MSc Business Informatics

A thesis presented for the degree of Master of Science

An adaptation of an evaluation framework for software testing techniques and tools

Betül Sezer 0884065

First Supervisor: Dr. Fabiano Dalpiaz

Second Supervisor: Dr. Fernando Castor

2023

Utrecht University

Faculty of Science

Abstract

Software testing is essential for ensuring the quality and reliability of software systems. To make informed decisions about testing techniques, there is a need for suitable evaluation frameworks. However, existing evaluation frameworks may not be applicable in limited scenarios, where conducting a case study and fault injection are not possible.
The primary objective of this research is to adapt an existing evaluation framework to enable the evaluation of software testing techniques experimentally. The focus is on redesigning this evaluation framework, which proposes clear metrics to evaluate the effectiveness and efficiency of software testing paradigms and validate the adapted framework to assess its applicability.
To validate the framework, a quasi-experiment was conducted, comparing GUI testing paradigms by applying the adapted evaluation framework. Due to limitations in sample size, the quasi-experiment does not provide statistical evidence but offers valuable insights and initial validation of the framework. While statistical significance is not achieved, the findings contribute to understanding the strengths and weaknesses of the redesigned treatment. Further research with larger samples and statistical analysis is recommended to strengthen the validity and generalizability of the findings.

Acknowledgements

This research project concludes my very exciting and challenging journey of completing my Master's in Business Informatics at Utrecht University. During my time here, I have had the privilege of meeting inspiring people from all over the world, which has shaped my personal growth. I am grateful for the friendships I have made here and cherish the memories of our shared experiences.

First of all, I want to thank my second supervisor Fernando Castor, whose valuable feedback provided me with crucial insights during critical moments. Moreover, I want to thank my first supervisor Fabiano Dalpiaz for his patience and dedication throughout this project. Despite the challenges we encountered, his motivation and enthusiasm helped to shape this research project into something significant. I feel grateful for having the chance to work with him.

I am grateful for all the research participants who generously took part in my experiment. This study would not be possible without them.

Lastly, I want to thank my friends and family who have been my strength and motivation throughout these two years. Their unlimited support, encouragement, and belief in me have been the driving force behind my academic journey.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem statement

An increasing number of systems are more and more integrated with software, be it a life-critical system, for which a failure in the software could result in death or big damage to the environment, or other systems that could cause loss of money and time. According to a report by Synopsys (2022), costs resulting from poor software quality are estimated to be 2.41 trillion USD alone for the U.S. in 2022. To avoid these costs and critical damages among other software engineering activities, software needs to be properly tested. Studies estimate the overall cost of testing activities in a software development project to exceed 40% of the total cost of the project (Rahikkala, Hyrynsalmi, & Leppänen, 2015). This shows how crucial the testing phase is in the Software Development Life-cycle (SDLC). Testing is a complex process done through different levels of a system's software. There are four different testing levels: unit testing, integration testing, system testing, and acceptance testing. An in-depth definition of the testing levels can be found in Chapter 4. This research focuses on automated testing of the Graphical User Interface (GUI) of a System Under Test (SUT). A GUI represents the interface of a system designed for human-computer interaction and is considered as another abstraction level of system testing. A SUT is defined as the testing object, that can refer to a software product, an application, or a system (Bourque, Fairley, & Society, 2014).
Testing activities can be done manually or semi-automatically by testing tools, that still require human intervention in creating test oracles and assessing the outcome, see Section 4.4.1. Manual testing is a process where a tester writes test cases and executes them manually. It is a time-consuming activity that requires the tester to be patient to perform repetitive testing tasks that can become even more difficult with larger software applications (Sharma, 2014). Manual testing is practical when there is no need to repeatedly test an application. In cases where repeating tests over a long period is necessary, repetitive tasks can become costly. Automating these testing tasks is a solution that aims to reduce costs and time and minimize human error (Ammann & Offutt, 2016). Some automated testing approaches, however, require technical knowledge that cannot be obtained very easily, especially script-based testing approaches that require programming knowledge like Selenium (Selenium,

2023a). In contrast, learning how to use a code-less test automation tool can be a lot easier in this case. Additionally, test automation still requires human intervention and effort to maintain test scripts for regularly changing test cases, especially in GUI testing.
Performing GUI testing activities is a challenging task itself as GUIs require constant changes which also increases the complexity of GUI systems and their testing. There exist several test automation tools with different levels of tolerance for GUI changes and different levels of automation that are based on different testing techniques.
In this thesis, we divide them into two categories of testing paradigms: script-based and script-less GUI testing. The aim is to evaluate the testing paradigms and to see how they differ in applicability in a practical setting. To do so, two different test automation tools were selected, that are based on each of the testing approaches. These tools will be applied in a practical setting to compare them with each other.
However, while working on this problem, we encountered an additional challenge. The initial plan was to apply the evaluation framework by Vos et al. (2012) to conduct a comparative study on GUI testing paradigms, but the case study we were planning did not materialize. This was due to the fact, that our specific context did not allow fault injection, and we did not have access to existing cases that could be used for our study. This limited scenario made it not feasible to conduct the predefined case study. Therefore, we turned this challenge into an opportunity to adjust the original evaluation framework. which focuses on case studies, into an experimental design. Thus, the following design problem can be defined:

How to adapt the evaluation framework proposed by Vos et al. (2012) **that proposes** guidelines to measure the effectiveness and efficiency of software testing paradigms **so that** a reliable comparison of software testing techniques can be done **in a context** where fault injection and conducting a case study are not possible?

1.2 Research objectives

Given the above circumstances, the research objective of this research can be formulated as follows: The research aims to adapt the original evaluation framework by Vos et al. (2012) into an experimental design within the context of the limited scenario explained in the problem statement.
Relying on the research objective, the following main research questions and sub-research questions were formulated:

1. How to compare software testing techniques experimentally in terms of effectiveness and efficiency?

The answer to this research question should provide an adapted evaluation framework that follows experimental guidelines and supports the evaluation of the effectiveness and efficiency of software testing techniques and tools.

- **RQ1.1**: *What are the state-of-the-art comparison approaches for software testing techniques* *in the literature?*

By answering this research question, this study aims to give an overview of the existing comparison approaches for software testing techniques and tools in the literature. Additionally, it will investigate the problem stated in the problem statement.

- **RQ1.2**: *How to compare multiple software testing techniques, without the possibility of fault injection and conducting a case study, in terms of effectiveness and efficiency?*

  This research question contributes to the adaptation of the chosen evaluation framework for the comparative study. The aim is to adjust the evaluation framework to fit the experimental guidelines.

2. How do script-based and script-less testing paradigms compare in practical settings?

This research question aims to validate the redesigned evaluation framework by conducting a comparative study of script-based and scriptless testing paradigms.

- **RQ2.1**: *What are the different automated GUI testing approaches in the literature?*

  This research question will be answered by the literature study in Chapter 4. The aim is to create an understanding of the different terminologies that exist within this field and also to study what different testing approaches exist in the literature. This research question focuses on the different testing techniques that exist at the GUI level of system testing, more specifically on automated GUI testing.

- **RQ2.2**: *What are the challenges of automated GUI testing in the literature?*

  This research question should highlight the existing challenges of automated GUI testing known and recognized in the literature. The goal is to create an understanding of the challenges that exist within this field. Additionally, it aims to show the challenges of the different testing paradigms that are mentioned in the study of the first research question.

- **RQ2.3**: *How well does the adjusted method support the comparison of the GUI testing paradigms?*

  To validate the adjusted evaluation framework, a quasi-experiment was conducted in a practical setting. This will be achieved by applying different test automation tools based on script-based and script-less testing paradigms.

1.3 Contribution

This thesis addresses the challenge of evaluating software testing techniques and tools in scenarios that are not covered by existing evaluation methodologies. By redesigning the original evaluation framework by Vos et al. (2012), this research enables the comparison of software testing approaches through experiments.
Furthermore, the adaptation of the evaluation framework provides a detailed and structured experimentation and execution process, that follows reporting guidelines for controlled experiments in software engineering. This ensures consistent and standardized reporting of experimental results, which improves the reproducibility of empirical research in the field of software testing.

1.4 Thesis organization

The following chapter consists of a background of the topic. It will give the reader an overview of the idea of software testing and the role of it in SDLC. Moreover, it will give the reader explanations of specific terms within the field of software testing, to create a better understanding. The chapter will conclude with a section about related work that consists of other existing studies that were conducted within this field.
Chapter 3 will present the research method that was used to do this research. First, it will shortly describe the method that was used to conduct the literature study and then it will continue with the introduction of the design science method used to redesign the evaluation framework.
Furthermore, Chapter 4 consists of a literature study that should create a basis for the following treatment redesign and validation. This chapter aims to answer research questions 1.1 *(What are the state-of-the-art comparison approaches for software testing techniques in the literature?)*, 2.1 *(What are the different automated GUI testing approaches in the literature?)* and 2.2 *(What are the challenges of automated GUI testing in the literature?)*.
Chapter 5 presents the adaptation of the evaluation framework and what guidelines were followed in the process. The method is illustrated in a Process-Deliverable-Diagram together with activity and concept tables. This chapter aims to answer the research question 1.2 *(How to compare multiple software testing techniques, without the possibility of fault injection and conducting a case study, in terms of effectiveness and efficiency?)*.
The treatment validation can be found in Chapter 6. Here, the reader is presented with a quasi-experiment report that discusses the experiment process and its results. This chapter focuses on research question 2.3 *(How well does the adjusted method support the comparison of the GUI testing paradigms?)*.
Chapter 7 discuss the limitations of the study and give a summary of this research.

Chapter 2

Background and Overview

Over the years, the definition of Software Testing has evolved and taken different forms. Hetzel (1988) provided the following definition of compliance of the system with its requirements:

"Testing is the process of establishing confidence that a program or a system is doing what is supposed to do".

Another definition by Myers (1979) focuses on the intention of finding errors and was defined as follows:

"Testing is the process of executing a program with the intent of finding errors."

Both of these definitions capture the basis of software testing as an activity that assesses the quality of software and examines whether it meets the requirements and is ready for use by end-users with the goal of identifying errors. Today, testing is a crucial part of the software development process, and there are many different types and levels of testing that can be performed. One area of testing that has gained significant importance in recent years is GUI testing, which focuses on evaluating the graphical aspects of software applications (Rodríguez-Valdés, Vos, Aho, & Marín, 2021). This is due to the fact that GUIs have become an everyday part of our lives through smart devices, evolving over time.
This chapter provides a general understanding of software testing and explains basic terms that are used throughout this thesis. Additionally, it gives an overview of the existing studies in the literature.

2.1 Role of Testing

Testing plays a significant role in the SDLC. The primary objective of Software Testing is to improve and assure the quality of software. **Quality** of software is defined as the degree to which the system meets its specified requirements, as stated by (IEEE, 1990). **Quality Assurance** (QA) is the process that includes activities designed to provide assurance that software or a system conforms to its requirements (IEEE, 1990). This process involves many different tasks with the goal of high customer satisfaction and benefit.

Several techniques for software quality assessment and assurance exist, which can be classified into two broad categories: static and dynamic analysis (Tripathy & Naik, 2011). Static analysis techniques consist of an examination of the content and structure of software, including specification documents, software models, and source code (Bourque et al., 2014). On the other hand, dynamic analysis techniques are based on the execution of the actual source code in order to detect any failures of software (Tripathy & Naik, 2011). Static analysis techniques are particularly useful during the early stages of software development, where they can help identify potential issues before code execution. This can save time and resources and prevent critical errors that can occur during software development. Dynamic analysis techniques, on the other hand, are more useful during the later stages of development when code has been written and needs to be tested based on use cases from the perspective of the end-users. These techniques can help to identify problems that may have been missed during static analysis.

2.2 Testing Levels

Testing activities have different levels of abstraction. The differentiation of testing levels is very important because each level has different testing objectives that require a different kind of knowledge and skills (Graham, Black, & van Veenendaal, 2021). A testing process already starts very early in a software development life cycle and its role is as important as the development process. The following section describes each level of abstraction starting with the lowest level and continuing to the highest level of abstraction.

2.2.1 Unit Testing

The first level of abstraction is Unit Testing. This level consists of testing software units or components that have been implemented in the programming phase. Unit testing only focuses on single software units and is solely based on component specifications and design (Graham et al., 2021). This isolation of software units is very important to prevent any external influences on units. The main testing objects are program units, such as functions, classes, and scripts. The detected failures during unit testing are based on the software units themselves.

2.2.2 Integration Testing

The next level of testing is Integration Testing. While unit testing focuses on the isolation of each unit during testing, testers combine several units into a subsystem and test the integration of these combined units, which is called integration testing. The goal is to check if the interface that combines different units works correctly and detects failures in the interaction of these units early in the development phase (Graham et al., 2021). A test object of integration testing can enclose interfaces that have database access and infrastructure components. In this case, integration testing examines if units can access the database correctly.

2.2.3 System Testing

The third level after integration testing is System Testing. The goal of system testing is to check if the system meets its specifications. The earlier levels were focusing on testing the system against technical specifications, while system tests examine the product from a user perspective based on functional and non-functional requirements (Graham et al., 2021). The test object for system testing is a complete system, that also includes a GUI - if it has one. A system is considered as a whole and is tested against functional and non-functional requirements. Thus, GUI testing can be considered as another abstraction level of system testing. An in-depth explanation of GUI testing can be found in Section 2.7.

2.2.4 Acceptance Testing

The last level of abstraction in testing is Acceptance Testing. In a software development project, the produced system can be developed for a customer or for the developer's own use. If there is a customer or a third person involved, the acceptance tests are mostly done by the customer or third person themselves. The testing activities of earlier testing levels are all the producer's responsibility, while the acceptance tests are the customer's responsibility (Graham et al., 2021). The test object is the system as a whole that is being tested from the customer's or user's perspective. The system is being tested against all specifications that describe the customer or user viewpoint.

2.3 White-box Testing

A white box, on the other hand, is a system whose internal contents and implementations are known and relevant to the test design (IEEE, 1990). White-box testing is based on the source code of the SUT, thus it is also often called structure-based or code-based testing (Graham et al., 2021). It is also possible to manipulate the source code for the white-box testing. Thus, white box testing is mostly done by the developer to test code units.

2.4 Black-box Testing

A black box is a system whose general functions, like inputs and outputs are known, but whose implementations are unknown (IEEE, 1990). Black box testing is solely based on the specifications of a system that describes the inputs and expected outputs, on which the test cases are based. Thus, black box testing is also called specification-based (Graham et al., 2021). The inner structure and design of the SUT are unknown and irrelevant to the test design. Black box testing focuses on the functionality of a SUT, which makes it a functional test.

2.5 Test Case Generation

To understand the testing process better, some terms related to testing activities need to be defined and explained. An important part of testing is test case generation. A test case is the documentation of test inputs, execution conditions, and expected results developed for a specific outcome to verify compliance with a specific requirement (IEEE, 1990). Test cases can be generated manually, which can be very time-consuming because manual creation of test cases needs maintenance and evaluation of compliance with specifications (A. Memon, Pollack, & Soffa, 2001). To overcome these issues, automated test generation techniques and tools were developed that are more efficient and reliable. Testing a complete GUI can be done by creating several test cases. A set of test cases is called a test suite. A test case execution results in a test outcome, which is called the test result, that differentiates between a failed and passed test result. To distinguish between these two outcomes there is a method or a mechanism called a test oracle, that defines the expected behavior of a SUT. In manual testing, a test oracle is a human, who decides on the result of a test execution, while on the other hand, in automated testing a test oracle is the mechanism that distinguishes between the outcomes (Baresi & Young, 2001). Test automation tools generate test suites based on different techniques. These techniques are part of script-based testing techniques. A detailed study of these techniques on the GUI level can be found in Chapter 4.

2.6 Regression Testing

Regression testing is defined as the *"selective retesting of a SUT to verify that modifications have not caused unintended effects and that the SUT still complies with its specified requirements"* (Bourque et al., 2014). Regression testing can be performed at all test levels and has to be documented well to make it reusable (Graham et al., 2021). There are many reasons to run regression tests, which are:

- defect retests, which are done after fixing a previously found bug,
- testing altered functionality, which is done after a specific functionality was changed or corrected,
- testing new functionality, which consists of testing newly integrated system parts,
- complete regression test, which is about testing the whole system.

Regression testing can be done both manually and automatically. Manual regression testing can be very time-consuming and tiresome, so automating repetitive test cases can save lots of time. That is why test automation is important during regression testing.

2.7 Automated GUI Testing

GUI testing, also known as GUI-based testing, is part of the system testing of software. The tests are executed to examine the front-end of software the user interacts with (Banerjee, Nguyen, Garousi, & Memon, 2013). Through GUI events like mouse clicks, selections, and typing, inputs are taken from users, which change the state of GUI widgets like buttons, drop-down menus, and text fields, and provide output in a graphical form (Banerjee et al., 2013). Every event that can be made on GUIs, creates an outcome, which depends on its internal state and external environment. Thus, each event executed on GUIs generates a different outcome in different states. Additionally, different sequences of an event can again create different outcomes. Relying on these characteristics, each event executed on GUIs needs to be tested on different states and with different sequences, which makes GUI testing a complex and challenging task (Banerjee et al., 2013).

To ensure effective GUI testing, testers should have an in-depth understanding of the software's requirements and functionality. Furthermore, there has to be a clear understanding of the different states that the software can have and the possible sequences of GUI events. This knowledge can provide a complete test coverage of a system. Overall, GUI testing is a crucial aspect of software testing on the system level that ensures that the software behaves correctly according to its requirements. It is a complex and challenging task that requires a comprehensive understanding of the software's requirements, functionalities, different states, possible outcomes, and sequences of GUI events.

2.8 Related Work

Automated GUI testing is a highly anticipated and continuously growing field that is evolving within software testing (Rodríguez-Valdés et al., 2021). There have been several case studies conducted that compare the script-less testing tool Testar against manual testing practices of industrial companies (Bauersfeld, Vos, Condori-Fernández, Bagnato, & Brosse, 2014; Martinez, Esparcia, Rueda, Vos, & Ortega, 2016; Chahim, Duran, Vos, Aho, & Condori-Fernández, 2020). In two of these studies, the researchers focused on the test effectiveness, efficiency, and learnability of the script-less tool. The results showed that the script-less testing approach outperformed the manual testing approach regarding test effectiveness and functional coverage and detected some critical failures.

Furthermore, a similar study to our research was conducted by Bons, Marín, Aho, and Vos (2023), where the researchers compared Testar with Selenium, a scripted testing tool. The metrics that were measured are test effectiveness, efficiency, and subjective satisfaction of the tools. The aim of the study was to measure the complementarity of different testing tools to perform automated system testing at the GUI level. The experiment was performed in a Dutch company, where the testing team was doing the regression tests manually. The study was divided into three different phases:

1. The first phase is the set-up and learning phase, where the subjects had to download, and install both of the tools and then learn how to use both of them.

2. The second phase is the testing phase, which consists of running tests and analyzing the results obtained by both tools.

3. The last phase is the subjective evaluation phase, where the researchers measured the subjective satisfaction of the subjects. The result showed that Selenium is better at detecting process failures, whereas Testar is better at detecting visible failures and reached a higher event coverage.

Regarding test efficiency, both tools showed similar results and overall, both of the tools were perceived as useful for the company and complementary. The authors concluded that scripted and script-less approaches can improve manual testing.

Another study compares a testing tool that is based on a script-less approach against three other open-source scripted tools (Di Martino, Fasolino, Tramontana, & Starace, 2020). The authors measured the effectiveness of the code coverage achieved by the different tools used during the experiments. The study was divided into two experiments done with computer science students. During the first experiment, the subjects (students) were not provided with any information about the SUT and were instructed to perform exploratory testing using all the tools. The results showed that the script-less tool showed similar code coverage to one of the scripted tools. The second experiment showed an increase in code coverage, which was obtained by providing more information about the SUT. The results of the second experiment showed that combining manual exploratory testing and automated testing can result in higher testing effectiveness.

A study that compared random, model-based, and systematic automated GUI testing techniques was done by Pezzè, Rondena, and Zuddas (2018). The authors selected multiple tools for each testing technique, one for the random technique, six for the model-based technique, and three for the systematic testing technique. The objective of the study relies on comparing automatically generating and executing GUI test cases for desktop applications that do not require any additional information but the SUT itself. The selection of the tools was mainly based on the public accessibility of each of them. The tools then were evaluated regarding their effectiveness in terms of the ability to reveal faults, sample the SUT execution space, and efficiency. The results of the study revealed that random test case generators are better at fault detection than testing techniques that are based on GUI models or machine learning algorithms. Moreover, the authors indicated that the performance of the current techniques could be enhanced in regard to their definition of automated oracles, which could detect more complex failures than crashes and uncaught exceptions. The study concludes by suggesting investigating the oracle problem for GUI testing more in the future.

Table 2.1 depicts some example metrics used in the aforementioned comparative studies.

Variables	Metrics
Effectiveness	Number of failures observed (+ injected faults) Number of failures discovered Code/Functional/Event coverage Number of test cases generated Severity and type of the detected failures
Efficiency	Time needed to design test suites Time needed to set up and configure tools Time needed to execute tests Reproducibility of the failures detected Time needed to analyze failures
Subjective satisfaction	Satisfaction Perceived usefulness
Learnability	Reaction to the learning process Learning (knowledge growth) Maintainability

Table 2.1: Example of metrics used in related work

Chapter 3

Research Method

This thesis is divided into two different phases. The first phase consists of a literature study, that aims to study the existing different approaches of script-less and scripted testing and additionally, the existing challenges of GUI testing in the literature. The second phase includes the redesign of a treatment and its validation with the help of a quasi-experiment, that will be conducted in a practical setting. Table 3.1 depicts the research question together with the research method that will be used to obtain the outcome, that it is intended to achieve. The design of both phases is elaborated on further in the following sections of this chapter.

Research Question	**Research Method**	**Outcome**
RQ1.1	Literature Study	Existing evaluation framework approaches
RQ1.2	Design Science	Treatment design
RQ2.1	Literature Study	Existing automated GUI testing techniques
RQ2.2	Literature Study	Existing challenges of automated GUI testing
RQ2.3	Quasi-Experiment	Treatment validation

Table 3.1: Research Questions and Methods

3.1 Literature Study

A literature study will be conducted to investigate the problem and the existing approaches in more depth. The reviewed literature will include existing testing approaches in script-based and script-less testing fields and state-of-the-art automated GUI testing tools. To understand the problem better, the literature review will consist of the challenges recognized in the literature. Furthermore, the study will include state-of-the-art evaluation approaches for software testing techniques. Relying on this literature study, we aim to address research questions 1.1 *(What are the state-of-the-art comparison approaches for software testing techniques in the literature?)*, 2.1 *(What are the different automated GUI testing approaches in the literature?)* and 2.2 *(What are the challenges of automated GUI testing in the literature?)*. The snowballing method will be used for the literature review. The method makes use of the reference list of papers or/and the citations to the paper to identify more papers (Wohlin, 2014). For this literature study, the two-way snowballing method was used, which makes use of both the reference list and the citations of the selected papers. This was essential to reach all the connected papers on the relevant topic. Google Scholar was used to find the citations of the papers. The list of papers from which additional papers will be identified is listed below.

Title	Author	Content
Scripted and scriptless GUI testing for web applications: An industrial case	Bons et al. (2023)	Related comparative study
Evolution of Automated Regression Testing of Software Systems Through the Graphical User Interface	Aho et al. (2016)	History of GUI Testing
Practical Model-based Testing: A tools approach	Utting and Legeard (2010)	Definition of Model-based Testing
Why many challenges with GUI test automation (will) remain	Nass, Alégroth, and Feldt (2021)	Recent mapping study about challenges of GUI Testing
A Methodological Framework for Evaluating Software Testing Techniques and Tools	Vos et al. (2012)	The evaluation framework to be adapted

Table 3.2: Selected papers for the snowballing method

Inclusion/exclusion criteria

Regarding the inclusion and exclusion criteria, it was important that the papers were discussing automated GUI testing techniques and evaluation techniques for software testing. Papers regarding automated GUI testing, that were published before 2000, were excluded since the testing techniques themselves evolved over time and it was important for this literature study to have the latest definitions and explanations of the testing techniques. Regarding the literature on evaluation techniques, we had to include papers that were published before 2000, since the first experiments in this field were conducted in the late 80s, which contributed to the evaluation of software testing techniques a lot.

Further Search

To find standardized definitions of important terminologies, we made use of glossaries and books that cover important aspects within the software testing field. The glossaries, that were used for standardized definitions are the "IEEE Standard Glossary of Software Engineering Terminology" (IEEE, 1990) and "SWEBOK V3.0" (Bourque et al., 2014). Additionally, the book "Foundations of Software Testing, ISTQB Certification, 4th edition" was used to get specific additional information about the terminologies used in the literature review to provide the reader with a more in-depth understanding.

3.2 Design Science

The adaptation of the evaluation framework will be done by following the Design Science Methodology by Wieringa (2014). The Design Science Methodology supports the design and investigation of a problem within a given context. This methodology proposes three different tasks, namely, problem investigation, treatment design, and treatment validation. This set of tasks is called the design cycle, which is part of a bigger cycle, in which the validated treatment from the design cycle is implemented and evaluated in the real world. This larger cycle is called the engineering cycle, which is depicted in Figure 3.1.

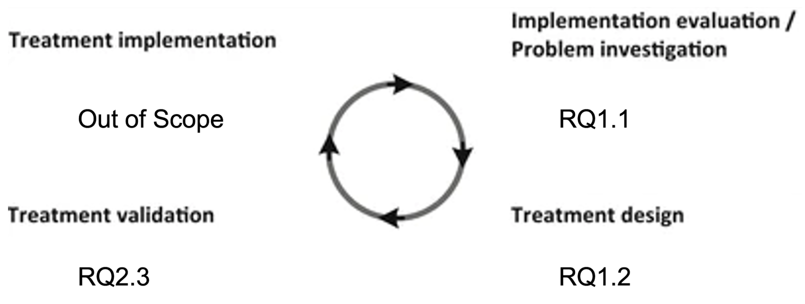


Figure 3.1: The Engineering Cycle by Wieringa (2014)

Problem Investigation

This task aims to investigate the problem in a given context. The problem this research is investigating is about how the existing evaluation framework by Vos et al. (2012) can be adapted, so a reliable comparison of software testing techniques and tools is possible, in a context where fault injection and conducting a case study are not possible. This task aims to answer research question 1.1.

Treatment Design

Treatment design has the objective of adapting the evaluation framework by Vos et al. (2012) following the results gained in the first task. This task aims to answer research question 1.2.

Treatment Validation

This task aims to answer the question if the designed treatment would solve the problem. This research will validate this question by conducting a quasi-experiment, which is being discussed further in the following section. This task aims to answer research question 2.3.

3.3 Quasi-Experiment

A quasi-experiment is an investigation to establish a quantitative relationship between several variables or alternatives under examination (Stol & Fitzgerald, 2018). This experiment is a quasi-experiment since the investigation consists of independent and dependent variables and the subjects are not randomly assigned to the treatments (Easterbrook, Singer, Storey, & Damian, 2008). The subjects will be able to select the treatment they want to apply during the experiment. The selection of the subjects is based on non-random sampling as their participation depends on their availability during the time of the execution of the quasi-experiment. The quasi-experiment will be conducted to validate the treatment designed for the sake of this research by evaluating different automated GUI testing approaches in a practical setting. To do so, two automated testing tools will be selected, that will represent each of the testing approaches, namely model-based for the script-based approach and monkey testing for the script-less approach.

Goals

The objective of this comparative experiment is to compare two different automated GUI testing paradigms in a practical setting and to validate the adjusted evaluation framework. The two selected testing approaches will be evaluated and compared by their test effectiveness, and efficiency. The research questions are formulated and elaborated on in Section 1.2.

Chapter 4

Literature Review

As mentioned in Section 2.1, testing covers many roles in SDLC. Software Testing is a broad field with different methods, abstraction levels, techniques, and terminologies. It is important to highlight that the literature review starts with the history of automated GUI testing, which is also called the generation of automated GUI testing. It gives an overview of a categorization, that is used within this field. This thesis focuses on script-based and script-less GUI testing. Thus, the literature study encompasses mainly automated software testing, disclosing manual methods. Software testing is a process that has different levels of abstraction and other abstraction levels within these levels. This chapter aims to give an overview of all the different testing approaches of the GUI level of system testing. Furthermore, it discusses the challenges of GUI-based system testing in the literature. The goal of this literature review is to answer research questions 1 *(What are the different automated GUI testing paradigms in the literature?)* and 2 *(What are the challenges of automated GUI testing in the literature?)*.

4.1 Generations of Automated GUI Testing

The history of automated GUI testing goes back to the late 80s (Rodríguez-Valdés et al., 2021). A categorization of the different automated GUI testing techniques has been proposed by Alégroth and Feldt (2014), classifying existing GUI testing approaches in three chronological generations. Furthermore, Aho et al. (2016) made a more in-depth categorization of the existing GUI testing techniques which is depicted in Figure 4.1. The vertical axis represents the tolerance for change in the GUI/SUT, which is divided into four levels and is also called the generation of GUI-based testing. The fourth highest level "combining visual and widget-based approaches" was added by the authors, but as it does not bring any novel approach to the field, it is called the 3.5th generation.

The horizontal axis shows the level of automation in regression testing through the GUI. These testing approaches are elaborated on and studied in-depth in the literature study in the following sections.

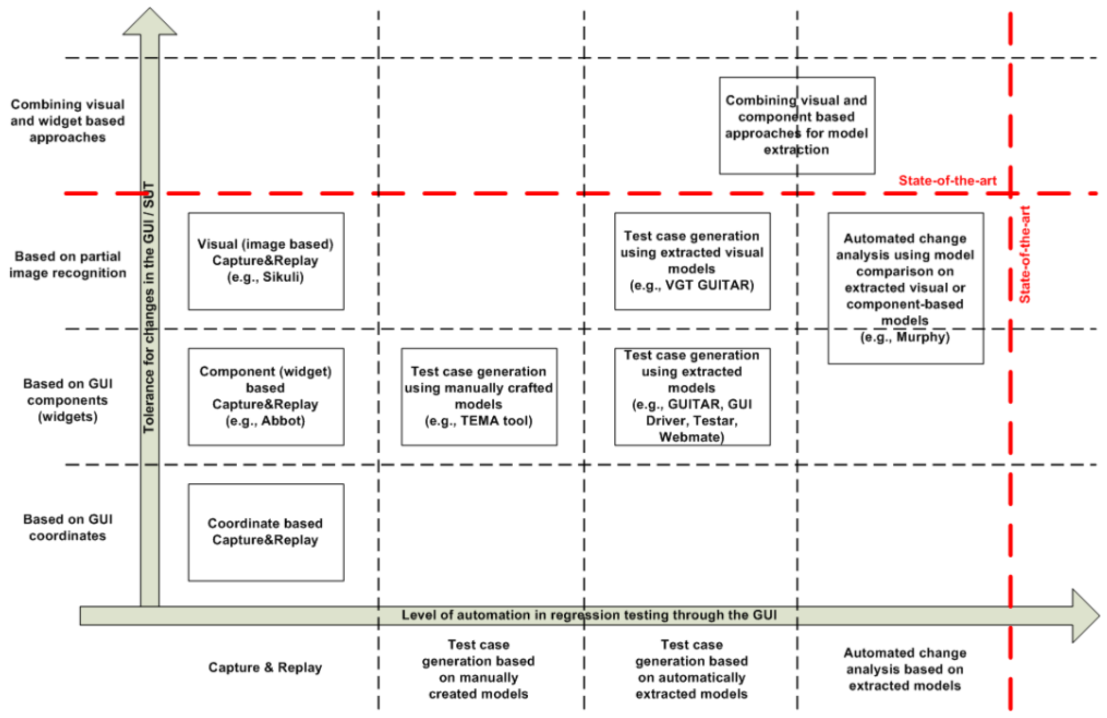

Combining visual and widget based approaches
Based on partial image recognition
Based on GUI components (widgets)
Based on GUI coordinates
Tolerance for changes in the GUI / SUT
Combining visual and component based approaches for model extraction
State-of-the-art
Visual (image based) Capture&Replay (e.g., Sikuli)
Test case generation using extracted visual models (e.g., VGT GUITAR)
Automated change analysis using model comparison on extracted visual or component-based models (e.g., Murphy)
State-of-the-art
Component (widget) based Capture&Replay (e.g., Abbot)
Test case generation using manually crafted models (e.g., TEMA tool)
Test case generation using extracted models (e.g., GUITAR, GUI Driver, Testar, Webmate)
Coordinate based Capture&Replay
Level of automation in regression testing through the GUI
Capture & Replay
Test case generation based on manually created models
Test case generation based on automatically extracted models
Automated change analysis based on extracted models

Figure 4.1: Categorization of automated GUI testing by Aho et al. (2016)

4.1.1 First generation

In their first paper Alégroth and Feldt (2014) refer to the first level of GUI-based testing as the first generation which they then later also call "the tolerance of changes in the GUI". The first generation or the first tolerance level of changes in the GUI is called as coordinate-based GUI-based testing, as it uses exact coordinates on the screen to interact with the SUT's GUI. The earliest versions of Capture & Replay (C&R) tools recorded user actions using exact mouse coordinates to generate and execute test scripts. This approach is not supported anymore by the current tools that are on the market and is categorized in the lowest level of tolerance for changes, just because even changing the screen resolution would break the generated test scripts, which requires a lot of maintenance.

4.1.2 Second generation

An upper level of tolerance for changes in GUI, or the second generation of automated GUI testing, is based on components or widgets of the GUI. A widget of a GUI can be defined as functionality for user input, e.g. button, text field, or drop-down menus. In contrast, a component or a property of a GUI is e.g. a background color, size or font of a widget (A. Memon, Banerjee, & Nagarajan, 2003). This level of abstraction consists of API-based approaches, which is more beneficial than the coordinate-based approach, as all the recorded user interactions are mapped into components or widgets of the GUI. This way the test scripts are more robust against GUI changes, which also lowers the maintenance costs of the test scripts.

4.1.3 Third generation

The third generation and the highest level of tolerance for change in the GUI is Visual GUI Testing (VGT). This approach uses image recognition on the screen captures in order to identify and interact with the GUI. This way of GUI interaction, also called bitmap interaction, allows for mimicking user behavior, where automated mouse and keyboard commands are given to the SUT, the output is observed and then compared to expected results (Borjesson & Feldt, 2012). This makes VGT more robust to layout changes, but also more dependent on the graphical representation of the GUI, e.g. change of image size, shape or color, than the second generation.

The following sections concentrate on the horizontal axis of Figure 4.1, which represents the level of automation of the GUI testing techniques by differentiating between script-based and script-less GUI testing.

4.2 Script-based GUI testing

A classification of automated GUI testing techniques has been presented in Section 4.1. The proposed classification by (Aho et al., 2016) is illustrated in Figure 4.1. The vertical axis of this graph, which represents the level of tolerance for GUI changes, also called generations of automated GUI testing, was discussed. This section of the literature review elaborates on the horizontal axis of the graph, which depicts the level of automation in regression testing through the GUI. As mentioned in the problem statement, this thesis is differentiating between script-based and script-less GUI testing. Starting with the script-based approaches in this section, which are also mentioned by (Aho et al., 2016), additionally, other GUI and script-based testing are discussed. A script-based testing technique is an automated testing technique that generates test scripts while executing test cases. It is important to note that a code-less test automation tool is not equal to a script-less tool. A tool that is code-less can also be script-based, as it might generate and run test scripts in the background. The following sections elaborate on the existing script-based GUI testing techniques in the literature.

4.2.1 Capture & Replay

The first level of automation in Figure 4.1 and one of the earliest and widely used approaches in automated regression testing is Capture & Replay (C&R), also called Record & Play. In C&R approaches, the tool acts as the name says as a recorder, that logs all the inputs, which is done by a manual tester. These inputs are various GUI events like a mouse click or a textual input, that are saved as a test script. The test script can be executed automatically by simply "playing it back" (Graham et al., 2021). These tools have two different modes: capture and replay mode. In capture mode, the tool captures and saves any kind of input given by the tester, which can be an event as well as an object's attributes like the color, position, name, etc., to identify the selected object. All these captured attributes are saved in a test script, which can be replayed several times in replay mode. In

C&R approaches, the expected result of a use case can be incorporated as checkpoints, that compare the expected result with the actual result. This way, it determines if the SUT is behaving correctly (Graham et al., 2021). This can happen during the capture mode or by editing the test script. A test fails when the actual result differs from the expected result at a checkpoint in the test script. More advanced and modern C&R tools can also record the behavior of the GUI, thus are able to notice changes in GUI in later versions (Aho et al., 2016).
The C&R approaches are generally easy to use, decrease the manual effort for regression testing, and deliver faster results. However, the problem with these approaches is that they are not tolerant of GUI changes. The test script, captured from the earlier versions, has to be maintained and it has to be re-captured for the new GUI. The maintenance of these scripts requires additional manual effort. There exist several open source and commercial C&R tools, like Ranorex (Ranorex, 2023), QF-Test (QFS, 2023), Squish (Squish, 2023), which not only support C&R testing but also code-based testing, that is further explained in the following section.

4.2.2 Code-based Testing

Another scripted testing approach is the one where test scripts have to be written manually. Two of the most popular test automation tools, that are based on manually written scripts are Selenium (Selenium, 2023a) for web applications, and Appium (Appium, 2023) for mobile applications. Selenium supports the automation of web browsers through a WebDriver, which is an API that defines an interface for controlling the behavior of web browsers. Implementing a WebDriver into a script enables the communication between Selenium and the browser. The open-source tool supports five programming languages: Ruby, Java, Python, JavaScript, and C#. Selenium also offers a C&R tool, which is the Selenium IDE (SeleniumIDE, 2023). Another scripted testing framework is the AutoIt (AutoIt, 2023), which has its own scripting language and supports only Windows applications.

An example of a code snippet of Selenium written in Java is given below. Line 5 creates a new session by defining the Driver with the WebDriver class object. On line 6 the navigation to the SUT is being created, which is the Selenium webpage in this case. Line 8 extracts the title of the webpage, where this title is compared to the expected result on line 9. Moving forward to line 10, this code line makes sure that the WebDriver will wait the browser to load the certain element, that is tried to be extracted. By doing so, WebDriver requests the DOM for a certain duration when trying to find any element (Selenium, 2023b). Lines 12 and 13 are commands that find an element and save them in the WebElement object. The command on line 15 sends an input "Selenium" in the text box and the input is submitted by the command on line 16. The expected output is compared with the element that has been found after submitting the input on line 20. The connection to the driver is closed by the command on line 22.

Listing 4.1: Selenium example extracted from Selenium (2023b)

```java
public class FirstScriptTest {

    @Test
    public void eightComponents() {
        WebDriver driver = new ChromeDriver();
        driver.get("https://www.selenium.dev/selenium/web/web-form.html");

        String title = driver.getTitle();
        assertEquals("Web form", title);
        driver.manage().timeouts().implicitlyWait(Duration.ofMillis(500));

        WebElement textBox = driver.findElement(By.name("my-text"));
        WebElement submitButton = driver.findElement(By.cssSelector("button"));

        textBox.sendKeys("Selenium");
        submitButton.click();

        WebElement message = driver.findElement(By.id("message"));
        String value = message.getText();
        assertEquals("Received!", value);

        driver.quit();
    }
}
```

4.2.3 Model-based Testing

Another approach, that is widely used is model-based testing (MBT). This approach differs from the previously mentioned approaches, as it creates the test scripts automatically. That means that not every codeless automation tool is a script-less tool at the same time, as it might generate test scripts in the background as MBT tools do. To understand MBT better, it is important to highlight all the different meanings of MBT. Model-based testing is an approach that has different meanings for different test generation techniques. It includes four different approaches, which are defined by Utting and Legeard (2010) as follows:

1. Generation of test input data from a domain model
2. Generation of test cases from an environment model
3. Generation of test cases with oracles from a behavior model
4. Generation of test scripts from abstract tests

The first approach is about automatic test input generation and involves the selection and combination of a subset of input values to produce test input data. Model-based testing for test input generation creates a model with information about the input values' domains. However, this approach does not create any information for test oracles, so it does not provide a test result.

The second approach of MBT creates a model, that describes the expected environment of the SUT. These environment models enable the generation of sequences of calls, that are made to the SUT. Like the previous approach, it is not possible to determine if the test fails or passes, as the environment model does not provide any information about the behavior of the SUT, so the output values can not be predicted.

The third approach of MBT generates test cases with oracles, that specify the expected output values of the SUT. The model that is created is called the behavior model, as the generated test cases include oracles, that define the behavior of the SUT, which is the relation of the input and output values. Unlike the other approaches, in addition to the input generation, it provides the tester with whole test cases and a test result.

The last approach focuses on the abstraction of a description of a test case, such as a UML or a sequence diagram, into a low-level test script that is executable. The model that is being created includes information about the API and the structure of the SUT.

An example of a test automation tool that is based on the model-based approach is Tosca by Tricentis (Tosca, 2023), for which a detailed explanation can be found in Section 6.1.1. As the main focus of this thesis is on test case generation and the whole test design of automated testing, the approach that is relevant for this research is the third approach of MBT. Relying on the categorization proposed by Aho et al. (2016) there are two different approaches to test case generation that are based on models. These are test case generation-based on manually created and automatically inferred models, which are explained further below.

Test case generation based on manually created models

As for the categorization proposed by Aho et al. (2016) MBT is also divided into two different categories. The first category is MBT which is used for test case generation and is based on manually created models. The focus here is on test automation tools where the test designer has to create a model of the GUI and its expected behavior. The tool then automatically generates the test cases based on the manually created behavior model. There are several techniques that require the manual creation of models and automatically generate test cases from them, like a keyword-driven model, faulty event sequence graph, AI planning, genetic models, probabilistic event-flow graph, latin squares, coverage arrays, hierarchical finite state machines, and UML diagram-based technique, for which an explanation can be found in A. M. Memon and Nguyen (2010).

Test case generation based on automatically inferred models

More modern approaches were introduced in MBT where GUI models are automatically extracted, also called model extraction, model inference, or GUI ripping (Aho et al., 2016). Earlier techniques for model extraction mostly used static analysis on the source code of the system, which couldn't capture the dynamic behavior of the GUI. Thus, dynamic analysis was introduced and the behavior of the GUI was analyzed during user interaction with the SUT, similar to C&R tools. Here, the interaction is able to simulate the user interaction, which makes it possible to automatically interact with the widgets of the GUI. There are many approaches to automatically extract models, like an event-flow graph, event interaction graph, and the feedback-based model extraction technique (A. M. Memon & Nguyen, 2010).

4.3 Script-less GUI testing

The script-less testing approach refers to a technique that does not require test case generation, unlike script-based approaches. Thus, there are no scripts generated that execute test cases, which makes them script-less. A script-less approach automatically generates sequences of user actions during run-time to explore the SUT by selecting and executing the actions of the discovered GUI states (Pastor Ricós, Slomp, Marín, Aho, & Vos, 2023). A typical tool that is based on a script-less approach works with the following techniques as explained in Vos, Aho, Pastor Ricós, Rodriguez-Valdes, and Mulders (2021). Script-less approaches and tools that are based on a script-less approach, first identify the available GUI widget of the SUT. Then, all the possible actions that can be performed with those GUI widgets are derived. The next step is to select some of these actions to build the test sequences by using an Action Selection Mechanism (ASM). This ASM is being performed randomly which is the random testing approach. The next section discusses this approach more in detail.

4.3.1 Random or Monkey Testing

Random testing, also called monkey testing or stochastic testing, is a script-less software testing approach where test cases are generated purely at random and during the test execution (Bourque et al., 2014). This technique involves exploring the SUT by generating random inputs and performing random actions with the goal of finding system failures. In most cases, test cases that are generated during testing are not being saved, since this approach only focuses on sequences that find failures in the SUT (Vos et al., 2021). The advantage of this approach is that there is no creation and maintenance of test cases. Unlike script-based approaches, monkey testing can discover bugs that cannot be discovered by scripted approaches. Microsoft reported that 10% to 20% of the bugs are found by test monkeys (Moreira, Paiva, Nabuco, & Memon, 2017). We use the term "test monkeys" to refer to test automation tools that are based on the random testing approach. These test monkeys usually explore the SUT in a different way in every test run, which makes them observe the SUT from a different perspective than a human tester. The main goal of test monkeys is to create test sequences that are purely random to make the SUT unresponsive and crash. A tool that is based on the monkey testing technique is Testar by Vos et al. (2021), for which an in-depth explanation can be found in Section 6.1.1. Monkey testing is differentiated into two types of test monkeys, which are smart monkeys and dumb monkeys (Nyman, 2000).

Smart monkeys

Smart monkeys are called "smart" because they have some knowledge about the SUT. They know in what action sequences a simple functionality can be done and how that one functionality should result (Nyman, 2000). If a functionality does not result with the expected result, it reports a failure. The first step of smart monkeys is to make them obtain knowledge about the SUT (Vos et al., 2021). A way to do this is to get the programmatic structure of the layout and widgets of the GUI using a technical API or image recognition to detect visible widgets of the GUI.
The second phase involves detecting and extracting the state of the GUI using the information obtained in the first phase. After the monkey obtains some knowledge about the GUI, it can derive a set of actions that will be executed. The aim of smart monkeys is to generate arbitrary input sequences to crash or hang the GUI (Vos et al., 2021). The knowledge about the SUT can be obtained by a state model, which guides the smart monkey through the GUI and helps with action selection. This is done by the action selection mechanism (ASM) (Vos et al., 2021). Selecting the right actions is a crucial step in script-less testing, as detecting failures depends on the actions that are being executed. Action selection can be done purely randomly or there can be put some intelligence on action selection to tell the test monkey what to do next. There are some strategies to make a smart monkey smarter, for example using techniques like reinforcement learning, ant colony optimization (ACO), and meta-heuristics (Vos et al., 2021). All these techniques have the objective of calculating the most optimal action to take by a test monkey in a given state of a GUI.

Dumb monkeys

Dumb monkeys, on the other hand, possess no knowledge about the SUT and act ignorantly throughout the testing process. They do not have a state model, thus, have no knowledge about what state the SUT is in, or what to expect after an action is executed. They act ignorant toward bugs, which create an unexpected result, that cannot be recognized by the dumb monkey. What they can recognize is for example obvious bugs, like crashes and hangs. The first dumb monkey test tool was created in the late 80s by Apple, to test how robust their application was in their operating environment (Nyman, 2000). Dumb monkeys are usually used to detect operating system bugs, but they can find application errors as well. These applications are cheap and easy to develop since they are completely automatic, which makes them more attractive to testers (Vos et al., 2021).

4.4 Challenges of GUI Testing

In the previous sections, we mentioned that GUI testing is a challenging task, as it is a continuously growing field within software testing. A recent systematic literature review by Aho and Vos (2018) presents the challenges within the GUI testing field, which we used to discuss the main challenges found by them. This section will address the existing challenges found in the literature and give an overview of the disadvantages of the aforementioned testing approaches. The first section talks about the general challenges of test automation that also impact automated GUI testing. The following sections focus on the challenges that are more specific to the aforementioned automated GUI testing techniques.

4.4.1 General challenges of automated testing

A major problem for system-level testing through the GUI is caused by the short development cycles of iterative and incremental processes and continuous integration practices in software development (Aho & Vos, 2018). Shorter development cycles like those used when following agile methodologies lead to iterative and incremental development, which also requires testing to take place in incremental stages. This shortens the quality assurance activities drastically, which involves the maintenance of test cases and scripts and regression testing. Insufficient time for regression testing can decrease the quality of testing and can cause inefficient test runs (Aho & Vos, 2018). Additionally, there is not enough time to automate test cases, thus, regression tests are being done manually which increases the time and costs used for the testing activities in a development cycle.

Furthermore, the increasing complexity of software systems that are supported by multiple devices and platforms increases testing activities, since every test run has to be done on every platform and device that is supported by the software system (Aho & Vos, 2018). This includes cross-browser testing, which is testing through different browsers that are supported by the SUT, and compatibility testing, which verifies whether the SUT can collaborate with different hardware and software facilities or with different versions or releases (Bourque et al., 2014).

Another challenge is the test oracle problem which is one of the main problems of test automation. A test oracle can be defined as an activity or functionality that determines whether a given activity sequence is an acceptable behavior of the SUT or not. A survey by Barr, Harman, McMinn, Shahbaz, and Yoo (2014) differentiates the test oracle problem into four different categories: specified, derived, implicit, and human test oracles. Specified test oracles are defined by mathematical logic, thus it requires a specification language. A derived test oracle can distinguish a SUT's correct behavior from incorrect behavior by deriving various artefacts like documentation or system executions and different system versions. Lastly, an implicit test oracle relies on implicit knowledge to distinguish between the correct and incorrect behavior of a SUT and does not require any domain knowledge and formal specifications. The human oracle is the tester who decides if the behavior of the SUT is correct or incorrect.

The main challenge Barr et al. (2014) address is that there is less attention and research for automating test oracles to achieve more test automation in software testing. Human intervention is still needed, to check and review the behavior of SUTs. For the aforementioned test oracle approaches the disadvantage is that there is a lack of a formal specification, which relies on the abstraction of models that can be imprecise or contain irrelevant behavior of, for example, the GUI model, that is not required. Another challenge of these approaches is to interpret the abstract outputs, which makes

it hard to read the test results and thus requires human intervention.
One challenge for scripted testing approaches is defining oracles manually. Test oracles that are defined manually, can result in ambiguous test results since defining expected behavior for any SUT state is not feasible (Aho & Vos, 2018).

4.4.2 Challenges of script-based GUI testing

The following section will elaborate on the challenges of script-based testing and each testing approach within that field. Additionally to the test oracle problem, issues that can be faced when using these testing approaches are discussed below.

Capture & Replay

Capture & Replay tools have the drawback that the maintenance of the test scripts is high and so are the costs for the maintenance of scripts (Moreira et al., 2017). As shown in Figure 4.1 and explained in Section 4.1, this approach has the lowest level of tolerance for GUI changes and also offers the lowest level of automation in regression testing. Hence, this approach has high maintenance and requires high manual effort to update test scripts, especially in regression testing. For this reason, C&R tools are mostly used for the first software releases to get faster feedback and are then discarded after new versions of the SUT (Moreira et al., 2017).
Another challenge that a tester might face when testing with C&R tools is when there is a need to add new test cases to a ready test suite (Nass et al., 2021). Merging new test cases into ready test suites might require programming and skills to do so, which again requires manual effort and additional programming skills. Considering the fact that C&R tools are being selected to get faster feedback for the first versions of a SUT, this challenge shows that new versions of a SUT can take more time to run tests and result in higher costs.

Code-based Testing

Code-based testing consists of writing test scripts in a scripting language as explained in Section 4.2.2. As the other script-based approaches, code-based testing also has the challenge of not being robust for GUI changes. For any update on the SUT, the test script needs to be manually extended with new test cases, and the increasing number of test cases will make the maintenance of these scripts even more complex which can make it prone to errors (Aho & Vos, 2018). A significant change in the GUI of a SUT might result in breaking the test execution as whole (Nass et al., 2021).
Additionally, scripted testing requires expertise and knowledge in programming. With this limitation, it is not possible for everyone to generate test scripts, thus experts and specialists with programming skills are needed for testing and maintaining extensive test suites (Nass et al., 2021).
A challenge of tools that are based on the code-based approach is that some tools still do not automatically detect GUI widgets, where the tester has to locate them according to their ID and

class name (Karam, Dascalu, & Hazimé, 2006). A change of these class names or IDs can crash the whole script and then it is required to update those attributes manually, which can take lots of time and be costly.

Model-based Testing

Model-based testing is a very popular testing approach, that is being adopted in many test automation tools, although its application depends a lot on tool support (Aho, Suarez, Memon, & Kanstrén, 2015). MBT approaches, like the other script-based testing techniques, are also not robust for GUI changes as test scripts have to be maintained regularly for regression testing. Test automation tools, that adopt the manually crafted GUI models technique, which is explained in Section 4.2.3, still require manual effort and specialized expertise to create the models. Usually, the tester has to take intense training on these tools to obtain the required knowledge. The needed high manual effort to learn how to use the tool and to create the GUI models to generate test cases automatically can be very costly and time-consuming (Grilo, Paiva, & Faria, 2010; Utting & Legeard, 2010).
Automated model extraction methods, on the other hand, face the challenge of requiring human intervention. For example, certain inputs, such as usernames and passwords, need to be manually predefined by a tester (Grilo et al., 2010). To reach the desired coverage of GUI models, the automatically generated model usually needs to be manually reviewed and corrected by a person, which shows that even though the manual effort is reduced, there is still a need for human intervention.
Another challenge is that irrelevant behavior of models are also being extracted, where a tester is only interested in the expected and required behavior of the SUT (Aho, Kanstrén, Räty, & Röning, 2014). Validating the correctness of such models is another challenge that needs to be mitigated in the field of MBT approaches.

4.4.3 Challenges of script-less GUI testing

Random or monkey testing is a script-less testing approach, that works on random action selection as explained in Section 4.3. The following section discusses the disadvantages of this approach found in the literature in addition to the test oracle problem explained in Section 4.4.1.

Monkey Testing

A major challenge for the monkey testing approach is created by its execution time being very long. A test automation tool that is based on this approach can have test runs that can last for many days since it tries to find the vulnerability that makes the SUT crash (Aho & Vos, 2018). A high execution time is necessary for monkey testing to achieve high testing coverage. A solution to overcome this issue is to enhance the resources and make parallel executions, which still do not guarantee a test execution time as fast as the other approaches. Many papers were published that experimented with the monkey testing approach and discuss its inefficiency in system level testing (Girard & Rault, 1973;

Thayer, 1978; Myers, 1979). These researchers are not convinced that monkey testing can detect severe bugs and see this approach as the least optimal testing approach comparing it to other existing approaches.
One major challenge of monkey testing following the long execution time is the reproducibility of the detected bugs (Nyman, 2000). It can take a test monkey several days to crash a SUT, which results in long and randomly created testing sequences by arbitrary events on the GUI. It can be very difficult to recreate these bugs for debugging purposes, which can be challenging for the developer to fix the found bug (Vos et al., 2021).
Monkey testing is differentiated between smart and dumb monkeys. Dumb monkeys don't have any knowledge about the SUT and its state. They do not have any idea what input and output are allowed or not allowed. Therefore, they are unable to recognize a bug when they detect one (Moreira et al., 2017). For smart monkeys, on the other hand, giving and maintaining SUT-specific knowledge increases the maintenance effort, which is a challenge that is tried to be mitigated by script-less testing approaches (Vos et al., 2021). Furthermore, the quality of a smart test monkey depends on the quality and accuracy of the state model and the action selection mechanism, which requires expertise and effort to develop (Moreira et al., 2017).

Overall, both automated GUI testing techniques have challenges that can be found summarized in the table below:

GUI testing technique	**Challenges**
General challenges of automated testing	Short development cycles Increasing complexity and number of supported platforms Test oracle problem
Script-based	High maintenance of test scripts High manual effort to create test scripts Requires technical knowledge Requires manual effort to detect GUI widgets Requires high effort to learn the tool/programming language
Script-less	Long execution time Difficult reproducibility of bugs Smart monkey: requires technical knowledge to create a test oracle

Table 4.1: Summary of the challenges of automated GUI testing

4.5 Evaluation frameworks for software testing techniques

In order to make an informed decision on what software technique [1] to apply, practitioners must have a comprehensive understanding of the software technique itself first. Then, they have to know which software technique performs better regarding effectiveness and efficiency. To compare software techniques, several evaluation frameworks are based on different evaluation methods. First, it is important to know the available methods to conduct scientific research in software engineering. In

[1] A software testing technique describes how the test cases are being generated (Bourque et al., 2014).

the field of software engineering, four research methods were presented by Basili (1993), which are the following:

- **Scientific:** *"The world is observed and a model is built based on the observation, for example, a simulation model."*
- **Engineering:** *"The current solutions are studied and changes are proposed, and then evaluated."*
- **Empirical:** *"A model is proposed and evaluated through empirical studies, for example, case studies or experiments."*
- **Analytical:** *"A formal theory is proposed and then compared with empirical observations."*

The following section gives an overview of existing evaluation techniques and the research method they are based on.

4.5.1 State-of-the-art evaluation techniques

In empirical studies within software engineering, two primary types of research methods are commonly distinguished: case studies and experiments. There exist several different studies that are based on the analytical method, but this research focuses on the empirical research methods, hence this section will focus more on techniques that propose an empirical method.

Empirical methods

An experiment in software engineering is designed as a controlled study. An experiment involves manipulating variables and imposing controls to systematically investigate the studied setting (Wohlin, 2014). It aims to establish cause-and-effect relationships by comparing different conditions or treatments and measuring their impact on specific outcomes. Unlike the case study, an experiment provides a more controlled environment. In contrast to an experiment, a case study in software engineering is a research method that examines a specific real-world software engineering situation using various sources of evidence (Runeson, Höst, Rainer, & Regnell, 2012). It focuses on understanding a particular software engineering phenomenon within its practical context. Guidelines for conducting case studies in software engineering are discussed further in several studies, such as Kitchenham, Pickard, and Pfleeger (1995), Verner, Sampson, Tosic, Bakar, and Kitchenham (2009) and Runeson, Höst, et al. (2012).

1. **Repeatable Software Engineering Experiments for Comparing Defect-Detection Techniques** by Lott and Rombach (1997) and a replication of Basili, Selby, and Hutchens (1986) proposes a characterization schema for software testing experiments, that was adapted from the scheme in Basili et al. (1986) for software testing evaluations. The schema is divided into four parts: scope definition, experimental planning, operation, and interpretation. It follows the

guidelines of experimentation in software engineering, however, it does not provide any metrics or guidelines on how to measure the software testing techniques.

2. **Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact** by Do, Elbaum, and Rothermel (2005), which builds upon their earlier work Do, Elbaum, and Rothermel (2004), propose an infrastructure called the Software Artifact Infrastructure Repository (SIR) to support controlled experiments. SIR provides a repository of benchmark programs that can be used to evaluate testing techniques, however, no clear guidelines are given on the measurements.

3. **A Framework for Comparing Efficiency, Effectiveness and Applicability of Software Testing Techniques** by Eldh, Hansson, Punnekkat, Pettersson, and Sundmark (2006) focuses on comparing the efficiency, effectiveness, and applicability of testing techniques using fault injection. It involves steps such as preparing code samples with known faults, selecting a testing technique, performing experiments, collecting and analyzing data, and potentially repeating the experiment. Although it proposes clear steps to evaluate software testing techniques, it is restricted to fault injection.

4. **A Methodology for Evaluating Software Engineering Methods and Tools** by Kitchenham (1993) is an organizational framework that targets the industry rather than the researchers. It is organization-dependent, meaning that the measurements are defined depending on the context of the study. The methodology provides guidance on nine different research methods and advises on which method to use for what context (Runeson, Host, Rainer, & Regnell, 2012). The guidance mostly focuses on how to conduct an empirical study in an industrial setting, but no specific metrics on how to measure certain variables are given.

5. **A Methodological Framework for Evaluating Software Testing Techniques and Tools** by Vos et al. (2012) proposes a guideline for conducting a case study to compare software testing techniques and tools. It provides a case study design with clear metrics that allows a better comparison of the results. The authors focus on the effectiveness, efficiency, and subjective satisfaction measurements of the software testing techniques and tools, and define clear metrics to measure them. It offers a case study protocol for seven different scenarios where scenario 1 consists of the most limited context and suggests a qualitative analysis of the software testing techniques or tools. The scenario depends on the answers given to some questions like "Ability to inject faults?" or "Known faults in the SUT?". However, these scenarios are limited in context and are only focused on case studies.

Analytical methods

In addition to the empirical study methods, there exist several methods for evaluating software testing techniques. A recent paper by Kumar and Kaur (2022) presents a hybrid method to evaluate software testing techniques that is based on a Multi-Criteria-Decision-Making (MCDM) approach. This method considers multiple factors such as cost, schedule, and resources. One paper by Neto and Travassos (2014) that uses the MCDM approach is restricted to model-based testing techniques only and aims to support the combined selection of model-based techniques for software engineering projects.

Another approach that is commonly used to evaluate test cases is called mutation analysis or mutation testing. A mutant in this case is a modified version of the SUT, that differs by artificial changes (Bourque et al., 2014). Each test case is tested by both versions and if a test case can identify the difference between the original SUT and the mutant, then the mutant is "killed". The ratio of killed mutants to the total number of generated mutants is used to measure the effectiveness of the generated test cases (Bourque et al., 2014). However, this approach is also only restricted to fault injection. Software testing technique evaluation approaches based on mutation analysis can be found in Gupta and Jalote (2008); Belli, Beyazit, Hollmann, and Guler (2011); Gopinath, Alipour, Ahmed, Jensen, and Groce (2016).

Chapter 5

Treatment Design

The need for adaptation of the methodological framework proposed by Vos et al. (2012) arises from the requirement to evaluate software testing techniques in a specific scenario not covered by the existing case study protocol scenarios. The methodological framework was designed to conduct case studies following case study design guidelines, and providing a structured approach applicable to various treatments, subjects, and objects. As mentioned in the problem statement, in a particular scenario, where fault injection and conducting a traditional case study were not feasible or applicable, the framework required adaptation to enable the comparison of software testing methodologies through experiments. This chapter presents the adaptation of the methodological framework.

5.1 The experimental guidelines

The experiment process in software engineering by Wohlin et al. (2012) follows a structured approach that allows for its instantiation in different contexts within the field of software engineering. Figure 5.1 provides an overview of the entire experiment process, which can be divided into several key activities.

1. **Scope Definition**: First, the scope of the experiment should be defined, encompassing the object of study, the purpose of the experiment, the quality focus, the perspective, and the context in which the study is conducted. These elements are crucial for the goal definition of the experiment.

2. **Experiment Planning**: Planning is a critical step to ensure the usefulness and validity of the experiment. It involves defining the variables, hypotheses, and experimental design, which includes choosing a suitable experimental design. Thorough planning is essential to avoid potential pitfalls and biases that could compromise the reliability of the results.

3. **Experiment Operation**: The operation phase contains three main steps: preparation, execution, and data validation. In the preparation step, subjects and necessary materials, such as data collection forms, are prepared. Participants are informed about the experiment's intentions and their consent is obtained. The execution phase involves conducting the experiment according

to the predetermined plan and design, including data collection. Data validation is carried out to verify the correctness and validity of the collected data, ensuring its reliability.

4. **Analysis & Interpretation**: The collected data is given as input for the analysis and interpretation phase. Descriptive statistics are often employed to gain a better understanding of the data, providing visualizations and aiding in the informal interpretation of the results. This step allows researchers to uncover meaningful insights and draw conclusions from the data.

5. **Presentation & Packaging**: The final activity involves documenting and presenting the experiment's findings. This can take the form of a research paper for publication, a lab package for replication purposes, or inclusion in a company's experience base. Effective documentation ensures that the lessons learned from the experiment are properly captured and presented.

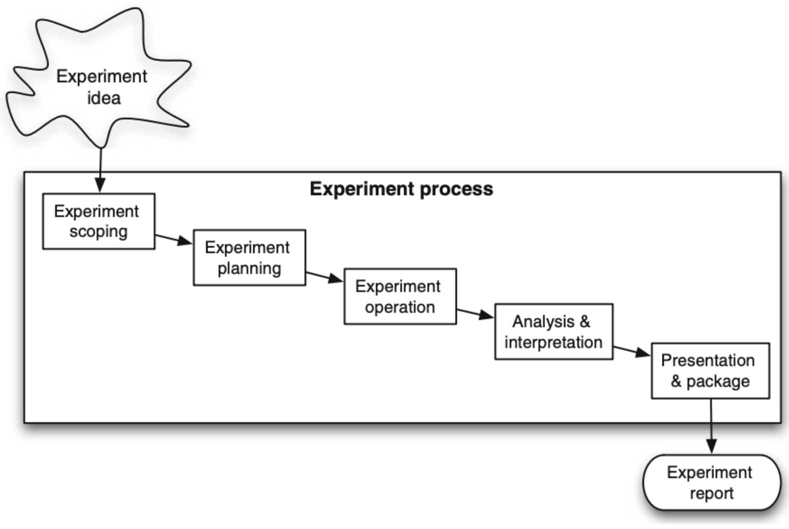


Figure 5.1: Experiment process by Wohlin et al. (2012)

5.2 Experimental reporting

The integration of study results into a body of knowledge faces a significant challenge due to the heterogeneity of study reporting. Inconsistent and non-standardized reporting often leads to difficulties in finding relevant information and missing contextual details, that are essential for further generalizability of the results (Wohlin, Höst, & Henningsson, 2003; Sjøberg et al., 2005; Dybå, Kampenes, & Sjøberg, 2006; Kampenes, Dybå, Hannay, & Sjøberg, 2007). To address this issue, Jedlitschka, Ciolkowski, and Pfahl (2008) have developed a unified reporting guideline specifically tailored for reporting controlled experiments and quasi-experiments. The guideline, presented in a structured table format, summarizes the essential elements that a report should include, which are further elaborated upon in its sub-elements. The detailed table of the reporting guideline can be found in Appendix A.

5.3 The adapted evaluation framework

As described in the problem statement, to evaluate software testing techniques in a limited scenario that is not covered in the case study protocol scenarios proposed by Vos et al. (2012), certain actions to adapt the methodological framework was needed. The original framework by Vos et al. (2012) was designed to execute case studies that follow the guidelines of case study designs. The adaptation of this framework will make it possible to compare software testing techniques by conducting experiments in a scenario where fault injection and conducting a case study is not possible.
To do so, the case study design in the original framework was transformed into an experimental design proposed by Wohlin et al. (2012). In addition, the adaptation will address the issues regarding the reporting of the experiments and will allow the reporting of the experiment designed by this framework in a standardized and consistent way. This was achieved by following the reporting guidelines by Jedlitschka et al. (2008). This adaptation ensures consistency in the evaluation process of software testing techniques by following established guidelines and standards for conducting controlled experiments in software engineering.

The adapted methodological framework is presented in a Process-Deliverable-Diagram (PDD) (van de Weerd & Brinkkemper, 2009). Modeling the framework in a PDD allows us to have an overview of the whole of the experiment and execution process (left) and the deliverables each activity of the process create (right). Furthermore, the activity and concept tables provide a detailed explanation of the activities and deliverables. The modeling of the adapted framework was divided into two different PDDs. Figure 5.2 presents the experiment process that consists of ten different phases, where one of the phases is an open activity, that is further discussed in a separate PDD, see in Figure 5.3.

5.3.1 The experiment process

The first phase of the adapted framework in the experiment process starts with the scope definition, as proposed by Wohlin et al. (2012) and depicted in Figure 5.2. This phase and the following four phases are essential for the experimental planning, which is a crucial part of the experiment itself and also the experiment report as suggested by Jedlitschka et al. (2008). Apart from that, the metrics for the dependent variables proposed by Vos et al. (2012) are defined in the first phase of the experiment. Moreover, the third phase consists of the definition of experimental material that delivers the treatment description table as proposed by Vos et al. (2012), alongside the SUT description and the relevant training material. The experiment operation is defined by the sixth and seventh phases, where the seventh phase is an open activity and is modeled and described further in Figure 5.3. The analysis and interpretation of the results are depicted in phases eight and nine. The last activity encompasses the definition of the conclusion, for which the researcher needs to follow the guidelines by Jedlitschka et al. (2008). The experiment process delivers the experimental protocol or the experiment report as the main deliverable. Further description of the activities and their sub-activities can be found in Table 5.1. For a detailed explanation of the deliverables, readers are referred to Table 5.2.

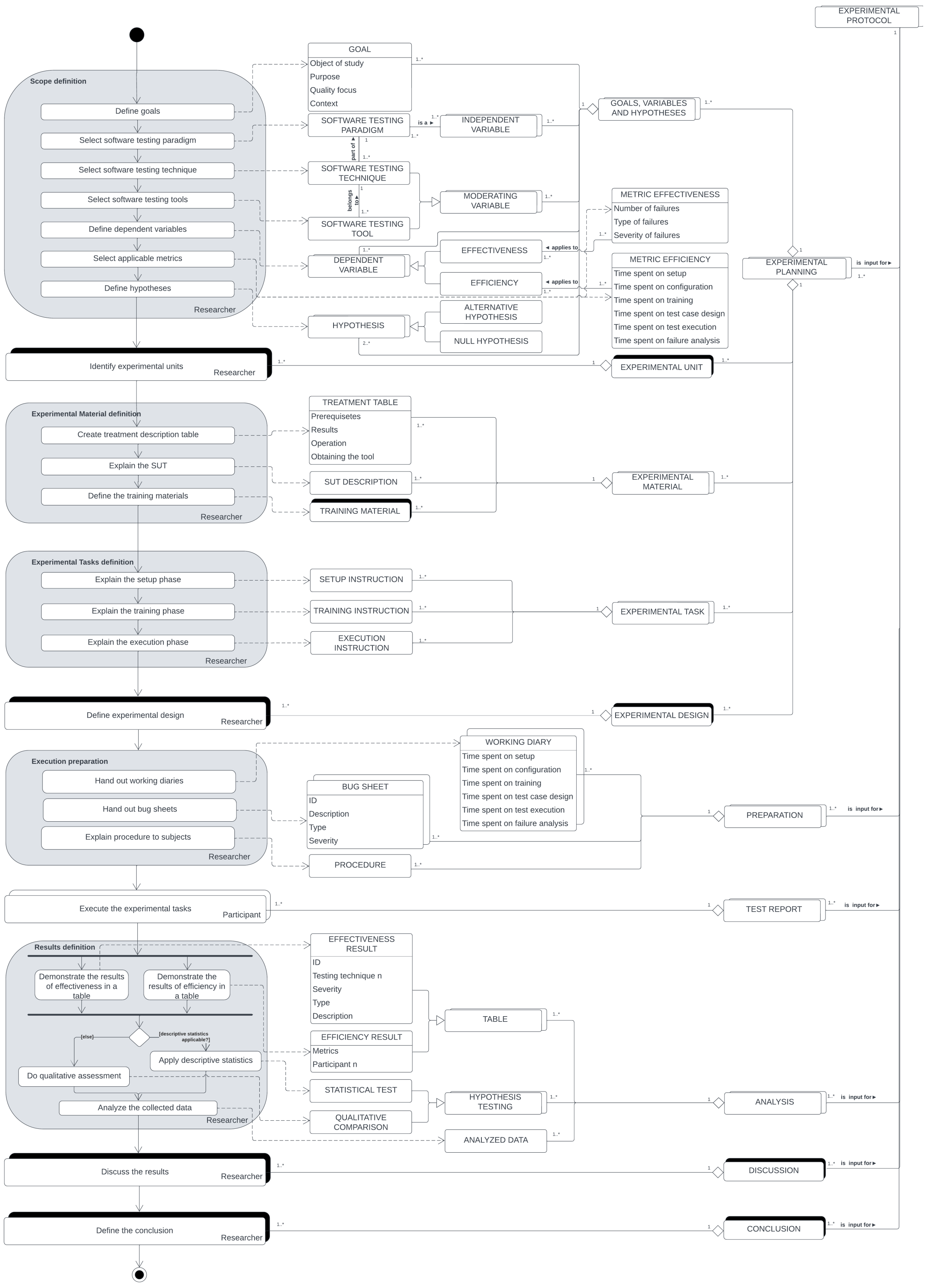
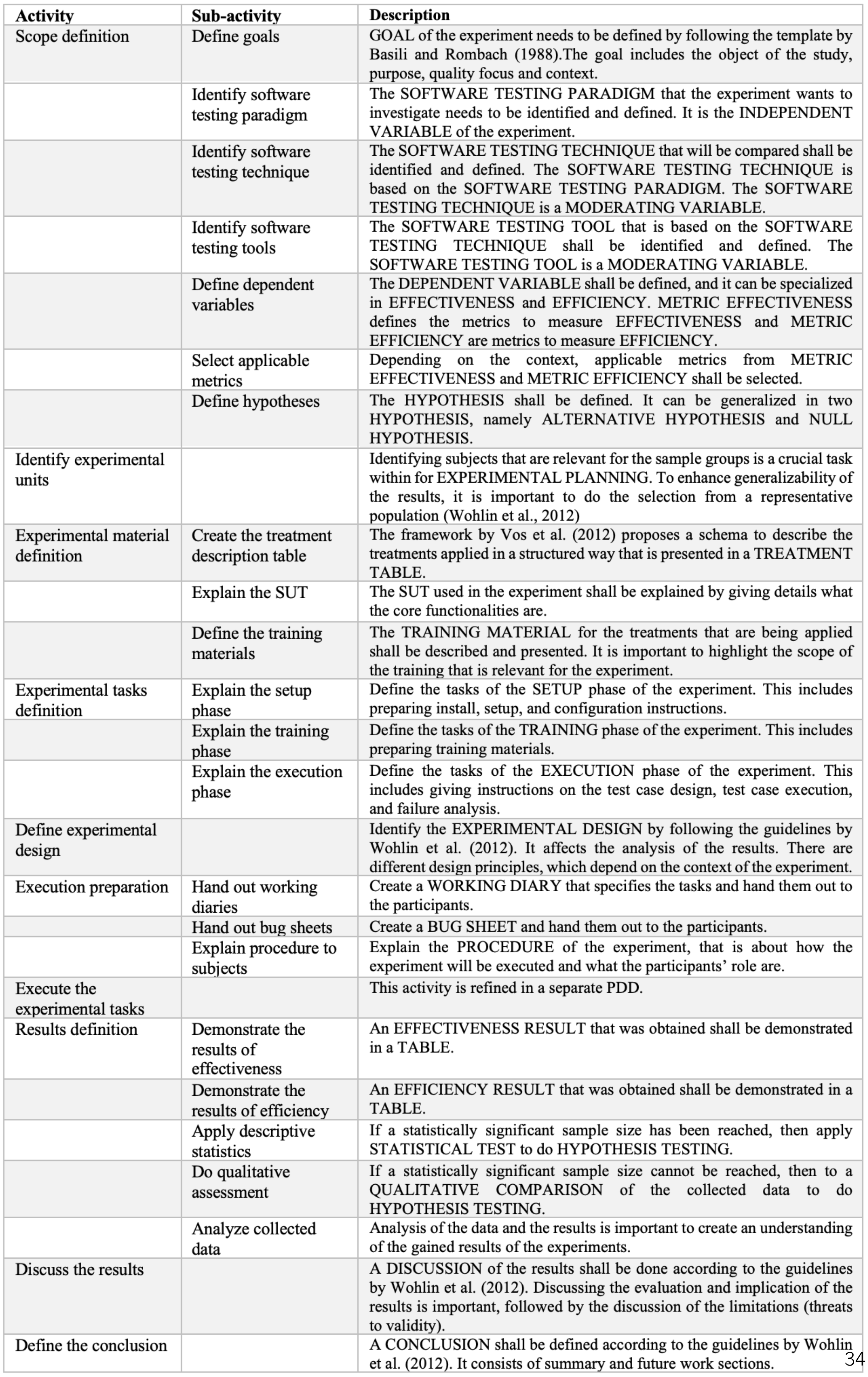


Figure 5.2: PDD Model of the experiment process

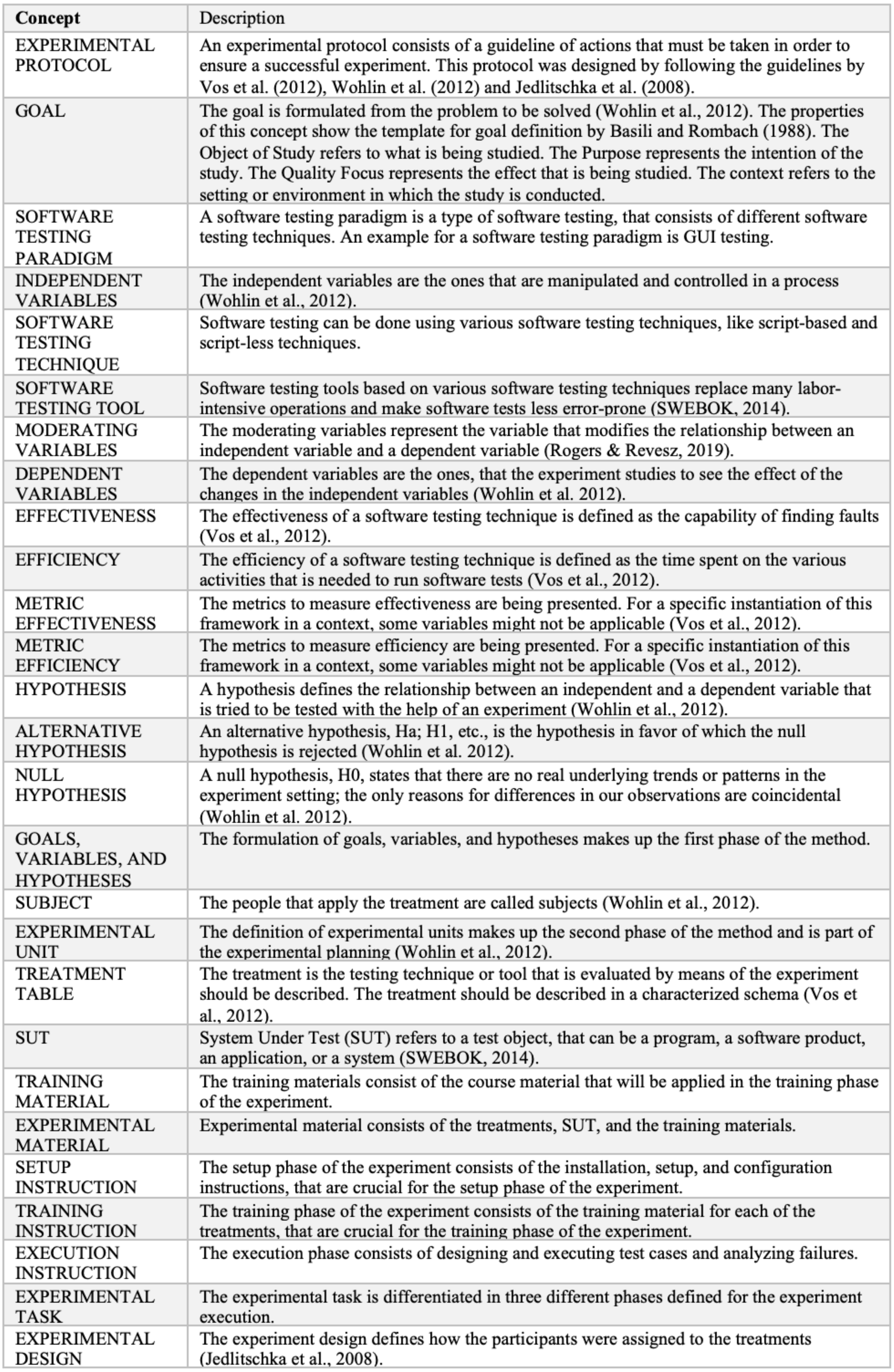
Activities of the experiment process



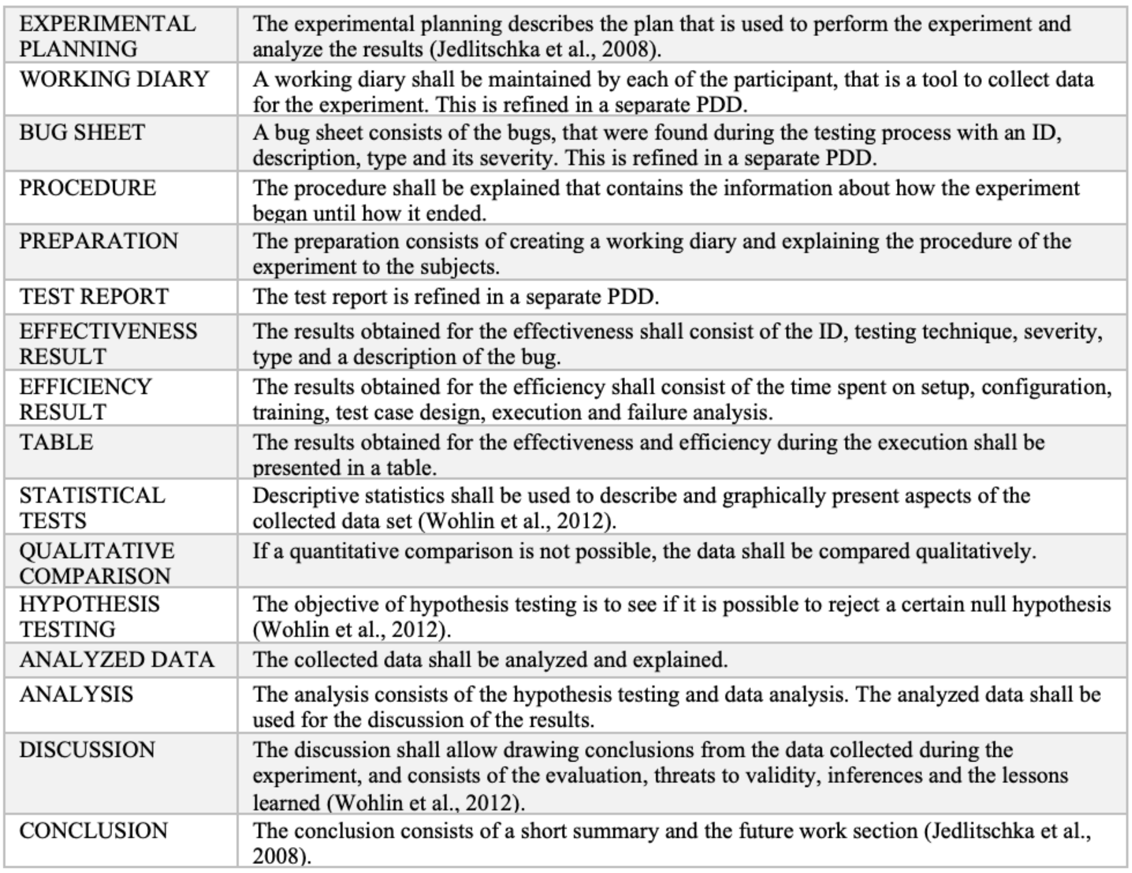
Activity	Sub-activity	Description
Scope definition	Define goals	GOAL of the experiment needs to be defined by following the template by Basili and Rombach (1988).The goal includes the object of the study, purpose, quality focus and context.
	Identify software testing paradigm	The SOFTWARE TESTING PARADIGM that the experiment wants to investigate needs to be identified and defined. It is the INDEPENDENT VARIABLE of the experiment.
	Identify software testing technique	The SOFTWARE TESTING TECHNIQUE that will be compared shall be identified and defined. The SOFTWARE TESTING TECHNIQUE is based on the SOFTWARE TESTING PARADIGM. The SOFTWARE TESTING TECHNIQUE is a MODERATING VARIABLE.
	Identify software testing tools	The SOFTWARE TESTING TOOL that is based on the SOFTWARE TESTING TECHNIQUE shall be identified and defined. The SOFTWARE TESTING TOOL is a MODERATING VARIABLE.
	Define dependent variables	The DEPENDENT VARIABLE shall be defined, and it can be specialized in EFFECTIVENESS and EFFICIENCY. METRIC EFFECTIVENESS defines the metrics to measure EFFECTIVENESS and METRIC EFFICIENCY are metrics to measure EFFICIENCY.
	Select applicable metrics	Depending on the context, applicable metrics from METRIC EFFECTIVENESS and METRIC EFFICIENCY shall be selected.
	Define hypotheses	The HYPOTHESIS shall be defined. It can be generalized in two HYPOTHESIS, namely ALTERNATIVE HYPOTHESIS and NULL HYPOTHESIS.
Identify experimental units		Identifying subjects that are relevant for the sample groups is a crucial task within for EXPERIMENTAL PLANNING. To enhance generalizability of the results, it is important to do the selection from a representative population (Wohlin et al., 2012)
Experimental material definition	Create the treatment description table	The framework by Vos et al. (2012) proposes a schema to describe the treatments applied in a structured way that is presented in a TREATMENT TABLE.
	Explain the SUT	The SUT used in the experiment shall be explained by giving details what the core functionalities are.
	Define the training materials	The TRAINING MATERIAL for the treatments that are being applied shall be described and presented. It is important to highlight the scope of the training that is relevant for the experiment.
Experimental tasks definition	Explain the setup phase	Define the tasks of the SETUP phase of the experiment. This includes preparing install, setup, and configuration instructions.
	Explain the training phase	Define the tasks of the TRAINING phase of the experiment. This includes preparing training materials.
	Explain the execution phase	Define the tasks of the EXECUTION phase of the experiment. This includes giving instructions on the test case design, test case execution, and failure analysis.
Define experimental design		Identify the EXPERIMENTAL DESIGN by following the guidelines by Wohlin et al. (2012). It affects the analysis of the results. There are different design principles, which depend on the context of the experiment.
Execution preparation	Hand out working diaries	Create a WORKING DIARY that specifies the tasks and hand them out to the participants.
	Hand out bug sheets	Create a BUG SHEET and hand them out to the participants.
	Explain procedure to subjects	Explain the PROCEDURE of the experiment, that is about how the experiment will be executed and what the participants' role are.
Execute the experimental tasks		This activity is refined in a separate PDD.
Results definition	Demonstrate the results of effectiveness	An EFFECTIVENESS RESULT that was obtained shall be demonstrated in a TABLE.
	Demonstrate the results of efficiency	An EFFICIENCY RESULT that was obtained shall be demonstrated in a TABLE.
	Apply descriptive statistics	If a statistically significant sample size has been reached, then apply STATISTICAL TEST to do HYPOTHESIS TESTING.
	Do qualitative assessment	If a statistically significant sample size cannot be reached, then to a QUALITATIVE COMPARISON of the collected data to do HYPOTHESIS TESTING.
	Analyze collected data	Analysis of the data and the results is important to create an understanding of the gained results of the experiments.
Discuss the results		A DISCUSSION of the results shall be done according to the guidelines by Wohlin et al. (2012). Discussing the evaluation and implication of the results is important, followed by the discussion of the limitations (threats to validity).
Define the conclusion		A CONCLUSION shall be defined according to the guidelines by Wohlin et al. (2012). It consists of summary and future work sections.

Table 5.1: Activity table of the experiment process

Concepts of the experiment process



Concept	Description
EXPERIMENTAL PROTOCOL	An experimental protocol consists of a guideline of actions that must be taken in order to ensure a successful experiment. This protocol was designed by following the guidelines by Vos et al. (2012), Wohlin et al. (2012) and Jedlitschka et al. (2008).
GOAL	The goal is formulated from the problem to be solved (Wohlin et al., 2012). The properties of this concept show the template for goal definition by Basili and Rombach (1988). The Object of Study refers to what is being studied. The Purpose represents the intention of the study. The Quality Focus represents the effect that is being studied. The context refers to the setting or environment in which the study is conducted.
SOFTWARE TESTING PARADIGM	A software testing paradigm is a type of software testing, that consists of different software testing techniques. An example for a software testing paradigm is GUI testing.
INDEPENDENT VARIABLES	The independent variables are the ones that are manipulated and controlled in a process (Wohlin et al., 2012).
SOFTWARE TESTING TECHNIQUE	Software testing can be done using various software testing techniques, like script-based and script-less techniques.
SOFTWARE TESTING TOOL	Software testing tools based on various software testing techniques replace many labor-intensive operations and make software tests less error-prone (SWEBOK, 2014).
MODERATING VARIABLES	The moderating variables represent the variable that modifies the relationship between an independent variable and a dependent variable (Rogers & Revesz, 2019).
DEPENDENT VARIABLES	The dependent variables are the ones, that the experiment studies to see the effect of the changes in the independent variables (Wohlin et al. 2012).
EFFECTIVENESS	The effectiveness of a software testing technique is defined as the capability of finding faults (Vos et al., 2012).
EFFICIENCY	The efficiency of a software testing technique is defined as the time spent on the various activities that is needed to run software tests (Vos et al., 2012).
METRIC EFFECTIVENESS	The metrics to measure effectiveness are being presented. For a specific instantiation of this framework in a context, some variables might not be applicable (Vos et al., 2012).
METRIC EFFICIENCY	The metrics to measure efficiency are being presented. For a specific instantiation of this framework in a context, some variables might not be applicable (Vos et al., 2012).
HYPOTHESIS	A hypothesis defines the relationship between an independent and a dependent variable that is tried to be tested with the help of an experiment (Wohlin et al., 2012).
ALTERNATIVE HYPOTHESIS	An alternative hypothesis, Ha; H1, etc., is the hypothesis in favor of which the null hypothesis is rejected (Wohlin et al. 2012).
NULL HYPOTHESIS	A null hypothesis, H0, states that there are no real underlying trends or patterns in the experiment setting; the only reasons for differences in our observations are coincidental (Wohlin et al. 2012).
GOALS, VARIABLES, AND HYPOTHESES	The formulation of goals, variables, and hypotheses makes up the first phase of the method.
SUBJECT	The people that apply the treatment are called subjects (Wohlin et al., 2012).
EXPERIMENTAL UNIT	The definition of experimental units makes up the second phase of the method and is part of the experimental planning (Wohlin et al., 2012).
TREATMENT TABLE	The treatment is the testing technique or tool that is evaluated by means of the experiment should be described. The treatment should be described in a characterized schema (Vos et al., 2012).
SUT	System Under Test (SUT) refers to a test object, that can be a program, a software product, an application, or a system (SWEBOK, 2014).
TRAINING MATERIAL	The training materials consist of the course material that will be applied in the training phase of the experiment.
EXPERIMENTAL MATERIAL	Experimental material consists of the treatments, SUT, and the training materials.
SETUP INSTRUCTION	The setup phase of the experiment consists of the installation, setup, and configuration instructions, that are crucial for the setup phase of the experiment.
TRAINING INSTRUCTION	The training phase of the experiment consists of the training material for each of the treatments, that are crucial for the training phase of the experiment.
EXECUTION INSTRUCTION	The execution phase consists of designing and executing test cases and analyzing failures.
EXPERIMENTAL TASK	The experimental task is differentiated in three different phases defined for the experiment execution.
EXPERIMENTAL DESIGN	The experiment design defines how the participants were assigned to the treatments (Jedlitschka et al., 2008).



EXPERIMENTAL PLANNING	The experimental planning describes the plan that is used to perform the experiment and analyze the results (Jedlitschka et al., 2008).
WORKING DIARY	A working diary shall be maintained by each of the participant, that is a tool to collect data for the experiment. This is refined in a separate PDD.
BUG SHEET	A bug sheet consists of the bugs, that were found during the testing process with an ID, description, type and its severity. This is refined in a separate PDD.
PROCEDURE	The procedure shall be explained that contains the information about how the experiment began until how it ended.
PREPARATION	The preparation consists of creating a working diary and explaining the procedure of the experiment to the subjects.
TEST REPORT	The test report is refined in a separate PDD.
EFFECTIVENESS RESULT	The results obtained for the effectiveness shall consist of the ID, testing technique, severity, type and a description of the bug.
EFFICIENCY RESULT	The results obtained for the efficiency shall consist of the time spent on setup, configuration, training, test case design, execution and failure analysis.
TABLE	The results obtained for the effectiveness and efficiency during the execution shall be presented in a table.
STATISTICAL TESTS	Descriptive statistics shall be used to describe and graphically present aspects of the collected data set (Wohlin et al., 2012).
QUALITATIVE COMPARISON	If a quantitative comparison is not possible, the data shall be compared qualitatively.
HYPOTHESIS TESTING	The objective of hypothesis testing is to see if it is possible to reject a certain null hypothesis (Wohlin et al., 2012).
ANALYZED DATA	The collected data shall be analyzed and explained.
ANALYSIS	The analysis consists of the hypothesis testing and data analysis. The analyzed data shall be used for the discussion of the results.
DISCUSSION	The discussion shall allow drawing conclusions from the data collected during the experiment, and consists of the evaluation, threats to validity, inferences and the lessons learned (Wohlin et al., 2012).
CONCLUSION	The conclusion consists of a short summary and the future work section (Jedlitschka et al., 2008).

Table 5.2: Concept table of the experiment process

5.3.2 The execution process

The execution process encompasses three phases: setup, training, and the execution phase as seen in Figure 5.3. These phases include activities that are part of the experimental tasks specified for the participants. The deliverables of these activities consist of the working diary, test report, and bug sheet, from which the working diary and the bug sheet are prepared and delivered by the researcher during the experiment process. The test report is generated by executing tests during the execution phase. Table 5.3 offers a deeper understanding of the activities and their sub-activities. Further description of the deliverables can be found in Table 5.4.

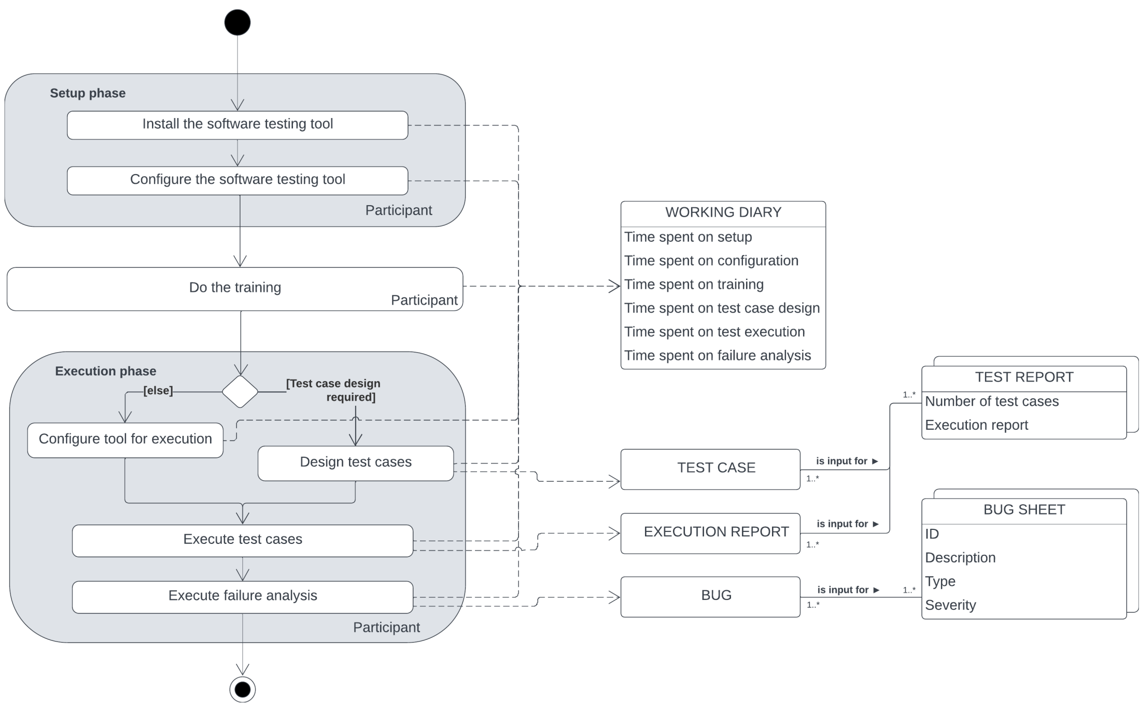


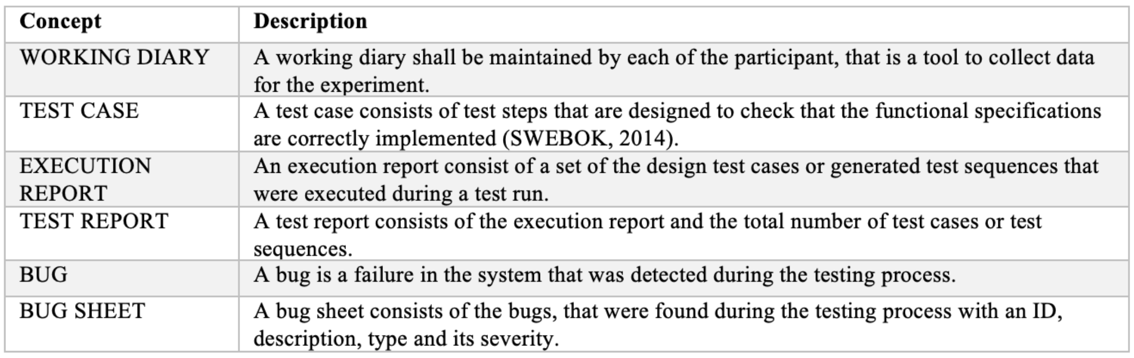
Figure 5.3: PDD Model of the execution process

Activities of the execution process

Activity	Sub-activity	Description
Setup phase	Install the software testing tool	The time spent on installation shall be recorded and is input for WORKING DIARY.
	Configure the software testing tool	The time spent on configuration shall be recorded and is input for WORKING DIARY.
Do the training		The time spent on training shall be recorded and is input for WORKING DIARY.
Execution phase	Configure tool for execution	The time spent on the configuration of the execution shall be recorded and is input for WORKING DIARY. If the software testing technique does not require manual test case generation, then the testing tool shall be configured to execute tests, e.g., test oracle definition.
	Design test cases	The participant is expected to design TEST CASEs, which is an input for the TEST REPORT. The time spent on test case design shall be recorded and is input for WORKING DIARY.
	Execute test cases	The time spent on test execution shall be recorded and is input for WORKING DIARY. An EXECUTION REPORT shall be generated after a test run, which is input for TEST REPORT.
	Execute failure analysis	The time spent on the failure analysis shall be recorded and is input for WORKING DIARY. A BUG found during the failure analysis shall be recorded in a BUG SHEET with the parameters ID, description, type and severity.

Table 5.3: Activity table of the execution process

Concepts of the execution process



Concept	Description
WORKING DIARY	A working diary shall be maintained by each of the participant, that is a tool to collect data for the experiment.
TEST CASE	A test case consists of test steps that are designed to check that the functional specifications are correctly implemented (SWEBOK, 2014).
EXECUTION REPORT	An execution report consist of a set of the design test cases or generated test sequences that were executed during a test run.
TEST REPORT	A test report consists of the execution report and the total number of test cases or test sequences.
BUG	A bug is a failure in the system that was detected during the testing process.
BUG SHEET	A bug sheet consists of the bugs, that were found during the testing process with an ID, description, type and its severity.

Table 5.4: Concept table of the execution process

Chapter 6

Treatment Validation

This chapter focuses on the validation of the adapted methodological framework for evaluating software testing techniques and tools. It provides the reader with the experimental protocol and report of the conducted quasi-experiment to validate the redesigned treatment. The goal of this treatment validation is to answer research question 2.3 *(How well does the adjusted method support the comparison of the GUI testing paradigms?)*.

6.1 Quasi-Experiment

This quasi-experiment aims to compare the script-based and script-less testing techniques within the GUI testing paradigm. By following the adjusted experimental guideline, this study investigates the relationship between the selected GUI testing techniques and their effectiveness and efficiency by utilizing software testing tools that are based on these selected software testing techniques.

6.1.1 Experimental Planning

Goals, Variables, and Hypotheses

The goal of conducting this experiment is to answer research questions 3 and 4. The research objective defined in the introduction is derived from the main research question. This section of the experimental planning will refine the specific goals of this experiment to answer research questions 3 and 4. The refined goals are defined as follows:

Goal 1: Analyze script-based and script-less GUI testing techniques
For the purpose of understanding their effectiveness
With respect to the number of defects detected, type of failures, and severity of failures.

Goal 2: Analyze script-based and script-less GUI testing techniques
For the purpose of understanding their efficiency

With respect to the time needed for designing and executing test cases, analyzing found failures, finishing the training, setting up the environment, and configuring the testing tool.

To reach these specific goals, two testing tools were selected, representing each of the GUI testing techniques this research is comparing. Table 6.1 gives an overview of the independent, moderating, and dependent variables that this experiment is investigating. Furthermore, the metrics that show how the variables will be measured are displayed in a separate column of the table. The independent variable is defined as the variable that influences and brings some variations of the dependent variable (Rogers & Revesz, 2019). On the other hand, moderating variables represent the variable that modifies the relationship between an independent variable and a dependent variable (Rogers & Revesz, 2019). The independent variable this experiment is investigating is the "GUI testing paradigms" that is expected to influence the dependent variables "effectiveness" and "efficiency" by using two different GUI testing paradigms, namely script-based and script-less GUI testing technique.

Independent Variable	**Moderating Variable**	**Dependent Variable**	**Metrics**
GUI testing paradigms			
	Script-based testing with TOSCA		
	Script-less testing with TESTAR		
		Effectiveness	• Number of failures detected • Type of failures • Severity of failures
		Efficiency	• Time needed to design the test cases • Time needed to execute the test cases • Time needed to analyze the found failures • Time needed to finish the training • Time needed to set up • Time needed to configure

Table 6.1: Variables and Metrics

Additionally, to investigate the research questions and allow this study to reach its specific goals, the following hypotheses were formulated:

- **RQ1: How do script-based and script-less testing paradigms compare in terms of test effectiveness?** Test effectiveness represents the ability of the selected approach to detect failures in the SUT (Vos et al., 2012). The metrics for measuring this variable can be found in Table 6.1.
  - *H1.0* There is no significant difference between the script-based and script-less GUI testing approach in terms of test effectiveness.
  - *H1.1* The script-less testing approach can detect more system failures like software crashes,

than the script-based GUI testing approach.

- **RQ2: How do script-based and script-less testing paradigms compare in terms of efficiency?** Test efficiency depicts the time required for the test activities, including activities from setting up the environment to analyzing the test results.
  - *H2.0* There is no significant difference between the script-based and script-less GUI testing approaches in terms of efficiency.
  - *H2.1* The script-based testing approach requires more time for the execution phase, which includes test case design and failure analysis, than the script-less GUI testing approach.

Experimental Units

The initial plan to conduct this quasi-experiment was to collaborate with a Quality Assurance team of a consulting company, that would make it possible to perform the experiment with many experienced testers and novice testers to reach a reasonable sample size. Due to time limitations and the availabilities of the potential subjects, only two testers were available at the time when the experiment was planned to be executed. For this reason, two other subjects outside of the company were included in the experiment.
One subject, who has testing experience of nine years and several years of test automation experience with Tosca, decided to participate in this experiment by applying Testar. He is part of the Quality Assurance team of the collaborating company for over a year now and has worked on several projects for test automation with Selenium.
Another subject, who has testing experience for more than a year, has started test automation with Selenium and has also gained some experience in manual testing. He selected Tosca for this experiment, as he had no experience with the tool, and was also suggested by supervisors to learn test automation with Tosca for his future projects within the company.
Two other Ph.D. students were willing to participate in the experiment, as they are currently focusing on their studies within the field of software testing and requirements engineering and wanted to gain experience in test automation with Tosca.

Experimental Material

There are several objects that this quasi-experiment requires. First of all, the treatments that are being used by the participants of the experiment. Moreover, there is the test object that is being tested. Last but not least, the training material, was used to prepare the subjects for the testing phase.

Treatments

The treatments that were used for this quasi-experiment represent the two different software testing tools Tosca and Testar. Tosca was selected to represent the script-based GUI testing technique and

Testar for the script-less GUI testing technique. An overview of both of the tools can be obtained in table 6.2. The following sections provide a more in-depth description of the tools.

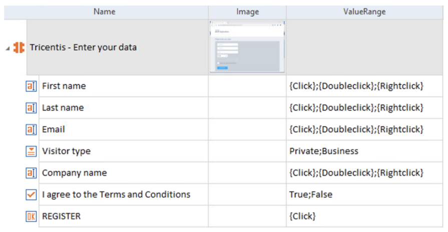
Prerequisites	**Tosca** [6.1.1]	**Testar** [6.1.1]
SUT type	Desktop, Web, API, Database, Browser, Mobile (Android and iOS)	Desktop, Web, Android
Lifecycle phase	GUI, API and Load testing	GUI testing
Environment	Tosca Engine 3.0 includes different Engines for different environments	Desktop (Windows), Web (Chromedriver), Android (Appium)
Input	Test suites including test cases, test steps, test data and test oracles	Protocol and configuration of action selection and test oracles
Knowledge	Knowledge in test case design and execution	Knowledge in GUI element structure
Experience	Advanced Tester	Advanced Tester
Results		
Output	Test cases, Test reports, Failures	Test sequences, Test reports, Failures, State models
Completeness	Depends on the number of the designed test cases	Investigated by van der Brugge, Pastor-Ricós, Aho, Marín, and Vos (2021)
Effectiveness	Investigated by this quasi-experiment	Investigated by van der Brugge et al. (2021); Bons et al. (2023); Bauersfeld et al. (2014); Martinez et al. (2016); Chahim et al. (2020) and this quasi-experiment
Defect types	Depends on the designed test cases	Failures, crashes, suspicious titles
Test suite size	Depends on the number of the designed test cases	Depends on the number of sequences and actions configured
Operation		
Information/Support	Webpage of the tool	Webpage of the tool, support by developers
Task applicability	Integration and System testing	System testing
Maturity	Widespread use outside of own organization	Academic research tool under development
Obtaining the tool	Commercial	Open source

Table 6.2: Description of the treatments following Vos et al. (2012)

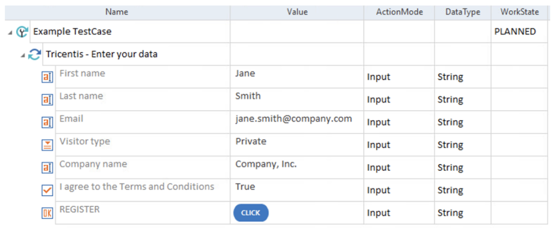
Tosca by Tricentis

Tosca by Tricentis is a model-based test automation tool that supports continuous testing for DevOps and agile management. It is a codeless automation framework, that scans the application's UI to create a business-readable automation model, which is needed to create test cases. These models can be combined and reused through the tests. Tosca supports not only GUI testing but also API and Load Testing. Tosca TBox Framework allows the application to test different environments and GUIs steering all Engines 3.0, like the XBrowser Engine 3.0, Excel Engine 3.0, SAP Engine 3.0, etc. The XBrowser Engine 3.0 is used for web applications and supports browsers like Internet Explorer, Firefox, Chrome, and Edge (Tricentis Documentation, 2023).

There are three elements that are needed to create automated tests with Tosca: Scanning, Modules, and TestCases (Tricentis Documentation, 2023). A test case design, first, starts with scanning through the GUI of the SUT. While scanning through the GUI, Tosca retrieves all the technical information on these GUI elements and saves all this information as a Module. These Modules contain the technical information that Tosca needs to interact with the SUT. A TestCase can be created by using the Modules, that were saved through the scan. After specifying a Module for a TestCase, TestSteps are created. Figure 6.1 shows an example of a Module used in a TestCase. Then, the test steps need to be defined in the "Value" column. All these elements create an automated test on Tosca.



Name	Image	ValueRange
Tricentis - Enter your data		
First name		{Click};{Doubleclick};{Rightclick}
Last name		{Click};{Doubleclick};{Rightclick}
Email		{Click};{Doubleclick};{Rightclick}
Visitor type		Private;Business
Company name		{Click};{Doubleclick};{Rightclick}
I agree to the Terms and Conditions		True;False
REGISTER		{Click}



Name	Value	ActionMode	DataType	WorkState
Example TestCase				PLANNED
Tricentis - Enter your data				
First name	Jane	Input	String	
Last name	Smith	Input	String	
Email	jane.smith@company.com	Input	String	
Visitor type	Private	Input	String	
Company name	Company, Inc.	Input	String	
I agree to the Terms and Conditions	True	Input	String	
REGISTER	CLICK	Input	String	

Figure 6.1: Modules and TestCases example retrieved from Tricentis Documentation (2023)

Testar

Testar is an open-source tool that applies a script-less approach for completely automated test generation at the GUI level for Web, Android, and Windows desktop applications (Vos et al., 2021). The script-less testing technique allows Testar to create test cases only during the test execution, meaning that no test case design is needed. The underlying principles as shown in Figure 6.2 are as follows.

Firstly, it starts with detecting all the available widgets in the current state. A state of a GUI is represented by a widget tree that consists of a widget that has properties with values (Vos et al., 2021). Then, it derives all possible actions, starting with deriving all "actionable widgets". These are widgets, that are enabled, unblocked, not filtered by the user, and expect user interaction (click, drag and drop, etc.). The next steps are selecting an action and executing it. Action selection happens randomly since Testar is based on the random testing technique. Before starting Testar, the number of sequences and actions needs to be defined. So, a test sequence ends after reaching the defined length of actions or after detecting a faulty state, and a test run ends after reaching the defined length of sequences. The last step is to check the test oracles in the new state. After every execution

of an action, Testar creates a new state of the GUI. The Testar oracle checks each of these states, to determine whether or not it is a faulty state.
A faulty state can

- be a crash, that returns a verdict of an *unexpected close*,
- be a freeze, that returns a verdict of an *unresponsive state*, or
- contain a *suspicious title*, that is defined in the Testar oracle (Vos et al., 2021).

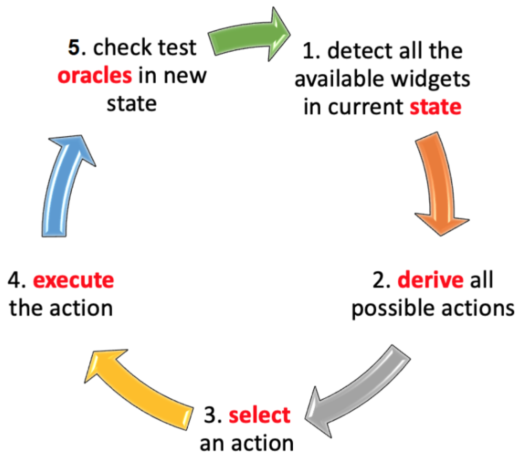

1. detect all the available widgets in current state
2. derive all possible actions
3. select an action
4. execute the action
5. check test oracles in new state

Figure 6.2: Testar principles (Testar, 2023)

System Under Test

System Under Test (SUT) refers to a test object, that can be a program, a software product, an application, or a system (Bourque et al., 2014). The test object that was used for this quasi-experiment is the Demo Web Shop that was developed by Tricentis. This Demo Web Shop is a web application that represents a web shop that includes products like books, computers, electronics, apparel and shoes, digital products, jewelry, and gift cards. It is a demo website, as the name says, that is mainly used for training purposes. It has the main functionalities of a web shop, like registering, logging in and logging out, putting products into a shopping cart and wish list, ordering products, adding reviews on items, etc. Figure 6.3 shows the main page of the web shop.

Training Material

The training materials used during the experiment are all suggested by experts and developers of the tools. The training for Tosca is proposed by the developer company Tricentis on their website called *Tricentis Academy*. This online platform consists of different training and course materials regarding different functionalities and test levels that Tosca comprises. According to the training manager of the quality assurance team of the company, there is a training package for novices, that is suggested by Tricentis and covers topics, like GUI testing, requirements and test data management, and API testing. The training starts with courses for GUI testing and continues with courses about

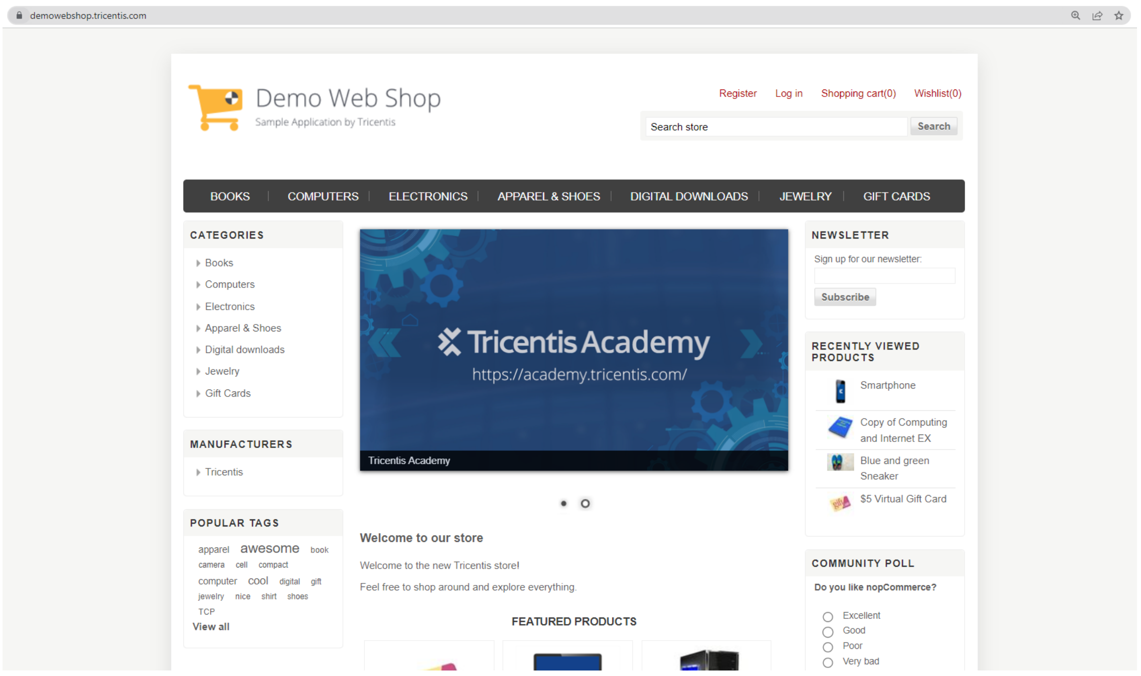


Figure 6.3: Tricentis Demo Web Shop (Tricentis, 2023)

requirements and test data management. The training that covers the course material of GUI testing and was used during this experiment with Tosca is *Automation Specialist 1* (Tricentis Academy, 2023). It consists of multiple video materials and exercises, that need to be done in order to be able to succeed in the quizzes, which are given to the student after each section to assess their state of knowledge.
The training for Testar consists of following a hands-on training manual created by the developers of Testar (Testar, 2023). It contains tasks that need to be executed to familiarize oneself with the tool's features and functionalities. It starts with instructions on how to install the tool and/or how to configure a remote environment of the tool and continues with tasks that specify the functionalities of the tool. The table of contents of both of the training materials can be found in appendix **??**.

Tasks

The experiment was divided into three different phases. Each phase has a contribution to reach the goal of the experiment and at each phase, different metrics were measured to reach the goal. Both treatment groups will have to follow each phase carefully by executing the activities step by step as suggested by the researcher. The three phases of the experiment are explained in further detail below.

- First Phase: Setup of the environment
  This phase includes tasks like installing the tool, requesting and activating the license, and requesting access to the remote testing environment. It is important to note that some tasks are tool specific like requesting and activating a license is a task that is Group Tosca specific, and requesting access to the remote testing environment is only a task that Group Testar has to

execute. Additionally, the phase includes following installation instructions, and ensuring that all dependencies are met. The installation instructions are all given by the corresponding websites of both of the tools.

- Second phase: Training
  The training phase consists of similar tasks for both of the treatment groups, however, they differ in training materials. Group Tosca will follow an online course, whereas Group Testar will follow the hands-on training created by the developers of Testar. Both treatment groups will have to study the training material thoroughly to understand the tool's features, capabilities, and usage guidelines.

- Third phase: Execution
  For Group Tosca, this phase involves creating models that represent the SUT's behavior and designing test cases based on these models. Additionally, they will define the expected outcomes or test oracles for each test case, specifying the conditions that determine whether the test passes or fails. Group Testar will have three iterations. The first iteration will start with a basic test oracle and depending on the behavior of the SUT and the test results, the test oracle will be defined in more detail to detect more failures for the next iterations.

Throughout all of these phases, all subjects will document every task they executed in a working diary they were provided by the researcher. The working diary will be in a form of a spreadsheet, that will also consist of a bug sheet, where the subjects will have to report the bugs found during the test execution. The working diaries can be found in appendix C. Furthermore, all subjects will be asked to document the time they have spent on each of the tasks, the problems they have encountered throughout the experiment, the bugs they have detected, and any kind of screenshots or test reports the tool provides. For the bug sheets, the reader is referred to appendix D.

Design

A quasi-experimental design will be employed using a between-subject design due to the context of the experiment and the limited number of participants. The between-subject design ensures that each participant belongs to only one group and will exclusively utilize the designated testing technique (Rogers & Revesz, 2019). This approach allows for a comparison between the two treatment groups. In this design, participants will be divided into two distinct groups: Group Tosca and Group Testar. The distribution of participants to these groups will be non-randomized due to the limited number of samples available and the limited availability of the participants for the study.
Group Testar will consist of a single participant, chosen to employ the monkey testing technique of Testar, which eliminates the need for human intervention. The decision to have only one subject running Testar is based on the understanding that having multiple subjects would not significantly impact the test outcomes, but it may impact their interpretation and create a validity threat. This experimental design aims to avoid potential biases associated with the former experiences of the

testers by not configuring more advanced test oracles within Testar. Hence, the tool will run tests as a dumb monkey, that has no specific knowledge about the behavior of the SUT, see section 4.3.1 for a detailed explanation. By adopting this experimental design and having a single participant in Group Testar, the test results gained by the tool will remain unaffected by the participant's experience or subject-specific knowledge.
Group Tosca, on the other hand, will consist of three subjects, who have different backgrounds and experiences. All of the subjects, however, do not have any test automation experience with Tosca, which is important to gain comparable test outcomes.
By utilizing a between-subject design within the quasi-experimental framework, this study aims to explore and compare the effectiveness and efficiency of the script-based testing technique with Tosca and the script-less testing technique with Testar. While a random assignment is not feasible due to the limited sample size, this design allows for a practical approach to investigate the research questions under the given context.

6.1.2 Execution

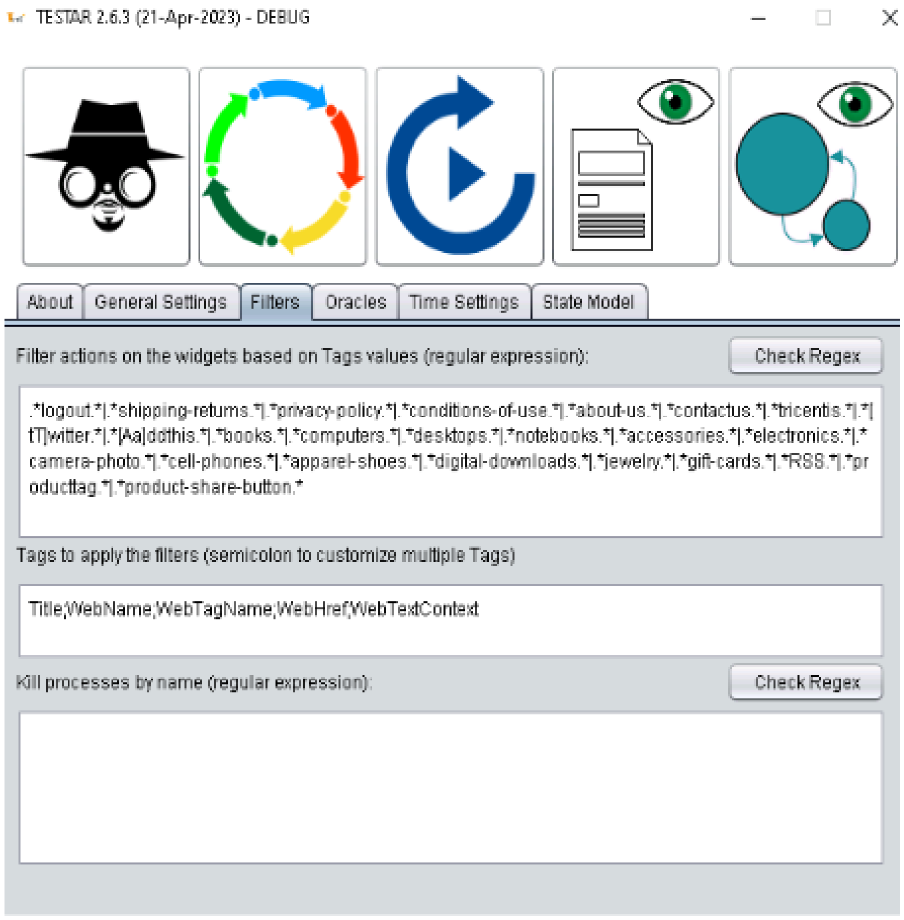
This section discusses the execution of the quasi-experiment. It starts with the procedure of how the experiment was executed. The first section explains the whole process of the experiment execution precisely. It describes every activity, that the subjects performed from the beginning until the end. The section continues with the preparations that were carried out by the researcher to execute the experiment. It concludes with deviations from the experimental planning.

Procedure

The experiment for both of the treatment groups started with setting up the environment. Group Testar got the instructions on how to set up the remote environment from the hands-on training manual. The request for access to the remote environment of Testar was made by the researcher. After receiving the login information, it was provided to the subject, and the setup phase began by following the instruction in section 2 of the hands-on manual. The training began after finishing the exercises in section 2 and the subject followed every task in each section until section 11, since testing Android systems is out of the scope for this experiment. The reader is asked to refer to appendix B for a detailed overview of the sections of the training.
The next and last phase of the experiment was the test execution phase. This phase began after the subject finished the training and was ready to start with the test execution. The first activity in this phase was to configure Testar for the test of the SUT, that was selected for this experiment. During the training for Testar, an example website was tested, for which a protocol was already provided by Testar. The subject took this existing protocol and adjusted it to the SUT of this experiment. After testing, if the protocol was working successfully, the first iteration of the execution phase with Testar started. The first iteration consisted of a basic test oracle, and a filter, that disabled the action "logout", which would limit the main functionalities of the website considerably. Testar

was configured to have 50 sequences and 25 actions per sequence for each of the iterations. The failure analysis followed the successful test run of the first iteration. This failure analysis consisted of tasks like analyzing bugs, if they are reproducible or duplicates, and, bug reporting. The bugs were documented in a spreadsheet, where information like the title, description, reproducibility, and additional comments were recorded.

The next two iterations had a similar procedure, where, in addition, only the test oracle and filters had to be refined after the failure analysis of the previous iterations. Some actions, that seemed to perform without failure, were filtered out, so it would limit the selection of actions, including actions, that were faulty and detected multiple times by Testar, and actions, that were executed multiple times in a row by Testar. The filtered actions included, product category pages, external links, or product tags, that were detected as faulty in the previous iterations. Figure 6.4 shows all the filtered actions on the last iteration.



Filter actions on the widgets based on Tags values (regular expression):

```
.*logout.*|.*shipping-returns.*|.*privacy-policy.*|.*conditions-of-use.*|.*about-us.*|.*contactus.*|.*tricentis.*|.*|
tT]witter.*|.*[Aa]ddthis.*|.*books.*|.*computers.*|.*desktops.*|.*notebooks.*|.*accessories.*|.*electronics.*|.*
camera-photo.*|.*cell-phones.*|.*apparel-shoes.*|.*digital-downloads.*|.*jewelry.*|.*gift-cards.*|.*RSS.*|.*pr
oducttag.*|.*product-share-button.*
```

Tags to apply the filters (semicolon to customize multiple Tags)

```
Title;WebName;WebTagName;WebHref;WebTextContext
```

Kill processes by name (regular expression):

Figure 6.4: Filter settings on Testar

The experiment for Group Tosca started with requesting a training license on the company's website and installing the application following the instructions on their website. After receiving the license, it needed to be activated within the application. The activation of the license made the tool ready to use and start the training. The training on the online platform of Tricentis Academy starts with navigating through the application and creating a workspace. The training is divided into modules, and after every module, there is an exercise and a quiz to complete. At the end of the training, there is a final exam, which needs to be positive in order to be eligible to receive a certification that the training was completed successfully. Taking the final exam, was not a requirement by the researcher, since the training consisted of modules, that were not necessary for the execution phase and some subjects decided to skip those modules. The first three modules of the training were relevant to the experiment, so it was expected to finish the first three modules only. One subject finished 5 out of 6 modules, and the other subjects finished 3 out of 6 modules. One subject, however, finished the training, since it was also suggested by the company to receive the certification at the end of the training. Here, the reader is referred to appendix B for the overview of the training.

After finishing the training, the subjects were ready to start with the execution. The subjects were asked to create a test design, that would encompass an end-to-end process on the webshop. They were provided with all the time they needed to create a test design, so there was no time limitation. At the end of this phase, a failure analysis was done, similar to Group Testar. The subjects also used the bug sheet to report bugs.

All subjects were told to record their time spent on each activity on a working diary, that was provided by the researcher. This working diary could also be used to record problems, the subjects encountered during the experiment. The data needed for the experimental analysis was collected through these working diaries and bug sheets and the test reports, the tools generated.

Preparation

The experiment required some preparations, that were needed to conduct the experiment successfully. These preparations were performed by the researcher, to make the execution run smoothly.

As mentioned in the previous sections, it was important to find subjects for the experiment. Additionally, the planning of the training and the training material required some preparations.

For the Testar experiment, it was required to request access to the remote environment of the testing tool. This was done by the researcher, following the information provided on the tool's website. The response from the developer team containing the login information was received within a week. During this time frame, the working diary and bug sheets were prepared for the documentation that is needed for data collection. The researcher also undertook the training for Testar, to be familiar with the tool and also to be able to support the subject during the experiment. This was also important to be able to interpret the results. In the case of the Tosca experiment, this was not necessary, since the researcher was already familiar with the tool and had the same training two years ago. The experience does not include practical experience with the tool.

A schedule for the experiment was not defined, since the availability of the subjects was limited, and it was important to avoid any time pressure on the subjects, so the quality of the experiment outcome wouldn't be influenced in any way.

Deviations

The main deviation from the initial plan was the sample size of the quasi-experiment. An ideal sample size could not be reached, because the experiment required a serious amount of time and availability of the subjects. The time needed to conduct the whole experiment for Group Tosca was approximately one working week. The time needed for the training alone is suggested to be 24 hours. The design of the test cases depends on the subjects. The time needed for the Testar experiment was less than the Tosca experiment since the training was shorter and no test case design was needed. Considering these factors, the sample size was kept small, as it was impossible to acquire more subjects for the experiment due to time limitations.

6.1.3 Data Analysis

This section presents the results of the quasi-experiment. It focuses on the data obtained to measure the effectiveness and efficiency of the script-based and script-less GUI testing tools. The working diaries and bug sheets that were used for data collection can be found in the appendix C and D.

Effectiveness of script-based and script-less GUI testing

To measure the effectiveness of both of the GUI testing paradigms, specific metrics were used. These are the number of detected failures, type, and severity of the failures. The severity of the found failures was assigned based on usability and visibility. Considering the main functionality of the demo webshop, high severity was assigned to failures when it blocked any access to a product on the website, a page failed to load, or the ordering process of a product contained a failure that blocked the user from buying it. Low severity was assigned to failures that are not visible to the user.
Table 6.3 presents the failures found by the different testing tools used in this experiment and shows the severity, type, and description of each of the failures. Testar found in its three iterations 16 bugs in total, from which 9 failures were decided to be either a duplicate or not a bug after the failure analysis by the participant. The 7 failures that are presented in Table 6.3 are all system failures, that are described either as a system failure, which can also be described as a crash, or a freeze. The failure with ID 1, has been described as not reproducible and a concurrent bug, that has no severity. 4 bugs out of 7 were assigned a high severity, which made the website either freeze or crash. 2 bugs were assigned a low severity since both of the bugs were only visible in the web console and not to the user. The functionality of the website was not impacted by the error message in the web console, therefore low severity was assigned.
Overall, Testar has detected in its three iterations, and a total of 147 test sequences, each with 25 actions, 4 bugs of high severity, 2 of low, and one with no severity.
Group Tosca has detected 16 bugs in total, from which 3 were decided to be duplicates after the failure analysis. Table 6.3 presents 13 bugs found by Tosca, from which two of them are categorized as system bugs and the rest as functional bugs. One of the system bugs was detected by both treatment

ID	Tosca	Testar	Severity	Type	Description
1		x	none	system	Suspicious tag "title" after every first login, a concurrent bug that is not reproducible
2		x	high	system	Access to a blog entry failed to load and caused an unexpected close
3		x	low	system	Twitter call responded with 403() error in the console
4		x	low	system	Twitter call responded with 401() error in the console
5		x	high	system	Access to a product tag failed to load and an empty page was opened
6		x	high	system	Another blog entry page failed to load and caused an unexpected close of the website
7	x	x	high	system	Downloaded products page failed to load, no access to downloaded products
8	x		high	functional	Missing "add to cart" button for the product "Fiction EX", even though the product is "in stock"
9	x		high	functional	Product price for desktops does not update after changing options on the product page but on the last page of the order
10	x		low	functional	Missing "add to cart" button for the product "Computing and Internet EX", but "out of stock"
11	x		high	functional	Missing "add to cart" button for the product "Desktop PC with CDRW", even though the product is "in stock"
12	x		high	functional	Missing "add to cart" button for the product "Elite Desktop PC", even though the product is "in stock"
13	x		high	functional	Missing "add to cart" button for the product "1MP 60GB Hard Drive Handycam Camcorder", even though the product is "in stock"
14	x		high	functional	Missing "add to cart" button for the product "Camcorder", even though the product is "in stock"
15	x		low	functional	Missing "add to cart" button for the product "Digital SLR Camera 12.2 Mpixel", but no information about stock availability
16	x		high	functional	Missing "add to cart" button for the product "High Definition 3D Camcorder", even though the product is "in stock"
17	x		high	functional	"Cash on delivery" as an option for payment for "downloadable products" should not be possible
18	x		high	system	Comments page for products is not responding and failed to load
19	x		low	functional	"Add to cart" buttons on a product category page don't work. The user has to open the product page and then it is possible to click on it.

Table 6.3: Failures detected by using script-based and script-less GUI testing techniques

groups, which has a high severity since it causes a system crash. 3 out of the 13 bugs are low-severity bugs since it has no major impact on the order process of an available product. The rest of the bugs found by Group Tosca are of high severity since it blocks the order process of an available product. The majority of the high-severity bugs are caused by missing "add to cart" buttons, for which the webshop shows a stock availability. Bug with ID 15 has a low severity compared to the bug with ID 8, 11-14 because there is no information about its stock availability, which could mean that the product is out of stock. In contrast, bugs with ID 8, 11-14 are in stock and have to be purchasable in the webshop. In general, Group Tosca was able to detect 13 failures, of which 11 of them are functional and 2 of them are system bugs. Regarding the number of test cases, one participant designed 10 test cases that represented an end-to-end test of one order process, where one bug was found. Another participant also designed 10 test cases, that covered multiple functionalities of the webshop, and discovered 12 bugs, where one of which was a duplicate of the bug found by the other participant. The last participant of the Group Tosca designed 9 test cases and detected 4 bugs, where two of which were duplicates of an already existing bug.

Efficiency of script-based and script-less GUI testing

Table 6.4 represents the data obtained to measure the efficiency of the software testing techniques. The data contains the time spent on executing the testing activities defined in Section 6.1.1. The first column depicts the metrics to measure the efficiency and the rest of the columns represent the data obtained from each of the participants.

The setup phase, which consisted of the setup and configuration activities, required 45 minutes in total for Group Testar, whereas Group Tosca required an average time of 40.33 minutes for the first phase. It has to be mentioned, that for Group Testar no installation of the tool was needed since the participant connected with the remote environment of Testar, where the tool was already installed and set up. So, this activity consisted of setting up the remote environment and configuring a successful connection.

For the training phase, Group Testar required 6 hours and 7 minutes to finish the suggested training material. The training material consisted of 11 sections, from which the first 9 sections were relevant for the context of this experiment, so Group Testar completed the training after finishing section 9. On the other hand, Participant 1 of Group Tosca finished the whole training material in 18 hours and 40 minutes, which also consisted of the time spent on the final exam of the training. Participant 2 completed 69% of the entire training by finishing the first 5 modules in 7 hours and 30 minutes. Similarly, Participant 3 completed 59% of the training by completing the first 3 modules in 8 hours and 56 minutes. The suggested training that covered the material relevant to the experiment consisted of the first 3 modules.

During the execution phase, Group Tosca focused on designing test cases, while Group Testar focused on executing tests. Participant 1 from Group Tosca spent 4 hours designing test cases specifically for an end-to-end test of the order process. Participant 2 from the same group dedicated 15 hours to test case design, covering core functionalities and additional features like navigating through blog and comment pages. Similarly, Participant 3 spent a total of 13 hours designing test cases, including more functionalities compared to Participant 1. In contrast, Group Testar spent a total of 3 hours and 57 minutes executing tests during three iterations. In addition to designing and executing test cases, the execution phase also involved failure analysis. Group Testar spent 2 hours and 20 minutes analyzing and documenting the identified failures in the bug sheet. Participant 1 from Group Tosca, who discovered one bug, spent 2 hours on failure analysis and reporting. Participant 2 spent 9 hours and 30 minutes analyzing the identified bugs and reporting them, while Participant 3 spent a total of 9 hours on the same activity.

Metrics	Testar	Tosca 1	Tosca 2	Tosca 3
Set up	10 min	30 min	30 min	16 min
Configuration	35 min	20 min	15 min	10 min
Training	6h 7 min	18h 40 min	7h 30 min	8h 56 min
Designing test cases	-	4h	15h	13h
Executing test cases	3h 57 min	-	-	-
Analyzing failures	2h 20 min	2h	9h 30 min	9h

Table 6.4: Time needed to execute testing activities

6.1.4 Discussion

This section provides an evaluation of the results of this quasi-experiment and some implications related to existing studies in this field. Furthermore, it explains the threats to the validity that this quasi-experiment could not mitigate due to its context. It concludes with some inferences and lessons learned.

Evaluation of results and implications

In terms of effectiveness (RQ1), Testar and Tosca demonstrated their ability to detect failures in the demo webshop. Regarding the severity of the failures, 57 % of the bugs found by Testar were of high severity, whereas for Tosca, 77 % of the bugs were of high severity. Tosca is particularly effective in identifying high-severity bugs related to the order process of available products. Testar was effective in discovering system failures, such as crashes and freezes, and was able to identify both visible failures that impacted the user experience and non-visible failures that were only visible in the web console. This aligns with findings from previous comparative studies on Testar, see Section 2.8. On the other hand, Tosca detected a mix of system and functional bugs, including one bug that was also detected by Testar.

When considering efficiency (RQ2), Testar showcased its efficiency in terms of setup and execution time. The setup phase for Testar, which involved configuring the remote environment, required less time compared to the average setup time of Tosca. Additionally, Testar spent relatively less time executing tests during the three iterations, which depends on the configured amount of executed test sequences and actions. Increasing the number of test sequences and actions would subsequently increase the execution time of Testar and possibly the number of found failures.

On the other hand, Group Tosca invested a significant amount of time in designing test cases, resulting

in a longer execution phase. The results imply that investing effort in comprehensive test case design can enhance the effectiveness of testing, especially when aiming to cover multiple functionalities of an application. Previous comparative studies on Testar indicate that, in general, this software testing technique typically spends more time on failure analysis compared to other testing techniques, except when compared to Tosca which is based on model-based testing. However, it is important to note that a considerable amount of the time spent on failure analysis by Group Tosca participants was due to analyzing bugs that were caused by erroneously designed test cases and fixing those test cases. This suggests that the increased time spent on failure analysis for Group Tosca was partly influenced by the need for adjustments and improvements in the test case design. Furthermore, the fact that Group Testar consisted of only one participant has implications for the interpretation of the results and introduces a threat to the validity of the study. The limited sample size of Group Testar reduces the generalizability of the findings and raises concerns about the reliability of the observed differences between the two groups.
In conclusion, these findings lead us to reject the null hypothesis H1.0, as a noticeable difference is evident in the number of detected system failures between both software testing approaches. This indicates that the tested approaches have varying effectiveness in detecting system failures. However, it is important to mention, that these results should be interpreted with caution considering the threats to the validity. Regarding H2.0, it is not possible to draw a definitive conclusion. Due to the inconsistent training, the participants received during the training phase, the results cannot be compared since the discrepancy in the amount of training the participants received is relatively high. This discrepancy may have influenced the time spent on failure analysis and could introduce bias into the results. Therefore, the null hypothesis H2.0 cannot be either rejected or accepted based on the available evidence.

Threats to validity

Conclusion Validity

Due to the limited sample size, descriptive statistics could not be applied to quantify and compare the results obtained from Testar and Tosca. The qualitative nature of the comparison may introduce subjectivity and limitations in drawing definitive conclusions. Therefore, the findings should be interpreted with caution, and further studies with larger sample sizes are needed to provide more robust and statistically significant conclusions.

Internal Validity

The skills and experience of the participants using Testar and Tosca may have influenced their performance and results. However, it is important to note that Group Testar consisted of only one participant. This is due to the nature of the software testing technique employed by Testar, which does not require any human intervention during test execution. Adding more subjects to Group Testar would have an impact on the first two phases of the experiment, which are the setup and training phases, but since creating an advanced test oracle was not asked for this experiment, which requires advanced programming skills, there would be no impact on the results of the execution phase.

External Validity

The findings are specific to the one SUT used in the experiment, and the generalization of the results on other applications should be done carefully. Different applications may show distinct characteristics that can affect the performance of the software testing techniques. In this study, two of the participants from Group Tosca were students, which introduces potential biases and limits the generalizability of the results to a more experienced population of software testers.

Construct Validity

Variation in the time spent on training among participants introduces potential differences in skills and understanding of the use of the software testing techniques, which hinders the comparison of the results. Additionally, the specific implementation and configuration of Testar and Tosca in the experiment may differ from real-world scenarios, potentially affecting the construct validity of the findings. An example could be choosing to download and configure Testar on a local environment, which can make the setup phase more complex.
Furthermore, the SUT was developed primarily to serve as a training platform for Tosca. This raises the question of whether the SUT adequately represents the real-world systems typically encountered in software testing scenarios.

Inferences and Lessons Learned

Inferences and lessons learned from this study should be interpreted within the context of these potential threats to validity. The limited sample size highlights the need for future studies, so a statistically significant conclusion can be drawn with a larger sample size that makes statistical hypothesis testing possible. This is necessary to mitigate validity threats and will enhance the generalizability of the findings. Furthermore, conducting experiments with diverse applications can provide a more comprehensive understanding of the effectiveness and efficiency of different software testing techniques.

6.1.5 Conclusion

This quasi-experiment aimed to compare script-based and script-less GUI testing techniques in regard to their effectiveness and efficiency. The effectiveness was evaluated based on the number, severity, and of failures detected, while efficiency was assessed by measuring the time spent on various testing activities. For the effectiveness evaluation, Group Testar detected a total of 7 bugs, with 4 categorized as high severity, 2 as low severity, and 1 with no severity assigned. On the other hand, Group Tosca detected 13 bugs, with 11 categorized as functional and 2 as system bugs. Although script-less testing detected more system failures, the difference in effectiveness between the two software testing techniques requires further statistical analysis. Regarding efficiency, Group Testar required 3 hours and 57 minutes for test execution and 2 hours and 20 minutes for failure analysis. Group Tosca spent considerable time on test case design and failure analysis. The qualitative comparison suggests that script-based requires more time for the execution phase since it requires additional effort in failure analysis due to issues arising from test case design.

Chapter 7

Discussion

7.1 Limitations

As with every study, this research also has its limitations. The threats to the validity of the study are differentiated into four categories: conclusion validity, internal validity, external validity, and construct validity (Wohlin et al., 2012). This section discusses the threats to the validity of the results in this research.

Conclusion Validity

The adapted framework was validated through a quasi-experiment with limited sample size. The small number of participants may restrict the generalizability of the findings. The small sample size also prevents the application of statistical analysis to obtain the significance of the results and validate the adapted framework. Thus, the conclusions drawn from the study should be interpreted with caution and may not be fully representative of the broader population or applicable in all contexts. Moreover, the framework incorporates subjective interpretation in some parts of the experiment, such as severity assignment and qualitative comparisons of testing techniques. The assessment of severity levels and the subjective evaluation of the effectiveness and efficiency of the testing techniques introduce the potential for bias. Different individuals may have varying interpretations and judgments based on their own perspectives and experiences, which can impact the objectivity and reliability of conclusions, that were obtained by following this adapted framework. The subjectivity involved in the interpretation of the findings creates a threat to the conclusion validity of the guideline. To address these conclusion validity threats, future research can focus on expanding the sample size. This would allow statistical analysis and enhance the generalizability of the findings. Additionally, efforts can be made to minimize subjectivity by providing clearer guidelines and criteria for severity assignment and qualitative comparisons. Incorporating multiple evaluators can help mitigate the potential impact of subjective bias on the conclusions drawn by applying this framework.

Internal Validity

The small sample size and specific characteristics of the participants may introduce selection bias, limiting the generalizability of the findings. In this research study, the participants consisted of two software testing professionals and two Ph.D. students, with varying levels of software testing experience. The inclusion of participants from these specific backgrounds may not fully represent the broader population of software testers, introducing potential biases in the findings. Furthermore, variations in the completion of the training among the participants could impact the reliability of the outcomes that were achieved by following the adapted framework. In the treatment validation, there were discrepancies in the extent to which the participants completed the suggested training material due to some time limitations. One participant from Group Tosca completed the entire training, while the other two participants finished different amounts of modules. These differences in training completion and understanding may influence the outcomes. To mitigate these internal validity threats, future research should aim to increase the sample size and the diversity of participants to include a wider range of software testers. Additionally, efforts should be made to ensure consistency in the training received by participants, such as providing clear guidelines and instructions for completing the training material. This could include adding an evaluation of the training, which could be achieved through assessments or practical exercises at the end of the training phase. This would help to reduce the variability caused by differences in training completion and understanding, enhancing the internal validity of the research and improving the generalizability of the findings.

External Validity

The adapted framework was primarily validated in a specific context, focusing on the domain of automated GUI testing and specifically targeting a web application as the SUT. As a result, the external validity of the adapted framework may be limited in terms of its applicability to different contexts beyond automated GUI testing or other types of software systems. The findings of this research may have been influenced by the unique characteristics of the automated GUI testing paradigm and the specific web application used in the study. Different industries, organizations, or software systems with different technologies or user interfaces may present varying challenges that could affect the applicability of the adapted framework. To enhance the external validity of the adapted framework, future research should consider validating the framework in a broader range of contexts, such as mobile applications, desktop applications, or different industries. By including diverse contexts, the framework can be evaluated for its applicability to various software testing paradigms, allowing for more generalizable recommendations that consider a wider range of contexts and systems. This would increase the external validity and ensure that the framework remains valuable for the evaluation of different software testing paradigms.

Construct Validity

During the preparation for the quasi-experiment, participants were provided with working diaries to track the time spent on different activities. The diaries were intended to capture the exact time spent on each activity. However, it was observed that some participants did not provide precise time entries and instead provided rough estimates, especially for the time spent on training activities. This discrepancy in reporting may introduce a threat to the construct validity of the study. One possible reason for the estimated time entries could be the length and time-consuming nature of the training materials. Participants might have faced difficulties in accurately tracking and reporting the exact time they spent on each training activity due to the extensive duration of the specific activities. As a result, the reported time values may not reflect the actual time invested in the training phase, which potentially impacts the efficiency measurements of the software testing techniques. To mitigate this threat, future adaptations of the working diary suggested by the guideline could be considered. For example, providing participants with clearer instructions on how to track time accurately or using automated time-tracking tools, that can capture the duration of each activity. These options could enhance the accuracy of the efficiency measured by this experiment.

7.2 Conclusion

In this research, the primary goal was to compare script-based and script-less GUI testing techniques. However, an additional problem was encountered, which was the lack of an evaluation framework for a specific scenario where conducting a case study and fault injection was not possible. To address this, an adaptation to an existing methodological framework was made to create an experimental guideline that enables experiments using measurements and metrics proposed by the original framework, while following a standardized reporting guideline. This research has two main research questions and five sub-research questions, which will be answered below:

- **RQ1.1**: *What are the state-of-the-art comparison approaches for software testing techniques in the literature?*
  There exist different evaluation methods for software testing techniques that are based on empirical and analytical methods. However, they lack either a clear procedure or metrics to measure the effectiveness and efficiency of software testing techniques.

- **RQ1.2**: *How to compare multiple software testing techniques, without the possibility of fault injection and conducting a case study, in terms of effectiveness and efficiency?*
  The methodological framework presented by Vos et al. (2012) was adapted to align with the experimental guidelines proposed by Wohlin et al. (2012), following the standardized reporting guidelines outlined by Jedlitschka et al. (2008). This adaptation facilitates the conduction of controlled experiments in a limited scenario as described in the problem statement.

- **RQ2.1**: *What are the different automated GUI testing approaches in the literature?*

The GUI testing approaches are differentiated into two testing paradigms: script-based and script-less GUI testing. A script-based GUI testing is an automated testing technique that is based on test case generation, which generates test scripts while executing the test cases. The literature differentiates between three different script-based GUI testing techniques: Capture & Replay, code-based testing, and model-based testing. In contrast, script-less GUI testing does not require any test cases and generates them during the test execution, which makes it script-less. Random testing, also called monkey testing, is a script-less testing technique.

- **RQ2.2**: *What are the challenges of automated GUI testing in the literature?*
  In addition to the automated GUI testing-specific challenges, it is impacted by the general challenges of test automation, which are caused by shorter development cycles, increasing software complexity, and the test oracle problem. Furthermore, script-based GUI testing techniques face challenges due to high maintenance and the requirement of technical knowledge. In contrast, script-less GUI testing techniques have long execution time, which also makes it difficult to reproduce failures, which usually requires the following of long test sequences.

- **RQ2.3**: *How well does the adjusted method support the comparison of the GUI testing paradigms?*
  The adapted methodological framework ensures a consistent and standardized comparison and evaluation of software testing techniques while following experimental guidelines, which allows a controlled comparison of GUI testing paradigms. Furthermore, it specifies the variables and metrics required for data collection. However, it is important to acknowledge that further research is necessary to address the limitations discussed in Chapter 7.

7.3 Future work

In future work, it is recommended to expand the sample size to improve the statistical validity of the findings. A larger sample would allow for a more robust quantitative analysis to determine the applicability of the adapted method. Additionally, exploring a wider range of applications and systems would provide insights into the generalizability of the findings obtained by applying this adapted method. Furthermore, it is suggested that future research should focus on ensuring consistency in the training phase. This would help to reduce the high variability in training, which would have a negative impact on the execution phase. Finally, some adaptations in tracking the time of testing activities might be necessary to enhance the accuracy of the efficiency measured by this adapted framework.

References

Aho, P., Alégroth, E., Oliveira, R., & Vos, T. (2016). Evolution of automated regression testing of software systems through the graphical user interface. In S. Hamrioui & J. Lloret Mauri (Eds.), *Accse 2016* (pp. 16–21). International Academy, Research, and Industry Association (IARIA).

Aho, P., Kanstrén, T., Räty, T., & Röning, J. (2014). Chapter two - automated extraction of gui models for testing. In A. Memon (Ed.), (Vol. 95, p. 49-112). Elsevier.

Aho, P., Suarez, M., Memon, A., & Kanstrén, T. (2015). Making gui testing practical: Bridging the gaps. In *2015 12th international conference on information technology - new generations* (p. 439-444). doi: 10.1109/ITNG.2015.77

Aho, P., & Vos, T. (2018). Challenges in automated testing through graphical user interface. In *2018 ieee international conference on software testing, verification and validation workshops (icstw)* (p. 118-121).

Alégroth, E., & Feldt, R. (2014). Industrial application of visual gui testing: Lessons learned. In J. Bosch (Ed.), *Continuous software engineering* (pp. 127–140). Cham: Springer International Publishing.

Ammann, P., & Offutt, J. (2016). *Introduction to software testing*. Cambridge: Cambridge University Press.

Appium. (2023). *Appium.* Retrieved 2023-03-11, from https://appium.io/

AutoIt. (2023). *Autoit.* Retrieved 2023-03-11, from https://www.autoitscript.com/site/

Banerjee, I., Nguyen, B., Garousi, V., & Memon, A. (2013, 10). Graphical user interface (gui) testing: Systematic mapping and repository. *Information and Software Technology*, *55*.

Baresi, L., & Young, M. (2001). Test oracles. *Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and . . . .*

Barr, E., Harman, M., McMinn, P., Shahbaz, M., & Yoo, S. (2014, 01). The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, *41*, 1-1.

Basili, V. R. (1993). The experimental paradigm in software engineering. In H. D. Rombach, V. R. Basili, & R. W. Selby (Eds.), *Experimental software engineering issues: Critical assessment and future directions* (pp. 1–12). Berlin, Heidelberg: Springer Berlin Heidelberg.

Basili, V. R., Selby, R. W., & Hutchens, D. H. (1986). Experimentation in software engineering. *IEEE Transactions on Software Engineering*, *SE-12*(7), 733-743.

Bauersfeld, S., Vos, T. E., Condori-Fernández, N., Bagnato, A., & Brosse, E. (2014). Evaluating the testar tool in an industrial case study. In *Proceedings of the 8th acm/ieee international symposium on empirical software engineering and measurement* (pp. 1–9).

Belli, F., Beyazit, M., Hollmann, A., & Guler, N. (2011, 08). Statistical evaluation of test sets using mutation analysis. In (p. 180 - 183).

Bons, A., Marín, B., Aho, P., & Vos, T. E. (2023). Scripted and scriptless gui testing for web applications: An industrial case. *Information and Software Technology*, *158*, 107172.

Borjesson, E., & Feldt, R. (2012). Automated system testing using visual gui testing tools: A comparative study in industry. In *2012 ieee fifth international conference on software testing, verification and validation* (pp. 350–359).

Bourque, P., Fairley, R. E., & Society, I. C. (2014). *Guide to the software engineering body of knowledge (swebok(r)): Version 3.0* (3rd ed.). Washington, DC, USA: IEEE Computer Society Press.

Chahim, H., Duran, M., Vos, T., Aho, P., & Condori-Fernández, N. (2020). Scriptless testing at the gui level in an industrial setting. In *Research challenges in information science* (pp. 267–284). Springer Nature Switzerland AG. (The 14th International Conference on Research Challenges in Information Science, RCIS 2020)

Di Martino, S., Fasolino, A., Tramontana, P., & Starace, L. (2020, 10). Comparing the effectiveness of capture and replay against automatic input generation for android graphical user interface testing. *Software Testing Verification and Reliability*, *31*.

Do, H., Elbaum, S., & Rothermel, G. (2004). Infrastructure support for controlled experimentation with software testing and regression testing techniques. In *Proceedings. 2004 international symposium on empirical software engineering, 2004. isese '04.* (p. 60-70).

Do, H., Elbaum, S. G., & Rothermel, G. (2005). Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, *10*, 405-435.

Dybå, T., Kampenes, V. B., & Sjøberg, D. I. (2006). A systematic review of statistical power in software engineering experiments. *Information and Software Technology*, *48*(8), 745–755.

Easterbrook, S., Singer, J., Storey, M.-A., & Damian, D. (2008). Selecting empirical methods for software engineering research. In F. Shull, J. Singer, & D. I. K. Sjøberg (Eds.), *Guide to advanced empirical software engineering* (pp. 285–311). London: Springer London.

Eldh, S., Hansson, H. A., Punnekkat, S., Pettersson, A., & Sundmark, D. (2006). A framework for comparing efficiency, effectiveness and applicability of software testing techniques. *Testing: Academic & Industrial Conference - Practice And Research Techniques (TAIC PART'06)*, 159-170.

Girard, E., & Rault, J. (1973). A programming technique for software reliability. In *Proceedings of 1973 ieee symposium on computer software reliability* (pp. 44–50).

Gopinath, R., Alipour, A., Ahmed, I., Jensen, C., & Groce, A. (2016). Measuring effectiveness of mutant sets. In *2016 ieee ninth international conference on software testing, verification and validation workshops (icstw)* (p. 132-141).

Graham, D., Black, R., & van Veenendaal, E. (2021). *Foundations of software testing istqb certification, 4th edition*. Cengage Learning.

Grilo, A. M. P., Paiva, A. C. R., & Faria, J. P. (2010). Reverse engineering of gui models for testing. In *5th iberian conference on information systems and technologies* (p. 1-6).

Gupta, A., & Jalote, P. (2008). An approach for experimentally evaluating effectiveness and efficiency of coverage criteria for software testing. *International Journal on Software Tools for Technology Transfer*, *10*, 145-160.

Hetzel, B. (1988). *The complete guide to software testing*. QED Information Sciences.

IEEE. (1990). Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, 1-84.

Jedlitschka, A., Ciolkowski, M., & Pfahl, D. (2008). Reporting experiments in software engineering. In F. Shull, J. Singer, & D. I. K. Sjøberg (Eds.), *Guide to advanced empirical software engineering* (pp. 201–228). London: Springer London.

Kampenes, V. B., Dybå, T., Hannay, J. E., & Sjøberg, D. I. (2007). A systematic review of effect size in software engineering experiments. *Information and Software Technology*, *49*(11-12), 1073–1086.

Karam, M. R., Dascalu, S. M., & Hazimé, R. H. (2006). Challenges and opportunities for improving code-based testing of graphical user interfaces. *Journal of Computational Methods in Sciences and Engineering*, *6*(s2), S379–S388.

Kitchenham, B. (1993). A methodology for evaluating software engineering methods and tools. In H. D. Rombach, V. R. Basili, & R. W. Selby (Eds.), *Experimental software engineering issues: Critical assessment and future directions* (pp. 121–124). Berlin, Heidelberg: Springer Berlin Heidelberg.

Kitchenham, B., Pickard, L., & Pfleeger, S. (1995). Case studies for method and tool evaluation. *IEEE Software*, *12*(4), 52-62.

Kumar, A., & Kaur, K. (2022). Bw-topsis: A hybrid method to evaluate software testing techniques. *Journal of Communications Software and Systems*, *18*(4), 336–342.

Lott, C., & Rombach, D. (1997, 06). Repeatable software engineering experiments for comparing defect-detection techniques. *Empirical Software Engineering*, *1*.

Martinez, M., Esparcia, A. I., Rueda, U., Vos, T. E., & Ortega, C. (2016). Automated localisation testing in industry with testar. In *Testing software and systems: 28th ifip wg 6.1 international conference, ictss 2016, graz, austria, october 17-19, 2016, proceedings 28* (pp. 241–248).

Memon, A., Banerjee, I., & Nagarajan, A. (2003). What test oracle should i use for effective gui testing? In *18th ieee international conference on automated software engineering, 2003. proceedings.* (p. 164-173). doi: 10.1109/ASE.2003.1240304

Memon, A., Pollack, M., & Soffa, M. (2001). Hierarchical gui test case generation using automated planning. *IEEE Transactions on Software Engineering*, *27*(2), 144-155.

Memon, A. M., & Nguyen, B. N. (2010). Advances in automated model-based system testing of software applications with a gui front-end. In *Advances in computers* (Vol. 80, pp. 121–162). Elsevier.

Moreira, R. M. L. M., Paiva, A. C. R., Nabuco, M., & Memon, A. (2017). Pattern-based GUI testing: Bridging the gap between design and quality assurance. *Softw. Test., Verif. Reliab.*, *27*(3).

Myers, G. J. (1979). *The art of software testing.* John Wiley & Sons.

Nass, M., Alégroth, E., & Feldt, R. (2021). Why many challenges with gui test automation (will) remain. *Information and Software Technology*, *138*, 106625.

Neto, A., & Travassos, G. (2014, 10). Supporting the combined selection of model-based testing techniques. *IEEE Transactions on Software Engineering*, *40*, 1025-1041.

Nyman, N. (2000). Using monkey test tools – how to find bugs cost-effectively through random testing. *Software Testing & Quality Engineering*, 18–21.

Pastor Ricós, F., Slomp, A., Marín, B., Aho, P., & Vos, T. E. (2023). Distributed state model inference for scriptless gui testing. *Journal of Systems and Software*, *200*, 111645.

Pezzè, M., Rondena, P., & Zuddas, D. (2018). Automatic gui testing of desktop applications: An empirical assessment of the state of the art. In *Companion proceedings for the issta/ecoop 2018 workshops* (p. 54–62). New York, NY, USA: Association for Computing Machinery.

QFS. (2023). *Qf-test.* Retrieved 2023-03-11, from https://www.qfs.de/en/index.html

Rahikkala, J., Hyrynsalmi, S., & Leppänen, V. (2015, 10). Accounting testing in software cost estimation: A case study of the current practice and impacts..

Ranorex. (2023). *Ranorex.* Retrieved 2023-03-11, from https://www.ranorex.com/

Rodríguez-Valdés, O., Vos, T. E. J., Aho, P., & Marín, B. (2021). 30 years of automated gui testing: A bibliometric analysis. In A. C. R. Paiva, A. R. Cavalli, P. Ventura Martins, & R. Pérez-Castillo (Eds.), *Quality of information and communications technology* (pp. 473–488). Cham: Springer International Publishing.

Rogers, J., & Revesz, A. (2019, 07). Experimental and quasi-experimental designs. In (p. 133-143).

Runeson, P., Host, M., Rainer, A., & Regnell, B. (2012). *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons.

Runeson, P., Höst, M., Rainer, A., & Regnell, B. (2012). *Case study research in software engineering – guidelines and examples*. doi: 10.1002/9781118181034

Selenium. (2023a). *Selenium.* Retrieved 2023-02-15, from https://www.selenium.dev/

Selenium. (2023b). *Selenium.* Retrieved 2023-04-14, from https://www.selenium.dev/documentation/webdriver/getting_started/first_script/

SeleniumIDE. (2023). *Seleniumide.* Retrieved 2023-03-11, from https://www.selenium.dev/selenium-ide/

Sharma, R. M. (2014, July). Quantitative analysis of automation and manual testing. In (Vol. 4). International Journal of Engineering and Innovative Technology (IJEIT).

Sjøberg, D. I., Hannay, J. E., Hansen, O., Kampenes, V. B., Karahasanovic, A., Liborg, N.-K., & Rekdal, A. C. (2005). A survey of controlled experiments in software engineering. *IEEE transactions on software engineering*, *31*(9), 733–753.

Squish. (2023). *Squish.* Retrieved 2023-03-11, from https://www.qt.io/product/quality-assurance/squish

Stol, K.-J., & Fitzgerald, B. (2018, sep). The abc of software engineering research. *ACM Trans. Softw. Eng. Methodol.*, *27*(3).

Synopsys. (2022). *[analyst report] 2022 the cost of poor quality software.* Retrieved 2023-02-15, from https://www.synopsys.com/software-integrity/resources/analyst-reports/cost-poor-quality-software.html?intcmp=sig-blog-cisq22

Testar. (2023). *Testar hands-on training 2.6.4.* Retrieved 2023-06-02, from https://github.com/TESTARtool/TESTAR_dev/releases/tag/prev2.6.4#:~:text=Hands_on_TESTAR_2.6.4.pdf

Thayer, T. A. (1978). Software reliability. *TRW Series of Software Technology*.

Tosca, T. (2023). *Tricentis tosca.* Tricentis. Retrieved 2023-03-02, from https://www.tricentis.com/de/plattform/automate-continuous-testing-tosca

Tricentis. (2023). *Tricentis demo web shop.* Retrieved 2023-05-30, from https://demowebshop.tricentis.com/

Tricentis Academy. (2023). *Automation Specialist Level 1.* Retrieved 2023-06-02, from https://academy.tricentis.com/automation-specialist-level-1

Tricentis Documentation. (2023). *Tricentis documentation.* Retrieved 2023-06-02, from https://documentation.tricentis.com/tosca/1400/en/content/first_steps/get_to_know_tosca.htm

Tripathy, P., & Naik, K. (2011). *Software testing and quality assurance: Theory and practice*. John Wiley & Sons.

Utting, M., & Legeard, B. (2010). *Practical model-based testing: A tools approach*. Elsevier.

van der Brugge, A., Pastor-Ricós, F., Aho, P., Marín, B., & Vos, T. E. (2021). Evaluating testar's effectiveness through code coverage. *Actas de las XXV Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2021)*, 1–14.

van de Weerd, I., & Brinkkemper, S. (2009). Meta-modeling for situational analysis and design methods. In *Handbook of research on modern systems analysis and design technologies and applications* (pp. 35–54). IGI Global.

Verner, J., Sampson, J., Tosic, V., Bakar, N. A., & Kitchenham, B. (2009). Guidelines for industrially-based multiple case studies in software engineering. In *2009 third international conference on research challenges in information science* (p. 313-324).

Vos, T., Aho, P., Pastor Ricós, F., Rodriguez-Valdes, O., & Mulders, A. (2021, 05). testar – scriptless testing through graphical user interface. *Software Testing, Verification and Reliability*, *31*.

Vos, T., Marín, B., Escalona, M., Escalona, A., & Marchetto, A. (2012, 08). A methodological framework for evaluating software testing techniques and tools..

Wieringa, R. (2014). *Design science methodology for information systems and software engineering*. Springer. doi: 10.1007/978-3-662-43839-8

Wohlin, C. (2014). Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering* (pp. 1–10).

Wohlin, C., Höst, M., & Henningsson, K. (2003). Empirical research methods in software engineering. *Empirical methods and studies in software engineering: Experiences from ESERNET*, 7–23.

Wohlin, C., Runeson, P., Hst, M., Ohlsson, M. C., Regnell, B., & Wessln, A. (2012). *Experimentation in software engineering*. Springer Publishing Company, Incorporated.

Appendix A

Reporting Guidelines

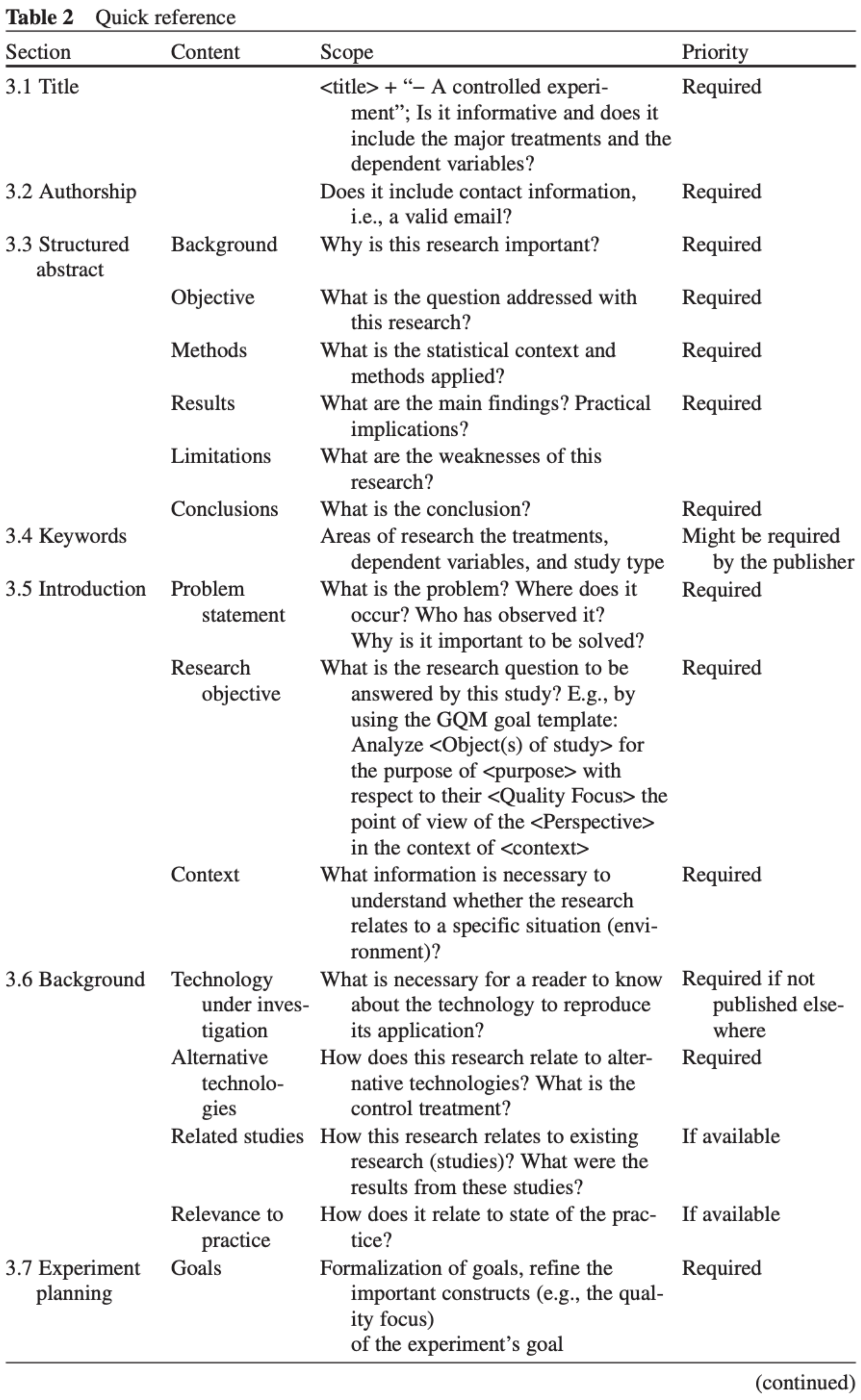


Table 2 Quick reference

Section	Content	Scope	Priority
3.1 Title		<title> + "– A controlled experiment"; Is it informative and does it include the major treatments and the dependent variables?	Required
3.2 Authorship		Does it include contact information, i.e., a valid email?	Required
3.3 Structured abstract	Background	Why is this research important?	Required
	Objective	What is the question addressed with this research?	Required
	Methods	What is the statistical context and methods applied?	Required
	Results	What are the main findings? Practical implications?	Required
	Limitations	What are the weaknesses of this research?	
	Conclusions	What is the conclusion?	Required
3.4 Keywords		Areas of research the treatments, dependent variables, and study type	Might be required by the publisher
3.5 Introduction	Problem statement	What is the problem? Where does it occur? Who has observed it? Why is it important to be solved?	Required
	Research objective	What is the research question to be answered by this study? E.g., by using the GQM goal template: Analyze <Object(s) of study> for the purpose of <purpose> with respect to their <Quality Focus> the point of view of the <Perspective> in the context of <context>	Required
	Context	What information is necessary to understand whether the research relates to a specific situation (environment)?	Required
3.6 Background	Technology under investigation	What is necessary for a reader to know about the technology to reproduce its application?	Required if not published elsewhere
	Alternative technologies	How does this research relate to alternative technologies? What is the control treatment?	Required
	Related studies	How this research relates to existing research (studies)? What were the results from these studies?	If available
	Relevance to practice	How does it relate to state of the practice?	If available
3.7 Experiment planning	Goals	Formalization of goals, refine the important constructs (e.g., the quality focus) of the experiment's goal	Required

(continued)

Table A.1: Reporting guidelines table part 1

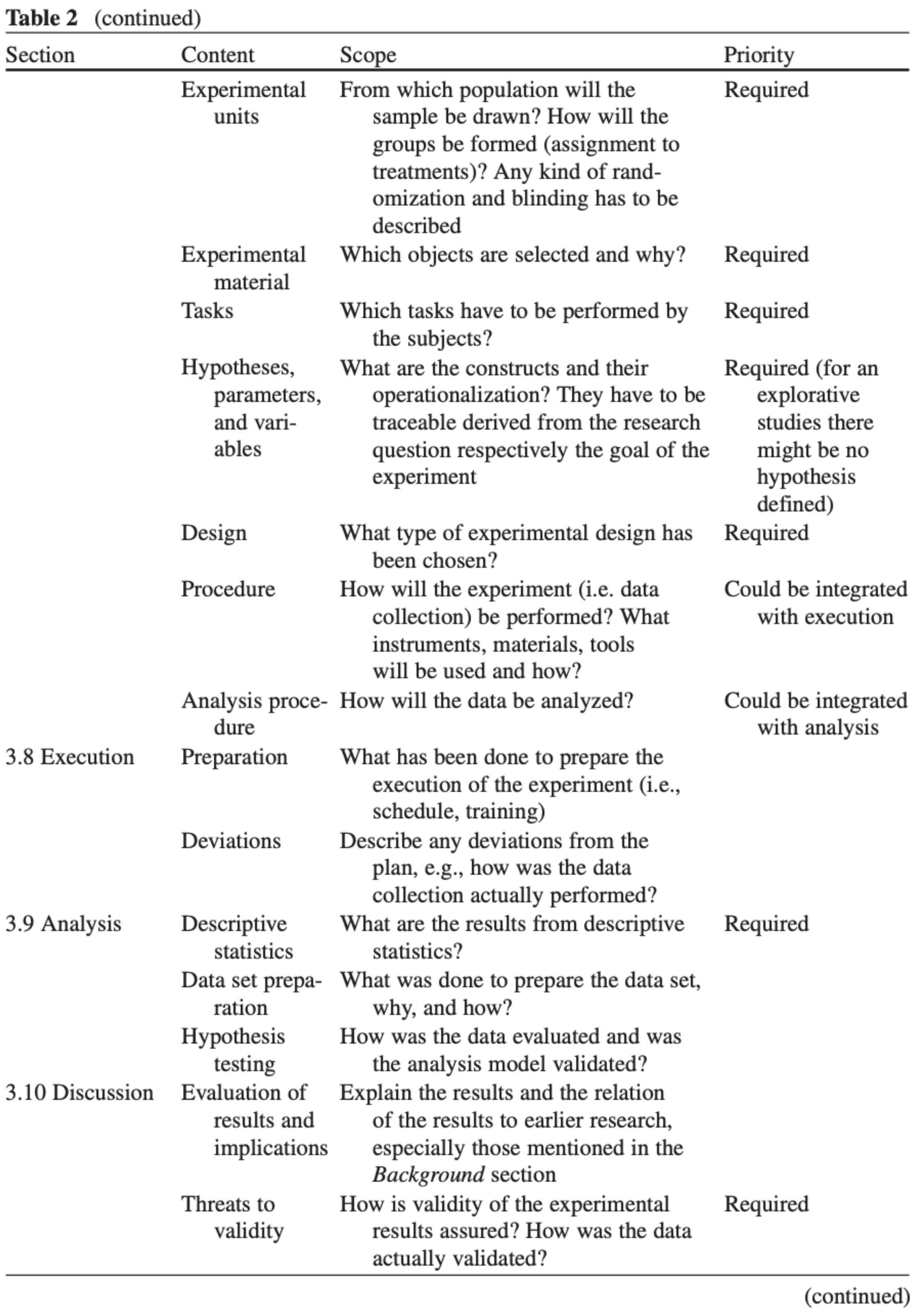


Table 2 (continued)

Section	Content	Scope	Priority
	Experimental units	From which population will the sample be drawn? How will the groups be formed (assignment to treatments)? Any kind of randomization and blinding has to be described	Required
	Experimental material	Which objects are selected and why?	Required
	Tasks	Which tasks have to be performed by the subjects?	Required
	Hypotheses, parameters, and variables	What are the constructs and their operationalization? They have to be traceable derived from the research question respectively the goal of the experiment	Required (for an explorative studies there might be no hypothesis defined)
	Design	What type of experimental design has been chosen?	Required
	Procedure	How will the experiment (i.e. data collection) be performed? What instruments, materials, tools will be used and how?	Could be integrated with execution
	Analysis procedure	How will the data be analyzed?	Could be integrated with analysis
3.8 Execution	Preparation	What has been done to prepare the execution of the experiment (i.e., schedule, training)	
	Deviations	Describe any deviations from the plan, e.g., how was the data collection actually performed?	
3.9 Analysis	Descriptive statistics	What are the results from descriptive statistics?	Required
	Data set preparation	What was done to prepare the data set, why, and how?	
	Hypothesis testing	How was the data evaluated and was the analysis model validated?	
3.10 Discussion	Evaluation of results and implications	Explain the results and the relation of the results to earlier research, especially those mentioned in the *Background* section	
	Threats to validity	How is validity of the experimental results assured? How was the data actually validated?	Required

(continued)

Table A.2: Reporting guidelines table part 2

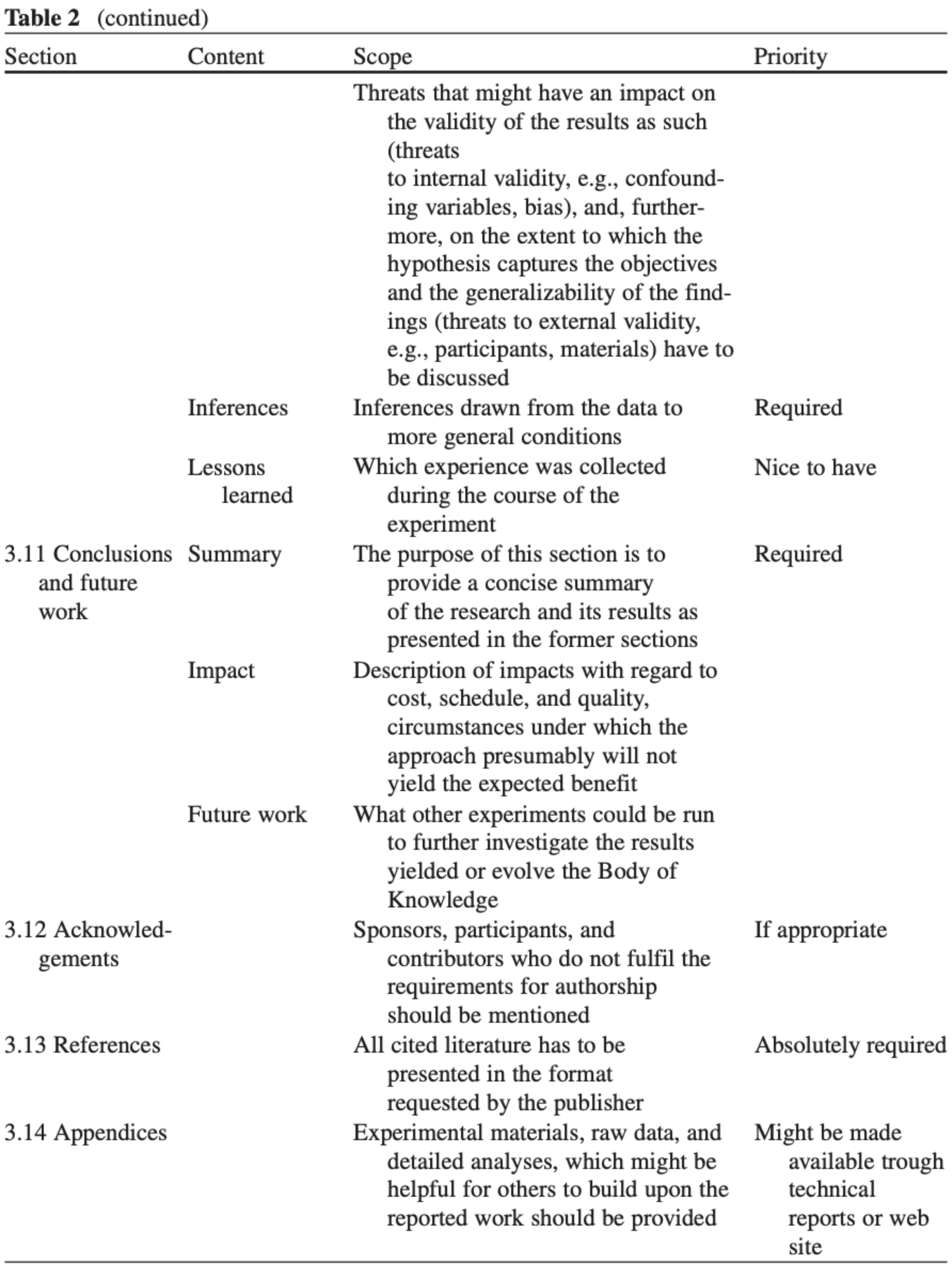


Table 2 (continued)

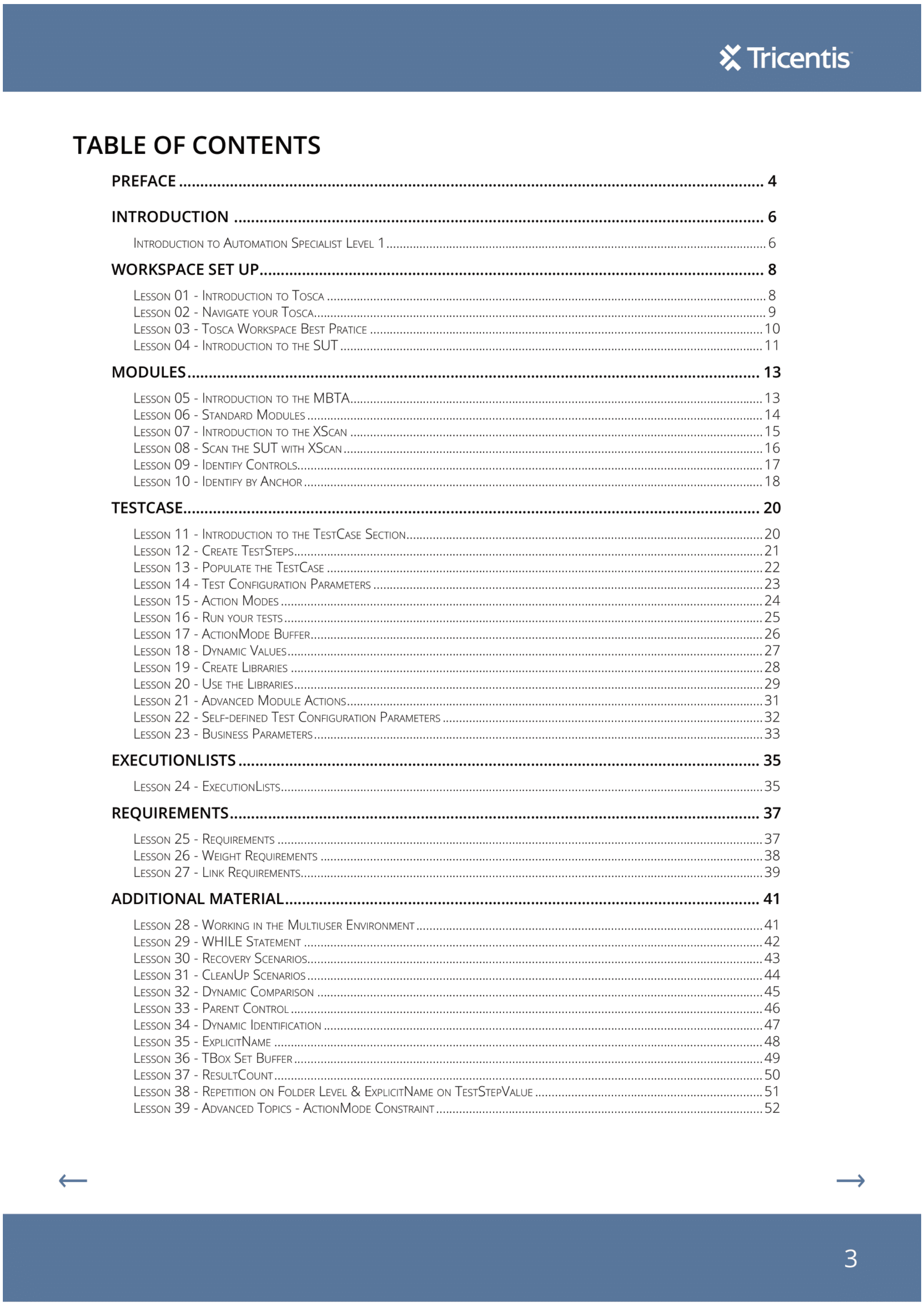
Section	Content	Scope	Priority
		Threats that might have an impact on the validity of the results as such (threats to internal validity, e.g., confounding variables, bias), and, furthermore, on the extent to which the hypothesis captures the objectives and the generalizability of the findings (threats to external validity, e.g., participants, materials) have to be discussed	
	Inferences	Inferences drawn from the data to more general conditions	Required
	Lessons learned	Which experience was collected during the course of the experiment	Nice to have
3.11 Conclusions and future work	Summary	The purpose of this section is to provide a concise summary of the research and its results as presented in the former sections	Required
	Impact	Description of impacts with regard to cost, schedule, and quality, circumstances under which the approach presumably will not yield the expected benefit	
	Future work	What other experiments could be run to further investigate the results yielded or evolve the Body of Knowledge	
3.12 Acknowledgements		Sponsors, participants, and contributors who do not fulfil the requirements for authorship should be mentioned	If appropriate
3.13 References		All cited literature has to be presented in the format requested by the publisher	Absolutely required
3.14 Appendices		Experimental materials, raw data, and detailed analyses, which might be helpful for others to build upon the reported work should be provided	Might be made available trough technical reports or web site

Table A.3: Reporting guidelines table part 3

Appendix B

Training Materials

B.1 Tosca Training



Tricentis

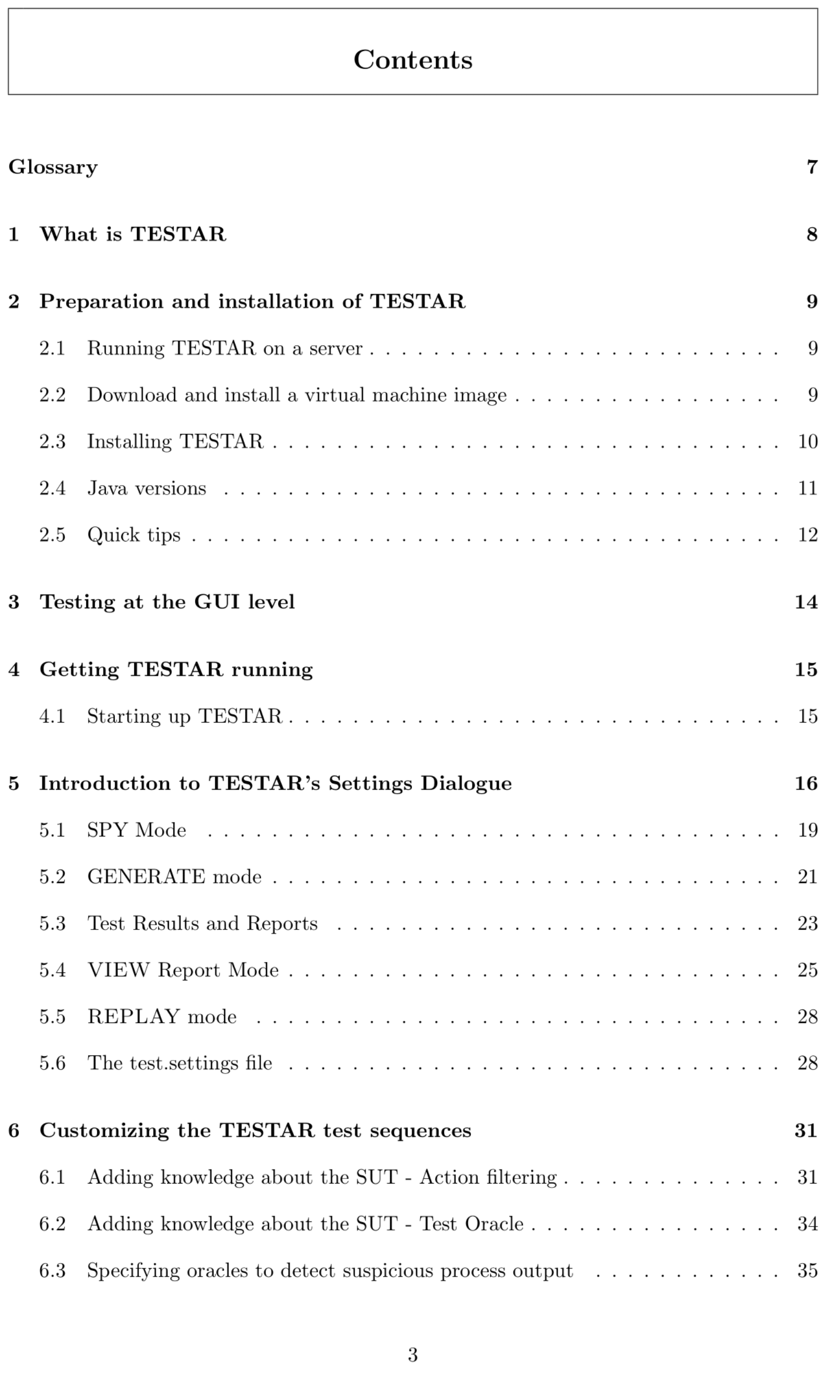
TABLE OF CONTENTS

← →

3

Figure B.1: Tosca Training - Table of contents

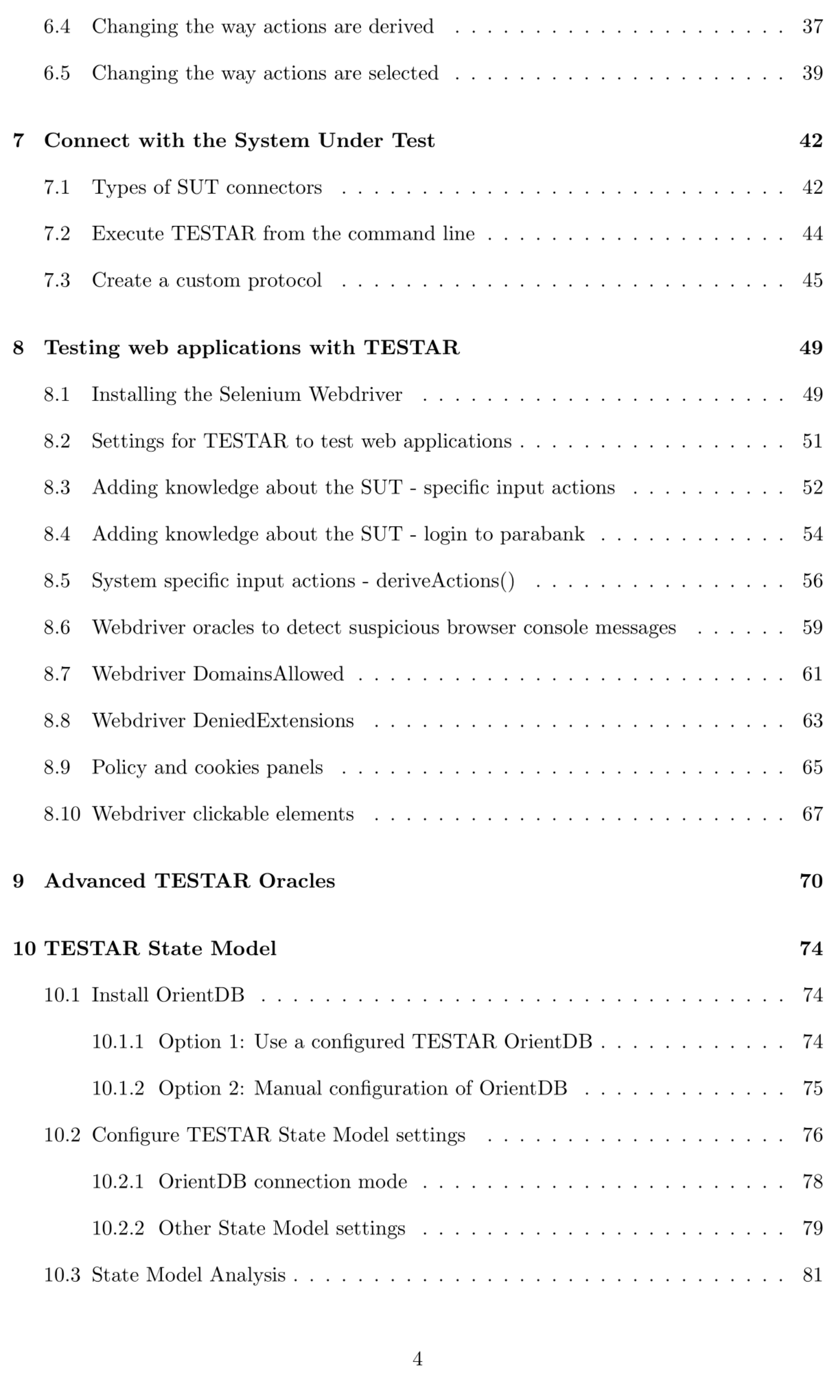
B.2 Testar Training



Contents

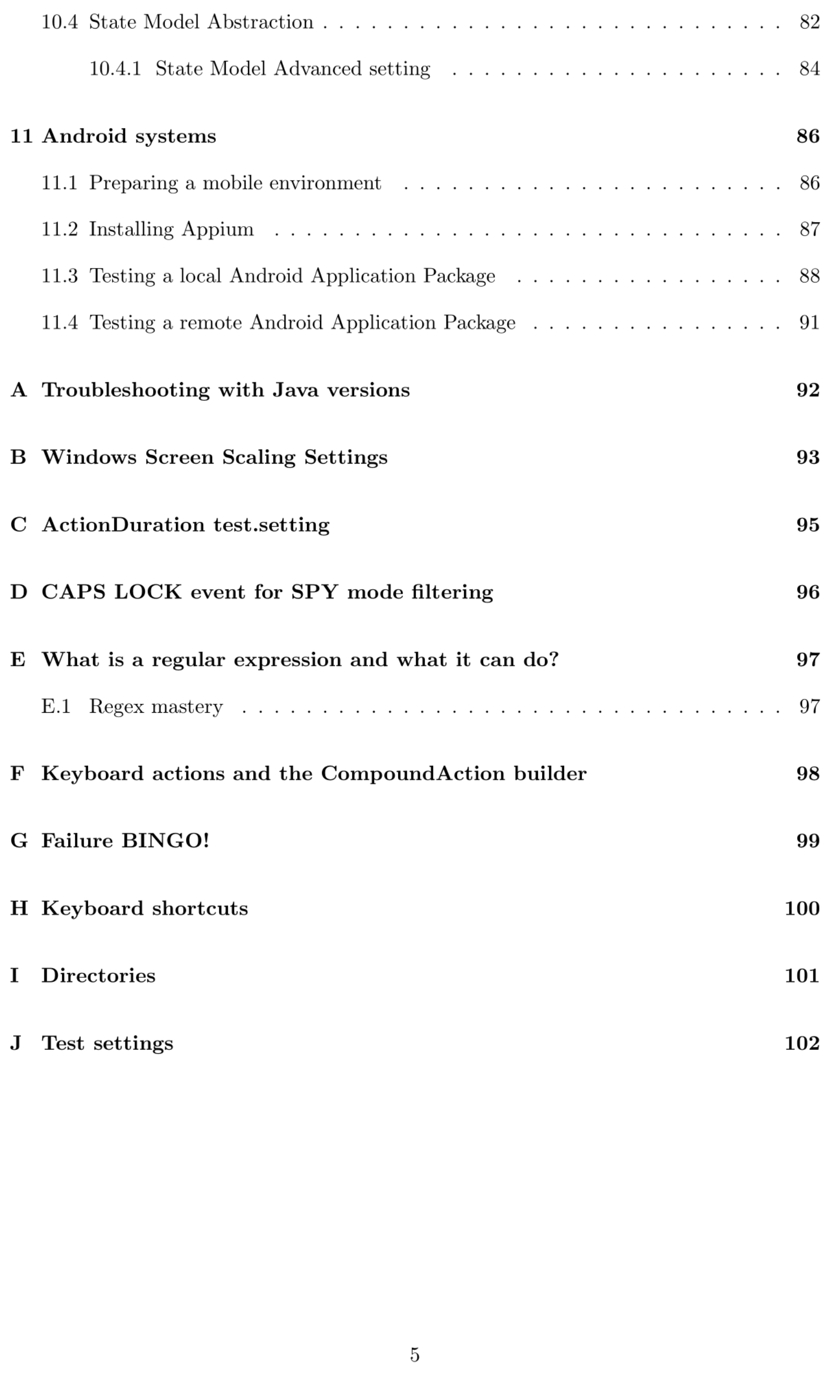
3

Figure B.2: Testar Training - Table of contents part 1



4

Figure B.3: Testar Training - Table of contents part 2



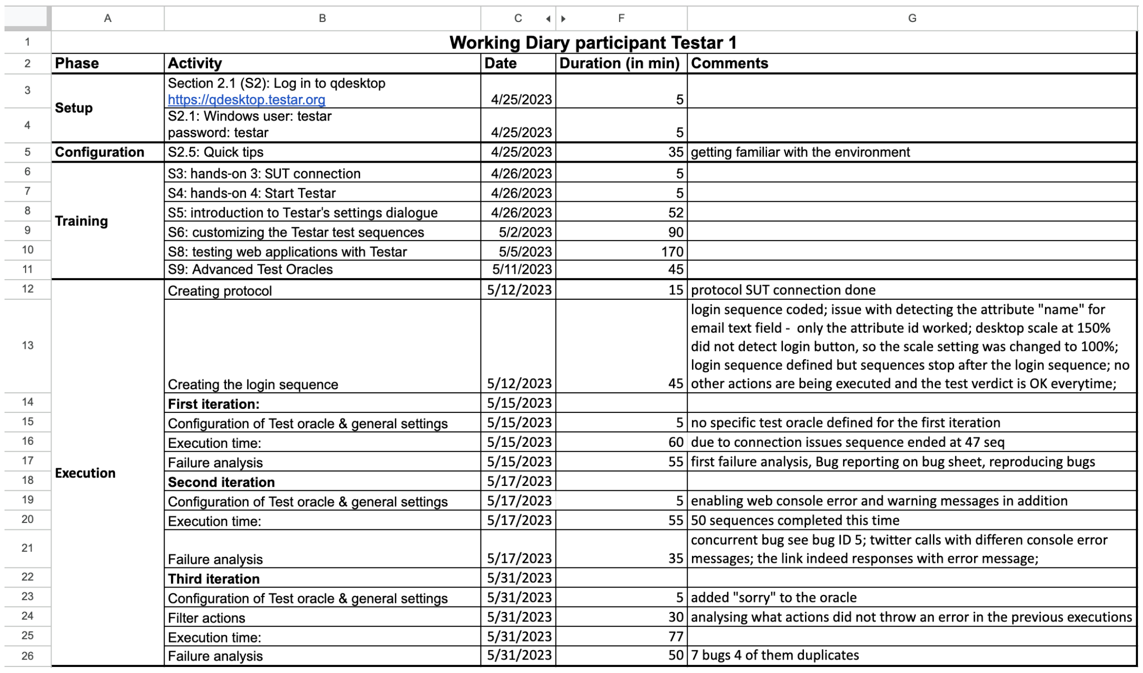
5

Figure B.4: Testar Training - Table of contents part 3

Appendix C

Working Diaries

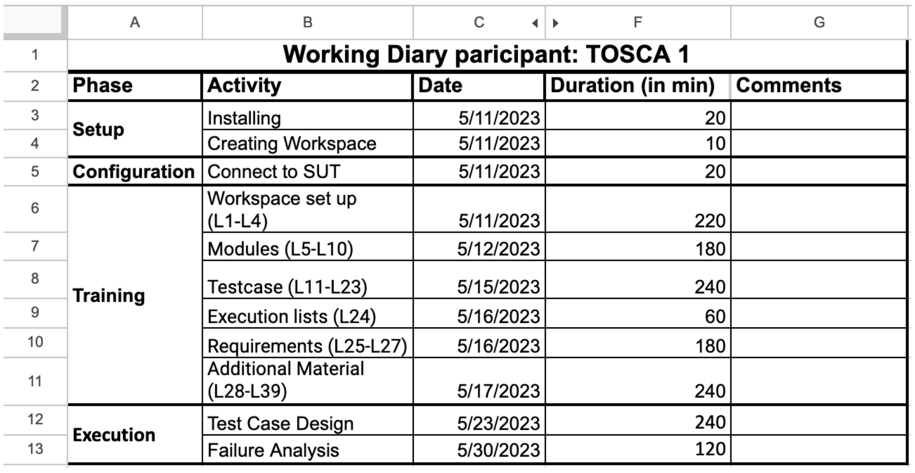
C.1 Group Testar



	A	B	C	F	G
1	**Working Diary participant Testar 1**				
2	**Phase**	**Activity**	**Date**	**Duration (in min)**	**Comments**
3	**Setup**	Section 2.1 (S2): Log in to qdesktop https://qdesktop.testar.org	4/25/2023	5	
4		S2.1: Windows user: testar password: testar	4/25/2023	5	
5	**Configuration**	S2.5: Quick tips	4/25/2023	35	getting familiar with the environment
6	**Training**	S3: hands-on 3: SUT connection	4/26/2023	5	
7		S4: hands-on 4: Start Testar	4/26/2023	5	
8		S5: introduction to Testar's settings dialogue	4/26/2023	52	
9		S6: customizing the Testar test sequences	5/2/2023	90	
10		S8: testing web applications with Testar	5/5/2023	170	
11		S9: Advanced Test Oracles	5/11/2023	45	
12	**Execution**	Creating protocol	5/12/2023	15	protocol SUT connection done
13		Creating the login sequence	5/12/2023	45	login sequence coded; issue with detecting the attribute "name" for email text field - only the attribute id worked; desktop scale at 150% did not detect login button, so the scale setting was changed to 100%; login sequence defined but sequences stop after the login sequence; no other actions are being executed and the test verdict is OK everytime;
14		**First iteration:**	5/15/2023		
15		Configuration of Test oracle & general settings	5/15/2023	5	no specific test oracle defined for the first iteration
16		Execution time:	5/15/2023	60	due to connection issues sequence ended at 47 seq
17		Failure analysis	5/15/2023	55	first failure analysis, Bug reporting on bug sheet, reproducing bugs
18		**Second iteration**	5/17/2023		
19		Configuration of Test oracle & general settings	5/17/2023	5	enabling web console error and warning messages in addition
20		Execution time:	5/17/2023	55	50 sequences completed this time
21		Failure analysis	5/17/2023	35	concurrent bug see bug ID 5; twitter calls with differen console error messages; the link indeed responses with error message;
22		**Third iteration**	5/31/2023		
23		Configuration of Test oracle & general settings	5/31/2023	5	added "sorry" to the oracle
24		Filter actions	5/31/2023	30	analysing what actions did not throw an error in the previous executions
25		Execution time:	5/31/2023	77	
26		Failure analysis	5/31/2023	50	7 bugs 4 of them duplicates

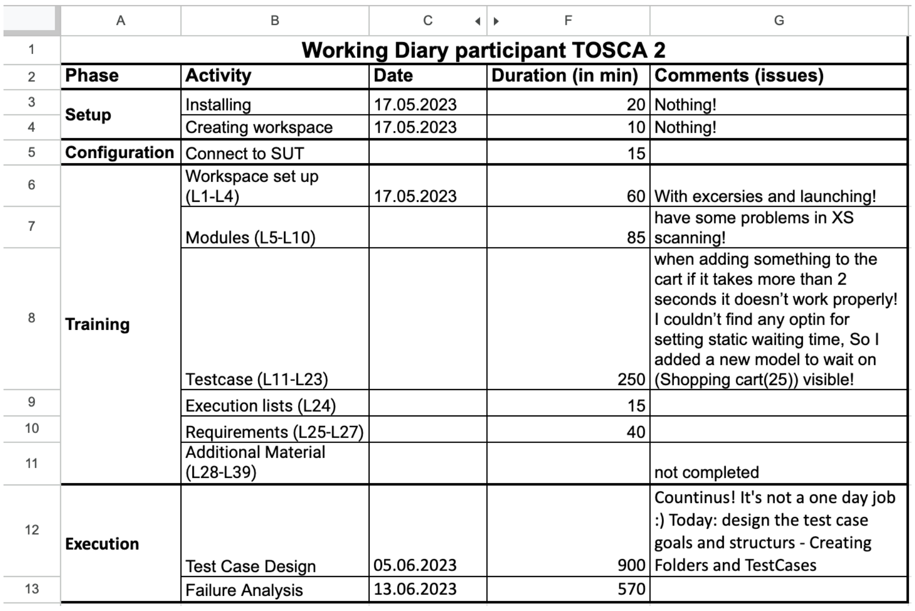
Table C.1: Working diary of participant Testar

C.2 Group Tosca



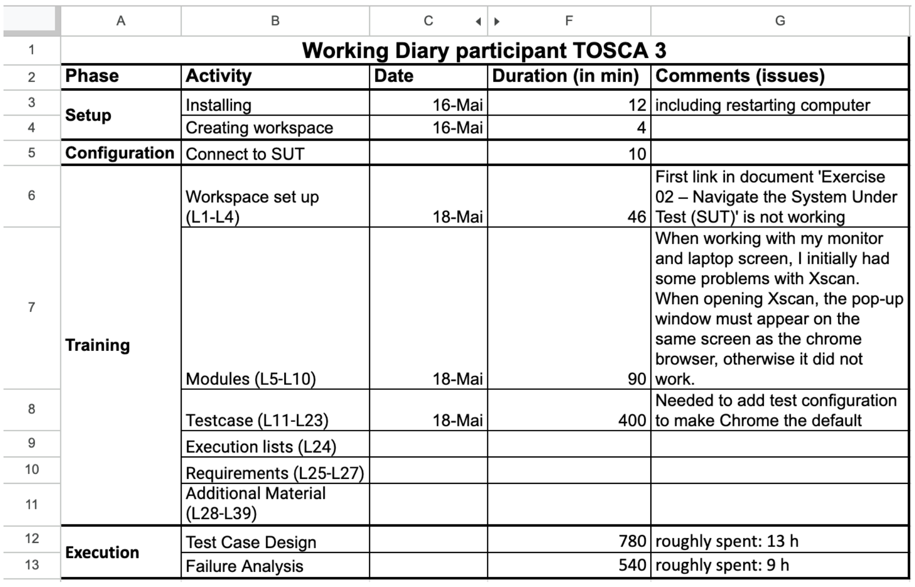
Working Diary paricipant: TOSCA 1				
Phase	**Activity**	**Date**	**Duration (in min)**	**Comments**
Setup	Installing	5/11/2023	20	
	Creating Workspace	5/11/2023	10	
Configuration	Connect to SUT	5/11/2023	20	
Training	Workspace set up (L1-L4)	5/11/2023	220	
	Modules (L5-L10)	5/12/2023	180	
	Testcase (L11-L23)	5/15/2023	240	
	Execution lists (L24)	5/16/2023	60	
	Requirements (L25-L27)	5/16/2023	180	
	Additional Material (L28-L39)	5/17/2023	240	
Execution	Test Case Design	5/23/2023	240	
	Failure Analysis	5/30/2023	120	

Table C.2: Working diary of participant Tosca 1



Working Diary participant TOSCA 2				
Phase	**Activity**	**Date**	**Duration (in min)**	**Comments (issues)**
Setup	Installing	17.05.2023	20	Nothing!
	Creating workspace	17.05.2023	10	Nothing!
Configuration	Connect to SUT		15	
Training	Workspace set up (L1-L4)	17.05.2023	60	With excersies and launching!
	Modules (L5-L10)		85	have some problems in XS scanning!
	Testcase (L11-L23)		250	when adding something to the cart if it takes more than 2 seconds it doesn't work properly! I couldn't find any optin for setting static waiting time, So I added a new model to wait on (Shopping cart(25)) visible!
	Execution lists (L24)		15	
	Requirements (L25-L27)		40	
	Additional Material (L28-L39)			not completed
Execution	Test Case Design	05.06.2023	900	Countinus! It's not a one day job :) Today: design the test case goals and structurs - Creating Folders and TestCases
	Failure Analysis	13.06.2023	570	

Table C.3: Working diary of participant Tosca 2



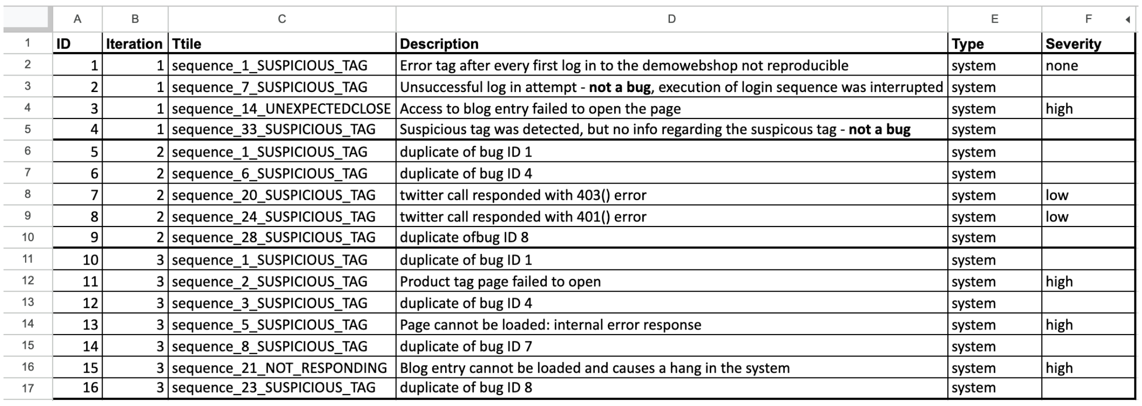
	A	B	C	F	G
1	**Working Diary participant TOSCA 3**				
2	**Phase**	**Activity**	**Date**	**Duration (in min)**	**Comments (issues)**
3	**Setup**	Installing	16-Mai	12	including restarting computer
4		Creating workspace	16-Mai	4	
5	**Configuration**	Connect to SUT		10	
6	**Training**	Workspace set up (L1-L4)	18-Mai	46	First link in document 'Exercise 02 – Navigate the System Under Test (SUT)' is not working
7		Modules (L5-L10)	18-Mai	90	When working with my monitor and laptop screen, I initially had some problems with Xscan. When opening Xscan, the pop-up window must appear on the same screen as the chrome browser, otherwise it did not work.
8		Testcase (L11-L23)	18-Mai	400	Needed to add test configuration to make Chrome the default
9		Execution lists (L24)			
10		Requirements (L25-L27)			
11		Additional Material (L28-L39)			
12	**Execution**	Test Case Design		780	roughly spent: 13 h
13		Failure Analysis		540	roughly spent: 9 h

Table C.4: Working diary of participant Tosca 3

Appendix D

Bug Sheets

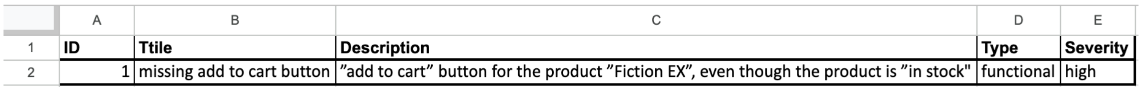
D.1 Group Testar



	A	B	C	D	E	F
1	**ID**	**Iteration**	**Ttile**	**Description**	**Type**	**Severity**
2	1	1	sequence_1_SUSPICIOUS_TAG	Error tag after every first log in to the demowebshop not reproducible	system	none
3	2	1	sequence_7_SUSPICIOUS_TAG	Unsuccessful log in attempt - **not a bug**, execution of login sequence was interrupted	system	
4	3	1	sequence_14_UNEXPECTEDCLOSE	Access to blog entry failed to open the page	system	high
5	4	1	sequence_33_SUSPICIOUS_TAG	Suspicious tag was detected, but no info regarding the suspicous tag - **not a bug**	system	
6	5	2	sequence_1_SUSPICIOUS_TAG	duplicate of bug ID 1	system	
7	6	2	sequence_6_SUSPICIOUS_TAG	duplicate of bug ID 4	system	
8	7	2	sequence_20_SUSPICIOUS_TAG	twitter call responded with 403() error	system	low
9	8	2	sequence_24_SUSPICIOUS_TAG	twitter call responded with 401() error	system	low
10	9	2	sequence_28_SUSPICIOUS_TAG	duplicate ofbug ID 8	system	
11	10	3	sequence_1_SUSPICIOUS_TAG	duplicate of bug ID 1	system	
12	11	3	sequence_2_SUSPICIOUS_TAG	Product tag page failed to open	system	high
13	12	3	sequence_3_SUSPICIOUS_TAG	duplicate of bug ID 4	system	
14	13	3	sequence_5_SUSPICIOUS_TAG	Page cannot be loaded: internal error response	system	high
15	14	3	sequence_8_SUSPICIOUS_TAG	duplicate of bug ID 7	system	
16	15	3	sequence_21_NOT_RESPONDING	Blog entry cannot be loaded and causes a hang in the system	system	high
17	16	3	sequence_23_SUSPICIOUS_TAG	duplicate of bug ID 8	system	

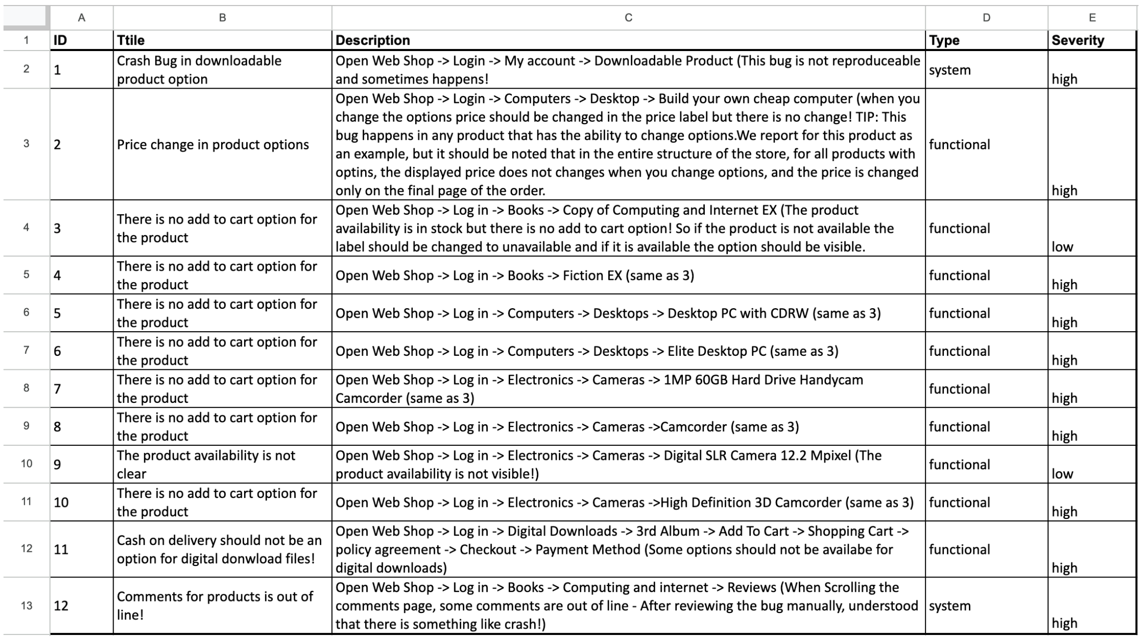
Table D.1: Bug Sheet of Group Testar

D.2 Group Tosca



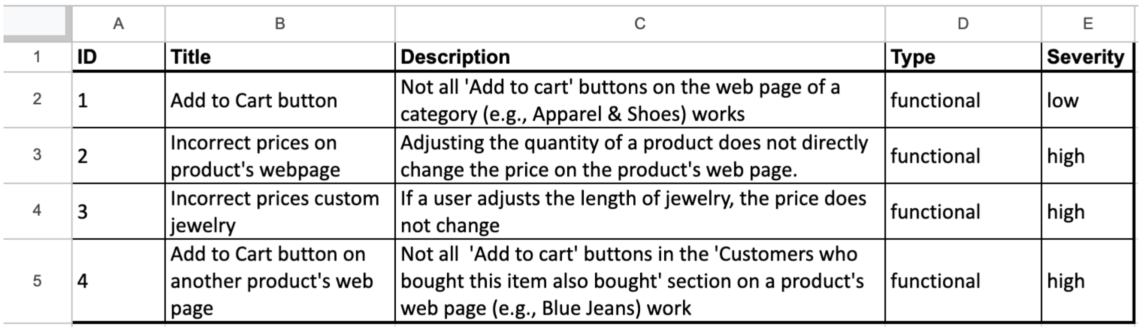
	A	B	C	D	E
1	**ID**	**Ttile**	**Description**	**Type**	**Severity**
2	1	missing add to cart button	”add to cart” button for the product ”Fiction EX”, even though the product is ”in stock"	functional	high

Table D.2: Bug Sheet of participant Tosca 1



ID	Ttile	Description	Type	Severity
1	Crash Bug in downloadable product option	Open Web Shop -> Login -> My account -> Downloadable Product (This bug is not reproduceable and sometimes happens!	system	high
2	Price change in product options	Open Web Shop -> Login -> Computers -> Desktop -> Build your own cheap computer (when you change the options price should be changed in the price label but there is no change! TIP: This bug happens in any product that has the ability to change options.We report for this product as an example, but it should be noted that in the entire structure of the store, for all products with optins, the displayed price does not changes when you change options, and the price is changed only on the final page of the order.	functional	high
3	There is no add to cart option for the product	Open Web Shop -> Log in -> Books -> Copy of Computing and Internet EX (The product availability is in stock but there is no add to cart option! So if the product is not available the label should be changed to unavailable and if it is available the option should be visible.	functional	low
4	There is no add to cart option for the product	Open Web Shop -> Log in -> Books -> Fiction EX (same as 3)	functional	high
5	There is no add to cart option for the product	Open Web Shop -> Log in -> Computers -> Desktops -> Desktop PC with CDRW (same as 3)	functional	high
6	There is no add to cart option for the product	Open Web Shop -> Log in -> Computers -> Desktops -> Elite Desktop PC (same as 3)	functional	high
7	There is no add to cart option for the product	Open Web Shop -> Log in -> Electronics -> Cameras -> 1MP 60GB Hard Drive Handycam Camcorder (same as 3)	functional	high
8	There is no add to cart option for the product	Open Web Shop -> Log in -> Electronics -> Cameras ->Camcorder (same as 3)	functional	high
9	The product availability is not clear	Open Web Shop -> Log in -> Electronics -> Cameras -> Digital SLR Camera 12.2 Mpixel (The product availability is not visible!)	functional	low
10	There is no add to cart option for the product	Open Web Shop -> Log in -> Electronics -> Cameras ->High Definition 3D Camcorder (same as 3)	functional	high
11	Cash on delivery should not be an option for digital donwload files!	Open Web Shop -> Log in -> Digital Downloads -> 3rd Album -> Add To Cart -> Shopping Cart -> policy agreement -> Checkout -> Payment Method (Some options should not be availabe for digital downloads)	functional	high
12	Comments for products is out of line!	Open Web Shop -> Log in -> Books -> Computing and internet -> Reviews (When Scrolling the comments page, some comments are out of line - After reviewing the bug manually, understood that there is something like crash!)	system	high

Table D.3: Bug Sheet of participant Tosca 2



ID	Title	Description	Type	Severity
1	Add to Cart button	Not all 'Add to cart' buttons on the web page of a category (e.g., Apparel & Shoes) works	functional	low
2	Incorrect prices on product's webpage	Adjusting the quantity of a product does not directly change the price on the product's web page.	functional	high
3	Incorrect prices custom jewelry	If a user adjusts the length of jewelry, the price does not change	functional	high
4	Add to Cart button on another product's web page	Not all 'Add to cart' buttons in the 'Customers who bought this item also bought' section on a product's web page (e.g., Blue Jeans) work	functional	high

Table D.4: Bug Sheet of participant Tosca 3