to text field inputs. Other than bug 10, which is a keyboard shortcut bug, they involve error messages and character restrictions. The inputs and outputs for the test cases in question are simple and easy to produce. The bug 12 however probably is not a real bug, but if it was, it could be classified as a bug related to account security. Many software have similar features; not accepting password change, if the new password matches the current one.

Test cases, where performing actions in different orders of combination was essential, were: 2, 3, 4, 5 and 8. The function, related to bugs 3, 4 and 5, could be performed by a manual tester, when narrowed down to broad classes of combinations. Otherwise, both them and the bugs 2 and 8 would have an overwhelming amount of individual test phases and require very high amount of tracking, attention, effort and time for a manual tester to accomplish, because of the sheer amount of different combinations.

A Variable, which was included in one of the test cases, was speed. The steps in bug 1 had to be performed within a short time window for the bug to occur. In addition, this bug could only be reproduced with one specific browser. There were two bugs in total; bug 1 and 2, which were browser specific. As for minor bugs, 6 and 15 were such. They were easily detected UI bugs.

Table 4. Bug descriptions and bug qualities

#	Bug Description	Quality
1	This bug is related to a feature, which is used to buy a service with a number of different options. Selecting the options was part of one of the four phases in the procedure of ordering the service. These options can be included with the service by selecting the checkbox for each individual option. The bug occurred by selecting two of these checkboxes within a short time window. The bug occurred only with one specific browser.	Major
2	This bug is related to a feature, which is used to buy a service. There are four phases in ordering the service. Performing these phases in one particular order and reverting the changes, caused the current state of the procedure to be shown incorrectly.	Minor

3	This bug is related to a feature, which is used to buy a service. The cost of the different parts of the service are itemized separately. Some prices were separated with a comma and others with a dot, for example 1,0 € and 1.0 €.	Minor
4	This bug is related to a feature, which is used to buy a service. The cost of the different parts of the service are itemized separately. One name of the service was hard to distinguish from its price, since the name contained numbers, therefore part of the name could be read as part of the price.	Trivial
5	This bug is related to a feature, which is used to buy a service. One option was to buy the service in units of measurement (e.g. in weight), this unit of measurement was shown three times in a single sentence when only one would suffice.	Minor
6	This bug is related to a feature, which is used to buy a service. One option was to buy the service in units of measurement (e.g. in weight). The text field, where the unit of measurement was to be entered had the same unit of measurement printed twice.	Minor
7	This bug is related to a feature, which is used to buy a service. One text field, where the units of the product to be ordered were to be set, was not able to handle its value to bet set as 1. This value was shown incorrectly as 0 in the overview.	Major
8	This bug is related to a feature, which is used to buy a service. Selecting the target destination for the service caused an error with any specific location. This prevented the ordering of the service to this location.	Critical
9	This bug is related to account settings for the account used to log in to the web service. Password could be changed through this account setting page. The character limitation for the password were set to A-Z. The characters Ö, Ä, Å could be used in the password, however.	Minor
10	This bug is related to account settings for the account used to log in to the web service. Password could be changed through this account setting page. Trying to change the password by selecting keyboard key 'enter' caused the password change to fail, instead another setting in the same page was saved.	Trivial
11	This bug is related to account settings for the account used to log in to the web service. Password could be changed through this account settings page. Entering the new password for password change, the new password must be entered twice. Error message was missing when trying to enter a new password, which were not matching.	Minor
12	This bug is related to account settings for the account used to log in to the web service. Password could be changed through this account settings page. Trying to enter the current password as the new password was not prevented.	Trivial

13	This bug is related to account settings for the account used to log in to the web service. Password could be changed through this account settings page. Entering the new password field with restricted values or length, the error message was difficult to read.	Minor
14	This bug is related to account setting for the account used to log in to the web service. The account's phone number could be changed through this page. The phone number field had character restrictions, yet, any value could be entered as the phone number.	Minor
15	This bug is related to a feature which would list different kinds of documents. One of the documents had incorrect language code.	Minor

Table 5. Reconstructed ISTQB test cases, which are numbered from 1 to 15. The test cases' numbers in the steps, expected results and actual results represent different steps and their equivalent expected results and actual results.

Test Case	Steps	Expected Results	Actual Results
1	1. Access Feature X from the main menu. 2. Select a particular text field from a group of text fields. 3. Enter any value and press 'Enter' from keyboard. 4. Within a very small time window after pressing 'Enter' select any of the any of the checkboxes included with Feature X.	1. Feature X is accessed. 2. The text field is selected. 3. The value is written into the text field and the text field is deselected. 4. Checkbox is selected and the state of the widget showing the current progress in Feature X is updated to show the changes made in the text field and checkbox.	1. Feature X is accessed. 2. The text field is selected. 3. The value is written into the text field and the text field is deselected. 4. Checkbox is selected, but the widget could not handle the processing of two simultaneous changes in the progress: the progress for checking the checkbox cannot be seen in the widget. This occurs only on one specific browser.
2	1. Access Feature X from the main menu 2. Perform the four different steps of Feature X in different orders of combination. Additionally, revert the changes for every combination in every step of the order in question.	1. Feature X is accessed. 2. According to the current state of the progress, the progress widget shows the corresponding status in every different case.	1. Feature X is accessed. 2. Performing the fourth step as the first step and reverting back to the default state, the progress widget incorrectly shows the fourth step as completed. This occurs only on one specific browser.

3	1. Access Feature X from the main menu. 2. Select different combinations for dropdown list A and B.	1. Feature X is accessed. 2. For every different combination, the representation of the combination values match between other values in the overview widget.	1. Feature X is accessed. 2. Some of the different values represented in the overview use dot and some use comma.
4	1. Access Feature X from the main menu. 2. Select different combinations for dropdown lists A and B.	1. Feature X is accessed. 2. For every different combination, the values presented are clearly readable in the overview widget.	1. Feature X is accessed. 2. One specific name for a value (containing letters and numbers), in the overview, could easily be mixed with the value as there were no separators for the names and values.
5	1. Access Feature X from the main menu 2. Select different combinations for dropdown lists A and B.	1. Feature X is accessed. 2. For every different combination, the values are presented properly in the overview widget.	1. Feature X is accessed 2. In one specific information field in the overview, one value was shown three times in the same sentence, and one value had unnecessary brackets.
6	1. Access Feature X from the main menu 2. Enter any value for text field A	1. Feature X is accessed. 2. The same value is presented in a separate field.	1. Feature X is accessed. 2. The unit of measurement for the value is shown twice.
7	1. Access Feature X from the main menu. 2. Enter values to one specific text field of a group of three fields.	1. Feature X is accessed 2. The entered values correspond with the values in the overview widget.	1. Feature X is accessed 2. In the overview screen the value is shown as 0, when it is set as 1. This causes a problem calculating another value in the overview widget as well.
8	1. Access Feature X from the main menu 2. Select a value from dropdown list A, and finish the objective of Feature X. Repeat this process with all of the values in the list. (There are roughly one hundred items in list A)	1. Feature X is accessed 2. The function of Feature X can be completed with every item from list A.	1. Feature X is accessed 2. One specific item in the list caused an error and prevented the use of Feature X.

9	1. Access Feature Y from the main menu. 2. Select change password function. 3. Enter passwords of required length containing all different characters, including special characters, lower and upper case, to the new password field. 4. Select change password button.	1. Feature Y is accessed 2. Change password is selected 3. Password fields contain the information 4. The character limitations for passwords are presented in the error message; only characters A-Z and numbers are valid. No other characters are allowed for use.	1. Feature Y is accessed 2. Change password is selected 3. Password fields contain the information 4. Characters Ö, Ä and Å can be used in the password. These are not included within the character limitation A-Z.
10	1. Access Feature Y from the main menu 2. Select change password function. 3. Enter values within character limitations for the change password function. 4. Press 'Enter' from keyboard	1. Feature Y is accessed 2. Change password is selected 3. Password fields contain proper passwords 4. Password is changed and text "Password has been changed" can be seen next to the change password button	1. Feature Y is accessed 2. Change password is selected 3. Password fields contain proper passwords 4. Password is not changed, instead the entered passwords are wiped out, and a following info text can be seen: "User information has been saved"
11	1. Access Feature Y from the main menu. 2. Select change password function. 3. Fill the new password fields with a new password. Fill the old password field with a wrong password. 4. Select change password.	1. Feature Y is accessed. 2. Change password is selected. 3. The password fields are filled. 4. An Error message notifying of incorrect current password can be seen next to the change password button.	1. Feature Y is accessed. 2. Change password is selected. 3. The password fields are filled. 4. The error message is a generic error message: "Error in the function, if the error persists contact technical support." (Unlike other types of error messages)
12	1. Access Feature Y from the main menu. 2. Select change password function. 3. Fill both the old and the new password fields with the current password. 4. Select change password.	1. Feature Y is accessed. 2. Change password is selected. 3. The password fields are filled. 4. Password change is prevented and error message notifying that the user is trying to change the password to the same as	1. Feature Y is accessed. 2. Change password is selected. 3. The password fields are filled. 4. The current password is accepted as the new password.

		the current one.	
13	1. Access Feature Y from the main menu. 2. Select change password function. 3. Enter passwords outside the length and character restrictions. 4. Select change password to produce error message for both the old and the new password fields.	1. Feature Y is accessed. 2. Change password is selected. 3. The password fields are filled. 4. Error message is displayed and it is easy to read.	1. Feature Y is accessed. 2. Change password is selected. 3. The password fields are filled. 4. The error message is hard to read as the different restrictions are not separated; they are shown as a continuous stream of characters.
14	1. Access Feature Y from the main menu. 2. Select change user details function. 3. Enter phone number field with characters, which are not within the character restrictions: 0-9, +. 4. Select save button.	1. Feature Y is accessed. 2. Change user details is selected. 3. Any value can be typed into the phone number field. 4. Values within character restrictions are accepted as phone numbers and saved, and values not within the restrictions are not accepted.	1. Feature Y is accessed. 2. Change user details is selected. 3. All values can be typed into the text field. 4. All characters are accepted for the phone number and saved successfully: there are no character restrictions.
15	1. Access Feature Z from the main menu. 2. Go through the documents shown in the feature and that their language codes match their language.	1. Feature Z is accessed. 2. The language codes match their language.	1. Feature Z is accessed. 2. One of the languages had a language code, which does not exist.

6.5. Analysis

Comparing the time spent on finding the fifteen documented bugs to writing them into test cases, it took almost the same time to find the bugs and to reconstruct them into written test cases: 4.5 hours and 5 hours. Both in the testing and the reconstruction of the test cases, the responsible person was relatively inexperienced with no formal education in software testing, i.e. certification. The experience could be slightly in favor of the reconstruction part, as there were more knowledge and testing experience at the time of the reconstruction.

If the test cases are to be followed in specific detail, as is supposed in ISTQB, and the same level of detail, as in bugs 3, 4 and 5, is brought to all functions and aspects of the software, the amount of test cases would end up being massive, and large in their size. Even if it were somehow possible to produce all of the test cases with such

detail, it is very likely that a great deal of the target detail would be missing. In the alternative, where all the expected outcomes are documented in this detail, the size of the actual results would also increase greatly.

Bug 1 has the factor speed in its test steps. It is very hard to image a test case designer taking speed as a factor for test case design in this instance, or at all. This brings out two questions. Firstly, how to select the test cases, which need the factor speed? Secondly, what other similar factors are there to be taken into consideration? If there would be test cases taking speed or other similar factors into test case design, this would also generate a huge additional amount of test cases.

Since the tested software was browser based, there were bugs related to default browser functions and browser specific bugs. The bug 10 involved the keyboard key enter. This yet increases the potential scope of the test cases. Should there be keyboard key related test cases for every single part of the software? Browser and keyboard bugs could be grouped as platform bugs. These yet increase the number of dimensions to be included into test cases.

Some of the test cases, namely bugs 9 to 14, are probably easy to create using specifications as they are relatively simple and straightforward bugs. In instances like these, when there are multiple clear restrictions, test cases (or checklists) could prove to be effective. For some, there is not necessarily any specifications or they are too vague, some specifications may even be flawed. For example, specifications about, how the software should not behave. This can be problematic for creating test cases through test case specifications, as that is one of the main sources of test cases in ISTQB testing [25 p.37].

The number of combinations, required to test comprehensively the test cases 2, 3, 4, 5 and 8, is huge. In these particular test cases, it could result being cheaper to automate the testing of these, if not for the first test round, then in the regression testing. This suggests that 33% of the reconstructed test cases would benefit from automation.

Taking the factors brought up, there should be a massive amount of test cases, to find these fifteen particular bugs. Platform based test cases, extra factors for test cases and the level of detail, each individually add a theoretical potentiality to exponential increase in the volume of test case documentation. Thus, the time consumed would increase multifold, when the time consumed, solely for the writing of the fifteen test cases, was roughly the same as for finding and documenting the bugs.

Based on these results, the quality of the discovered bugs cannot be compared between RST and ISTQB testing methods, since no ISTQB testing was performed. However, research comparing exploratory testing and test case testing conducted by Prof. Juha Itkonen suggests that there are no big differences in the quality of discovered bugs between the two approaches [43].

7. CONCLUSION

The goal of the thesis was to compare the cost efficiency of International Software Testing Qualifications Board's (ISTQB) test case testing compared to Rapid Software Testing (RST). The thesis contains information on current state of the field of testing, core structures of ISTQB certificated testing and Rapid Software Testing and experiments for comparing the cost effectiveness of ISTQB test case testing and RST. As for the experiments; RST testing was conducted resulting in documented bugs, which were reconstructed into text book ISTQB test cases required to find those specific bugs. The experiments attempted to point out the complexity of the test cases required to find relevant bugs; the numerous potential variables of the test cases to be taken into account. The time spent on testing and writing the test cases combined with test case complexity were used as metrics to point out potential differences in effectiveness. Higher complexity and details were expected to lead into increased time consumption, which would convert into higher software testing costs.

The experiments resulted in a number of conclusions: variables were identified, which have the potential to increase the time spent; instances were identified, where test automation would be very beneficiary and other instances, where manual test cases could prove to be relatively efficient; and creating test cases is considerably more time consuming over the testing process.

There are test cases, which are very hard to design without interaction with the tested software, solely based on specification documents, such as taking the speed factor into test case design. For some of the created test cases manual testing is very effortful for the huge amount of test combinations. 33% of the reconstructed test cases were identified as benefiting considerably from test automation. Manual testing would likely result in wasted resources compared to automated testing in these cases, at least in the case of regression testing. Automation could reduce the required testing effort, at least in the long run. In instances where test cases are algebraic in nature and requiring many niche tests, such as text field inputs, written test cases could prove to be efficient. In these experiments the test process and the reconstruction of test cases took roughly the same time (5h and 4.5h). Since these fifteen reconstructed test cases can be expected to be only a portion of the total amount of test cases that are to be designed, test case design documentation can be expected to take a great deal of time to perform in contrast to the actual test process. It should be noted, however, that the time comparison does not fully represent the different test methodologies, as the experiments were not performed in a controlled environment and the test reconstruction does not equal real test design process. Increasing the number of variables encountered in this experiment for test case design, namely, environment factors: built-in browser functions; external factors: speed and the level of detail of the documentation, the test case documentation has the potentiality to increase exponentially in size. This can lead to major increases in software testing costs in ISTQB testing.

8. REFERENCES

[1] Ebert C & Dumke R. (2007) Software measurement. Springer.
[2] Tassey G. (2002) The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology, RTI Project 7007(011).
[3] Engel A & Last M. (2007) Modeling software testing costs and risks using fuzzy logic paradigm. J Syst Software 80(6): 817-835.
[4] Britton T, Jeng L, Carver G, Cheak P & Katzenellenbogen T. (2013) Reversible debugging software. Judge Bus.School, Univ.Cambridge, Cambridge, UK, Tech.Rep .
[5] Jorgensen PC. (2016) Software Testing: A Craftsman's Approach. : CRC press.
[6] Myers GJ, Sandler C & Badgett T. (2011) The Art of Software Testing. : John Wiley & Sons.
[7] Lee J, Kang S & Lee D. (2012) Survey on software testing practices. IET software 6(3): 275-282.
[8] Harrold MJ. (2000) Testing: A Roadmap. Proceedings of the Conference on the Future of Software Engineering. : ACM: 61-72.
[9] Glass RL, Collard R, Bertolino A, Bach J & Kaner C. (2006) Software testing and industry needs. IEEE Software 23(4): 55.
[10] Juristo N, Moreno AM & Strigel W. (2006) Guest Editors' Introduction: Software Testing Practices in Industry. IEEE Software 23(4): 19.
[11] Bertolino A. (2003) Software Testing Research and Practice. International Workshop on Abstract State Machines. : Springer: 1-21.
[12] Bertolino A. (2007) Software Testing Research: Achievements, Challenges, Dreams. 2007 Future of Software Engineering. : IEEE Computer Society: 85-103.
[13] Whittaker JA. (2000) What is software testing? And why is it so hard? IEEE Software 17(1): 70-79.
[14] Juristo N, Moreno AM & Vegas S. (2004) Reviewing 25 years of testing technique experiments. Empirical Software Engineering 9(1-2): 7-44.
[15] Lenberg P, Feldt R & Wallgren LG. (2015) Behavioral software engineering: A definition and systematic literature review. J Syst Software 107: 15-37.
[16] Garousi V & Mäntylä MV. (2016) A systematic literature review of literature reviews in software testing. Information and Software Technology 80: 195-216.
[17] Lenberg P, Feldt R & Wallgren L. (2014) Towards a Behavioral Software Engineering. Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering. : ACM: 48-55.
[18] Social and Behavioral Sciences Team 2016 Annual Report. [Online] URL: https://www.whitehouse.gov/sites/whitehouse.gov/files/images/2016%20Social%20and%20Behavioral%20Sciences%20Team%20Annual%20Report.pdf (11.11.2016).
[19] Pressman RS. (2005) Software Engineering: A Practitioner's Approach. : Palgrave Macmillan.
[20] Hergenhahn BR & Henley T. (2013) An Introduction to the History of Psychology. : Cengage Learning.
[21] Bret Pettichord. (2007) Schools of Software Testing. [Online] URL:

https://www.prismnet.com/~wazmo/papers/four_schools.pdf (24.7.2016).
[22] Kaner C. (2006) Schools of Software Testing. [Online] URL: http://kaner.com?p=15 (17.12.2016)
[23] Valle PHD & Barbosa EF. (2015) CS Curricula of the most Relevant Universities in Brazil and Abroad: Perspective of Software Testing Education. 2015 International Symposium on Computers in Education (SIIE). : IEEE: 62-68.
[24] International Software Testing Qualifications Board. [Online] URL: http://www.istqb.org/ (12.7.2016).
[25] Müller T, Friedenberg D & ISTQB WG Foundation Level. (2011) International Software Testing Qualifications Board: Foundation Level Syllabus. 2016 [Online] URL: http://www.istqb.org/downloads/syllabi/foundation-level-syllabus.html (15.8.2016).
[26] Itkonen J, Mäntylä M & Lassenius C. Test Better by Exploring: Harnessing Human Skills and Knowledge. IEEE Software 33(4): 90-96.
[27] Tinkham A & Kaner C. (2003) Learning Styles and Exploratory Testing. Proceedings of the Pacific Northwest Software Quality Conference.
[28] Itkonen J & Rautiainen K. (2005) Exploratory Testing: A Multiple Case Study. 2005 International Symposium on Empirical Software Engineering, 2005. : IEEE.
[29] Bach J. (2003) Exploratory testing explained. [Online] URL: http://jamescorne.com/cms/wp-content/uploads/2011/10/et-article.pdf (10.11.2016)
[30] Bourque P & Fairley RE. (2014) Guide to the Software Engineering Body of Knowledge (SWEBOK (R)): Version 3.0. : IEEE Computer Society Press.
[31] Bach J & Bolton M. (2016) Rapid Software Testing. Version (3.3.0), [Online] www.satisfice.com (17.12.2016).
[32] International Software Testing Qualifications Board Syllabi. 2016 (13.7.2016).
[33] Graham D, Van Veenendaal E & Evans I. (2008) Foundations of Software Testing: ISTQB Certification. : Cengage Learning EMEA.
[34] Smith M, Homès B & et al. (2012) International Software Testing Qualifications Board: Advanced Level Syllabus Test Manager. 2016 [Online] URL: http://www.istqb.org/downloads/category/10-advanced-level-syllabus-2012.html (27.09.2016).
[35] Bath G, Black R & et al. (2011) International Software Testing Qualifications Board: Expert Level Syllabus Test Management. 2016 [Online] (27.09.2016).
[36] Black R, Claesson A & et al. (2014) International Software Testing Qualifications Board: Agile Tester Extension. 2016 [Online] URL: http://www.istqb.org/downloads/send/5-agile-tester-extension-documents/41-agile-tester-extension-syllabus.html (27.9.).
[37] Kaner C. (2006) Exploratory Testing. Quality Assurance Institute Worldwide Annual Software Testing Conference. [Online] URL: http://mooc.ee/MTAT.03.094/2015_fall/uploads/Main/SE2014-handout10.pdf (13.9.2016)
[38] Kaner C. (2008) A tutorial in exploratory testing. [Online] URL: http://www.kaner.com/pdfs/QAIExploring.pdf (13.9.2016)
[39] Itkonen J, Mäntylä MV & Lassenius C. (2007) Defect Detection Efficiency: Test Case Based Vs. Exploratory Testing. First International Symposium on

Empirical Software Engineering and Measurement (ESEM 2007). : IEEE: 61-70.
[40] Kaner C, Bach J & Pettichord B. (2008) Lessons Learned in Software Testing. : John Wiley & Sons.
[41] Itkonen J, Mäntylä MV & Lassenius C. (2013) The role of the tester's knowledge in exploratory software testing. IEEE Trans Software Eng 39(5): 707-724.
[42] Prakash V & Gopalakrishnan S. (2011) Testing Efficiency Exploited: Scripted Versus Exploratory Testing. Electronics Computer Technology (ICECT), 2011 3rd International Conference on. : IEEE 3: 168-172.
[43] Itkonen J. (2008) Do test cases really matter? An experiment comparing test case based and exploratory testing. Licentiate, Helsinki University of Technology .
[44] ISTQB Glossary. 2016. [Online] URL: http://astqb.org/glossary/ (28.9.2016).