Exploratory GUI Application Testing and Productivity

Martin S. Zechner

University of Tampere
Department of Computer Sciences
Computer Science
M.Sc. thesis
December 2004

University of Tampere
Department of Computer Sciences
Computer Science
Martin S. Zechner: Exploratory GUI Application Testing and Productivity
M.Sc. thesis, 53 + 6 pages
December 2004

Abstract

In today's hectic and fast culture with short product cycles, less and less emphasis on explicit requirements and design it becomes generally more and more important to focus testing activities and to employ an efficient testing mindset. Badly tested software can very easily break a company's reputation.

Traditionally, graphical user interface (GUI) application design is quite difficult and the only reliable way to achieve good interfaces is through iteration, taking feedback from testers and end-users into consideration. It is exactly this iteration and change to the interface which makes GUI application testing demanding. Requirements may be vague and become clearer once the system evolves, or they may change altogether. Scripted testing, a testing method whereby test cases are generated early during software development, based on requirements and other relevant specifications and then executed, may prove to be an inefficient way on its own.

Exploratory testing, sometimes referred to as ad hoc testing and best described as "learning, test design, and test execution at the same time", may be one new testing approach. Currently there exists no research related to the possible benefits in terms of testing productivity of exploratory testing over scripted testing, except for some anecdotal evidence.

This work concentrates, by means of a case study, on whether the use of exploratory testing in addition to the more "traditional" script based testing, within the scope of GUI-based application testing, could result in more and relevant defects found than with traditional script-based testing alone, i.e. could result in a higher testing productivity.

Key words and terms: ad hoc testing, exploratory testing, productivity, scripted testing, GUI application testing.

Acknowledgements

I should like to thank Professor Markku Turunen from the University of Tampere for being my supervisor and providing invaluable feedback during the master's thesis work and Mrs. Hanna Pihlajarinne from Nokia Corporation for being my contact person.

I should like to thank my parents for the APOLO][, which I received for Christmas '83. Had it not been for that computer, I might never have ended up in the field of computer science.

Finally, I should like to use this opportunity to thank my wife Minna and my children, Malva and Reetu, for their incredible patience and encouragement and my colleagues at Nokia Corporation for their support.

Tampere, December 2004

Contents

"We shall not cease from exploration
And the end of all our exploring
Will be to arrive where we started
And know the place for the first time."

-T. S. Eliot [1975]

1. Introduction

In today's hectic and fast culture with short product cycles, less and less emphasis on explicit requirements and design (adequately termed "headless chicken" mode of development [Goldsmith and Graham, 2002]) it becomes generally more and more important to focus testing activities and to employ an efficient testing mindset. Whatever model of software development is being adhered to, testing remains a crucial activity within that model. Badly tested software can very easily break a company's reputation. Graham [2002] argues that a better link between requirements and testing would eventually result in improved tests and improved software, but what if the software developed is of such a nature that requirements simply cannot be very explicit or static and not change?

Graphical user interface (GUI) design is quite difficult and the only reliable way to achieve good interfaces is through iteration [Myers, 1995], taking feedback from testers and end-users into consideration. Traditional user interfaces (UIs) were function-oriented [Nielsen, 1993], but already by 1995 command line user interfaces like e.g. the UNIX command prompt, almost completely vanished and had made way for direct-manipulation visual GUIs, like e.g. Microsoft Windows [Myers, 1995]. Direct manipulation refers to the fact that GUIs are object-oriented [Nielsen, 1993] and users need to choose objects first and then action. In non-direct manipulation interfaces, an action has to be chosen before the object.

During the early days of visual interfaces, critics coined the acronym WIMP for such interfaces, standing for windows, icons, mouse and pull-down menu [Shneiderman, 1998, p. 207]. Windows are used by programmes as output of text or graphics, icons are representations of an object or an action, a mouse is a pointing device which enables manipulation of the interface and a menu is a list of commands. Typically, a GUI is made up of the following elements: controls, also called widgets (widgets are interface components [Shneiderman, 1998]), a window, a menu bar, layout and interaction.

GUIs evolve through many iterations it is precisely the iterations and change to the interface which makes GUI application testing demanding. Requirements may be vague and become clearer once the system evolves, or they may change altogether. Kaner *et al.* [2002, p. 72] argue that even if a product was fully designed in advance, people do not fully understand the system until it is built, implying a certain mismatch between design specifications and actual system, unless of course the specifications are always

kept up-to-date. This, together with the many GUI iterations, puts pressure on testing. Scripted testing, a testing method whereby test cases are generated based on requirements and other relevant specifications and then executed later, may prove to be an inefficient way on its own, because changes in requirements, together with iterations bring forth a need for constant documentation updates, including test case updates and those updates may temporally lag behind the actual implementation, provided they are even done.

In a study by Memon and Soffa [2003], 74% of test cases became unusable from one GUI version to the next. Even though they provided a novel technique for GUI regression testing, this technique is only applicable to automated GUI testing and has some limitations. Memon and Soffa also did not take a position on functionality changes of the underlying programme and the influence on existing test cases. Nevertheless, what is relevant from their work is that some test cases became unusable due to the GUI changes.

Would there thus be a benefit in looking at testing from a different perspective, a perspective that could cope better with changing or vague requirements and evolving GUI applications? This is not to say that scripted testing would become unnecessary, but rather that a new testing approach could be used in addition to the scripted approach. Exploratory testing could be such a new testing perspective. It is sometimes referred to as ad hoc testing and can best be described as ”learning, test design, and test execution at the same time” [Bach, 2003a]. This is in stark contrast to the more “traditional” scripted testing, where tests are designed beforehand.

Currently there exists no research related to the added benefits in terms of testing productivity of exploratory testing over scripted testing, except for some anecdotal evidence mentioned by Bach [2002, 2003a] and Agruss and Johnson [2000].

In an attempt to fill the gap in current research, the research problem this work will concentrate on is whether the use of exploratory testing in addition to the more “traditional” script based testing, within the scope of GUI application testing, could result in more and relevant defects found than with traditional script-based testing alone. In other words, would there be an increase in testing productivity if exploratory testing were to be used in addition to script based testing? Finding more faults does alone not necessarily constitute higher productivity [Kaner, 1999; Kaner *et al.* 2002, p. 71] and therefore this research will look not only at the mere number of faults found, but also at their severities.

The research problem will be worked on by means of a case study. Test reports generated during a specific time frame while testing a specific GUI application will be analysed. The error reports contain information about error severity and information about whether the problem reported is due to a test case or due to exploratory testing. Section 4 introduces the case study and details of the approach that has been taken in working towards the research problem. The section furthermore contains the data analysis and results and a part on limitations in the study. The section is concluded by a summary of findings.

The emphasis of this work is purely on the actual results achieved through exploratory testing and not on the time spent to achieve these, meaning that even if exploratory testing is used in addition to scripted testing, the amount of time spent is not considered in this work. Further work could concentrate on attributes such as time and also on issues of how much feedback exploratory testing results could feed back into the iterative and incremental software design. Section 5 contains an outlook on future work and also concludes this work.

Since testing is intrinsic to the process of software development, several software development process models that have at least to some extent relevance to this work, will be described in Section 2 to better enable the reader to position testing within these models and to see how changing requirements could influence the testing work.

Thereafter the focus shifts to testing and Section 3 will clarify terminology related to software defects and will set out how the different terms (error, fault, failure, defect and mistake) will be applied within this work. Furthermore, several levels of testing, test case generation, testing oracles, principles of testing as well as the principles of exploratory and scripted testing will be presented. The presentation of a testing model concludes the section.

2. Software development models

Software does not only refer to an actual program, but includes also requirements, specifications, design and programs, user manuals, guides and other related documentation [Mills, 1980].

The role testing plays within the software development process is very much dependent on the kind of process model one adheres to. Software development process models are used to determine the framework within which software is developed, a framework for guiding the software process. They principally specify what needs to be done and how it will be done.

Different software development process models place quite different emphases on testing. Even within a certain process model, there may be different approaches towards how software is being created and tested. In order to shed light on the role of testing within the software development process and on the ability of the process to react to changing requirements, two categories of software development process models will be described: specification-driven single-pass, and iterative and incremental development models. Most software development models fall into either of these two categories and some can even be considered falling into both. In addition to the categories, software development models can also be sequential or concurrent in nature, which will be explained in Section 2.3.

2.1. Document-driven single-pass development models

Single-pass models treat the software development process as a cycle of development, without real iteration (evolution). The development of the software is started with the specifications at some point and ends at another with the final product being ready, in one single pass. Development is document-driven, because there is great emphasis, as will be shown in the following section, on documentation, including requirements, specifications, and test plans.

2.1.1. The waterfall model

One of the earliest software development process models, is a model that views the development of software as a series of steps or phases that follow each other sequentially, implying that the tasks of a step are largely completed before progressing, through a quasi "gate" to the next step. Iteration is possible, but this is rather between preceding and succeeding steps and rarely with more

remote steps [Royce, 1970]. This model falls into the category of document-driven single-pass software development models and is known as the waterfall model, comprised of the system and software requirements stage, analysis and program design stages, coding, testing and operations stages.

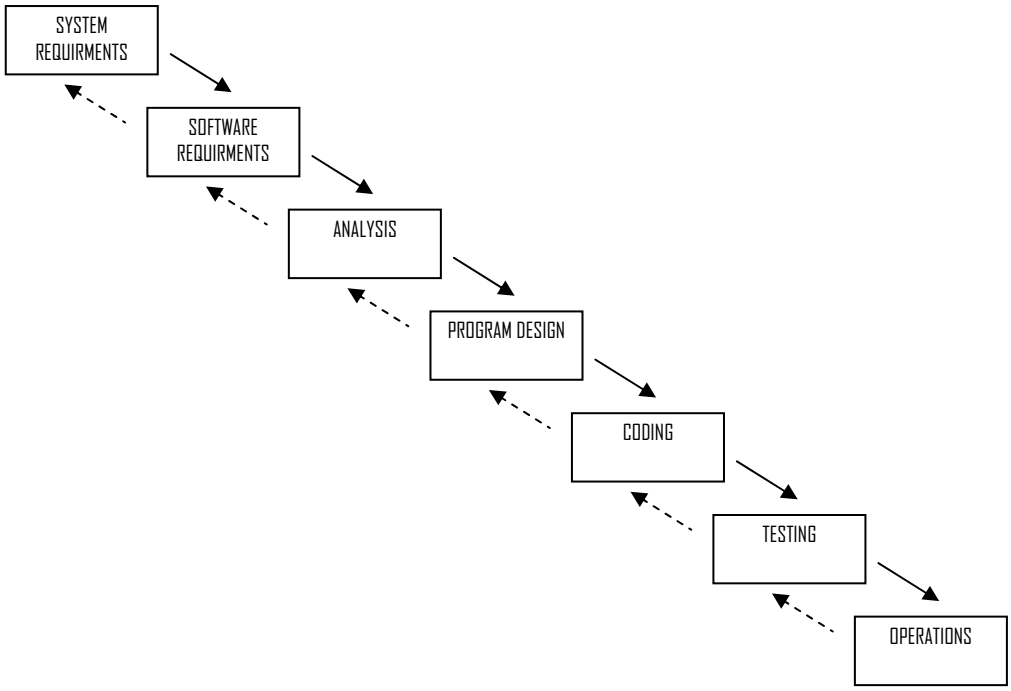

Figure 1. The basic waterfall model [Royce, 1970].

Numerous variations of this "basic" process model exist. The system specifications in this model are the first stage and also the source of later testing activities. The specifications and actual process documentation drive every stage of development. Before passing through the "gates" to the succeeding stage checklists are used and precisely defined documents have to be available.

Royce [1970] suggested that, if the software to be developed is developed for the first time, then it should be done twice, implying an element of original iterativeness and incrementality in the process, something which has been lost in later adaptations of the waterfall model, where is rather viewed as a single-pass model. The model is considered such also within this work.

The model has been criticised for not reflecting the realities of software development, which does not happen in such clearly defined stages and in a linear fashion. Even though iterations are possible in the waterfall model (i.e. going back, indicated by dotted-lined arrows in Figure 1) there is a tendency to nevertheless "freeze" certain parts of the development and continue with the next stage [Sommerville, 1992, p. 7]. This in turn makes this model very rigid

and slow to react to fast changing customer demands [Kaner *et al.*, 1993, p. 259]. It may nonetheless be well suited for smaller software projects that are not subjected to many changes in requirements during the development.

A clear disadvantage from a testing point of view is that testing is situated at the end of the "waterfall" and is not an intrinsic part of each stage of the development cycle. Interestingly, even Royce pointed out the risks involved in that "[testing] occurs at the latest point in the schedule when backup alternatives are least available, if at all" [Royce, 1970]. Testing activities may find defects, but in some way it is already too late [Craig and Jaskiel, 2002, p. 8; Kaner *et al.*, 1993, pp. 258-259] and the only way to rectify the defects is through a change in the requirements or a change in the design.

The need to have all documentation fully ready (e.g. requirements, specifications, and test specifications) before implementation may be a hindrance when developing interactive end-user applications [Boehm, 1988].

Owing to the nature of the waterfall model, i.e. deliverables are easy to define for each stage and lend themselves to measurement, it has been the most widely adopted model despite its shortcomings and criticism [Sommerville, 1992, pp. 12-13].

2.1.2. The V-model

The V-model, yet another document-driven single-pass development model developed in the 1980s by Rook addresses the testing phase issues of the waterfall model elegantly and gives equal weight to both development and testing [Rook, 1986]. The software development, including testing, is broken down into a series of distinct phases, each with well defined products. Once all required products of a phase have been successfully achieved, they form the foundation or so-called baseline for the succeeding phase. This continues for each phase and with each new baseline confidence in the software grows. The development phases, based on those of the basic waterfall model, and the testing phases are named project initiation, requirement specification, structural design, detailed design, code and unit test, integration and test, acceptance test, operation and maintenance and product phase-out. The phases and baselines are arranged in a V-shape (Figure 2) and for each design or specification phase on the left side of the "V", a corresponding integration phase can be found on the right side of the "V". In the diagramme depicted in Figure 2, the rectangular boxes represent the phases and the oval boxes represent the baselines.

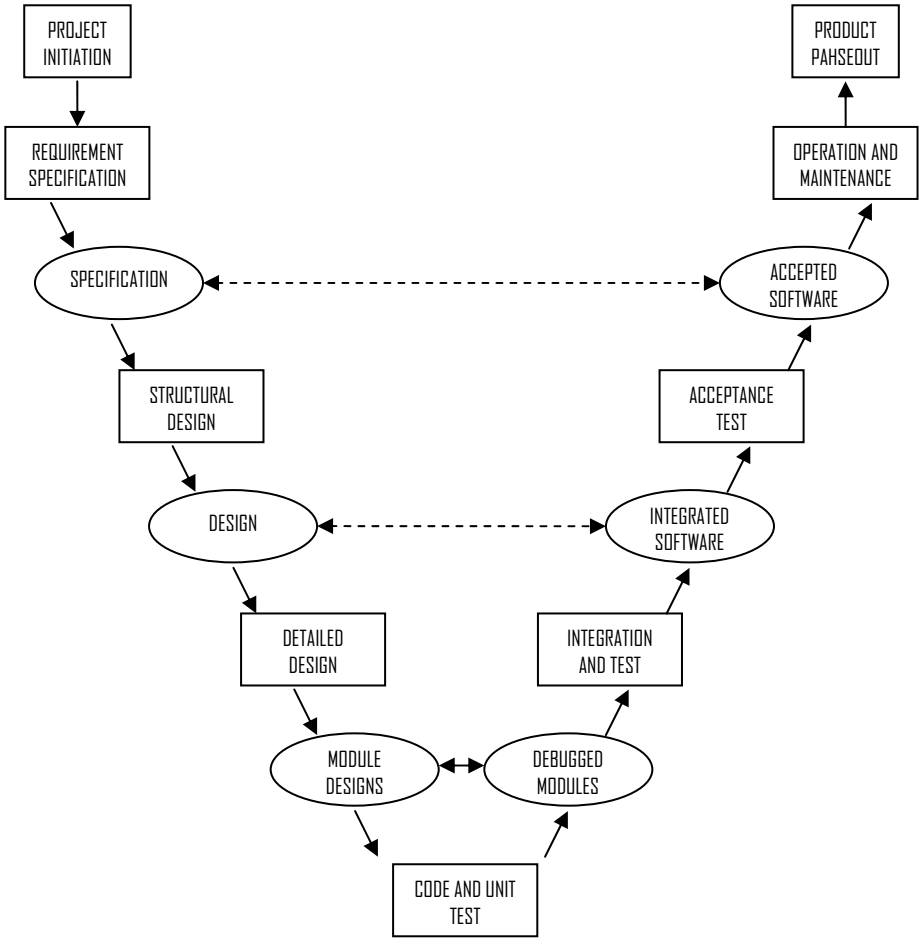

Figure 2. The phases and baselines of the V-model [Rook, 1986].

Testing is present at each phase of integration. This is important, because it in turn implies that there are several levels of testing and testing is not only something that is performed at the end. Several levels of testing also suggest that problems can be found early and also corrected early, which is always cheaper. Levels of testing will be explained in more detail in Section 3.3. As can be seen in Figure 2, the lowest level of testing is unit testing (in the code and unit test phase) and thereafter the modules can be verified against the module designs, the integrated software against the design and the accepted software against the specification. The V-model does not provide for the testing of requirements and assumes that requirements do not change, i.e. there is no testing phase in the model that would test the requirements. Graham [2002] argues that testing ought to start already with the requirements, something which the V-model does not cater for at all, nor does it cater for the testing of specifications or designs for that matter. Testing in the V-model starts with unit testing.

The V-model furthermore does not adequately handle all the facts of development and testing, such as frequent repeated builds, the change of requirements or even the lack of good requirements. It further presupposes that tests are designed from single documents and not modified by later or earlier documents or feedback [Goldsmith, 2002; Marick, 1999], meaning that iterations in design are not well supported by such a model. It furthermore suffers from the same criticism as the waterfall model, regarding the passage of software development in well-defined stages and in a sequential fashion. It must nevertheless be stated that Rook himself [1986] was fairly critical and stated that it would be unrealistic to interpret the model in a simplistic way (meaning rigorously), especially for large software projects, also precise breakpoints between the phases are not easy to define clearly and they depend to some extent on project management decision; implying that they are not necessarily natural. Rook [1986] additionally stated that the project plan may call for incremental development; a sort of development which is described in more detail in the next chapter and a sort of development which is not usually associated with the V-model, which is also the reason why this model has been treated as sequential within this work.

2.2. Iterative and incremental software development models

The roots of iterative and incremental software development (IID) can be traced back to the 1960s [Larman and Basili, 2003].

An incremental approach to software development, as for example described by Dyer [1980], is an approach which does not place the core emphasis on specifications, but rather starts from a basic outline of the system that needs to be built and in which software is partitioned into increments whose development is scheduled over the total development cycle. The software can be said to be evolving with each increment.

In today's fast world, there is a drive to produce software rapidly and with flexibility, something which a strict specification-driven single-pass model only allows with difficulty, if at all. IID models of software development are, due to their nature, more suited for such development, especially if the software to be developed is interactive software.

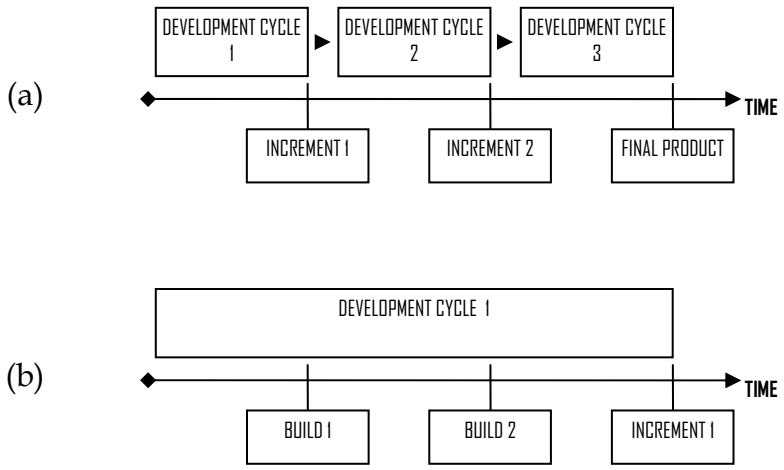

Figure 3. (a) Iterative and incremental software development model; (b) Iterations leading to increment 1.

Both the waterfall and the V-model can utilise an IID approach, i.e. incremental software development can be seen as a series of waterfall or V-model development cycles, each cycle producing an increment, until the final product is complete. This is depicted in Figure 3, (a). Development cycle 1 yields increment 1 after which development cycle 2 starts, yielding increment 2. This process continues until the final product is ready. Within the development cycles, there may be iterativeness. (Iteration should be understood as a doing over as well as taking received feedback into consideration). This is illustrated in Figure 3, (b), where development cycle 1 produces 2 builds before producing the actual increment 1. A software build refers to integrated software, meaning that all relevant components have been integrated into a whole software package.

Iterative and incremental software development is no longer driven entirely by specifications, but rather by received feedback from testers or users after each build or increment, which in turn may alter the specifications and requirements for the next iteration and, as a result, the build or increment.

From a testing point of view, the iterative and incremental approach brings with it the possibility to test builds and increments before the complete and final product is integrated and ready, as well as feed back into the process and requirements, which is a clear advantage over the single-pass waterfall model, because it provides more flexibility and does not attempt to specify everything in advance and then build everything at once. The IID models, just like the waterfall and V-models, may require a set of overall requirements to be defined

at the beginning, but these do not necessarily need to be of an exhaustive nature, which in turn may complicate a traditional specification-based testing approach and may favour an exploratory approach in addition, because an exploratory approach does not require fully defined requirements. Exploratory testing will be discussed in detail in Section 3.6.

One well known iterative and incremental development model, the spiral model, will be described next.

2.2.1. The spiral model

The spiral model [Boehm, 1988] is an iterative and incremental software development model and is based very much on risk analyses and contains elements of the basic waterfall and V-model approach. There are four major process areas with several sub-areas. The major areas are the determination of objectives, alternatives and constraints, the evaluation of alternatives and the resolution of risks (risk analysis), the development and verification of a next-level product and the planning of the next phases. Sub-areas contain elements such as the development of prototypes, unit, integration and acceptance testing, software requirements and several others which are not be mentioned here in order to avoid a level of detail that is not directly related to this work.

The most important aspect of the spiral model is that the process goes round and in each round, identified risks are analysed and resolved and certain tasks are performed. In round 0 there would be a feasibility study, in round 1 a concept of operations, in round 2 a top-level requirements specification, in succeeding rounds there would be a move towards the actual development (including the development of prototypes) and testing of the software. After each such round, the results are fed straight back into the next round, which again would go through evaluating and resolving risks, building a prototype, testing and implementation, until a final version of the software is obtained. As is evident, the final software will have evolved over several iterations and increments.

Out of the spiral model, an approach to software development, known as rapid application development (RAD) has evolved. RAD would start out with a rough overall set of requirements and would go through several iterations and produce prototypes that can be evaluated by testers and user groups. RAD is used e.g. in the development of speech recognition systems, but is less used and less appropriate for complex projects and not relevant for the software tested within this research.

2.3. Sequentiality and concurrency

Typically, both the specification-driven single-pass development models and the iterative and incremental development models are applied sequentially, but there is no constraint, except for added complexity, preventing such models to be applied also concurrently.

A concurrent development process model, created during the mid 1980s, introduces concurrency to software development and hence focuses on the concurrent execution of multiple processes with the main goal of reducing development time [Aoyama, 1993]. Concurrency could be introduced to any of the previously described sequential software development process models, but Aoyama [1993] chose to use the waterfall model as a basis for adaptation towards a concurrent model. In this particular model software is delivered in releases over time, each release incorporating certain enhancements over the previous release, something which this model has in common with iterative and incremental development models.

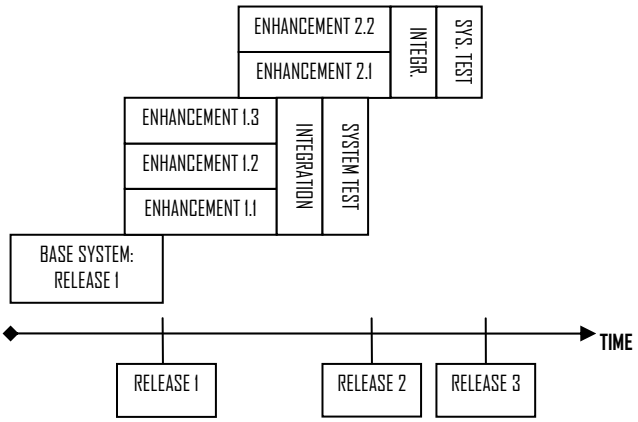

Figure 4. Concurrent development model [Aoyama, 1993].

Figure 4 shows that development work on enhancements 1.1, 1.2 and 1.3 starts while the base system is still being developed. After integration and system testing, release 2 is delivered. Again, while development on enhancements 1.1, 1.2 and 1.3 is still ongoing, development on enhancements 2.1 and 2.2 commenced already.

Managing such concurrent software development is much more complex than managing sequential software development and this includes the complexity of managing testing tasks. Because the concurrent development model is based on the waterfall model, testing suffers from not being an intrinsic part of the whole development chain, on the other hand the model contains elements of IID, but in the concurrent development model described

by Aoyama [1993], testing is performed at the end of the development of each enhancement and also when all enhancements are integrated into a software release (see Figure 4). It is theoretically possible that testing could assume a larger role in the model described. This could be achieved by e.g. viewing testing as an overall activity without being a specific "phase" in the model and providing testing for each activity that is being performed during the complete development process, starting from the requirements definitions.

2.4. Summary

Several software development process models were presented with the aim of positioning testing within the models and bringing forth the challenges testers operating within such models face when requirements are vague or even non-existent at the beginning of a software development project and only gradually evolving. The software application this research is focusing on is developed within an iterative and incremental software development model, based on the V-model, with slight elements of concurrency and is therefore affected by the described challenges. Defining a set of test cases early during the software development project only aggravates the challenges of testing and hence scripted testing may not be adequate on its own to ensure a properly tested application. The following section will look at testing in a broader sense and will introduce exploratory testing.

3. Testing

Testing is an extremely creative and intellectually challenging task and may even be destructive, but at the same time adding value to the application under test. Less errors and stability are likely to have a positive effect on end-users.

In this section, definitions for several terms that are often used interchangeably (errors, faults, failures, defects and mistakes) will be offered, followed by several definitions of "testing" and a discussion. Thereafter several levels of testing will be discussed and some basic concepts explained. Finally, exploratory and scripted testing will be presented and a summary concludes the section.

3.1. Errors, faults, failures, defects and mistakes

It is very common to hear some people talk about software having defects or errors; others may be talking about faults that have been found, yet others about failures. Terms appear to be used interchangeably. It is important to clearly define each of these, because this work deals with software testing and they are therefore at the very foundation of testing.

Human action during software development or operation leads to mistakes, which in turn manifest themselves as software faults. As a result of such faults, failures occur. A failure is the inability of a system or component to perform its required functions within specified performance requirements, resulting in an incorrect result. (According to Beizer [1995, p. 9] in good software developed under a good process, failures are rather caused by complexity and to a lesser extent by mistakes of individual programmers.)

The amounts by which the results are incorrect are the errors. [IEEE, 1990]. The definition of failure includes a reference to "specified performance requirements", which in turn implies that such must have been specified before any actual testing commences and must be available during testing. This does not necessarily need to be so in exploratory testing, where testers may test software not only against specified requirements, but also against certain guidelines (heuristics) and against the expected outcome. Given that, the definition of failure could be shortened to "the inability of a system or component to perform its required function resulting in an incorrect result", without losing the essence.

The term "error" is commonly used to denote mistakes, faults, failures and errors as defined above [IEEE, 1990]. An "error" is also known as a "bug". The

term "defect" is used to denote both faults and failures and will be used also in this work in that way.

3.2. Definitions of testing

Testing definitions usually fall into four groups, based on different primary goals. These could range from "demonstrative" (demonstrating that application behaves as it should), "destructive" (trying to break the application), and "evaluative" (gaining insight into the functionality) to "preventative" (trying to prevent errors from occurring in the future). This brings with it that testing is usually defined in several ways, depending on what the primary goal is set to be.

Myers [1979, p.5] defines testing as "the process of executing a program with the intent of finding errors", which clearly points towards a destructive [Gelperin and Hetzel, 1988] mindset. Kaner *et al.* [1993, pp.124-125], like Myers, see the purpose of testing in "finding errors" (here meaning faults, failures and mistakes) and define one of several characteristics a good test should have as a "reasonable probability of catching an error". Also this is a destructive point of view about the primary goal of testing.

The IEEE standard glossary of software engineering defines testing as "the process of analysing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items" [IEEE, 1990]. This definition focuses on both the bug-finding (destructive) aspect and an evaluative aspect. It is unclear whether the evaluative aspect is meant to demonstrate in some way or other that the software under test is working satisfactorily. This definition does not include where the information about the "required" condition may be found and hence one may argue that this does not necessarily have to mean specifications, but could include for example guidelines (heuristics) which a tester may be using.

Beizer [1995, p.3] views testing as "the act of designing, debugging, and executing tests". Test in this case refers to a sequence of one or more subtests executed as a sequence because the outcome and/or final state of one subtest is the input and/or initial state of the next. Beizer's definition lacks the intent, which is stylishly expressed in Myers', the IEEE´s and Kaner *et al.*'s definitions, but includes test design and debugging. Even though not stated in his definition, software, according to Beizer, is tested for several reasons. Amongst those are breaking the software (taken from Myers' definition), demonstrating that it works, and providing information that can be used for prevention of

mistakes in the future. The latter is seen as highest goal of software testing. [Beizer, 1995, p.7].

Myers [1979] states that testing cannot demonstrate that errors (here meaning faults and failures) are not present, which is starkly contrasted by Beizer [1995, p.7] who makes a distinction between dirty tests, i.e. tests designed to break the software and clean tests, i.e. tests designed to demonstrate the software's correct working. This reflects a difference in mindset, i.e. non-demonstrative vs. demonstrative, or what Pettichord [2004] would term an adherence to a different software testing school. According to Myers a possible definition of testing as being the process of demonstrating that a program does what it is supposed to do, i.e. satisfies its specification, is no good, because even if it does what it is supposed to do, there may still be errors present, causing the program to do things it is not supposed to do. [Myers, 1979, p.7]. An application may, for example, have been tested and all functionality is satisfactory, i.e. the application does what it is supposed to do. Unfortunately, the application does not accept any input when it has been running for more than one hour, i.e. it does also something it is not supposed to do, which cannot be spotted by concentrating only on verifying those things it is supposed to do.

What is elegant about Beizer´s definition is that it allows and incorporates three goals of testing, namely the "breaking of the software", the "verification", and the "information provision". There is no reason to believe that these activities need to be mutually exclusive.

Psychologically a definition of testing as a process of finding errors is superior to any definition claiming the contrary as this could become a self-fulfilling prophecy, i.e. if we set out to show that there are no errors we will use a different mindset as when the goal is to show that there are errors. [Myers, 1979, pp.4-7]

It is generally recognised that the earlier that defects are found, the lower are the costs of correcting them [Beizer, 1995]. Hence code inspections (a set of procedures and error detection techniques for group code-reading) and walkthroughs (similar to code inspection, but with different procedure [playing computer] and different error detection techniques) are seen as a good start of any "testing" [Myers, 1979]. Such testing activities, including requirements reviews, can be both evaluative and preventative in nature [Gelperin and Hetzel, 1988], even though their goal is still to find faults.

These methods may find 30% to 70% of logic design and coding errors in typical programs (percentage of total number of found errors.) [Myers, 1979,

pp. 18-33]. The percentages indicated by Myers need to be viewed in context of the time when Myers published these, namely during the 1970s. Whether the indicated figures are still true of current software projects could not be established.

Depending on the kind of software development process models followed (waterfall, incremental) the timing of testing takes on different roles. In the nowadays considered antiquated [Kaner, 2003] waterfall model, testing linearly follows coding and typically comes at the end of the software development process when coding is complete. Myers [1979] tends to adhere to this model as also he views testing as following coding and does not leave room for intertwined evolution of coding and testing, something which is considered essential [Hetzel, 1988].

Pettichord [2004] in a meeting of the Austin Software Process Improvement Network defined four different "schools" of software testing, each with different values and basic techniques associated with them and each with different views about testing. Even though Pettichord's paper has not been published except on his own web site, he presents some fresh thoughts worthwhile considering also within the context of this research. Nevertheless, since this work deals with both scripted testing and exploratory testing, it cannot adopt any one particular testing movement, but rather a mixture of two. It is questionable whether any real-life project can be categorised neatly within one specific testing "school" and real-life testing projects may adhere to a multitude of different movements. People may for example say that it is important to measure testing accurately, but at the same time favour an exploratory testing approach in addition to a scripted testing approach.

One of the "schools", the context-driven school, with an emphasis on people (as setting the context), on finding bugs and providing information is strongly associated with exploratory testing. The other, the routine school, places a lot of emphasis on manageability, i.e. testing must be predictable, repeatable and planned and will validate the product and fits in very much with scripted testing and waterfall-based stage software development models. Yet for this work, an adherence to a particular "school" or "schools" is not of primary importance; the emphasis is on testing, namely scripted and exploratory testing. Pettichord showed pleasingly that there is no one correct view about testing.

3.3. Levels of testing

Levels of testing refer to structuring the testing of software in such a way that the whole complex task of testing is split into several smaller tasks. This level-thinking fits in very well with several process models of software development such as for example the V-model.

Since many definitions of levels of testing and concepts in literature are based on Myers [1979] and Beizer [1990, 1995] and since they are also suitable to this study, their definitions will be used here. (Beizer has very much refined Myers's concepts and added his own work.) Myers [1979, pp. 77-120] proposes several levels of testing depending on the main focus of the testing.

Module testing (sometimes referred to as unit testing) is a process of testing the individual subprograms, subroutines, or procedures in a program. [Myers, 1979, p. 77].

Component testing is not mentioned by Myers [1979], but Beizer [1990, p. 21] defines components as the integrated aggregate of one or more units. Tests for component testing would be based on functional specifications. Arguably what has been defined by Beizer [1990] as component testing is very much related to what Myers [1979] refers to as integration testing, especially if performed incrementally.

Integration testing is not explicitly defined by Myers [1979], but viewed as part of module testing, i.e. testing that occurs when putting together more and more modules to form a complete system. This can be done incrementally or all at once. Beizer [1990, p. 21] defines integration testing as testing done to show that even though the components were individually satisfactory, as demonstrated by successful passage of component tests, the combination of components are incorrect or inconsistent.

Function testing is a process of attempting to find discrepancies between the program and its external specifications. An external specification is a precise description of the programme's behaviour from the point of view of the outside world (e.g. its user) [Myers, 1979, p. 108]. Beizer [1990, pp. 10-11] does not view function testing (or functional testing) as a testing process per se, but rather as a point of view from which test design emerges. It regards the programme or system as non-transparent as opposed to a structural view which regards the system as transparent. Myers [1979, p. 108] on the other hand states that except for small programmes, function testing is usually a "black-box" oriented activity, meaning that testers do not have a visibility into the source code. Even though there is a slight discrepancy, important is that verification would

happen against external specifications, i.e. test cases would be based on (functional) specifications.

System testing is testing with the purpose of comparing the system or program to its original objectives [Myers, 1979, p. 110]. It is concerned with issues and behaviours that can only be exposed by testing the entire integrated system or a major part of it [Beizer, 1990, p. 22]. System testing includes testing for facility, volume, stress, usability, security, performance, storage, configuration, compatibility, installability, reliability, recovery, serviceability, documentation, and procedure [Myers, 1979, pp. 110-118; Beizer, 1990, p. 22]. On a system testing level there may also be guerrilla testing (attacking the programme), scenario testing (testing a realistic and important set of features bundled into a scenario) and long sequence testing (several hours, days or weeks) [Kaner *et al.* 2002, p. 43]. Such testing may be of a scripted or exploratory nature.

Even if every feature of a program or system had been tested during unit and integration testing, the fact that within the whole system the order in which things can happen can no longer be predicted with certainty, redoing functional testing on a system testing level is required [Beizer, 1995, p. 10]. Marick [1999, p. 5] maintains that it might sometimes even make sense to defer both unit and integration tests until the whole system is at least partly integrated. The distinction between unit, integration and system tests begins to break down.

3.4. Test cases

Test cases are the instructions for testers about how they should test and what they should test. It is impractical or often impossible to find all errors in a program, because exhaustive input testing is impossible or just plainly not feasible. Neither is it possible to create error-free software, even though such was still believed by some in the early 1970s [Royce, 1970].

If the testing mindset is a destructive one, then a successful test case is one that finds an error and an unsuccessful test case is one that causes a programme to produce the correct result. [Myers, 1979, p. 16]. A good test case is consequently one that has a high probability of detecting an as-yet undiscovered error / that detects an as-yet undiscovered error. This means conversely that once the error has been discovered, the test case's usefulness decreases since the possibility of finding the same error again most likely is smaller [Bach, 2003a]. If, on the other hand, the testing mindset is a demonstrative one, then a successful test case is one that does not find an error

and would hence show that the item that was tested does indeed function as expected.

A necessary part of a test case is a definition of the expected output or result. Test cases must also be written for invalid and unexpected as well as valid and expected input conditions [Myers, 1979, pp. 12-14]. The requirement about the need of an expected output or result is tricky when it comes to exploratory testing, because as shall be shown in Section 3.6, due to the nature of exploration, specific and detailed outputs or results cannot always be anticipated.

Test cases can be generated in several ways, the most usual being through black-box or white-box methods, described in the following section.

3.4.1. Black-box and white-box

According to Myers [1979, pp. 37-75] there are two basic testing methodologies, namely black-box and white-box testing that can be employed when creating test cases. Myers' work has formed the basis of subsequent works on software testing and for that reason, his classifications are included here, followed by Beizer's more contemporary view.

Black-Box testing (data-driven or input/output driven testing), also called behavioural testing [Beizer, 1995, p. 8], is a testing strategy whereby the tester views the programme as a black box and is not concerned with the internal behaviour or structure thereof. Test data is derived solely from the specifications. According to Myers [1979], it includes equivalence partitioning (partitioning the input domain of a programme into finite number of equivalence classes such that one can reasonably assume that a test of a representative value of each class is equivalent to a test of any other value), boundary-value analysis (one or more elements are selected such that the edge or boundary of the equivalence class is the subject of a test), cause-effect graphing (a systematic method of generating test cases representing combinations of conditions), error guessing (the basic idea being to enumerate a list of possible errors or error-prone situations and then write test cases based on the list).

In Beizer's more contemporary view [Beizer, 1995], the following are part of black-box testing: control-flow testing (similar to cause-effect graphing), loop testing (a heuristic technique based on experience that has shown that bugs often accompany loops, i.e. a repetitive or iterative process), data-flow testing (based on data-flow graphs), transaction-flow testing (based on transaction-

flow graphs), domain testing, syntax testing (for testing command-driven software) and finite-state testing (based on finite-state machine models, which are much used in object-oriented software design)

White-box testing (sometimes referred to as glass-box testing [Kaner *et al.*, 1993, p. 41]) is based on the structure of the application under test, basically referring to the source code [Beizer, 1995]. Underlying structures of the software are by definition not known to the tester in black-box testing. White-box testing includes statement coverage, decision coverage, condition coverage, decision/condition coverage, multiple-condition coverage [Myers, 1979]. White-box testing methods are more appropriate for lower levels of testing, like e.g. module testing, but not very appropriate for testing on a system level, because the scope of system testing should be much wider. Figure 5 illustrates the difference in visibility towards the source code and underlying structure, a tester has when employing white-box and black-box techniques. With black-box techniques there is no visibility into the source code and with white-box ones there is.

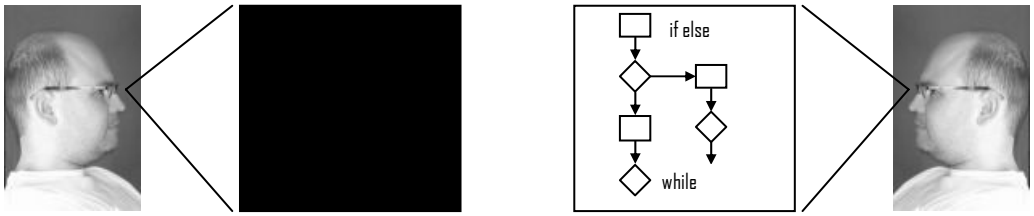

Figure 5. Visibility in black-box and white-box testing techniques.

Hybrid testing combines both black-box testing and white-box testing techniques [Beizer, 1995], thereby enabling testers to know something about the underlying structure of the application and at the same time also using methods that do not care about the structure, thereby enriching their overall arsenal of testing techniques.

Black-box and white-box methods are not described in any more detail, because they are not the focus of this work, but do nevertheless represent techniques that can be used when using an exploratory approach to testing and are therefore mentioned.

3.5. Script-based (scripted) testing

Script-based or scripted testing is testing performed by using a script or collection of step-by-step pre-defined test cases. These test cases are typically, but not necessarily, defined early during software development and are usually based on software requirements and internal logic, depending upon which procedure in their conception has been adhered to (black-box, white-box or a combination). Test cases are then executed when the application becomes available for testing.

Consider a hypothetical GUI application called "Opener" that consists of only one button which, when pressed, opens another application "A". One scripted test case for "Opener" could look like this:

Precondition:	*Application "Opener" is open.*
Test Steps:	*1. Move the mouse pointer over the button.*
	2. Press the button.
Expected result:	*Application "A" is opened.*
Exceptions:	*Application "A" is not opened.*

A benefit of scripted testing is that it lends itself very well to measurement. A total number of test cases can be calculated, test metrics can be created, measuring e.g. how many test cases have been run compared to how many have been planned. The measurement is something, which fits very well with document-driven development models. It is all about measurement and testing progress can be measured easily. It is also easy to assign testing resources, once the overall testing effort is known through the test specifications, greatly helping project management.

Another advantage is that from a testing skills perspective, nearly anyone can execute basic scripted tests, because that what is required is broken down into clear steps, which are easily followable. Executing scripted tests hence does not necessarily require very experienced and highly skilled testers and even testers just starting their testing career can cope. Designing test cases, on the other hand, does require skill, but because there is a difference in time between the creation of test cases and their execution, different testers can design and execute test cases. Tests can therefore be designed by the more skilled testers or even by one skilled tester and executed by several less skilled ones. In short, there is no need to have only skilled testers on a testing team.

One drawback of scripted testing is that if one test case finds a defect in the software, then after the defect has been fixed, the probability of finding another

defect with the same test case is much lower than the first time [Bach, 2003a]. This is turn implies that test cases lose their effect to some extent and new test cases may be needed, i.e. a change in the script may be required. Scripted test cases do not only lose their power the longer they are used, but they also may get outdated. This requires regular updates to the scripted test cases to keep up with the ongoing software development.

Another disadvantage of scripted testing is that in practice the software being developed usually changes from the time of the first requirements being written down to the time when it will be released to the market.

In a highly competitive environment adjustments are required in order to respond to challenges unknown at the time of writing requirements to make the software as "good" as possible from and end-user perspective. (Such changes in requirements over the development cycle of the software are very typical for interactive applications and GUIs are usually developed through many iteration cycles [Myers, 1995].)

Scripted test cases may also limit the testers' creativity and exploration, because they are bound by the already laid out test cases, unless of course deviations from the laid out test cases are specifically allowed and testers have the required skills to do so. Going beyond the obvious and exploring the application is fundamental to exploratory testing, which will be explained in the next section.

3.6. Exploratory testing

Exploratory testing, also sometimes called ad hoc testing, can be best described as "learning, test design, and test execution at the same time" [Bach, 2003a]. Learning here refers to what is observed and retained in memory, i.e. learned during the actual testing and serving as input for new tests to be planned and executed. It is important to know that exploratory testing is not a testing technique as such, but rather a different approach to testing – a different way of thinking about testing [Kaner and Bach, 2004], as compared to the "traditional" way of considering testing as an execution of pre-defined test scripts. Nonetheless, this does not appear to be viewed as such by all, since e.g. a chapter on exploratory testing by Bach in "The Testing Practitioner" [Van Veenendaal, 2002] is located in the section of test techniques. Even though such apparent contradictions exist, these do not matter within the scope of this work. Any of the already mentioned testing techniques can also be used in an exploratory way.

Exploratory testing as such is not a completely new idea and most likely testers do perform some exploratory testing occasionally, because they are after all not mere robots but human beings [Bach, 2002] – what is new though, is that exploratory testing is emerging as a subject of its own alongside the more "traditional" views on testing in handbooks on software testing, as for example in "The Testing Practitioner" [Van Veenendaal, 2002].

The earliest mention of a way of thinking one could relate to some extent to exploratory testing is by Myers [1979, pp. 73-75], called "error guessing" with the basic idea being the enumeration of a list of possible errors or error-prone situations and then writing test cases based on such a list.

This would technically make it scripted testing, but one may argue that it is also at least hypothetically possible, to design a test through error guessing, writing it down and executing it and that would make it exploratory in nature. The main difference in what Myers says and true exploratory testing is that Myers requires the creation of more than one test case prior to any testing and in that way the interactiveness and the feedback, i.e. the "information gained while testing" as Bach [2003a] calls it, will not lead to any new test cases generated on the fly and exactly that is the essence of exploratory testing. Furthermore, even though Myers mentions "error guessing" (a technique which can be very valuable also for exploratory testing), he is absolutely not in favour of ad hoc testing and is rather of the opinion that "throw-away test cases should be avoided unless the program under test is a throw away program" [Myers, 1979]. One reason for that is that after testing, test cases will be "lost". This does not have to be so, because documenting ad hoc test cases can overcome the problem of "loss", even though within exploratory testing very detailed test cases are usually not written down, but rather notes of what has been done, so that in case of error a test may be repeated [Bach, 2003a; Agruss and Johnson, 2000].

Microsoft's exploratory test procedure (based on Bach) for testing third-party applications for Microsoft Windows compliance explicitly states that very detailed test cases should not be written down, but rather an outline of what was done suffices, because such activities take too much time and interrupt the flow of testing [Microsoft, 2004]. Bach [2002] states that Kaner *et al.*'s [2003] position is that the production of materials or procedures by an exploratory tester for re-use is not required. This could not be explicitly verified by checking Kaner *et al*. [1993]; they have rather been vaguer and merely state that what has been done and what has happened should always be written down when testing in an exploratory way. If it is on the other hand not stated "how"

the testing has been performed then re-use is indeed not possible. Re-running exploratory (ad hoc) test cases no longer make the process exploratory but merely scripted, but for the sake of accountability and error reproduction testing notes may be required. The point is that in any case, an exploratory testing approach to testing brings forth much lighter documentation than a purely scripted testing approach, which may for example follow the 52-page IEEE standard for software test documentation 829-1998 [IEEE, 1998]. This IEEE standard describes the basic test documents that are associated with software testing and is waterfall-oriented.

Exploratory testing can be thought of as a "journey" with a desired outcome or a purpose (charter), but with no clear path defined and with many possible ways to reach the destination. Incidences during the "journey" will be used as guidance (direct feedback) in deciding how to continue.

Consider again the hypothetical GUI application called "Opener" that consists of only one button which, when pressed, opens another application "A". An exploratory testing session could look like this:

> *Purpose: Test all functions of the application "Opener".*
>
> *The tester would open application "Opener" take the mouse, move over the button, press it and notice that application "A" opens. The tester however also notices that the opening takes some time and that it feels slow. This observation makes the tester come up with a new test, one where the button would be pressed several times in succession, because that may or may not cause some problems, given the observed delay. After the tester has executed this new case, she is amazed to find that application "Opener" has crashed.*

This is what exploratory testing is all about; starting testing with a certain goal, learning about the product under test while testing and using gained information and feedback in the further testing design on-the-fly.

Exploratory testing is not against the idea of scripting [Bach, 2003a], it may complement it and considering script-based testing and exploratory testing as the end-points of a continuum would be most appropriate. Such a continuum would range from pure script-based testing, with tests described in advance in every detail, to pure exploratory testing, with every test emerging at the moment of execution [Bach, 2003a]. A purely "scripted" tester would not be allowed to deviate from the planned testing, whereby one with some "exploratory" traits would be allowed to do so. Conversely, a purely "exploratory" tester would not be allowed to do any planning before

embarking on the testing, whereby one with a slight "scripted" trait would be allowed to do so. This has also been referred to as a narrow and broad view of exploratory testing [Bach, 2002], whereby a narrow view would mean no scripting and a broad view would allow scripting and offer a possibility for testers who use scripting to deviate and improvise on the scripted tests. It is the broader view that forms the basis of the empirical part of this work.

Exploratory testing is not bound by the kind of software development processes or models being followed and can hence be used independently of these. Neither is it bound by the testing methods used for scripted testing. Such testing methods can also be used in an exploratory way [Kaner and Bach, 2004].

Exploratory testing requires a lot of mental work and careful observation, critical thinking and good ideas or guidelines are essential. More specifically, Bach [2003b] has defined 8 key elements that distinguish an expert exploratory tester from an amateur. These are test design skills, careful observation skills, critical thinking skills, diverse idea generation, rich resources inventory, self-management skills, rapid learning skills and status reporting skills. All these are elements that require training and are more likely to be found in experienced than in inexperienced testers. Highly trained and experienced testers may not be so readily available in any given software project, which is one of the criticisms exploratory testing has received [Van Veenendaal, 2004].

3.7. Testing oracles

An oracle can be said to be "any human or mechanical agent which decides whether or not a program behaved correctly in a given test, and accordingly produces a verdict of "pass" or "fail" [SWEBOK, 2004], or as Microsoft [2004] puts it "an oracle is a strategy for determining whether an observed behaviour of the product is or is not correct. An oracle is some device that knows the 'right answer'".

When the testing approach is a scripted one, then the determination of whether a program behaves correctly is already intrinsic to the test cases, which will include a mention of an expected result. This expected result serves as the oracle and will guide the tester. With an exploratory approach this is not so straightforward, because no expected results are defined. This in turn implies that nevertheless the tester needs to have some guidance, an oracle, to tell whether the observed behaviour is as intended or not. A list of known errors provided to the tester at the beginning of a testing session can serve that purpose, but there are also situations, where what is observed is not due to a known error. This is where the experience of the tester is crucial. Experienced

testers are able to compare observed behaviour to that of other applications they may have seen or tested in the past and can decide based on that. They can also employ guidelines, so-called heuristics, in their testing. A set of guidelines for exploratory testing, based on work by James Bach, has been published by Microsoft [2004].

Whittaker [2003] proposes the "fault model" to guide testing. He claims that "understanding what Software does -- and how it may fail doing it -- is crucial to being an effective tester." There are 4 fundamental capabilities of Software: 1. Software accepts input, 2. Software produces output, 3. Software stores data internally, 4. Software performs computations using input and stored data. The fault model is thus "if Software does any of these four things wrong, it fails." This is one of many models that can guide testers and there is no restriction on which models and techniques may be used for exploratory testing and each new model or technique will just add to the richness of the available inventory of resources.

Less experienced testers may lack such a rich set of guidelines or experiences with similar products (e.g. Windows applications). Exploratory testing requires hence a greater level of experience than would be required for scripted testing, which has also been identified as a disadvantage of exploratory testing [Van Veenendaal, 2004], because such experience level is usually not found abundantly in software projects. Within this study the experience level of testers related to the number of non-test case based error reports has been analysed and the results are presented in Section 4.2.5.

It is important to understand that exploratory testing, due to its exploratory nature, can provide information about an application under test on a different level than scripted testing. It is not simply a matter of "pass" or "fail" in many cases, but there is also an element of feedback. Testers can provide feedback about their testing session regarding observations they have made, irrespective of whether they relate to intended functionality or not. This in turn implies that the oracle does not necessarily have to be present at the time of testing but could be used after an exploratory testing session. There is then a slight danger that issues will be reported that are not really failures as such, or are already known problems, but they will be nevertheless very valuable product feedback. In order to reduce the number of reports of already known issues, such can be provided in a list at the beginning of a test session.

Since the probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that segment

[Myers, 1979] such information may also aid an exploratory tester and may add to the already existent repertoire of methods and techniques.

3.8. Six principles of testing

Hetzel [1988] presents six principles of testing, elements of which are based on earlier work by others. Following are the six principles and thereafter a short reflection on how these principles apply to a scripted testing approach and an exploratory testing approach.

1. Complete testing is not possible: This is due to practical limitations and also a theoretical impossibility. The total number of possible test cases may be infinite which makes it practically impossible to try them all. There will furthermore never be a way to be sure that we have a perfect understanding of what a program is supposed to do.

2. Testing work is creative and difficult: Creativity, business knowledge and testing experience and methodology are required.

3. An important reason for testing is to prevent deficiencies from occurring: Testing is not a phase, but it must be an intrinsic part of the software development model, whichever one is chosen.

4. Testing is risk-based: Risk is to serve as the basis for deciding what and how to test.

5. Testing must be planned: Ad hoc testing does not provide enough information to reasonably measure software quality and may even be harmful due to the possibility of it leading to a false sense of security.

6. Testing requires independence: The tester should be unbiased and independent in spirit.

Principles 4 and 6 are relevant for both scripted testing and exploratory testing alike.

Principle 1 does not directly relate to which view is taken regarding testing; this principle is relevant for both exploratory and scripted testing, with a possibility of the exploratory testing being less "complete" by nature than the scripted testing.

Exploratory testing requires even more experience and creativity than may be required for scripted testing, but nevertheless principle 2 is relevant for both scripted testing and exploratory testing alike.

Principle 3 clearly fits with a preventative testing mindset. Due to the rather destructive nature of exploratory testing, principle 3 does not apply. Exploratory testing very often deals with complete systems and not with requirements or specifications, something which is also a downside [Van Veenendaal, 2004]. Problems that are spotted early (in requirements or system specifications) can also be fixed early at a much lower cost. Graham for example advocates a very tight link between testing and requirements [Graham, 2002]. This nevertheless presupposes that requirements and specifications are readily available and detailed enough. Especially within the field of GUI application development with its numerous iterations such may not be the case and none of the presented software development process models presented in Section 2, even provide for the explicit testing of requirements.

Scripted testing does not necessarily have to abide by testing principle 3 either, because a destructive mindset may also be employed there. Principle 3 is, as a result, very much dependent on the overall organisational view of testing, whether it is seen as a destructive process whereby errors need to be found or as a preventative process whereby the goal is on prevention. Both are possible as well, of course. Also the possibility of employing an exploratory way of testing throughout the software life-cycle are theoretically possible, but not within the scope of this work, where the application is limited to system testing.

Principle 5 is the one principle that exploratory testing does not abide by at all. Even though exploratory testing is test planning and execution at the same time, this is a different planning than is implied by Hetzel, who sees great relevance and benefits in thinking through and defining expected outcomes. Within exploratory testing expected outcomes are not defined per se, because by definition it is an exploratory process with an unknown result. Problems may or may not be encountered, formerly unknown issues may appear.

Principle 6 is relevant for both scripted testing and exploratory testing alike.

3.9. Testing models

Even though the process of testing may and should be viewed as being an inherent part of software development, and consequently part of the software development models, this does not imply that specific testing models are not required. Several testing models have been described by Gelperin and Hetzel [1988] and they differ basically along the dimensions of activity scope and primary goals.

The scope of activity has very much to do with the software development process model that is being adhered to. As in the waterfall model, the scope of testing is typically at the end, whereas in the V-model testing is performed on each stage of integration and the scope is hence much wider. Typically, as depicted in Figure 6, a test model would include test planning, test design, test implementation, test execution, test result gathering and test maintenance [Gelperin and Hetzel, 1988; Hetzel, 1988].

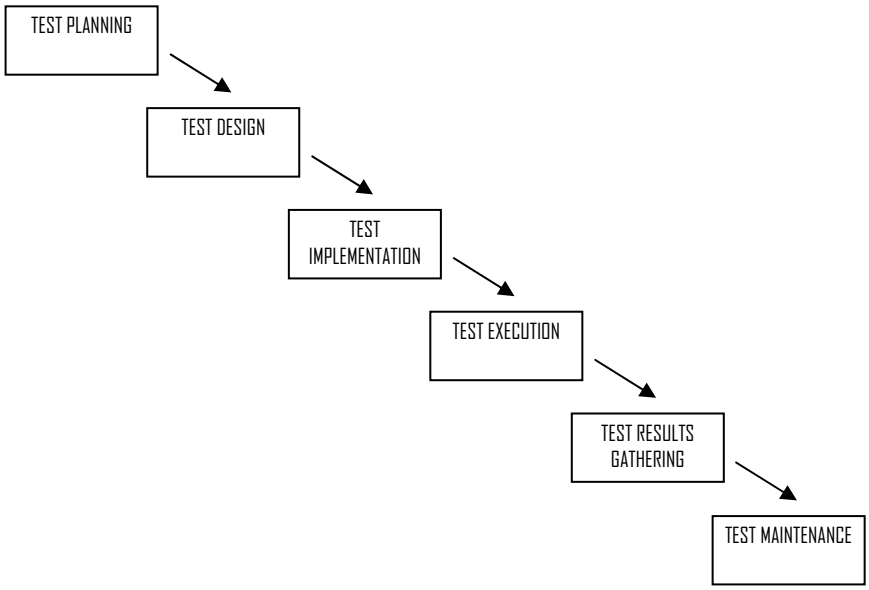

Figure 6. A typical model of testing [Hetzel, 1988].

With an exploratory testing approach, all of the steps depicted in Figure 6, happen at the same time, during the testing session. Test maintenance is the only step that is not necessarily required with exploratory testing, as was discussed in Section 3.6. With a scripted approach to testing, the test planning and test design steps happen much earlier than actual test execution. There is hence not much possibility of feedback acquired during testing to flow back into the test design.

3.10. Summary

A definition of commonly used terms to denote a failure in an application (error, fault, defects and mistakes) was provided, followed by a definition of testing. It was shown that different definitions of testing are possible, depending on what is being seen as the primary goal. Several levels of testing and the concepts of black-box and white-box were explained to prepare the reader for the empirical part of this work, which has concentrated on integration and system level black-box testing. Scripted testing and exploratory

testing were explained and discussed, and the concept of a testing oracle was clarified to point out the difficulty an exploratory tester, or a scripted tester without an up-to-date test specification, is faced with when having to decide whether what has been observed is intended functionality or indeed a failure. Furthermore, the six principles of testing were presented together with a short discussion on their application to scripted and exploratory testing. A general testing model description was provided to offer a framework specifically for testing, irrespective of whether such testing is performed in a scripted or exploratory way.

4. Case study

Since scripted testing seems obviously challenged within the context of iterative GUI application development, and changing requirements and specifications, it is imaginable that a different approach to testing could be beneficial. Such an approach may well be exploratory testing. Exploratory testing, due to its nature, cannot fully replace scripted testing, but can be used in addition to a scripted testing approach. The following section provides the reasoning that led to the case study, together with relevant background information. The results of the study are presented in Section 4.2.

4.1. Background

Scripted testing needs accurate requirements and specifications as input. If requirements and specifications are vague or changing during the time of development, scripted testing becomes challenging, because already existing test cases need to be modified to reflect the changes. Several software development models were described in the first sections of this work and changes in requirements were handled better by some and worse by others, but within all models testing was influenced by such changes.

It was pointed out in Section 2 that successful GUI application development requires iterations. These iterations in turn lead to a need for application specification updates and these in turn lead to a need for test specification updates. As is self-evident, documentation updates are time consuming. There may hence be times, when the application under development is different to what it ought to be, based on requirements, specifications and test specification documentation, simply because these had not been updated yet at that specific point in time. Moreover, through several iterations and evolving software, issues may appear, that could not have been conceived of earlier with only requirements and specifications as base and in turn requiring an update of relevant specifications. Furthermore, requirements may change during the development period, due to e.g. a change in customer demand or through simply becoming clearer and more detailed as the application development progresses, also requiring documentation updates. An exploratory approach to testing, due to its nature, might provide an increase in testing productivity compared to a purely scripted testing approach.

In order to work on the problem of whether exploratory testing in addition to scripted testing would yield higher testing productivity, it is argued that if it were possible to measure for a particular application, the number of error

reports received from testers based on exploratory testing and contrast these with the number of error reports received from testers, based on scripted testing then this would be one part of the answer. In addition a careful analysis of received error reports, especially their severities would be necessary.

This work will focus on error reports made during system and integration testing of the Nokia Text Message Editor, because for these testing levels, exploratory testing is mostly applicable. System and integration testing in this context refer to manual GUI application testing, taking all elements of the GUI into consideration, as well as the underlying functionality of the application. All testing is purely of a black-box nature and only non-duplicate errors reported for the English version (English UK) of the Nokia PC Suite, described in the next section, will be considered.

4.1.1. Nokia PC Suite

The Nokia PC Suite is a package of Windows-based GUI applications developed especially for use with Nokia phones. It can be downloaded without charge from http://www.nokia.com/pcsuite. The Nokia PC Suite is comprised of 10 applications and is localised into 33 languages, in addition to English.

One of the applications, the Nokia Text Message Editor is of relevance to this study [PC Suite, 2004]. The Nokia Text Message Editor enables users to open and edit, receive and send text messages, reply to text messages, as well as forward and print them. They can also organise text messages into folders using the Messages view offered in the Nokia Phone Browser. An application screenshot is provided in Figure 7.

Figure 7. Nokia Text Message Editor User Interface of PC Suite 6.4.

The Nokia Text Message Editor was part of the Nokia PC Suite 6.3 for the first time and is hence quite a recent addition to the application portfolio of the Nokia PC Suite. It was chosen as target for this study, because it was a new application and had as a result never been tested before. There was therefore the same starting point for both scripted and exploratory testing measurements. With any of the other applications in the Nokia PC Suite portfolio that would not have been possible, because earlier testing had already been performed or was ongoing. Test cases were created for the Nokia Text Message Editor as part of the normal scripted testing approach and the test cases were used for the testing of the Nokia Text Message Editor, irrespective of the Nokia PC Suite version, i.e. the test cases remained the same for all the versions studied and were updated only twice during the study.

The Nokia PC Suite is developed within an iterative and incremental software development model, based on the V-model, with slight elements of concurrency. Testers can test features not yet in any officially available Nokia PC Suite version by means of special development versions which are made regularly. Due to this, the testing of the Nokia Text Message Editor application has been possible already before it was officially launched.

4.1.2. Error report analysis time frame

The application of interest to this research, the Nokia Text Message Editor, was released for the first time as part of the Nokia PC Suite 6.3, which was launched on the Nokia web site during week 36/2004. Error reports between the release of the Nokia PC Suite 6.1 (week 18/2004) and the launch of the Nokia PC Suite 6.4 (week 40/2004) will be analysed. The release date of the Nokia PC Suite 6.1 has been chosen as starting date, because testing work on the Nokia Text Message Editor started thereafter. The launch times, together with the error analysis time frame are shown in Figure 8.

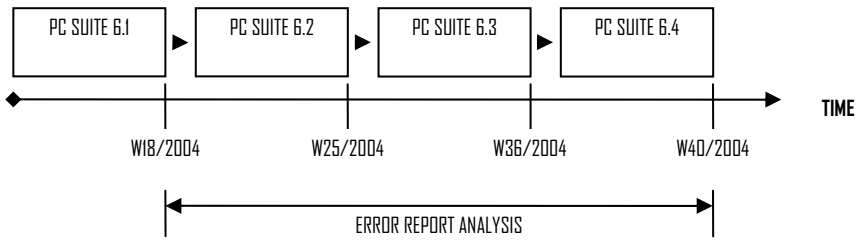

Figure 8. Nokia PC Suite launch dates and error report analysis time frame.

4.1.3. Changes to the error reporting template

Errors found by Nokia PC Suite testers are entered into an error database through an error reporting template. This template was modified already during 2003 and one new field was added, called "Test Method". The values for the field could be either "Test Case" or "Ad hoc" and could be chosen by means of a radio button. The default was "Test Case". All testers were informed that whenever they reported an error they had to also specify whether the error reported was a result of an existing test case ("Test Case") or obtained through other means ("Ad hoc"). This change to the error reporting template would enable later tracking of "Test Case" error reports and "Ad hoc" error reports.

The term "ad hoc" was chosen instead of "exploratory", because exploratory testing is also known as ad hoc testing [Bach 2003a] and it was felt at the time that it would be a more descriptive term and its meaning more intuitively known to testers at the time when the change was made. It was furthermore reasoned that by only making a distinction between reported errors based on test cases and those not based on test cases that those errors based on "purely" exploratory testing would inevitably be part of the group of non-test case based ones and such an easy distinction would not be difficult for the testers.

4.1.4. Error severities

Application errors submitted to the error database are classified according to their severity by testers at the time of reporting and re-classified if needed by an error manager, to prevent false classification. Table 1 lists the four severity levels that have been used in the analysed error reports together with a short explanation in terms of observed symptoms.

SEVERITY	OBSERVED SYMPTOMS
Soft crash	Application is inoperable or crashes.
Major	Some feature is inoperable; data is destroyed; no workaround exists.
Minor	Inconvenience is caused; the malfunctioning has a minor end-user impact; a workaround exists.
Cosmetic	Purely cosmetic, no impact on operation.

Table 1. Four severity levels and their explanations.

In addition to the severity of an error, it may also important to know something about its probability of occurrence. One error may be serious, but

not very likely or even improbable to occur, whereas another may also be serious, but very likely to appear frequently. It is imaginable, that for an end user, the latter is much more serious than the former. Several probability levels are used in Nokia PC Suite error reports, but for this research, during data analysis the element of probability is not considered, because the emphasis is on whether a non-scripted approach to testing could yield a relevant increase in reported errors than would a purely scripted approach. Also, due to practical reasons this information has not been available for analysis.

4.1.5. Information and material for testers

Testers were informed about the changes to the error reporting template and that as a result, better visibility to the origins of error reports would be obtained; they were not told that the data could form the basis of a possible later study in order to not influence them in any way. The change to the error reporting template had been done already during 2003 and by the time data was being collected for this research during 2004, the field was nothing new to testers any more and had become something they filled every time they made an error report, i.e. it had become part of the normal error reporting process.

The testing team leader was informed about the ongoing study. The Windows Applications Exploratory Test Procedure document [Microsoft 2004] was distributed to testers and the reporting of errors not based on test specifications was encouraged and had been encouraged already during 2003.

4.2. Results

The results are exhibited in several ways, emphasising different aspects. In Section 4.2.2, the mere number of reported errors based on a test case and those not based on a test case for every week in the analysis time window, will be presented and discussed.

In Section 4.2.3, the ratio of all errors reported based on a test case and all those not based on a test case by severity (“cosmetic”, “minor”, “major” and “soft crash”) will be presented and discussed. Additionally, the share of the individual severities by test method (“Test case” and “Ad hoc”) for all error reports will be examined.

In Section 4.2.4, the ratio of errors based on a test case and those not based on a test case by week will be introduced followed by a discussion.

In Section 4.2.5, the testers' experience levels will be looked at by presenting their experience in years, against their respective ratio of "Test case" and "Ad hoc" error reports.

In Section 4.2.6, the actual increase in productivity will be presented and discussed. In the subsequent sections possible limitations will be contemplated and the findings summarised.

4.2.1. Error reports by testers vs. all reports

Typically, error reports are not only received from testers within the actual testing project, but also from developers and testers in other related projects that also perform some Nokia PC Suite testing as part of their testing. In many cases, such reports are not entered directly, but transferred by the error manager.

Whether a test case has been based on a script or not cannot be determined with certainty in such a situation. For this reason, the data analysis is made only based on error reports reported by testers within the testing project, i.e. the testers specifically assigned to Nokia PC Suite testing and not from all error reports, including reports from people outside the testing project. Within the group of testers in the testing project all are assigned to Nokia PC Suite testing, but not all are assigned to testing the Nokia Text Message Editor.

4.2.2. Test case vs. ad hoc error reports by week

Following is an analysis of those error reports made only by testers in the testing project during each week between weeks 18 and 40. The emphasis is on the number of error reports based on a test case (”Test case”) against the number of error reports not based on a test case (”Ad hoc”). The actual numerical value is not supplied, but a ratio of the number of error reports and the number of the Nokia Text Message Editor's non-commented source statements (lines of code), divided by 1024, also known as KNCSS. KNCSS represents a count of all source lines excluding comments and blank lines [Grady and Caswell, 1987]. As is common practise, error reports marked as duplicate are not considered in the analysis.

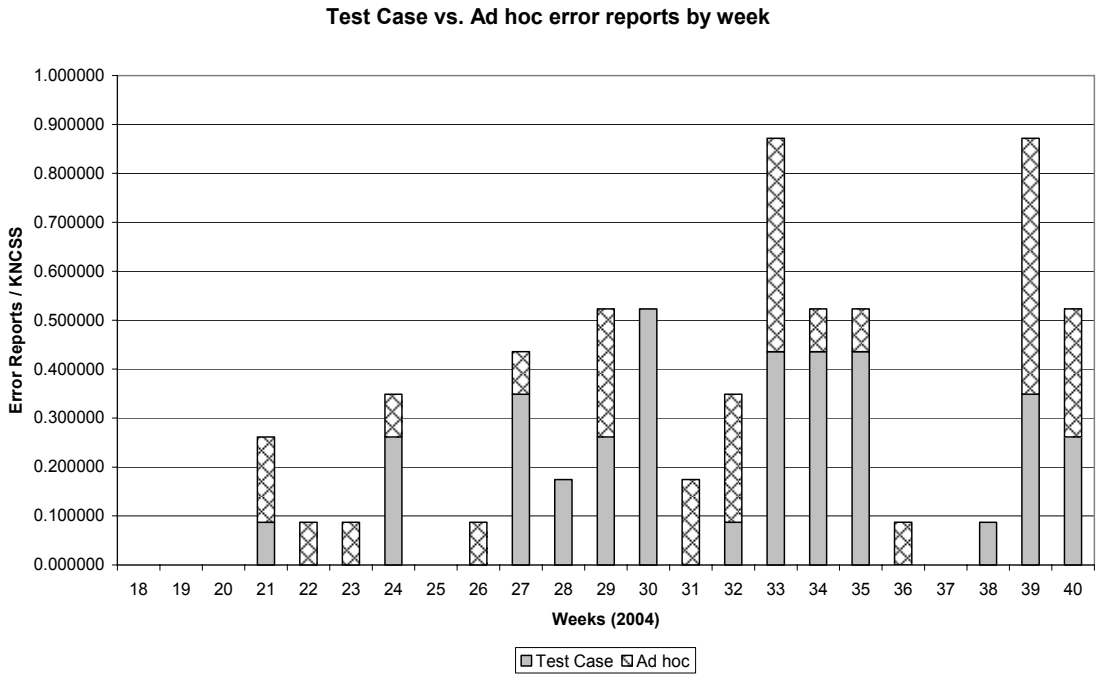

Figure 9. ”Test Case” vs. ”Ad hoc” error reports by week.

The data presented in Figure 9 suggest that no testing was performed prior to week 21. Furthermore, no error reports were created during weeks 25 and 37, but no clear reasons could be found for that. It is nevertheless evident from the data in Figure 9 that an exploratory testing approach did result in more reported errors than a scripted approach alone.

It must be noted that there is a possibility that not all errors reported under ”Ad hoc” were truly found through an exploratory testing session and may have been found purely by ad hoc means, i.e. simply by accident. The error reports considered in this analysis are of unique errors, i.e. duplicate error reports have not been considered. Testers were informed on many occasions about exploratory testing techniques and exploratory testing sessions were conducted during the testing of the Nokia Text Message Editor. It can therefore reasonably be assumed that testers were aware of the concepts of exploratory testing at the time of testing.

It is not important for this work, whether all the ”Ad hoc” results have been obtained exclusively by means of exploratory testing sessions, because what is important is that testing activities apart from the scripted, i.e. ”Test case” approach have even occurred. Based on these findings, as will be again mentioned in the conclusion of this work, exploratory testing skills of the testers need to be developed further, because non-scripted testing appears to be very relevant for GUI application testing.

In all weeks, except weeks 28 and 38, some ”Ad hoc” error reports have been made. During weeks 22, 23, 26, 31 and 36 only ”Ad hoc” error reports were received. This tends to suggest that either testers did not perform any scripted test cases during these weeks or they simply did not find any errors based on their test cases, but there is no data available to say that with certainty. The data available regarding test cases is that during week 21 and week 27 the test specifications were updated. This fact might explain the higher than before number of “Test case” error reports for week 27 and absolutely no “Ad hoc” error reports during week 28. It could be argued that testers were “keen” on running through the updated test specifications and that there was no time for any other testing activities. Overall, most exploratory “Ad hoc” error reports were received during week 39, but no sound explanation for this could be found.

Whether exploratory testing in addition to scripted testing could yield a benefit in testing productivity, within the confines of GUI application system testing cannot yet clearly be answered. An insight not only into the mere number of reports, but also into their severities is required and this is the focus of the following section.

4.2.3. Test case vs. ad hoc error reports by severity

In order to look at actual productivity as it was defined earlier as not only the number of errors found, but also to their severities, an analysis of severities is required.

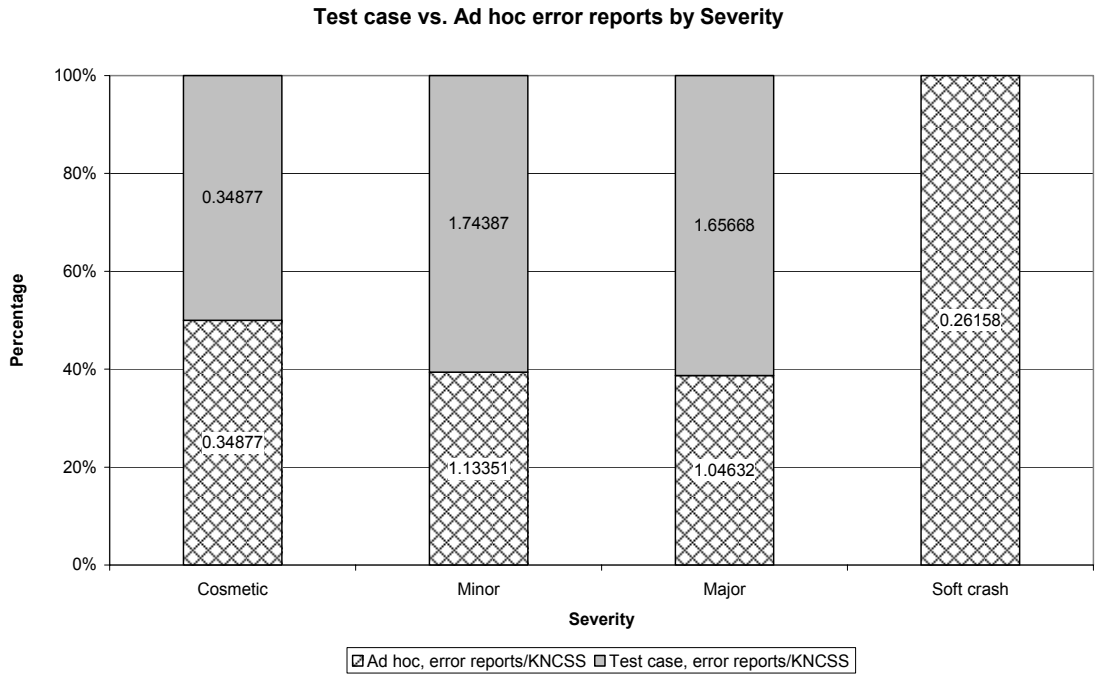

Figure 10. Test case vs. Ad hoc error reports by Severity.

Figure 10 shows the ratio of ”Test case” error reports and ”Ad hoc” error reports by severity. Within the individual bars, the number of error reports divided by KNCSS is presented to provide an insight into the number of reports and thereby set the ratios into perspective.

The data suggest that errors based on a non-scripted testing approach (”Ad hoc”) are reported for every severity and are not limited to a certain severity. Furthermore the ratio of ”Test case” errors to ”Ad hoc” errors is higher for cosmetic and soft crash errors than for minor and major errors. It has to be borne in mind though, that the actual number of error reports in the cosmetic and soft crash categories are much less than those in the minor and major categories. For cosmetic errors the ”Ad hoc” part is slightly below 50% and for soft crash errors all (100%) of the reported errors in that severity category are ”Ad hoc”, i.e. not based on a test case. This suggests that within the confines of this study, no soft crashes, i.e. very severe errors, were found based on a test case, but were found exclusively based on an exploratory approach. This finding is also in line with the common notion that complicated interactions of components and modules when everything has been integrated into one application cannot easily be grasped and for that matter documented in scripted test cases, before actual implementation.

Most errors are reported within the minor and major categories. In order to get a slightly better visibility into the actual share of the four severities broken

down by test method, the respective data is represented in from of a pie chart in Figure 11.

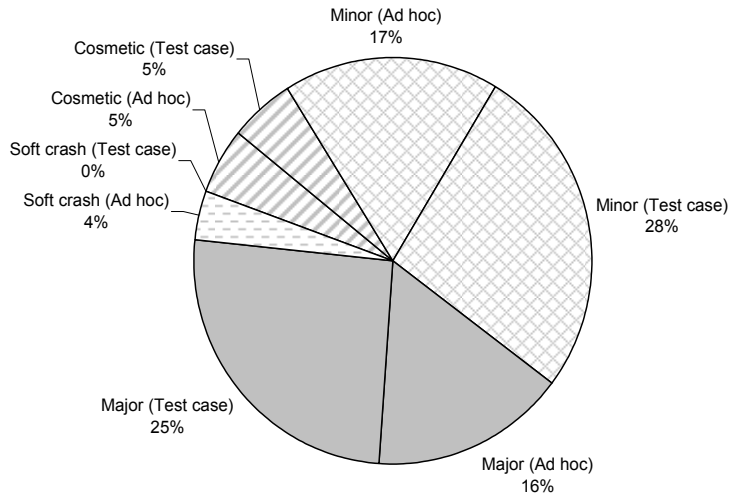

Figure 11. Share of severities by test methods.

The pie chart in Figure 11 breaks down the severities by test methods for all reported error reports reported by testers in the testing project. This shows that out of the total number of reported errors by testers, 10% are cosmetic, 45 % are minor, 41% are major and 4% are soft crash. Interesting here is that the share of cosmetic ”Ad hoc” and cosmetic ”Test case” is equal at 5% each. The share of the number of minor ”Ad hoc” and major ”Ad hoc” error reports is nearly the same at 17% and 16% respectively and also the share of minor ”Test case” and major ”Test case” error reports is nearly the same at 28% and 25% respectively. This means that the ratio of ”Ad hoc” error reports to ”Test case” error reports within the severities minor and major is about 50%.

4.2.4. Ratio of test case vs. ad hoc error reports by week

Figure 9 provided an insight into the number of error reports made per week, based on either a test case ”Test case” or none ”Ad hoc”. This showed that an exploratory approach to testing does yield more reported errors. Next, information about the actual ratio of ”Ad hoc” vs. ”Test case” error reports per week is provided in Figure 12, followed by a discussion.

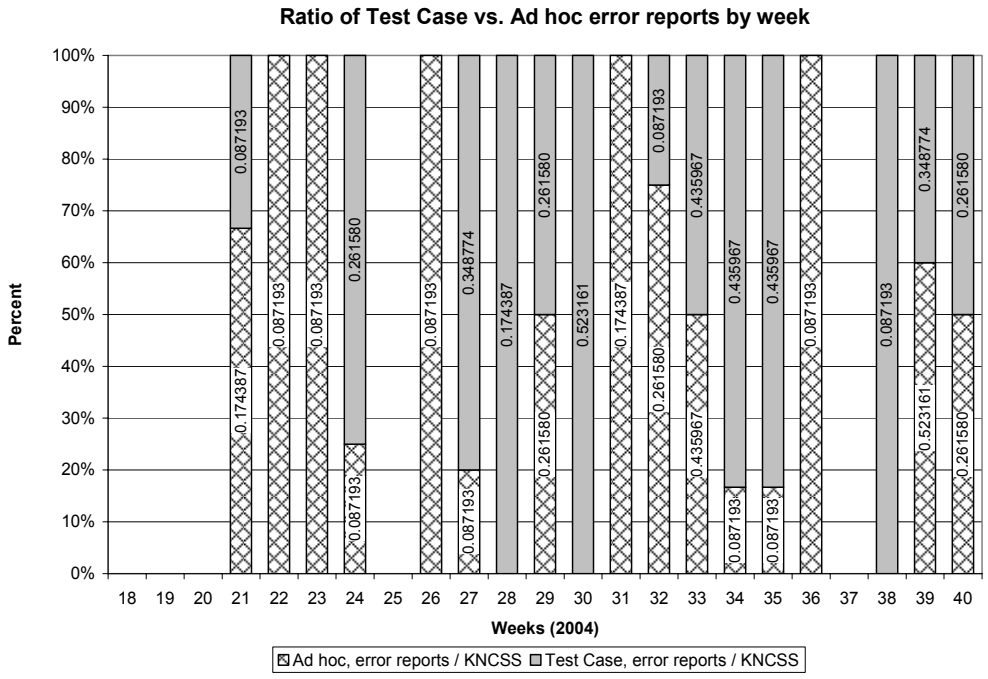

Figure 12. Ratio of ”Test case” vs. ”Ad hoc” error reports by week.

During weeks 34 and 35, the percentage of error reports based on a test case, ”Test case”, is over 80%, and the error reports per KNCSS numerical value is quite low at 0.087193, which suggests very intensive testing before the release of the Nokia PC Suite 6.3 (week 36), and not much extra time for exploratory testing, but there is no data available to explicitly support the claim. Numerically (error reports per KNCSS), there is no difference between the number of “Test case” errors submitted during weeks 33, 34 and 35 (0.435967), but there is much more exploratory testing in week 33, than in weeks 34 and 35. A decline in the ratio of ”Ad hoc” error reports vs. “Test case” error reports can be observed already from week 31 onwards up to week 34, but interestingly, numerically there has actually been an increase in reported “Ad hoc” errors from week 31 to week 33. Then, during the release week (week 36) only non-scripted testing was performed and during week 37 no testing at all.

Numerically, the overall largest amount of “Ad hoc” error reports have been received in week 39, even though “Ad hoc” testing makes only 60% of all errors reported for that week. No clear data-supported explanation could be found for that, except for the speculation that shortly before the release of PC Suite 6.4 (week 40) testers tested very intensely and were working hard at “breaking” the software.

Overall, the percentage of error reports made, not based on a test case ”Ad hoc” is over 15% for every week, except for those weeks with no testing activities (18-20, 25 and 37) and weeks 30 and 38. It is 20% and more, if not counting weeks 34 and 35. The percentage of ”Ad hoc” error reports made by severity, has been discussed in the previous section.

4.2.5. Experience and ratio of test case vs. ad hoc error reports

It has been argued by e.g. Van Veenendaal [2004] that exploratory testing requires skills that inexperienced testers may not possess. The ratio of error reports classified as ”Test case” and ”Ad hoc” is contrasted in Figure 13 against the experience level of testers A to P, who were part of the testing project.

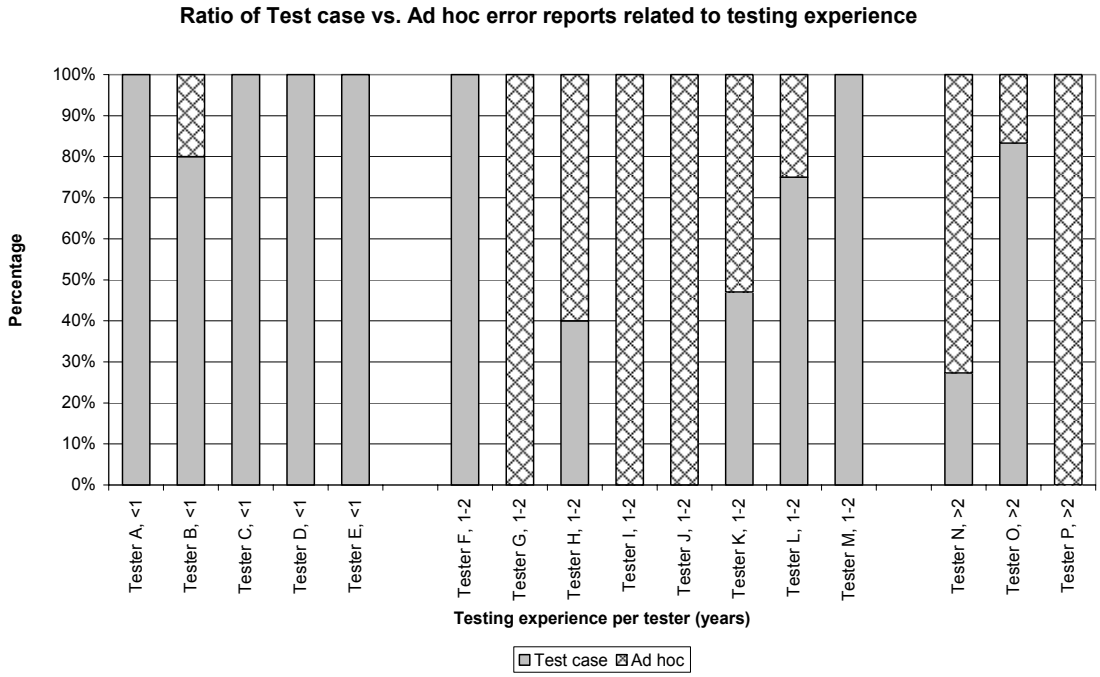

Figure 13. Test case vs. Ad hoc, related to experience.

It can clearly be seen in Figure 13, that those testers with less than 1 year of testing experience have reported hardly any errors not based on a test case, whereas those with more testing experience have reported more such errors, which tends to support the claim that exploratory testing requires a certain amount of testing experience.

The 100% level of ”Ad hoc” reports for the testers G, I and J in the group of testers with a testing experience of between 1 and 2 years may be indicative of a tester who has not tested the Nokia Text Message Editor as a main application, but who has embarked on some non-test based testing of the application. There is no data available to confirm whether this is the case. For those in the same

group, who have tested also with test cases, the ratio of ”Ad hoc” to ”Test case” is higher than for those testers with less than one year experience.

The testers with more than two years experience have a very varied ratio of ”Ad hoc” vs. ”Test case” error reports. Tester P has only reported ”Ad hoc” errors which may be an indication that the tester had not been specifically assigned to testing the Text Message Editor, but there is no data available to clearly support this.

Overall the data suggest that testing experience plays a role in how testers are able to deal with non-test case based errors and that exploratory testing requires skills that may not be available in testers with little to no testing experience.

4.2.6. Increase in productivity

Figure 14 presents the actual increase in productivity, i.e. the actual increase in error reports for each severity, calculated as an increase in percent, taking as baseline the number of error reports received in the ”Test case” category.

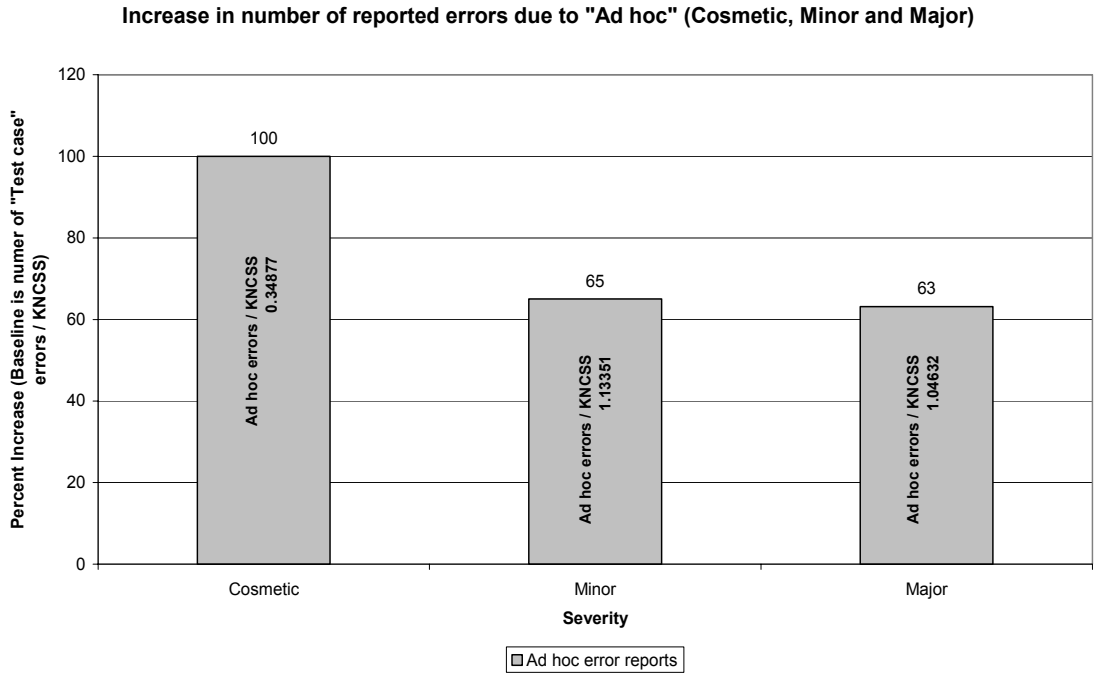

Figure 14. Increase in number of reported errors due to ”Ad hoc”.

For the cosmetic severity, a 100% increase (0.34877 errors per KNCSS) has been observed when taking the number of ”Test case” errors per KNCSS as baseline and calculating the increase using the number of ”Ad hoc” cosmetic error reports. The increase for minor is 65% (1.13351 errors per KNCSS) and for

major 63% (1.04632 errors per KNCSS). There is no percentage increase for soft crash ”Ad hoc” error reports, as no soft crash errors have been reported based on test cases. This means that overall, the use of exploratory testing, as measured through the reporting of errors as ”Ad hoc”, has yielded 0.26158 soft crash errors per KNCSS, as compared to no such reported errors in the “Test case” category. To put these numbers into perspective, it can be said that the increases in each category is numerically less than 50 errors.

4.2.7. Limitations

The population of testers whose error reports were analysed is not homogenous and their educational as well as experience levels vary. It may be argued that better skilled testers are better able to test in an exploratory fashion than testers with little or no testing experience. As a result, if there are many experienced testers in a testing team, more ”Ad hoc” error reports could be expected. The experience levels of testers contrasted against their “Ad hoc” vs. “Test case” error reports has been presented in Section 4.2.5. The experience levels have however not been taken into account in the data analysis, where results from all testers were combined and considering those as an attribute would therefore be a good starting point for future work. It can be assumed that within any testing project a variety of testers with different educational and experience levels are usually present and thus the results obtained in this case study may be of relevance also to other, similar projects. Below a chart depicting the experience levels of the testers who were part of the case study presented in this work.

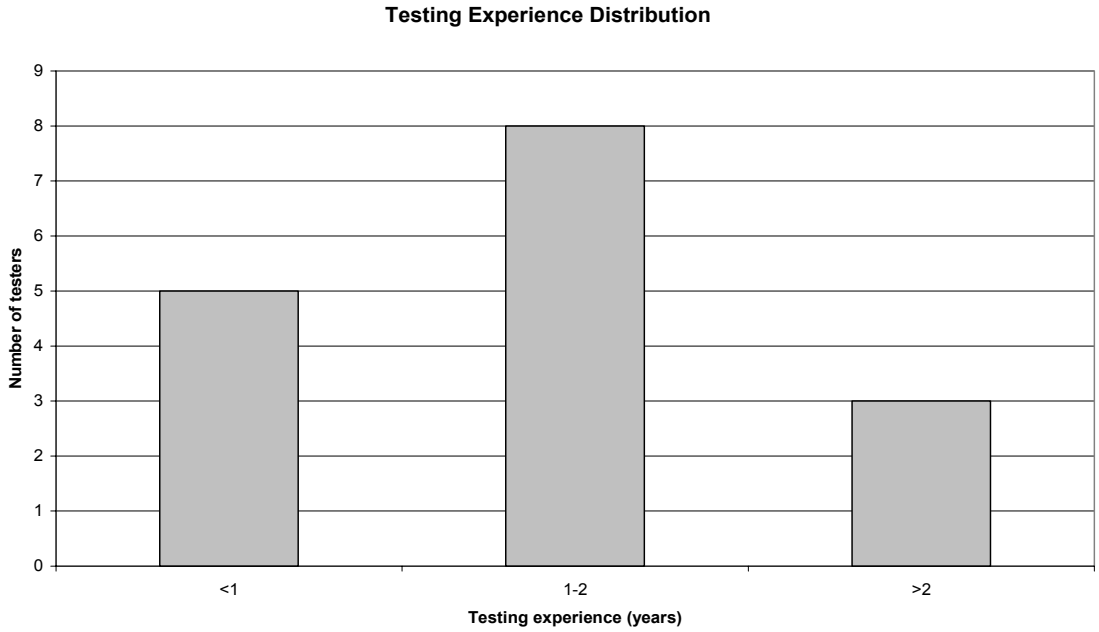

Figure 15. Testing experience distribution.

From a total number of 16 individual testers, the majority fall within the 1 to 2 years experience category. Five testers have less than 1 year of testing experience and 3 have more than 2 years experience.

Not all the 16 testers who submitted error reports were assigned to test the Nokia Text Message Editor application and they may have consequently observed problems with the application while testing their own assigned application(s). Such error reports would then have been marked as ”Ad hoc”. Since the emphasis of this work has been on whether exploratory testing could increase testing productivity, no effort has been made to concentrate only on the testers specifically assigned to test the application, but a more holistic approach was taken. This approach was to consider all testers within the Nokia PC Suite testing project who contributed to the Nokia Text Message Editor testing, because it was felt to be important to consider the testing project as close to real life as possible. It was furthermore reasoned that had the testers who were not specifically assigned to test the Nokia Text Message Editor application been ”forbidden” to report any errors not based on any test case, no ”Ad hoc” error reports would have been received from them. The fact that they were received shows the great potential of exploratory testing to increase the overall testing productivity.

There have been no control measures in place to check whether testers had performed exploratory testing properly before they were allowed to report an

"Ad hoc" error and therefore there is a possibility that the "Ad hoc" classified error reports include both errors found by true exploratory means and errors found by pure chance, i.e. not within an exploratory testing session. There is a very fine line between such "chance" findings and exploratory testing and it is primarily about control in the process. Exploratory testing is a controlled process with a testing charter and testers had been informed about exploratory testing guidelines on several occasions. It was reasoned that it is always possible to observe a failure without having done anything, i.e. without having started an exploratory testing session and it is important to report such observations. Such an observation would have resulted in an "Ad hoc" error report. Not having absolute certainty that all "Ad hoc" error reports were really based on true exploratory testing, was therefore not considered a hindrance in this research.

Test specifications were updated in weeks 21 and 27. The "Test case" error reports received prior to week 27 are therefore based on test specifications whose content is not exactly the same as the content of the test specifications used during and after week 27. Since the differences in content are minor and since the question this work tries to answer is whether exploratory testing in addition to scripted testing could bring about an increase in productivity, it was reasoned that the update of test specifications would be acceptable and that data would be analysed from week 18 to week 40.

4.2.8. Summary

It was found that non-scripted testing does find errors. Of all errors reported, 43%, or nearly half, were categorised as "Ad hoc", i.e. not based on any test case. (The "Ad hoc" error reports include errors found by exploratory means).

An increase in non-test based error reports was found for all severities, not only for certain ones. The actual increase in number of errors found in addition to purely scripted testing was 100% (0.34877 errors per KNCSS) for "cosmetic" errors, 65% (1.13351 errors per KNCSS) for "minor" errors and 63% (1.04632 errors per KNCSS) for "major" errors, i.e. the highest numerical increases were for "minor" and "major" errors. Numerically these numbers correspond to increases of less than 50 error reports per severity. No "soft crash" errors were found with a scripted testing approach, whereas an exploratory testing approach yielded 0.26158 "soft crash" errors per KNCSS. Numerically this also corresponds to less than 50 such reports. This finding suggests that very severe errors cannot be detected with the help of scripted test cases, but are better detected using an exploratory testing approach. This may be so, because the

complexity in the interplay of components and modules leading to possible failures is not easy or even impossible to grasp and translate into scripted test cases during an early phase of software development, but becomes more apparent once everything has been integrated. This claim would be supported also by Kaner *et al.*'s argument [2002, p. 72] that even if a product was fully designed in advance, people do not fully understand the system until it is built.

The highest percentage increase (100%) was found for the "cosmetic" severity, supporting a conclusion that scripted test cases are not particularly good at catching cosmetic errors. Cosmetic errors are very likely caused by application design iterations. These iterations are essential for good GUI application design and hence exploratory testing appears to be a superiour approach to testing such applications.

Van Veenendaal's [2004] claim that exploratory testing requires a certain level of experience and skill could be confirmed. Testers with less than one year of testing experience reported less errors based on exploratory methods than did testers with more experience.

Overall testing productivity has been higher when allowing and employing an exploratory approach to testing than with only a scripted testing approach.

5. Conclusion

The development of GUI applications is challenging and only possible through numerous iterations [Myers, 1995]. This continuous change, together with a market-driven change in requirements, makes the creation of appropriate test cases demanding. Test specifications simply cannot be written once during application design and remain static until application delivery. There is also a certain danger that test specifications, even if updated regularly, are never quite corresponding to the application's design and functionality at that same moment in time. This problem has been described in the introductory section.

In Section 2, software development models were presented to allow the reader to position testing within the different models and to see the challenges that a change in the requirements put on the various models in general and on testing in particular. In Section 3 several testing definitions and principles were shown, including scripted and exploratory testing. The section concluded with the presentation of a testing model.

It was argued throughout this work that exploratory testing could possibly help in a situation of constant change, because test design and execution happen at the time of testing and are not predefined at an earlier stage, as prescribed by all presented software development models. In Section 4, the research problem was presented together with the approach that was used in this research for arriving at a conclusion for the question of whether an exploratory approach to testing, within the realms of GUI application testing, could indeed result in an increase in testing productivity than a scripted approach alone.

The analysis of the data allows for a conclusion that an exploratory approach to testing does increase testing productivity, when compared to a strictly scripted testing approach. It was found that all of the very severe or so-called "soft crash" error reports were reported as based on exploratory testing and out of all errors reported, 43% were based on exploratory testing. The results of this research also allow for a conclusion that an exploratory testing approach was generally very powerful across all severity categories (cosmetic, minor, major and soft crash) and that the approach clearly increased the testing productivity.

Testers must be given the opportunity and be encouraged to participate in relevant training to increase their skills and become better explorers, because it was found that testers with less than one year of testing experience reported

less errors based on exploratory methods than did testers with more experience.

The findings of this research should serve as an incentive to employ exploratory testing methods in GUI application testing projects and to increase awareness of exploratory testing.

5.1. Future work

The influence of testers' educational level and testing experience may have an influence on how readily exploratory testing can be performed and errors found. Future work could take the experience and educational levels of testers into consideration and analyse what the effects on exploratory testing are.

Learning styles may also play a role in exploratory testing and research in that area is currently performed by Tinkham and Kaner [2003]. Their paper discusses their initial work into examining a tester's learning style as an indication of the types of actions she might use while doing exploratory testing. It may be interesting to apply their findings also within the context of productivity, i.e. whether certain learning styles may have an effect on testing productivity.

The issue of time spent on exploratory testing as compared to time spent on scripted testing has not been taken into consideration in this work and could form an interesting base for future research in this area.

References:

[Agruss and Johnson, 2000] Chris Agruss and Bob Johnson, Ad hoc Software testing – a perspective on exploration and improvisation. Available at http://www.testingcraft.com (Checked June 2004)

[Aoyama, 1993] Mikio Aoyama, Concurrent-development process model. *IEEE Software* **10**, 4 (July 1993), 46-55.

[Bach, 2002] James Bach, Exploratory testing. In: Erik van Veenendaal (ed.) *The Testing Practitioner*, UTN(Uitgeverij Tutein Nolthenius) Publishers, 2002, 209-221.

[Bach, 2003a] James Bach, Exploratory Testing Explained v1.3.4/16/03. (First published as a chapter in The Testing Practitioner [Van Veenendaal, 2002], but v1.3.4/16/03 is different from the version published). Available from http://www.satisfice.com (Checked December 2004).

[Bach, 2003b] James Bach, Inside the mind of an exploratory tester. *Software Testing and Quality Assurance (STQE)* **5**, 6 (November/December 2003), 16-23.

[Beizer, 1990] Boris Beizer, *Software Testing Techniques 2nd Edition*. International Thomson Publishing, 1990.

[Beizer, 1995] Boris Beizer, *Black-Box Testing: Techniques for Functional Testing of Software and Systems.* John Wiley & Sons, 1995.

[Boehm, 1988] Barry W. Boehm, A spiral model of software development and enhancement. *IEEE Computer* (May 1988), 61-72.

[Craig and Jaskiel, 2002] Rick D. Craig and Stefan P. Jaskiel, *Systematic Software Testing*. Artech House Publishers, 2002.

[Dyer, 1980] M. Dyer, The management of software engineering. Part IV: Software development practices. *IBM Systems Journal* **19**, 4 (1980), 451-465.

[Eliot, 1975] T.S. Eliot, *Collected Poems 1909-1962*. Faber and Faber, 1975.

[Gelperin and Hetzel, 1988] David Gelperin and Bill Hetzel, The growth of software testing. *Communications of the ACM* **31**, 6 (June 1988), 687-695.

[Goldsmith and Graham, 2002] Robin F. Goldsmith and Dorothy Graham, The forgotten phase. *Software Development* (July 2002). Available from http://www.sdmagazine.com (Checked August 2004)

[Goldsmith, 2002] Robin F. Goldsmith, This or that, v or x. *Software Development* (August 2002). Available from http://www.sdmagazine.com (Checked August 2004)

[Grady and Caswell, 1987] Robert B. Grady and Deborah L. Caswell, *Software Metrics: Establishing A Company-Wide Program*. Prentice-Hall, 1987.

[Graham, 2002] Dorothy Graham, Requirements and testing: seven missing-link myths. *IEEE Software* **19**, 5 (September/October 2002), 15-17.

[Hetzel, 1988] Bill Hetzel, *The Complete Guide to Software Testing 2nd Edition.* John Wiley & Sons, 1988.

[IEEE 1990] *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Std 610.12-1990.

[IEEE 1998] *IEEE Standard for Software Test Documentation.* IEEE Std 829-1998 (Revision of IEEE Std 829-1983).

[Kaner, 1999] Cem Kaner, Don´t use bug counts to measure testers. *Software Testing and Quality Engineering* **1**, 3 (May/June 1999), 79-80.

[Kaner, 2003] Cem Kaner, How many lightbulbs does it take to change a tester. Presented at: *Pacific Northwest Software Quality Conference 2003*. Available from http://www.kaner.com/articles.html (Checked September 2004)

[Kaner *et al.*, 1993] Cem Kaner, Jack Falk and Hung Q. Nguyen, *Testing Computer Software 2nd Edition*. International Thomson Publishing, 1993.

[Kaner *et al.* 2002] Cem Kaner, James Bach and Bret Pettichord, *Lessons Learned in Software Testing: A Context-Driven Approach.* John Wiley & Sons, 2002.

[Kaner and Bach, 2004] Cem Kaner and James Bach, The nature of exploratory testing. *Slides presented in Tampere, Finland, May 7th 2004*. Available from http://www.kaner.com/articles.html (Checked September 2004)

[Larman and Basili, 2003] Craig Larman and Victor R. Basili, Iterative and incremental development: a brief history. *IEEE Computer* **36**, 6 (June 2003), 47-56.

[Marick, 1999] Brian Marick, New models for test development. Paper presented at *Quality Week '99*. Available from http://www.testing.com (Checked September 2004)

[Memon and Soffa, 2003] Atif M. Memon and Mary Lou Soffa, Regression testing of GUIs. *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering* **28**, 5 (September 2003), 118-127.

[Microsoft, 2004] Windows Applications Exploratory Test Procedure v. 3.1 as part of Windows Application Compatibility Toolkit 3.0, 2004. Available from http://www.microsoft.com (Checked September 2004)

[Mills, 1980] H.D. Mills, The management of software engineering. Part I: Principles of software engineering. *IBM Systems Journal* **19**, 4 (1980), 414-420.

[Myers, 1995] Brad A. Myers, User interface software tools. *ACM Transactions on Computer-Human Interaction* **2**, 1 (March 1995), 64-103.

[Myers, 1979] Glenford J. Myers, *The Art of Software Testing*. John Wiley & Sons, 1979.

[Nielsen, 1993] Jakob Nielsen, Noncommand user interfaces. *Communications of the ACM* **36**, 4 (April 1993), 83-99.

[PC Suite, 2004] Nokia, *User's Guide for PC Suite 6.4*. 2004. Available from http://www.nokia.com/pcsuite (Checked August 2004)

[Pettichord, 2004] Bret Pettichord, Four schools of software testing. Presented at: *Austin SPIN (Austin Software Process Improvement Network, http://www.ovpro.net/aspin/) February 2004*. Available from http://www.pettichord.com (Checked September 2004)

[Rook, 1986] Paul E. Rook, Controlling software projects. *IEE Software Engineering Journal* **1**, 1 (January 1986), 7-16.

[Royce, 1970] Winston W. Royce, Managing the development of large software systems. In: Proceedings *IEEE WESCON* (August 1970), 1-9.

[Shneiderman, 1998] Ben Shneiderman, *Designing The User Interface: Strategies for Effective Human-Computer Interaction 3rd Edition*. Addison-Wesley, 1998.

[Sommerville, 1992] Ian Sommerville, *Software Engineering 4th Edition*. Addison-Wesley, 1992.

[SWEBOK, 2004] Alain Abran and James W. Moore, *Guide to the Software Engineering Body of Knowledge (SWEBOK)*. IEEE Computer Society, 2004.

[Tinkham and Kaner, 2003] Andy Tinkham and Cem Kaner, Learning styles and exploratory testing. This paper was originally prepared for and presented at the *Pacific Northwest Software Quality Conference 2003*. Available at http://www.kaner.com (Checked November 2004)

[Van Veenendaal, 2002] Erik van Veenendaal, *The Testing Practitioner*. UTN(Uitgeverij Tutein Nolthenius) Publishers, 2002.

[Van Veenendaal, 2004] Erik van Veenendaal, Exploratory testen – zinvol of onzin. *Automatisering Gids* 14 (2. April 2004).

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