CHALMERS

A system for semi-automated testing in industrial practice

Master of Science thesis in Software Engineering

Duy Huynh
Stefan Spasov

Chalmers University of Technology
University of Gothenburg
Department of Computer Science & Engineering
Gothenburg, Sweden 2015

Duy Huynh, June 2015
Stefan Spasov, June 2015

Supervisors: PhD Emil Alégroth, Dr Richard Berntsson Svensson
Examiner: Matthias Tichy

Chalmers University of Technology
University of Gothenburg
Department of Computer Science & Engineering
SE-412 96 Gothenburg
Sweden
Telephone + 46 (0)31-772 1000

Acknowledgements

First and foremost, we would like to gratefully and sincerely thank PhD Emil Alégroth and Dr Richard Berntsson Svensson for their invaluable feedback and dedicated guidance. We would also like to extend the gratitude to our industrial supervisor, Fredrik Sjöberg, who helped bring this research project to fruition.

The Authors, Gothenburg 15/06/2015

Abstract

In the recent years the complexity of software products has increased tremendously. Many software companies strive to achieve higher quality software with shorter delivery time. Today software testing is used by the companies to increase the quality of their products and to ensure that the client requirements are met. In the industrial practice, verification and validation activities are often performed with pricey, laborious and error-prone manual test practices. Developers are usually under the pressure to release faster, thus, quality often becomes hindered. A possible solution to this problem is test automation, but the available tools and techniques bear a lack of generic applicability. One way to mitigate these problems is to unify various test methods and practices. This thesis presents a design science research, which explores the challenges of unification. Additionally, it tries to find a solution to the problems that these challenges present by developing a system for semi-automated testing (SeAT). The thesis also analyzes the effects of SeAT on the testing processes at a case company.

Keywords: manual testing; automated testing; software; unification challenges

Contents

List of Figures

List of Tables

1

Introduction

Many software companies today use software testing to increase the quality of their products and to ensure that the client requirements are met. Verification and validation in industrial practice are often performed with costly, tedious and error-prone manual test practices. Under the pressure to release faster, quality often becomes hampered. Test automation has been proposed as a solution, but the available tools and techniques experience a lack of generic applicability.

Furthermore, efficient automation requires alignment between verification and validation activities, which is poorly supported in the current state-of-practice solutions, where various test technologies are being used. Some examples for technologies which could be used together are unit testing frameworks, visual GUI testing, in-house developed solutions. This diversity makes it difficult for the testers to switch from one technology to another when they conduct test cases consisting of several test technologies. These insights reveal the need for an unified test approach.

On one hand is the unification between manual and automated testing. Manual testing is the process of manually checking the system for defects. In some test cases which require many repetitive steps automation could be applied to control the execution. In practice manual and automated testing techniques are being unified, meaning that a test case is designed to contain steps executed by a person and steps executed by a machine.

In addition, testing on different abstraction levels could be unified. In the Software Engineering Body of Knowledge [2] four main methods of testing are identified and grouped by the test target: unit, integration, system, and acceptance testing. The different views on test methods bear their own advantages and disadvantages. A combination could potentially provide a better overall view on the defects in a system and the system's compliance with its requirement specification. Thus, to obtain the advantages and ultimately mitigate the potential drawbacks it is reasonable to take a hybrid approach for testing the system on various levels of test target abstraction.

In the different domains there are reasons to use diversified test technologies. Therefore unification is encouraged so that the maximum balance between cost, time and errors found is achieved. For instance, in the safety-critical systems domain using only manual testing is not a proper choice because of the high level of human factor. Nevertheless, an enormous part of the testing in industries is done in a manual manner, e.g. scenario-based manual testing [3], exploratory testing [4]. Currently automation is not applied because of its high cost and its time consumption.

There are some domains in which the system under test involves, not only software components, but actual hardware units. In these cases, manual testing could be automated partially, but the test cases involving hardware components are not feasibly automated. The solution is designing test cases, consisting of manual and automated testing all together. However, this solution implies that the testers would have to synchronize their work with the execution of the scripts and that they have the required knowledge to operate the needed software tools for automation. In addition, time is wasted for starting the test framework, finding the correct script and checking the result after its execution.

Similar to the case of safety-critical domains is the case with domains containing distributed systems. Due to the distributed nature of a system, processes inside this system happen asynchronously and the sequential ordering of events is often unknown. In such scenarios, output data on different devices should be evaluated to ensure the success of testing activities. Therefore, test automation could be sometimes tedious and a human oracle could be needed. This fact makes the mixture between manual and automated test approaches and the related complications a requirement.

Furthermore, in some cases full automation is applicable, but still the potential benefits of several test tools are desirable. For instance, the power of visual GUI testing [5] on system level could be combined with stimulating lower level components through their external interfaces or observing the output of such. This creates a very versatile and not so costly testing technique. Nevertheless, this technique comes with the drawback that orchestration during execution is needed, in the form of starting the correct test tool at the right time, running scripts consecutively, observing outputs and registering results. This orchestration is time consuming and currently is performed by testers with the right level of knowledge.

The identified problem in the context of a legacy distributed system is the integration of new automation test technologies. The software market is continuously being updated with different test framework with added functionality. Companies would prefer to stay up to date with the newest test software, thus harvesting its benefits. Consequently, a framework or approach for unification should be able to easily integrate new test automation tools with yet unknown characteristics.

The aim of this thesis is to further explore the challenges of unifying different test technologies and techniques, through a design science research at company X. Additionally, it tries to find a solution to the problems that these challenges present by developing a system for semi-automated testing (SeAT). The thesis also analyzes the effects of the solution to the industrial practice.

The remainder of this report is organized as follows: in the next section, the background and the related studies are presented. In chapter 3, a description of the case company where this study was conducted is presented along with the current test practices in that company. In chapter 4, the research objectives and the methodology for data collection and analyses are introduced. In chapter 5 the implemented system for semi-automated testing (SeAT) is described in details. Chapter 6 shows the results of the data collection followed by a discussion of the results in chapter 7. This report ends with a conclusion and possible future work (chapter 8).

2

Background and Related Work

The purpose of this chapter is to provide the background of this study and a review of the related literature. First, the background section is introduced, where different viewpoints on the process of testing are discussed, followed by a related work section showing the advantages and disadvantages of these technologies as analysed by other researchers.

2.1 Background

Software testing is an analysis performed to assess the quality of the system under test [6]. There are various test techniques with the intent of reassuring that a component or a system under test meet their requirement specification, behave properly to plenty of inputs and achieve the results desired by the stakeholders. Various strategies for software testing are used to select different test techniques. That is because the amount of possible tests are infinite and time and resources are not. The models of software development regulate the time and the techniques for testing. In the more traditional Waterfall model [7] testing appear only after the system is designed and implemented completely. In comparison, if an agile approach is used during the development of the system, the testing phase is conducted concurrently with the implementation phase.

In this chapter various methods for testing are further explained. They can be grouped by different criteria in several groups and one method can belong to one or many groups. Each group has its positive sides and potential drawbacks. In many cases, these groups are complementary and only by combining them can the testers address their drawbacks as well as thoroughly verify and validate the system.

Firstly, taking into consideration the point of view of the test engineer who designs the test cases, software testing methods are divided into three groups: white-box, black-box and grey-box testing [8]. On the one hand, designing a white-box test case requires internal perspective of the system. On the other hand, black-box testing assesses the

functionality of the system under test without requiring any knowledge of the internal implementation. Grey-box testing is a combination of white- and black-box where knowledge of internals of a program is used to design test cases, while the execution of tests is at the user level.

Additionally test methods could be divided by their level of abstraction of the test target. On the lowest level is unit testing [9] which assures that each part of the software fulfils its designation correctly in isolation. These tests are usually written by the developers themselves before or after writing the code. White-box testing most commonly relates to the unit level of abstraction due to its detailed nature. On the second level is integration testing, where software components are combined and tested together as a group. Its primary goal is to detect defects during interactions of individual components. Consecutively, when the integrated system is tested from end to end to assure that it meets its requirements, system tests are conducted. The difference between them and the following acceptance tests is that the latter are performed on the client site where the system is tested in the environment where it will operate. The test cases differ, too, since test cases for acceptance tests should involve only steps which are relevant and important from the customer's viewpoint.

The levels of abstraction of the tests are closely related to the independent procedures that they are involved in — either validation or verification. The validation procedure is to assure that a system meets the needs of the stakeholders. On the contrary, the verification procedure evaluates whether a system complies with its requirement specification. The lower-level test techniques are part of the verification activities while the upper-level are part of the validation.

Moreover, on the unit level of abstraction exists a form of software testing that does not require the system to be running. To be specific, that is static testing, during which either the developer who wrote the code checks for sanity of the code, the documentation and the used algorithms, or experts conduct code reviews, inspections or walkthroughs. In opposition to static testing is dynamic testing in which the software must actually be compiled and run. It requires interacting with the system, providing input values and analysing if the output is as expected.

Test techniques could also be divided if they use support tools, scripts or other software during the execution or not. Manual testing is referred in cases that the test cases are executed without any assistance of automation software. The tester follows a written test plan and goes through different test cases to warrant that all of the features of an application have correct behaviours. This process can be automated which means that special software is used to control the execution of the test cases and compare the actual results with predicted ones [10]. All of the previously mentioned test methods could be either manual or automated. For instance even static testing can be automated. In that case there is a static test suite consisting of programs to be analyzed by an interpreter or a compiler that asserts the program's syntactic validity.

All of these different views of the test techniques could be used to extract various advantages from the combination between them. Most of the companies are currently using diverse methods to assure the quality and the functionality of the developed sys-

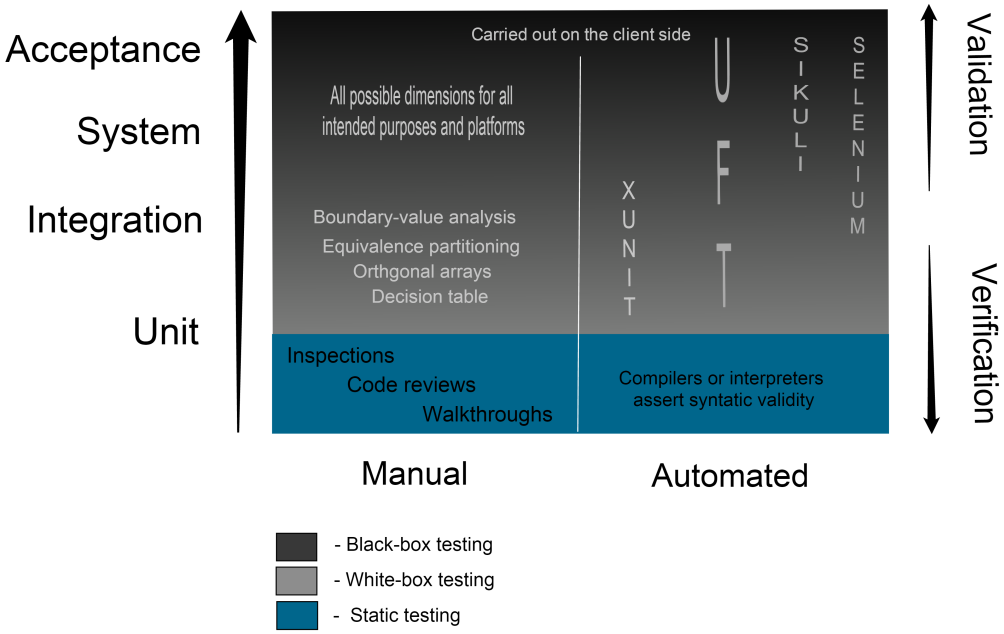

Figure 2.1: Different views on testing

tems. Unification is the process of combining different test techniques in order to take advantage of their strong sides and lower the impact of their drawbacks on the testing process. There are many related benefits to unification. Mainly it could shrink the learning curve required to test both user interface and application logic and ultimately connect verification and validation activities from end to end. Testing teams usually lack the software infrastructure that could guide their efforts. Thus, testers need a unification tool to allow them to efficiently pick between existing and quickly integrate new test technologies for efficient verification and validation of requirement conformance. Furthermore unification eases ongoing test asset maintenance and encourages test asset reuse rather than duplication of effort. On top of that, simplification of the process of test automation is needed to leverage existing investment in manual testing.

2.2 Related Work

Testing is the most widespread validation and verification technique for assuring adequate software behavior. Although it is widely used in industry it is largely ad-hoc and expensive. Bertolino [11] states that software testing involves a lot of activities, conducted in different manner and aiming at different goals. Hence, software testing faces a number of challenges. In Bertolino's work a roadmap of the challenges which are common in industry is created. While the challenges identified in Bertolino's paper should be taken into consideration by the companies they are not clearly focused on the

problem of unifying different test activities (manual scenario-based testing, unit testing, Visual GUI Testing) which is the main topic of this study.

Certainly, to be able to conclude what are the most efficient combinations of test technologies, their positive and negative sides have to be known. In fact, the research in the area of software testing has investigated the differences between contradicting groups of test methods. A research conducted by Khan [12] makes a comparison between black–, white– and grey–box testing. He concludes that white–box testing forces the tester to reason carefully about implementation and it helps in removing extra lines of code, which can bring in hidden defects. However, it is very expensive as it requires a skilled tester to perform it. Besides, this approach misses cases that are in the scope of the system's requirement specification but are not covered by the code. In contrast, black-box testing examines the fundamental aspects of the system and it is much more effective on larger units of code. The testers need no knowledge of implementation, including specific programming languages and they can be non-technical people. Above all, black-box testing helps expose any ambiguities or inconsistencies within the specifications and therefore black-box test cases could be designed right after the requirements for the system are elicited. Although this approach does not provide reasons for a failure when one is found and may leave many program paths untested. Unifying black- and white-box testing has proven to be efficient as shown in Khan's research. Namely that is grey-box testing and is provenly useful for designing excellent test scenarios.

Moreover, knowledge of the advantages and disadvantages of the methods, which are lying within different levels of abstraction of the test target, is potentially beneficial for their unification. On the one hand, unit testing ensures that the small individual components of a system are verified in isolation and they do something the way they are supposed to do it without any defects [13]. Therefore unit testing should be done in conjunction with other test methods, as they can only show the presence or absence of particular errors but they cannot connect these errors to their effect on the functionality of the system. In addition, unit tests are referred as a complementary documentation for developers who are trying to understand what functionality is provided by a specific unit [14]. While unit testing has its benefits, its implementation requires available experts with knowledge of the code and that is considered as a concern because of the time and cost involved in it. Automation support for unit testing could lower the time that testers are involved in this stage and hence lower the cost. The research of Singh [15] compares the available automation tools and categorizes them so that the testers could easily choose an appropriate unit testing tool.

On the other hand, system testing is the best at evaluating the system's compliance with its requirement specification. Integration, release and acceptance testing can be viewed as subcategories of system testing [13]. In fact, the integration testing's priority is to find defects while the system testing's priority is to validate that the system meets its requirements. However, in practice, the processes of validation testing and defect testing are interleaved. Release testing is usually black-box testing and it assures that the system does what it is supposed to do. As for acceptance tests they can be viewed as release tests in which the customers are involved and the focus is on functionality,

usability and performance [13].

In practice, tools have been developed to automate the system tests of software through different interfaces. There are two main approaches at the GUI level the first of which is Record and Replay (R&R), where user interaction is recorded via direct references to the GUI components, or with the use of the components' coordinates. A bright example of R&R is the Selenium test automation tool [16, 17, 18]. The second approach is visual GUI testing (VGT), which is similar to R&R, but it uses image recognition. For instance, Sikuli is a tool using this approach. In recent research Alégroth and Feldt [19] have proven that Visual GUI testing (VGT) could be applied in the industry but that there are several problems that have to be solved first. Additionally, systems could be tested based on the component interfaces which they provide. External software could connect to these interfaces to check the inputs or the outputs. Usually this software requires internal knowledge of the system and it is developed "in-house".

Altogether, system testing displays which parts of the functionality of the system do not meet their specification but does not pinpoint where the errors are in the code, conversely to unit testing. Although the two strategies are used in complementarity in practice, the tools for testing are usually different, leading to decreased productivity and reliability of the testing process.

In like manner, there are three considerable benefits of static analysis according to [13]. Firstly, during dynamic testing errors could be easily hidden or masked. That is because once an error is found, the tester cannot be sure if other observed anomalies are side effects of the same error. Static inspections mitigate the concern of interactions between errors. Secondly, during static analysis there is no need for the whole system and all of its parts to be developed, consequently removing the need for development of specialised stub subsystems, which are needed for a subsystem to be run and tested. The most important advantage of static analysis is that it allows the tester to look for inappropriate algorithms and poor programming style that could make the system difficult to maintain and update. Static analysis are actually a very old idea. In a report from 1986 Fagan [20] shows that more than 60% of the defects in a system could be found using informal inspections. There are two studies [21, 22] which conclude that static code reviewing is cheaper and more effective than dynamic testing. However, static techniques take time to arrange and they seem to hinder the development process and that is why it is difficult to implement formal static reviewing in the industry. There are automation tools for static analysis [13] which parse the source code of a program and detect incorrect statements and check the control flow of a program. Static analysers are very effective when there are weaknesses in the design of the programming language which was used. However, some of the more modern programming languages have less error-prone design which minimizes the effects of external static analysers.

As mentioned in [13], both static and dynamic testing have benefits and drawbacks and should be used together in the verification and validation process. A good example, as suggested by Gilb and Graham [22], is to review the test cases for a system so that problems with the testing could be discovered early in the testing process. The best practice is to perform static analysis early in the implementation process of a system and

later, when the system has already been integrated, to validate it against its requirement specifications.

Another research conducted by Leitner [10] has shown a positive result related to the unification of manual and automated testing (being unit testing in that case). The two strategies are seemingly contrasting, but as the research shows they are complementary in a way that each has flaws which the other outdoes. The disadvantage of manual testing is that it allows testers to perform only sequential test runs, hence increasing the time needed to complete the tests [23]. On the other hand, an important advantage related to this approach is that developers can use their reasoning to develop useful test cases and construct complicated input data. As for automated testing, it can potentially save time as it provides the testers the capability to perform parallel testing. Additionally, automated tests are not dependent on testers' working hours and they could run constantly [23]. Due to automation testing's efficient nature, it is implied that test coverage is increased, since there is more time for testing. Moreover, automated testing tools are useful during test case execution, test case generation and test result verification [10].

Paradoxically, one of the disadvantages of this approach is the lack of human input in cases where human reasoning is required, for instance in the manual selection of use-cases, which is believed to find additional bugs. Moreover, developing and maintaining automated tests is challenging and expensive in projects which are prone to change. In summary, the strong side of the manual method is to cover test cases in details but it is not efficient for extensive coverage, whereas automation can potentially execute a huge number of cases in brief time. Leitner's research [10] resulted in a test tool which combines the advantages of both strategies in a unified fashion. However, the findings have been limited to two types of testing, and little has been discovered about the consequences of unifying different testing activities, especially when they are in different abstraction levels of the test target.

To conclude with, there are advantages and disadvantages related to all of the testing techniques and different combinations amongst them is crucial for the successful testing of a system. Consequently, there are many issues bound to the mixed testing activities. Therefore, the intent of this thesis is to further investigate the issues by creating a framework on which different test technologies can be executed together in order to benefit from their strengths.

3

Case Company and System

The study was conducted with the collaboration of a large international security organization which has more than 10,000 employees and has operations on all continents. The site where the research was conducted was a department of this organization with approximately 80 employees.

The system which was chosen to be the unit of analysis is an Air Traffic Control (ATC) system. The ATC is a distributed safety-critical system which contains various components built upon different technologies. Such system needs to maintain a high level of availability and to be able to recover within a second without any information losses if it was to encounter a failure. Data transmission amongst subsystems or between the ATC and external services has to be guaranteed with correctness and precision in real time. Moreover, any detected failures have to be reported to the ATC officers. In this particular system, the absence of data is worse than the invalidity of data.

Undeniably, testing is an essential part in the development process of any safety-critical systems. Unit testing, integration testing and system testing are mandatory. In this particular case, in order to assist the testing process, there are specialized applications which are being used as simulators, for instance flight plan simulator, radar simulator or weather simulator. These simulators work on component or system level of abstraction of the test target. The open source framework NUnit is being used for unit and integration testing while Sikuli is being used for visual GUI testing. All of these co-existing technologies form an ecosystem within the current test environment of the ATC system. On top of that, extensive acceptance test still needs to be performed to meet the quality assurance standard ED–153 — a set of software safety assurance guidelines to be implemented by air navigation service providers to conform with the commission regulation EC 482 of the European communities.

Lastly, as adhered to the attributes of the aviation and avionics industry, the development cycle spreads in a period of years to decades. This fact results in a huge amount of legacy code. Additionally, the co-existence of several branches specially built

for different customer sites adds up to the management burden of software source code and tests. As the project evolves, regression testing takes longer and longer, and only by introducing more automation in testing can the verification and validation activities be focused on new features and be more efficient.

4

Research Approach

This chapter describes the research approach based on which the *design science research* [1] was conducted. It starts with describing research objectives (section 4.1) and explaining the design science research methodology which was used as well as providing arguments about why such methodology was chosen (section 4.2). It, then, continues with the research procedures (section 4.3) and based on that, threats to validity are discussed (section 4.4).

4.1 Research Objectives

The main purpose of the study is to investigate if the developed system for unifying test technologies is more efficient than the current processes and practices.

The following research questions were addressed:

- RQ1 — What challenges are experienced in practice when different test technologies and practices are used together for system verification and validation?
- RQ2 — How could different test technologies and practices be unified in a framework?
- RQ3 — How does the unified test framework affect the verification and validation activities in industrial practice?

4.2 Research Methodology

This thesis reports on a six-month (January 2014 – June 2014) study at the case company. The thesis was conducted following the design science research framework which was proposed by Hevner in 2010 [1]. Design science is fundamentally a problem-solving paradigm [1], which provides analytical techniques and perspectives for performing and

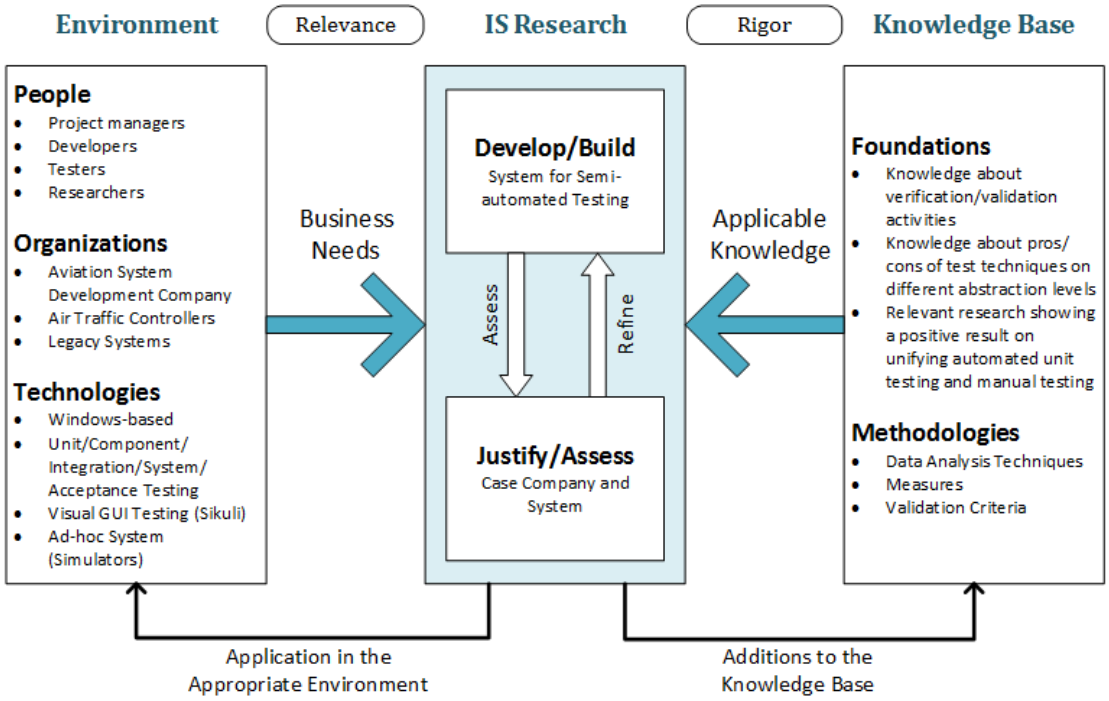

Figure 4.1: Methodology overview using Hevner's framework [1]

research in information systems. It centers around the innovative creations of IT artifacts upon which the usages, performance and efficiency of new ideas, practices and technical capabilities applied to information systems are evaluated. IT artifacts are broadly defined as constructs (vocabulary and symbols), models (abstractions and representations), methods (algorithms and practices), and instantiations (implemented and prototype systems) [1].

As the intention of the study is to develop a system for unification of testing technologies and practices, the design science methodology is highly applicable. Furthermore, the qualitative approach was chosen since its nature is to examine in-depth "purposive samples" to better understand a phenomenon [24]. Subsequently, the findings can be further validated using quantitative research.

An overview of the thesis methodology is illustrated in Figure 4.1. This model was constructed based on Hevner's framework for design science research [1]. Details about **Environment** including people, organizations and technologies are discussed in Case Company and System (chapter 3), whereas **Knowledge Base** is indicated in Background and Related Work (chapter 2) and in Data Analysis section of this chapter. The implementation of SeAT, which is the IT artifact, is discussed in Technical Implementation (chapter 5). The challenges of unification of test practices identified in this research are considered as the **Business Needs** and solving them would contribute to the companies to improve their testing process. The addition of this study to the **Knowledge**

Base is presented in the Discussion (chapter 7).

4.3 Research Procedures

This section describes the research approach used to answer each of the questions stated in the objectives. The main activities include the exploration of challenges in practice (subsection 4.3.1), the development of SeAT (subsection 4.3.2), assessment of the effects of SeAT on the validation and verification process (subsection 4.3.3) and the analysis of the collected data (subsection 4.3.4).

4.3.1 Challenges Exploration

Based on the challenges, which are mentioned in chapter 1, initial ideas about a system for semi-automated testing have been created. However, more input from the studied context was needed to elicit essential business needs for such a system. Therefore, a thorough exploration of the demands and challenges which the organization encountered, having both manual and automated tests scattered amongst different abstraction levels, was performed.

According to Stake [25] triangulation is needed to increase the validity of the research and to make it more precise. This means that several perspectives should be taken into consideration to get a broader picture of the challenges in testing through multiple activities. Data or source triangulation [25], is the collection of data from different sources. It was used in the exploration of the challenges, consisting of the following steps: observing testers who were conducting verification activities, building knowledge on the unit of analysis by reading source code and related documentation, eliciting functional and quality requirements of the system by unstructured interviews with various stakeholders.

Observation: In order to make sense of the difficulties the testers confronted while performing verification activities, an observation [26] was conducted during a major system test before a delivery of the system under analysis. Because people are not always aware of what they really do and how they do it, observing what is being done is one way around this mental blindness, and vastly improves the knowledge of the current work and some of the associated work problems [26]. In the beginning of the research, to be able to quickly familiarize themselves with the system, the authors used a variant of interviewing and observation, called task demonstration. A tester was asked to explain the system and to demonstrate how to perform a specific task. During the test execution, the tester was told to think aloud what has to be done, why and how it could be done. By using this approach, some usability problems were detected, too. For instance, why did it take longer time for some particular tasks or why certain mistakes were made during the execution. One of the objectives of this research is to find a way for testers to effectively switch amongst testing technologies without considerable hindrance. Hence, usability is definitely one of the quality criteria needed for SeAT.

Document study: The fact that technicality played a major role in the development of SeAT made document study [26] an important step in the process. Furthermore,

document study is a way to cross-check the interview and observation information [26]. There were two documents under investigation, which were the System/Subsystem Specification (SSS) and the Acceptance Test Description (ATD). Through this process, basic information about which framework SeAT should be built on and which technologies it should support were identified. However, such findings were just a starting point for developing the system. In order to increase the generalizability, common testing technologies being used in practice ware also investigated and considered to be incorporated into SeAT.

Unstructured and semi-structured interviews: Being in the exploration phase, unstructured and semi-structured interviews [26] allowed new ideas to be brought up as a result of what the interviewees said, thus, increased the amount of valuable information elicited. Simultaneously with the aforementioned activities, unstructured interviews were conducted with a variety of stakeholders to ensure that different aspects of the demands and challenges were correctly addressed and validated based on different viewpoints. Such stakeholders included developers, project managers, testers, university researchers and students. The authors decided not to limit the study within the context of the organization. Hence, not only industrial experts have been interviewed but also university researchers and students, which led to increase of the external validity. The methodology for semi-structured interviewing and the used subjects are the same as in (subsection 4.3.3), where they are described in detail.

One of the outcomes of the exploration were the challenges being experienced in practice when different test technologies and practices are used together for system verification and validation, resulting to be the answer for the first research question (RQ1).

4.3.2 SeAT Development

SeAT is the main information system artifact from Hevner's model for design science research [1] and it played a featured role in this study. The information elicited from the previous phase in combination with the theory foundation formed a list of functionalities and quality attributes which should be supported by SeAT. Since SeAT is a framework for unifying test technologies, it has to be general and flexible enough for future extensions when it might need to support unforeseen technologies with minimum effort to customize. Being aware of the importance of such quality attributes, the authors decided to construct a simplified version of a quality model following the guidelines provided by the ISO standard [27]. The simplified quality model addresses the basic quality attributes SeAT needed to conform to and lists the metrics for defining how to gauge these quality attributes in order to validate the system after it has been developed. Another work product during this phase was an initial architectural design for SeAT which would be revised as the system evolves.

Inspired by the Agile methodologies, the authors developed SeAT iteratively and incrementally. The very reason for an incremental and iterative strategy is to allow for people's inevitable mistakes to be discovered relatively early and fixed in a tidy manner [28]. No matter how good the requirements elicitation is, there are always

gaps between each pair of consecutive stages where information is being transferred and transformed, e.g. what the real demands are and what the stakeholders perceive, what they understand and what they describe, what is documented in the specification and what is developed. The idea was to have, as fast as possible, a working prototype of the system which could be used to validate the ideas with the stakeholders. Based on that, necessary modifications could be made incrementally. The approach and design which were chosen to implement SeAT were the answer to the second research question (RQ2).

4.3.3 Effects of SeAT

In order to answer the main research question (RQ3), which was to investigate the influences of unifying test technologies on industrial verification and validation activities, semi-structured interviews [29] were used as a method of data collection. This type of interviews was suitable because it provides space for the researcher to explore and to investigate further topics, related to the main subject. The main topics were thought in advance and an interview protocol was created, containing the predefined questions. Following this protocol strictly was not a necessity [29], although it was very helpful. The authors used the the funnel model for creating the interview protocol [30], which begins with open questions and moves toward more specific questions. The questions were not asked in the same order and the interviewers often influenced the interview to make a flowing conversation. This approach helped them explore issues that have not been thought in advance [30].

The subjects were chosen from industry and from the academia to ensure a better distribution of viewpoints. Additionally, employees with different roles and experience were chosen. SeAT was installed in laboratory workstations at the company. These workstations replicated the ATC system under test which was used in reality by the testers. To show the wide range of applicability of the framework not only the ATC system was used during the interviews. A simple web-site with relatively complex graphical interface was used as system under test during the interviews with the students. Due to non-disclosure agreement with the case company their system could not be shown to the students.

The selection criterion for the test cases, which were used during the interviews, was the complexity of the test cases. According to this criterion the example test cases were categorized in four categories. The first category are test cases in which all steps could be automated using only one tool for automation. In this case - Sikuli. The second category are test cases, which have steps that could be automated using different automation tools, namely simulators and Sikuli. The third category are test cases with steps that cannot be fully automated and require a person to perform some of them. The fourth category are completely manual test cases. The decision if a step could be automated and with what tool was taken with the support from the experts at the company. Test cases from all the categories were presented at the interviewees.

All the employees at the department and students in the IT field were invited by email through a common email template containing a short description of the purpose of the study and the intention of the interview. A total of 9 interviews were performed for this

Subject	Current role	Industrial experience (years)	Familiar with manual testing	Familiar with automated testing
S1	System Developer	10–20	Yes	Yes
S2	Developer Consultant	1–3	Yes	No
S3	Product Development Manager	10–20	Yes	Yes
S4	Test Specialist	10–20	Yes	Yes
S5	System Developer	4–9	Yes	Yes
S6	Senior System Developer	4–9	Yes	Yes
S7	Master Student	1–3	Yes	Yes
S8	Master Student	1–3	Yes	Yes
S9	Master Student	1–3	Yes	Yes

Table 4.1: Subjects demographics

study. All of the interviewees were familiar to manual testing prior to the interview and almost all of them, with the exception of one, were familiar with automated testing. This was important for the study because the subjects were able to understand the interview questions easily. Table 4.1 contains details about the interviewees such as their current role at a company or if they are students and their years of experience in the IT industry. The interview protocol (Appendix A) was inspired by the work of Runeson [30].

As suggested by Yin [31], a pilot interview was conducted so that the sanity and consistency of the interview protocol could be checked. The pilot interview was used to see if the asked questions are easily understood by the subject. During the pilot interview special attention was given to the way interview questions are formulated. The subject was asked how difficult the questions are to be answered and how understandable. Additionally, the time that the interview requires was recorded to assure that it is not too long for the employees at the company, who had definite time for the interview. The feedback from this pilot interview was used to make small changes to the interview protocol and formulate the questions in a more proper way.

The interviews were performed on both the company site with the employees and in campus Lindholmen (Chalmers). The average time for one interview was about one hour. Both authors were present during the interviews. One of the authors was asking questions and the other was documenting more important answers. At the beginning of each interview the interviewees were presented with the purpose of the study and how it fits a company needs. Moreover, the interviewees were informed that the interview is being recorded and that it is anonymous [29]. All of the interviews ended with wrap-up questions so that the interviewers could see if something was omitted in the research or if the interviewee could contribute with some relevant information.

After performing an interview the researchers summarized the answers of the questions and took out some relevant quotes. This approach is commonly used as an alternative to full transcription [32]. This technique assures that the researcher understood correctly the answers of the interviewees. With the same purpose, the summaries were send via email to the interviewees and verified by them. No changes were required after the summaries were verified by the interviewees.

In addition, one of the key quality requirements for SeAT is its extensibility, meaning that a new test technology could be easily added to it. The extensibility of SeAT makes the framework applicable in various domains, thus is very important for the generalisability of the study. At the end of the study the authors decided to add support for Microsoft Unit Testing Framework, which is a unit level test framework to SeAT and record the time it takes. Both authors were not familiar with Microsoft Unit Testing Framework prior to the implementation, however they were familiar to similar unit level frameworks.

4.3.4 Data Analysis

The data collection and analysis were conducted in three consecutive steps. Keeping a clear chain of evidence [33], conclusions were drawn in each step and used as input for the next step. This implies that the authors should present relevant information in each step in the research and justify the crucial decisions that were taken. Inspired by Runeson's model [29] the chain of evidence is illustrated with on Figure 4.2. A structured approach is vital in qualitative analysis [29]. This requires that the used instrumentation must be kept and links between data must be documented.

The first step of the research was the identification of the challenges, experienced in industry when different test technologies and practices are used. The data collected from unstructured and semi-structured interviews, observations and document study was coded, based on Robson's guidelines for qualitative analysis [32]. Coding is the process of giving a code representing a certain construct to a part of text. The level of formalism was chosen to be *editing approach* according to Robson's pattern [32], which guides codes to be defined by the researchers during the analysis. The *editing approach* eases the reader to understand the clear chain of evidence and to interpret the results from the analyses. Additionally, comments were added to the codes by the researchers to help them categorize the data.

Patterns in the collected data, related to the challenges, were identified in the first step of the research. These patterns were used to formulate the functional and quality requirements for the unification framework. The implementation, which is based on the elicited requirements, represents the second step in the the research. This structured approach of identifying patterns and relationships in the data and concluding a baseline for a framework makes the chain of evidence visible to the reader and helps him understand the trustworthiness of the conclusions [29].

The implemented framework was used as input for answering the third question, namely how such framework affects the validation and verification process in industry. In the third step of this research the effects of the framework were assessed with the help

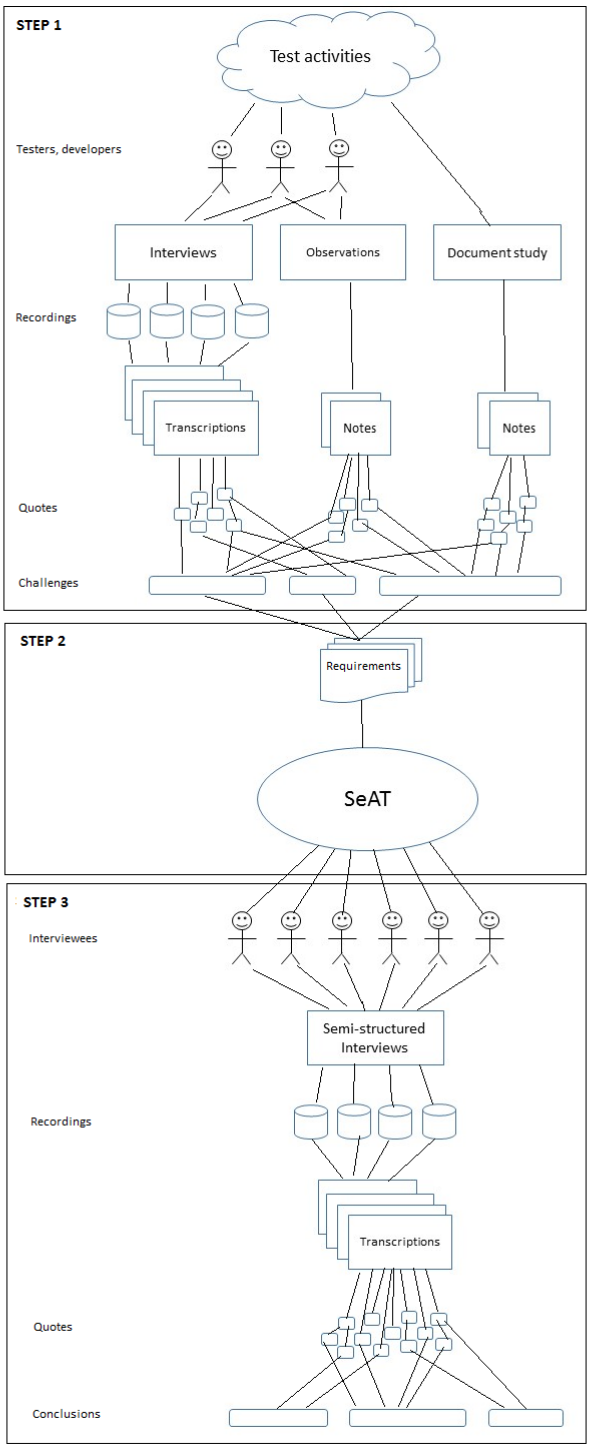

Figure 4.2: Chain of evidence

of semi-structured interviews. The interviews were transcribed into text files and parts of the text were marked by codes [32]. According to Runeson [29] tabulation is a very useful technique for the analyses of data, and hence the data from the interviews was presented in a table. Standard spreadsheet tool was used for managing the textual data. Using the tabulation approach, data can be overviewed easily because it is organized in rows of interest according to the codes. The different columns in the table represent the interviewees. This approach allowed the researchers to generalize the finding, resulting into formalized body of knowledge and it was presented as the final result of the research.

In order to reduce the bias of the individual researchers on the findings, a beneficial approach which was used in this research is the coding of the data by the two authors individually. After this process, the results from each author were merged into common analysis results. This approached helped for the increase of the validity of the results [29] and also broadened the codes, thus, the inferred generalizations.

4.4 Threats to Validity

The trustworthiness of qualitative research can be ensured by providing evidence that the findings were concluded from the data and not the authors' opinions [34]. According to Shenton [34] the researchers should always strive to provide information on the context of the research so that the readers could decide if it is applicable to other setting. Moreover, the researchers should enable possible repetition of the study, although this task is difficult in qualitative studies. To make the proof of trustworthiness possible, four categories for addressing validity threats are used in unison with Runeson and Höst model [29]. This model was designed to be more suitable for qualitative research.

4.4.1 Construct Validity

Construct validity indicates to what extend the researcher's perception of the studied subject is being investigated by the research questions [29].

To ensure source triangulation [25] during the process of data collection various approaches on different sources were used such as observation of the testers, document study on several key documents for the case company, interviews and discussions with diverse interviewees. To ensure observer triangulation [25] both authors were present on every observation or interview. Although the data collected from observations and documents was of tremendous value for the authors to understand the challenges in the given context, the main method for drawing conclusions were the interviews.

Having this in mind that the data from the interviews was precious, several approaches were undertaken to remove any possible bias. The selection of subjects was not based on the authors preferences since a common email with description of the study was sent to all employees in one department of the case company and interviews were conducted to all employees who responded to it, assuring *convenience sampling* to the research [35]. Similar email was sent to a large group of students and all respondents were interviewed.

Additionally, a pilot interview [29] was conducted with one of the students to assure that the questions from the interview protocol were understood the same way the authors understood them. The student was asked to evaluate the difficulty of answering the questions. The chosen interviewee was a student because students usually have less experience than the professionals and if a student does not have difficulties in understanding the questions the possibility that an employee would not have is high. Some minor fixes were made on the formulation of the questions after the pilot interview.

Before each interview the interviewees were introduced to the purpose of the study so that a common understanding could be achieved [29]. Moreover, the interviewees were asked about different key terms related to testing (such as unit, integration and system testing) and short discussions were conducted on each term mainly because even the literature does not fully agree on the exact meaning of these terms. To enable the interviewees to give their true opinions, the authors guaranteed their full anonymity before each interview. To keep the chain of evidence [33] all interviews were recorded with the permissions of the interviewees and are available on request.

The study was limited to only nine interviews, due to the fact that only nine people volunteered and the researchers were limited in terms of resources to find more interviewees. In spite of this fact, saturation of data [29] was reached, which means that it is unlikely that the researchers would extract any new information even if they had continued with the collection of data.

4.4.2 Internal Validity

To be able to identify the threats to the internal validity of a study, every researcher has to answer two key questions according to Feldt [36]: *Does the solution affect the outcome and are there any other factors that might have an effect.*

This study was trying to find the effects of SeAT on the verification and validation process in a company and the drawn conclusions were that it positively affects the efforts and the needed time for testing as well as it encourages the testers to use more automation. These findings were concluded from the opinions of experts and students during the interviews. Reducing the effects of the previous experience of the interviewees is the fact that all of them had different backgrounds and positions. During the interviews, the interviewees were not provided any hints or guided in any way towards answering positively about SeAT's qualities and effects. Additionally, all key terms were discussed prior to the interviews so that no misunderstandings could effect the answers of the interviewees.

4.4.3 External Validity

External validity indicates to what extend the findings of this research could be generalised and used in other companies [30].

To improve the external validity, the employees who responded to the invitational emails had different positions in the company and the students had different education background in IT. The limitation is that all employees were from the same company

and all students were from the same university, which affects the generalisability of the study negatively. However, conducting a qualitative study, the indicated challenges were abstracted from the context of the company and are applicable in other cases.

During the interviews with the employees, one system under test was used to show the capabilities of SeAT and another was used with the students to show that the developed framework is applicable in different domains and on different systems. Selection of the test cases was performed based on their complexity so that it would not be biased by the authors preferences, thus making the framework applicable to test cases with different complexity and nature. The automation tools which were used for comparison were selected according to the level of abstraction (Unit, Integration, System) that they work on, showing that the framework supports tools on all levels. However, the selection of these tools was affected by the company's preferences and their requirements, which could threaten the generalisability of the framework. To deal with this problem, the authors investigated the extensibility of the system through interviews and it was concluded that the framework is easily extensible and it is usable with any other tools for automation testing. Regarding the observations, used to identify the challenges in using different test approaches, they were conducted in micro scale and might lack representative sample [37]. Yet another limitation is that the framework is build on Microsoft .NET framework and currently only supports devices which have .NET framework installed.

4.4.4 Reliability

Possible threats to the validity of the study are the dependencies between the conclusions and the researches. A reliable study could be re-conducted with the same or similar results [29].

Robson [32] has introduced a list with methods on how to rise the reliability of a study. From that list several methods have been used by the authors. *Peer debriefing* relates to the group work of two or more researchers, in order to lower the risk of biasing the results. In that manner the two authors of this thesis were present in all observations, interviews and discussions with various stakeholders. *Member checking* is another method from Robson's list [32] and it refers to various materials obtained during the study to be reviewed by different participants. In unison with that method, the authors sent summaries from the interviews to the interviewees to ensure that nothing was misunderstood. Moreover, SeAT was reviewed on weekly basis by a company's employee who is in the position of a product development manager, in order to ensure that it addresses the challenges during validation and verification and that the framework conforms with its requirements. Importantly, all research design artifacts such as the interview questions protocol and SeAT requirements were reviewed by university researchers. Moreover, observations are generally regarded as less reliable [37] since they are very difficult to be replicated by other researchers in other contexts. Conforming with another method from Robson's list [32], namely *audit trail*, all collected data was documented and saved. Interview notes, audio recordings and observation notes were stored and are available on request, thus keeping a clean chain of evidence [33].

Furthermore, negative case analysis [30], which is the process of formulating alterna-

tive theories in order to improve the analysis, was used. The authors summarized and coded the transcripts from the interviews individually and then one set of codes was synthesized. No predefined codes were used, but instead all codes were induced from the collected data [32], thus the data was not forced into a concrete category.

5

Technical Implementation

This chapter starts by giving an overview of the system with a concrete example and screen-shots (section 5.1). Then, it describes in details the technical implementation of SeAT illustrated in four different architectural views [38], which are physical view (section 5.2), development view (section 5.3), logical view (section 5.4) and process view (section 5.5).

5.1 Overview

In general, SeAT is a system which supports semi-automation and unification of test technologies and practices. SeAT comprises two components, which are called Orchestrator and Client. Orchestrator is installed in a tester's workstation and it acts as a centralized server directing the test executions on various client machines. Client is installed in each of the subsystems under test. It receives and executes instructions from Orchestrator. This design decision is affected by the service orchestration approach where a central process (coordinator of the orchestration) controls the execution of different operations on the services participating in the process [39]. The Orchestration is usually compared to Choreography which, in contrast, does not depend on a central orchestrator and all participants know exactly when to become active. In order to make SeAT's Clients more agile and maintainable the Orchestration approach was chosen for SeAT's implementation. This choice was further encouraged by the fact that Orchestration also provides centralized management of the resource pool [40] — test descriptions, test scripts and other resources.

Furthermore, there was a choice of implementing the Clients as web service providers, which wrap the supported test tools in web services so that these services could be consumed by the Orchestrator. In fact, there are many existing frameworks, which support services and service Orchestration [41]. They are all based on Service-oriented architecture [42] to support the loosely coupling principle and they incorporate a subset

of Business Process Execution Language (BPEL) [43] to define a set of web service orchestration concepts. Some of the complete solutions by the major software vendors are:

- Oracle Fusion Middleware (OFM)
- IBM WebSphere Enterprise Service Bus (ESB)
- Microsoft Workflow Foundation (WF) with Windows Communication Foundation (WCF)
- SAP NetWeaver SOA Middleware

As a matter of fact, with the use of any of these software the goal of the study could be achieved. However, some of the quality attributes are that the developed system should be easy to learn and simple to be extended. In case of using external SOA frameworks, to add a new test tool would require from the users of SeAT to know and understand the frameworks for service orchestration. As SOA and the web service specifications practitioners expand, update and refine their output, it requires skilled people to work on SOA-based systems, including the integration of services and construction of services infrastructure. According to the interviewed developers, they prefer simplicity to extensive functionality. Since the external frameworks are built for general purposes, they do not specifically target the unification of test technologies. As a result, much more functionality than needed is implemented and this affects the performance in terms of execution time and space required. Therefore, SeAT is aimed to create a thin Client component which has as little impact as possible on the systems under test. In contrast to SeAT's non-intrusive approach, external SOA frameworks would require a server (IIS, Oracle WebLogic Server, WebSphere Application Server) and other software to be installed on each Client, which is time-consuming and requires a lot of effort. On top of that, these servers usually consume a big amount of computational resources ([44, 45]) from the system during execution and this might make the systems under test to behave differently while performing tests.

Importantly, the main research question is to evaluate the idea that having such a unified test framework will positively affect the verification and validation activities. Therefore, although the IT artifact plays an important role in this research, the effects of a unified test framework is validated regardless of the employed technologies. However, additional research is needed to evaluate if the adoption of the external frameworks for service orchestration would affect the results.

Additionally, in order to analyze the effects of a unified test framework on the verification and validation activities several systems for unified testing were investigated, but none of them fulfilled all the requirements for this study. One of the most used software for similar purposes is HP Unified Functional Testing (UFT) [46] which provides functional and regression test automation for software applications and environments, but lacks the ability to incorporate third-party test software. Another application is Test Director [47] which offers integration with third-party and custom testing tools, but on the other hand cannot control test cases distributively.

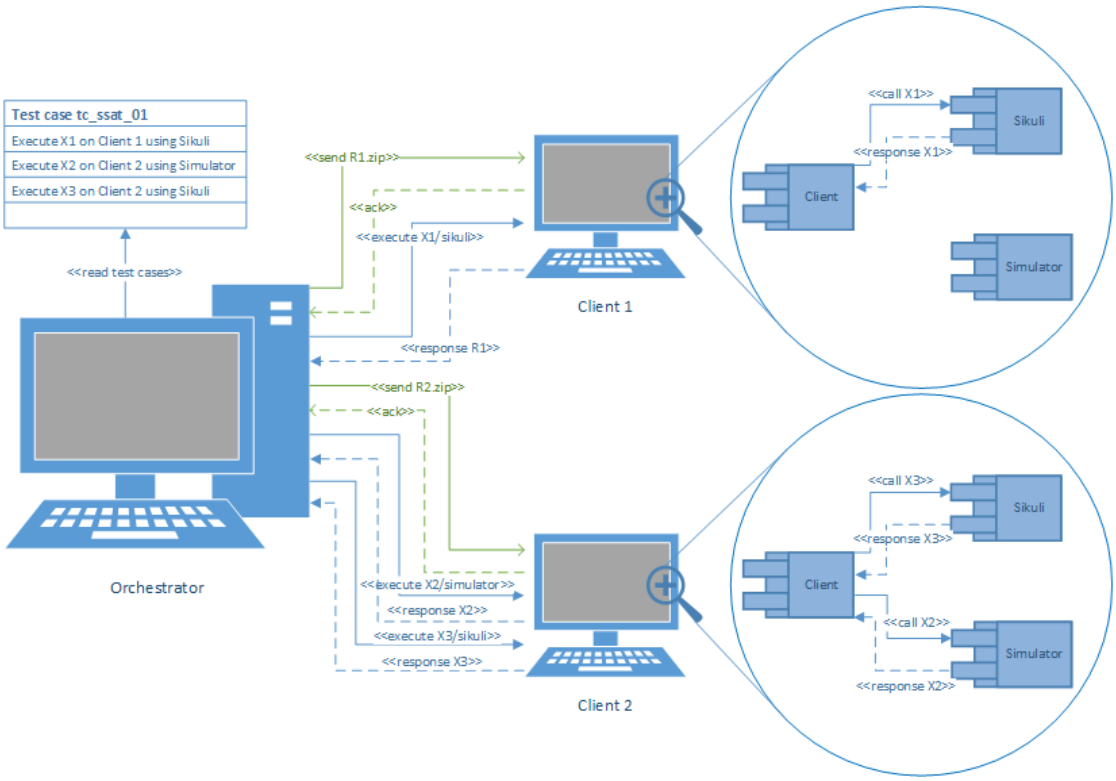

Figure 5.1: Test case execution: an example scenario

5.1.1 An example

Figure 5.1 shows an example scenario of executing a test case.

- *Environment setup:*
  The tester's machine is named Orchestrator and it has Orchestrator component installed and stores all test scripts. The system under test consists of two machines: Client 1 and Client 2. The two machines are installed with Client component.

- *Test case sample:*

  — Execute command X1 on Client 1 using Sikuli.
  — Execute command X2 on Client 2 using Simulator.
  — Execute command X3 on Client 2 using Sikuli.

  As can be seen in the sample test case, the test execution requires interactions with two machines. These interactions are on two different levels of abstraction — Sikuli on the acceptance level and Simulator on the system level.

```
<TestCase Id="123" Description="Test the correctness of the booking function." Name="tc1">
  <Steps>
    <Step Description="Create a package" IsCritical="true">
      <Operation Directive="Create a package with one flight and one hotel." Executor="Human" />
      <TargetClient Name="TRIP_2" />
    </Step>
    <Step Description="Check if a package appears on the screen" IsCritical="true">
      <Operation Directive="scripts.sikuli" Executor="Sikuli" />
      <TargetClient Name="TRIP_2" />
    </Step>
  </Steps>
</TestCase>
```

Figure 5.2: Test case example

- *Execution:*
  When the tester starts running the test cases, relevant resources (test scripts) are zipped and sent to the clients. The process of sending test scripts is denoted with green arrows in the Figure 5.1. Sending test scripts is independent on their execution, therefore, as soon as a test script is available on the Client, the Orchestrator can execute it. By this way, the Orchestrator does not need to wait for all test scripts to be sent to the Clients. The Orchestrator executes the steps sequentially. The executions are denoted with blue arrows. Not until a response of a particular step is received will the next step be proceeded. The dashed arrows are the responses of the interactions.

  Inside the Clients, the commands are processed by the Client component and they are translated to interactions with the test technologies using the corresponding executors. The executors are extensible and can be easily built/modified/added to be compatible with the target system under test.

5.1.2 Test case structure and semantics

Figure 5.2 depicts an XML representation of the structure of a test case. One test case contains a list of sequential steps, each of which is an operation performed on a particular target client.

Independence amongst test steps was noticed during the document study process at the case company. One reason for the independence is that the time consumed by setting up the environment for each test case is too long. Therefore, several test cases are combined in order to test one set of requirements at the same time. This observation led to the design decision that a test step inside SeAT can be *critical* or *non-critical*. A *non-critical* test step does not break the execution of a test case if it fails. In the case company only dependent or independent sequential steps were observed, however, in practice there might be test cases with more complex structures involving loops and conditions. Although, the complexity of the test cases is out of scope of this thesis, further research in this area might improve the applicability and efficiency of SeAT.

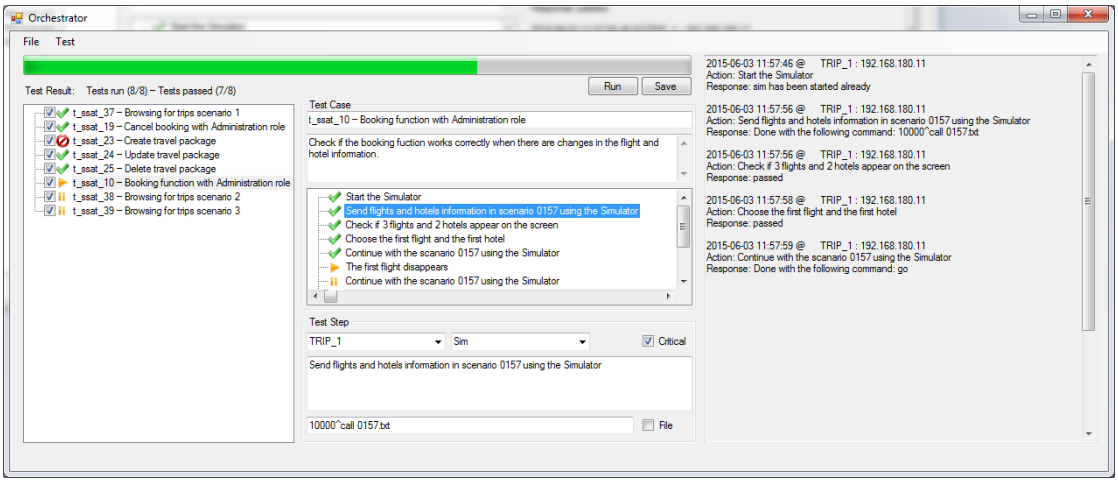

Figure 5.3: Orchestrator user interface

5.1.3 Test case manipulation

SeAT allows users to create/modify/delete test cases and test steps through the graphical interface. The screen-shot of the Orchestrator user interface (Figure 5.3) shows various controls for updating test cases and test steps. For example, the target client can be selected from the drop-down list which is currently set to `TRIP_1`; test description can be updated via the description text-box. Button 'Save' is used to save such kinds of modifications. Creation or deletion of test steps/cases could be performed by a contextual menu (Figure 5.4). Additionally, since the test cases are serialized into an XML file (shown in Figure 5.2), direct modification to such file also changes the test cases. The only requirement for this approach is that the users have to be familiar with the XML markup language.

5.1.4 Screen-shots

Figure 5.3 and Figure 5.5 illustrate Orchestrator and Client applications. Orchestrator is an Windows form application while Client is a console application. Figure 5.3 depicts how the test cases and test steps look like when running. An output windows on the right of Orchestrator shows the responses received from Client. Figure 5.5 portrays the directives obtained from Orchestrator and information of what is being operated in Client.

5.2 Physical View

This view encompasses the nodes that form the system's hardware topology on which the system executes; it focuses on distribution, communication and provisioning [38]. In this view, the deployment design will be demonstrated.

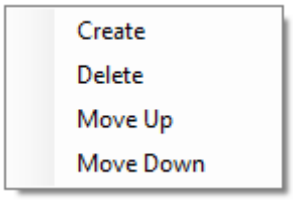

Figure 5.4: Contextual menu of test case/test step lists

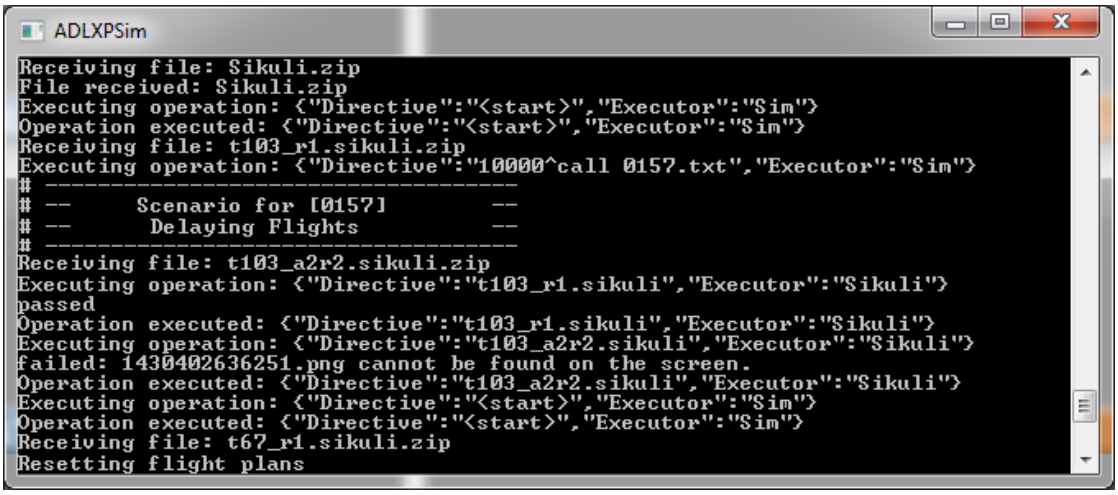

Figure 5.5: Client console application

In particular, SeAT consists of two components which are called Orchestrator and Client. They are implemented following the Server—Client design. Orchestrator is installed in a tester's workstation and acts as a server. Client is installed in each of the subsystems under test and acts as a client of Orchestrator. The deployment is illustrated in Figure 5.6.

Orchestrator retrieves test cases stored in database and shows them to the tester, who can then select what to run. Each of the test cases contains test steps. Each of the test steps contains an action directive stating how a step is done and an assertion directive stating how to validate the outcome of such step. Action and assertion directives can be manual or automated and performed on a specific client. Since the system under test is distributed and contains subsystems, every directive should have information about the target client. Manual directives are in the form of natural language whereas automated directives are test scripts. Once a test case is chosen to be run, its test steps will be performed sequentially by Orchestrator. If a directive within a step is manual, 2 dialog boxes will appear on both Orchestrator and target Client to show the description to the tester. These two dialogs are identical and the tester can interact with either of them. The only purpose of having multiple dialogs is to enhance the usability. After performing the action or assertion stated in the dialog boxes, the tester can choose to pass or fail

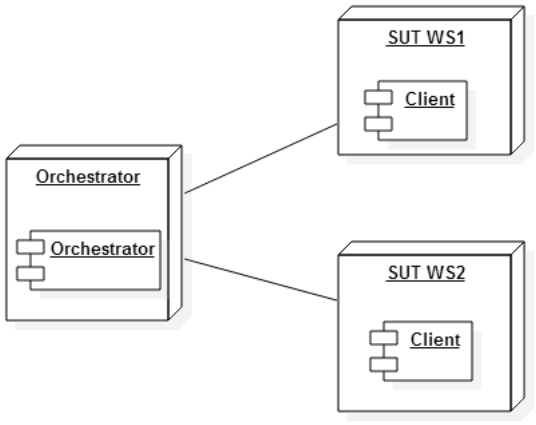

Figure 5.6: Deployment view of SeAT

the current directive. The tester can also add comments providing reasons why such directives are failed or additional information needed to be logged during verification and validation activities. If a directive is automated, it will be performed immediately at Client and the result will be sent back to Orchestrator. If any of the directives are failed, the related step and the related test case will be failed and terminated.

In order for Clients to be located, configuration is made and it can be changed at run-time. Due to the fact that test environment at the studied context is quite stable, which means that the subsystems under test have fixed locations and IP addresses for a long period of time, such configuration, despite being manual, does not hinder the whole process of testing. However, SeAT is open for changes and automated or semi-automated configuration with location detection, which is more user-friendly, can be easily incorporated in the future.

5.3 Development View

The development view focuses on the actual software module organization on the software development environment [38]. As described in the previous section (section 5.2), SeAT has 2 modules — Orchestrator and Client. The modules, sub-modules and exposed interfaces are depicted in Figure 5.7, Figure 5.8 and Figure 5.9.

5.3.1 Orchestrator

Orchestrator contains 4 sub-modules, namely `TestAccess`, `Controller`, `Organizer` and `UI`. `TestAccess` is in charge of retrieving test data from the database and transforming them if necessary. `UI` is a user interface module which has responsibility of displaying visualized data to the tester. `Organizer` enables the ability of setting up and tearing down the test environment during each test suite execution. `Organizer` depends on actual implementations of particular test technologies and the behavior is described in

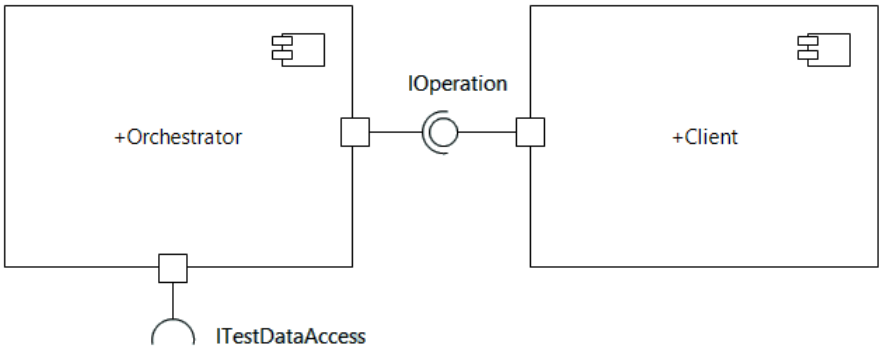

Figure 5.7: Development view of SeAT

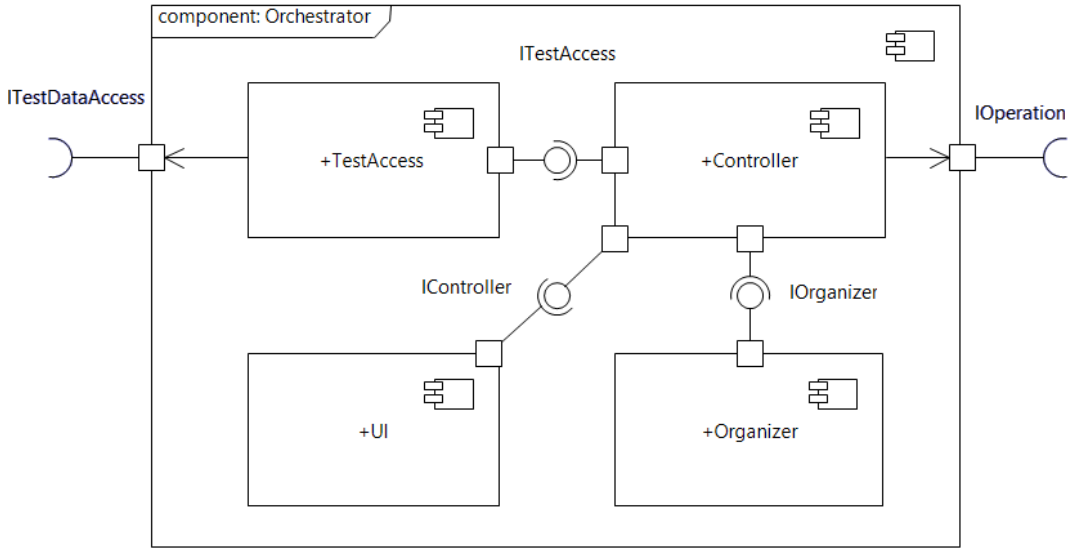

Figure 5.8: Sub-modules within Orchestrator

section section 5.4. `Controller` is the core module of Orchestrator. It connects all internal sub-modules and orchestrates the clients through an exposed interface `IOperation`. It is a "Thick Server—Thin Client" design and every control is centralized at Orchestrator while Client has no knowledge about the running test suite, test cases or test actions. It simply receives and performs any directives sent from Orchestrator.

5.3.2 Client

Client contains 2 sub-modules, namely `Coordinator` and `Executor`. `Coordinator` acts as a facade to receive commands from Orchestrator. Depending on which test technology is being used, `Executor` interprets and executes these commands accordingly. `Executor` is built in such a way that it can be easily extended for future integration of new test

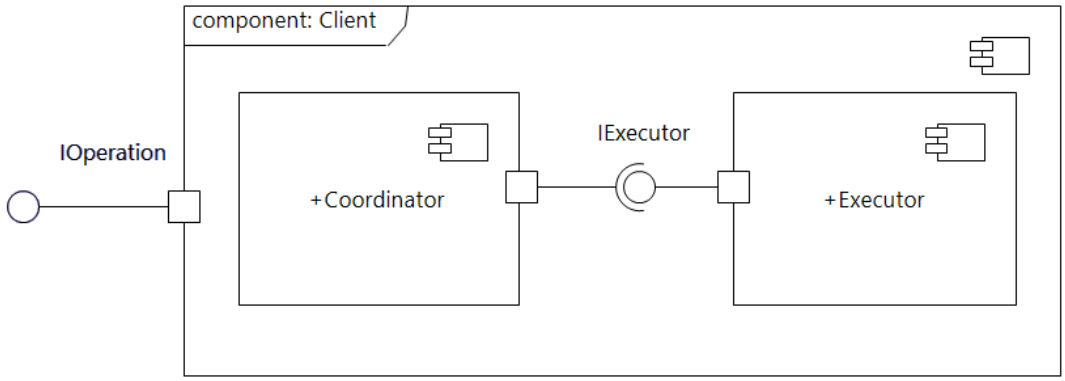

Figure 5.9: Sub-modules within Client

technologies. Such extension and the way to achieve it are described in details in the next section (section 5.4)

5.4 Logical View

In this view, the system is decomposed into a set of key abstractions, taken (mostly) from the problem domain, in the form of objects or object classes. They exploit the principles of abstraction, encapsulation, and inheritance [38]. Removing unnecessary details, the authors focus on describing the implementation of `Executor` and `Organizer` and how they are designed to maximize flexibility and extensibility. Figure 5.10 and 5.11 demonstrate the interfaces defined in SeAT and their current implementations to support testing against the ATC system. In general, the executors and organizers act as adaptors between SeAT and the test technologies which are required to be incorporated. In particular, the executors extend the Client's capabilities, while the organizers augment the Orchestrator's capabilities.

5.4.1 IExecutor interface

There are three methods declared in `IExecutor`, i.e. `Execute(string)`, `StartUp()` and `ShutDown()`, amongst which, `Execute(string)` is the most important. It executes every command string sent from Orchestrator and would be called once for each test action. By delegating the knowledge to the executors, it depends on certain cases, needs and testers to decide the execution commands and how they are interpreted. For instance, `"call 0123.txt:1000"` could be added as a test step command through the Orchestrator's user interface (Figure 5.3). Then this string is sent to the Client and the corresponding Executor parses the message into two parts — `"call 0123.txt"`, which is given to the simulator and instructs it to execute script 0123.txt, and `"1000"` which is the amount of miliseconds that the Client should wait for the simulator to finish executing the script. This is just an example and the actual execution commands and how they are handled

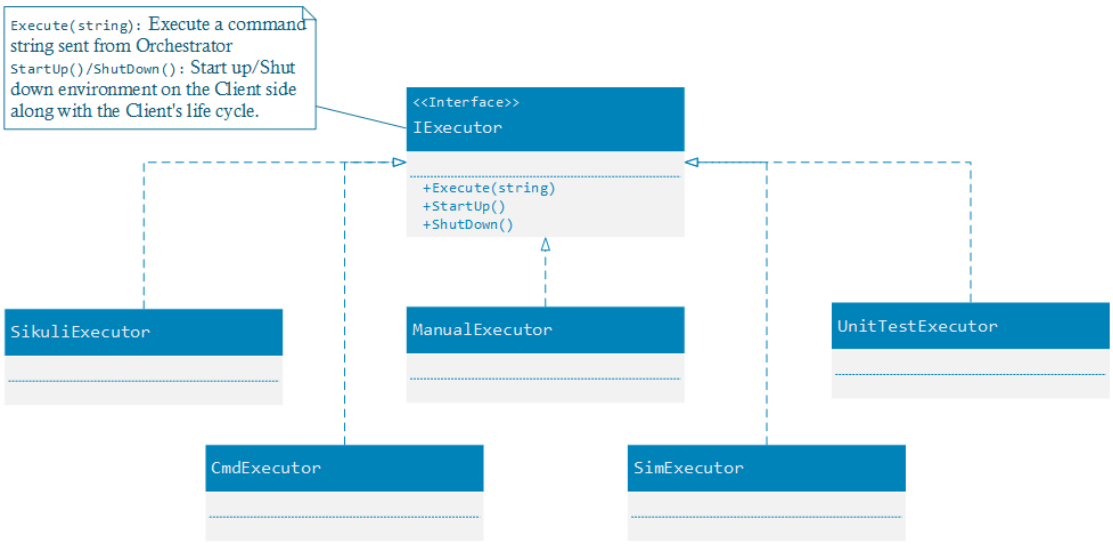

Figure 5.10: `IExecutor` interface and its implementations

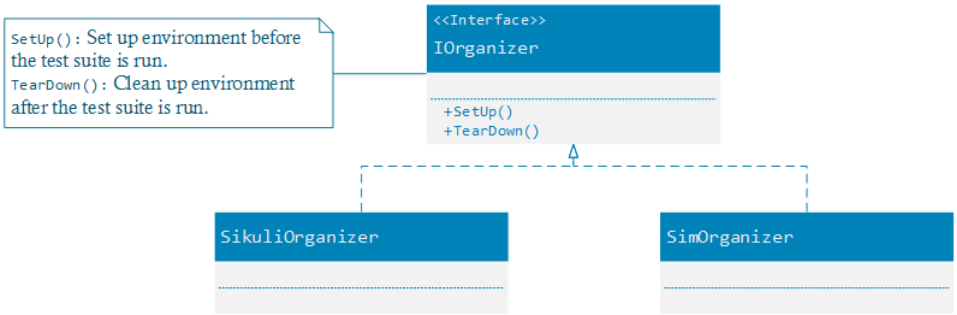

Figure 5.11: `IOrganizer` interface and its implementations

by the Executors are left for the users of the system to decide. For the purposes of this study, five different Executors have been implemented, but the system is designed to be easily extended by its users.

The reason why `string` was chosen to be the type of the input parameter is that almost everything can be passed as string in a concise and human-readable format. This enables the extensibility for unforeseen test technologies which might be incorporated into SeAT. More complex data structures, for example represented in forms of JSON or XML, could be sent from the Orchestrator to the Client instead of simple strings, but this may hamper the flexibility of the system. Furthermore, knowledge of these formats would be required for the `IExecutor` interface to be implemented. On the other hand, using complex messages, will be far more reliable and resistant to change than any custom string parsing. More test technologies should be investigated before a common structure for the parameter could be concluded.

Pertaining to the other two methods declared in `IExecutor`: `StartUp()` is called

when Client is started in order to kickoff dependent applications needed for a specific test technology; whereas `ShutDown()` is called when Client is stopped in order to terminate such dependent applications. This ensures the life-time of dependable components is in accordance with the life-time of Client.

As can be seen in Figure 5.10, `IExecutor` interface is actualized by `SikuliExecutor`, `CmdExecutor`, `ManualExecutor`, `SimExecutor` and `UnitExecutor`. The roles of the particular executors are:

- `SikuliExecutor` executes Sikuli scripts sent from Orchestrator. It takes quite some time to start Sikuli each run, therefore, a simple internal Python server which was named `SikHost` is started once and kept running to host the Sikuli executable during the whole test process. The commands are forwarded from `SikuliExecutor` to `SikHost` and results of the directives are sent back via this connection. Thanks to the *hot-swapping* technology embedded in the built-in Python libraries, any new commands or Python files sent can be executed during run-time. The `SikHost` instance is terminated once the test is done. With this approach, the running time of the Sikuli tests was sped up dramatically.

- `CmdExecutor` executes Windows commands sent from Orchestrator. It is a general executor and theoretically it can perform any directive in Windows systems.

- `ManualExecutor` displays a dialog box on the Client's screen showing action or assertion directives. The tester will manually perform test activities as instructed by the directives and let the system know if they are successfully done by clicking on Pass or Fail buttons.

- `SimExecutor` executes specialized commands on the simulators being used in the ATC system.

- `UnitExecutor` executes unit tests and returns results to SeAT.

5.4.2 IOrganizer interface

`IOrganizer` introduces 2 methods: `SetUp()` and `TearDown()`. These terms are borrowed from unit test terminology. The purpose of these methods, as the terms suggest, is to set up and clean up the test environment during each of the runs, each of which consists of many selected test cases. Figure 5.11 illustrates `IOrganizer` interface and its implementations, namely `SikuliOrganizer` and `SimOrganizer`. Purposes of such implementations are:

- `SikuliOrganizer` sends scripts which are reusable within one test run at the beginning of each run and clean up every script once all executions have been finished.

- `SimOrganizer` resets the simulator to its starting state each run.

```
<unity>
  <typeAliases>
    <!-- Lifetime manager types -->
    <typeAlias alias="singleton" type="Microsoft.Practices.Unity.ContainerControlledLifetimeManager, Microsoft.Practices.Unity"/>
    <typeAlias alias="perresolve" type="Microsoft.Practices.Unity.PerResolveLifetimeManager, Microsoft.Practices.Unity"/>

    <!-- Custom object types -->
    <typeAlias alias="IExecutor" type="TestEnvironment.Executors.IExecutor, TestEnvironment"/>
  </typeAliases>
  <containers>
    <container>
      <register mapTo="TestEnvironment.Executors.ManualExecutor, Adaptors.ManualAdaptor" name="ManualExecutor" type="IExecutor">
        <!-- Create new instance each resolve -->
        <lifetime type="perresolve"/>
      </register>
      <register mapTo="TestEnvironment.Executors.SikuliExecutor, Adaptors.SikuliAdaptor" name="SikuliExecutor" type="IExecutor">
        <lifetime type="singleton"/>
      </register>
      <register mapTo="TestEnvironment.Executors.SimExecutor, Adaptors.SimAdaptor" name="SimExecutor" type="IExecutor">
        <lifetime type="singleton"/>
      </register>
      <register mapTo="TestEnvironment.Executors.UnitTestExecutor, Adaptors.UnitTestAdaptor" name="UnitTestExecutor" type="IExecutor">
        <lifetime type="singleton"/>
      </register>
    </container>
  </containers>
</unity>
```

Figure 5.12: Register newly-built test technology adaptors to SeAT

5.4.3 Dependency injection

Above are the default and customized adaptors (executors/organizers) which were developed for the system under analysis. They implement the same interfaces and this enables any extension needed in the future. Each of these implementations is compiled into a dynamic-link library. Dependency Injection [48] (or sometimes known as Dependency Inversion Principle [49]) is, then, used to incorporate the libraries into SeAT. Additional configuration is necessary for SeAT to locate the libraries during run-time. Such configuration also specifies life-time of the components, such as singleton or non-singleton. With this approach, no recomplilation of the whole system is required for extensions. An example of the part of the configuration file containing the component registration is shown in Figure 5.12.

5.5 Process View

The process view addresses issues of concurrency and distribution, system's integrity, fault-tolerance. In addition, it shows how the main abstractions from the logical view fit within the process architecture, that is, on which thread of control is an operation executed [38]. An illustration of the communication between different components regarding the main flow of interactions is shown in Figure 5.13

During the test execution process, Orchestrator sends test's resources to Clients for each of the directives. Not until receiving an acknowledgement that corresponding resources have been sent correctly does Orchestrator start executing the directives. The resources can be images, text files or other test scripts. To optimize performance, the resource sending process is decoupled from the execution, in other words, they are done asynchronously. Therefore, Clients can receive resources for upcoming test steps before hand, while the previous ones are being run.

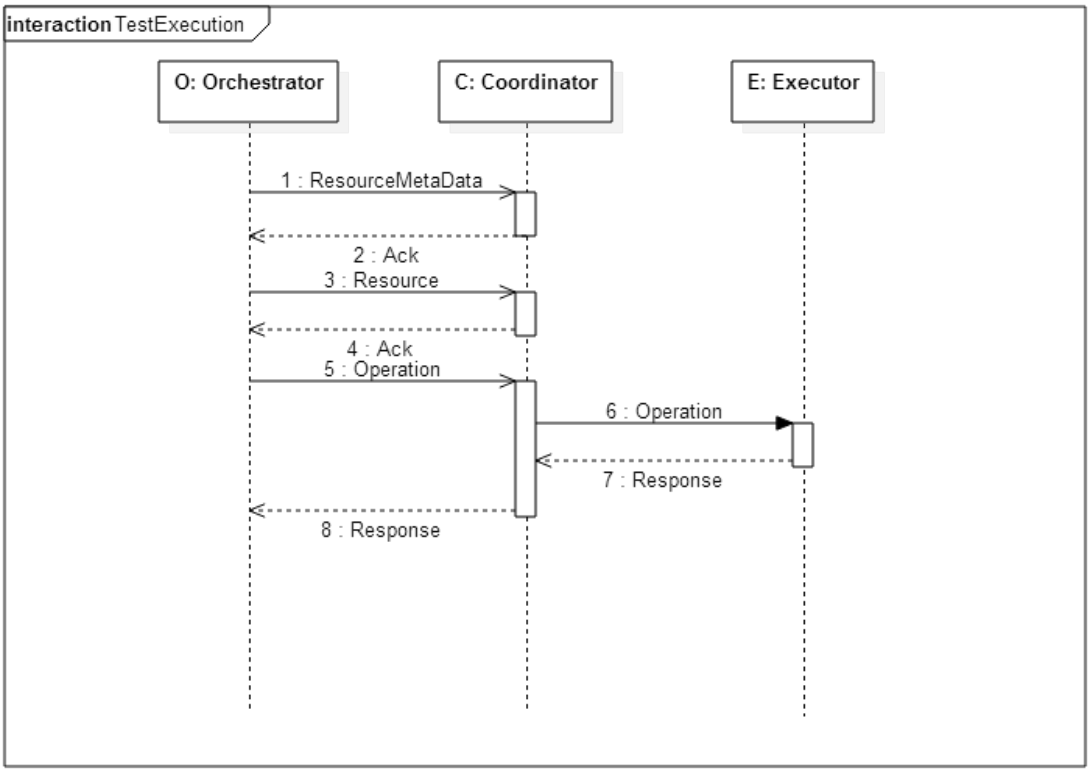

Figure 5.13: Test execution sequence diagram

6

Results

This chapter presents the findings of the study. Challenges of using different test technologies and practices are demonstrated in section 6.1. Based on the challenges detected, guidelines for developing a framework to unify test technologies and practices are revealed in section 6.2. Finally, assessment of the developed framework as well as its effects on the verification and validation activities are stated in section 6.3.

6.1 RQ1 — Challenges of using different test technologies and practices together

There are various challenges adhere in the testing process especially when it comprises different test technologies and practices within one test case. Amongst them, there are some significant issues which were observed, discovered and elicited during the course of the study.

6.1.1 Untidiness

First and foremost, the employment of different test technologies have caused a burden both on the creation and execution of tests. Since most of the industrial respondents had experienced the case where a mixture of such technologies was adopted within one test case, they came up with thoughtful ideas about the problems it brought. One is that automation supporting tools have to be installed and located on the system under test. The number of test scripts developed and executed on each tool is large — over 200 in one of the tools under investigation. It was observed that in many occasions, the tester struggled with searching for the correct test scripts, which are situated in various folders. One senior system developer admitted:

> *"It is often hard to find the right scenario (test script)."* — Senior System Developer

Another system developer added that it took time to setup such tools into correct states before performing the tests. This is one of the biggest problems they are facing today and it makes the whole testing process more costly.

> *"It's time consuming to setup the simulators and it's also hard to find the scenario files if they scatter all over the place."* — System Developer

Furthermore, according to a product development manager, things would get quite messy if many technologies are being used. By looking at a single script, one cannot see the relation with other scripts and/or a test case as a whole. Therefore, it would be good if there is a way to organize the test scripts of different technologies. He asserted:

> *"If you have a lot of automated things that would fit together and it's not clear where the result is, it would be a mess."* — Product Development Manager

Besides, integration was amongst the problems detected by a system developer when various technologies were employed:

> *"There could be integration problems among the test tools."* — System Developer

6.1.2 Insufficient knowledge

Having such a wide range of technologies to embrace, sometimes even skilled testers had problems conducting tests if there was quite some time since the last test activities. It was observed that after several months having not performed any test activities in a particular system, the tester found it hard to reminisce the right configurations for the simulators. Inevitably, for new-comers, the situations are worse, as stated by some interviewees:

> *"The tester (sometimes) does not know what the script does. After a failure of a test script, he is not sure where the problem comes from, such as misunderstanding of the test case, a bug in the system or a bug in the simulator."* — Product Development Manager

> *"The test cases do not describe really well what to do. If you are not familiar with the simulators and do something wrong the tests might not show what you expect."* — System Developer

Therefore, familiarity with the simulators is one of the requirements for testers of the ATC system. Needless to say, familiarity with the system is also a must due to the fact that test cases are usually described in a high level of abstraction. Additionally, test

cases use distinctive business terminologies, and testers need to understand the system in details to assert if a test case is passed or failed. Such finding was discovered during the document study performed in the beginning of the thesis. The senior system developer confirmed:

> *"Often you need to be very experienced with the system to see the problems. Often the test cases succeed but you see other issues that are not covered by the test. It is just some other things fail."* — Senior System Developer

6.1.3 Test reporting problem

The observation showed its strength when detecting a problem which is usually ignored unintentionally during interviews, that is the result reporting of the test cases. It was seen that the tester went through the test cases and tested the system step by step following a 300-page test specification. More often than not, the tester stopped to perform a complicated step and when he continued, he lost track of the current step and had to search for it. Upon his investigation, he noted down unexpected behaviours corresponding to each step on paper, then, created additional issues if applicable. Test results after each step were also written down in a notebook and input to a test management tool afterwards.

6.1.4 Inefficient time exploitation

There were challenges which made the efficiency of time used not maximized. In most of the test cases, the tester started a simulator and had to wait the whole time before proceeding to the next step. Additionally, setting up the system into a correct state before any test cases was time consuming. The product development manager confirmed:

> *"Setting the system to the correct state takes time but does not add value to the test."* — Product Development Manager

A distributed system with asynchronous communication caused another challenge. The tester interacted with many machines and had to switch workstation between the steps. Moreover, results of the interactions with one machine were sometimes expected to be shown in a different machine after an arbitrary amount of waiting time.

6.2 RQ2 — Method for unifying different test technologies and practices

In order to successfully unify different test technologies and practices a framework — which complies with the challenges from RQ1 which are actually the company business needs — has to be created. Moreover, certain quality requirements have to be met in order for the framework to be generalizable and can be used in other projects and

contexts. Such quality attributes are: applicability, usability, extensibility and flexibility. The evaluation of these attributes is presented in subsection 6.3.1.

The result for this research question is the developed framework for unifying test technologies and practices — SeAT. The technical solution to unifying different test technologies is described in the previous chapter (chapter 5). Some guidelines about how to unify test technologies and practices as well as rationales behind them are discussed in section 7.2.

6.3 RQ3 — Effects of the unified test framework on testing activities

This section discusses the effects of the developed unified test framework on testing activities. However, before any discussion being made, the assessment of the framework based on its quality requirements — which are conveyed in the results of research question RQ2 — is demonstrated in subsection 6.3.1 in order to make sense of the trustworthiness of the framework. Subsequently, various effects of the framework on the testing process are reported in the succeeding subsections.

6.3.1 Framework assessment

In this subsection, results about the assessment of SeAT's quality attributes, namely applicability, usability, extensibility and flexibility, are discussed.

— *Applicability*

In general, the system received appreciations from all of the interviewees and many agreed that the framework would certainly fit well to the testing process of their projects. As most of the practitioners (5/6) are working with legacy systems, where lacking of automated tests is quite common, the approach of reconciling manual and automated tests is highly appreciated. Because it takes time and effort to convert to a new way of writing and executing tests, SeAT shows it strength in backward compatibility with the traditional way of performing testing. Incremental changes can be made in order to fully utilize the framework capacity. The product development manager concluded about the applicability of SeAT:

> *"It (the framework) would certainly fit very well."* — Product Development Manager

When asked if the framework solved the problem of using many automation test technologies together in one test case, all of the interviewees responded 'Yes', especially when *"it comes to executing the tests"* — the product development manager added.

Similarly, all respondents were positive about the way the framework solved the problem of having manual and automated steps within one test case. The impression was that even though testers need to be present to perform semi-automated

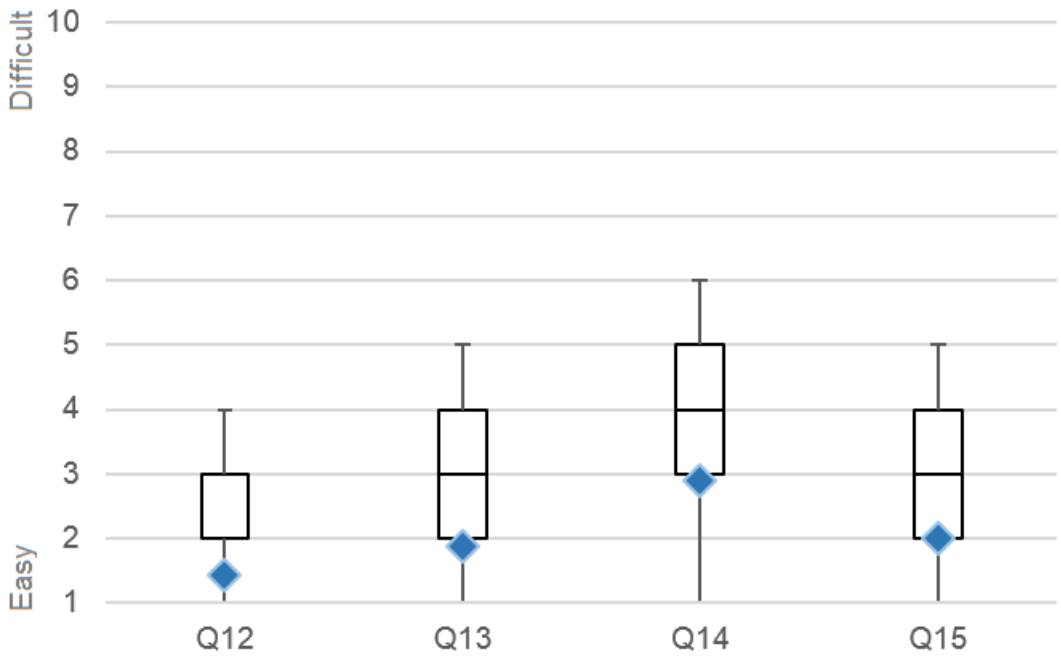

Figure 6.1: Subjects' assessment about SeAT's usability and extensibility

tests, the assistance of the framework makes it easier for them to execute the test cases.

> *"Even if you only use simulators and manual steps it can be still quite helpful"* — System Developer

— *Usability*

In combination with the open-ended questions to assess the usability of SeAT, the following questions were asked during the interview:

12. From 1 (easy) to 10 (difficult), how difficult is it to understand the purpose of the framework?
13. From 1 (easy) to 10 (difficult), how difficult is it to create test cases and related test steps?
14. From 1 (easy) to 10 (difficult), how difficult is it to get the result from the execution of a test case?

The full list of all interview questions are included in Appendix A.

Figure 6.1 is a box–plot diagram which illustrates the assessments of SeAT's usability and extensibility. The scale is from 1 (easy) to 10 (difficult). Q12, Q13 and Q14 stand for the results for questions 12–14. It can be seen that SeAT was found relatively easy to be understood and used for creating and executing test cases.

Regarding getting the results from the test cases, it was believed to be slightly more difficult than the rest (with median=4). The reason why it is still moderately difficult to export test results is because of the current system for archiving test results.

With respect to the test case execution, SeAT provides a way for testers to continue their tests even though some steps fail and log down their findings during the test execution. Such a feature ensures the compatibility of the framework to the existing way of performing tests in the company. Because the thorough reformulation of the test cases to fully adapt to a new approach of writing tests takes time and effort, this feature received positive feedback from most of the interviewees. A system developer claimed:

> *"It's good to have a way to say that you can continue testing anyway or not. Because sometimes it's worth continuing even though one step was failed."* — System Developer

Notwithstanding, there was some skepticism about the usefulness of the feature. Another system developer questioned if such a step can be skipped during execution, why should it be included in the beginning. The reason is that to save time due to the complexity of setting up the environment, many requirements were combined and tested once. Thus, there exist test steps which are independent of one another. As a result, skipping some of the steps does not affect the validity of the succeeding steps.

— *Extensibility*

The result for question 15, which is about extensibility of SeAT, is also included in Figure 6.1. The question is:

15. From 1 (easy) to 10 (difficult), do you think that a new test technology could be easily added to this framework? Example?

It can be noticed that adding a new test technology to the framework is relatively easy, especially for the developers. In addition, 3 of the interviewees, despite being students, found it simple to extend the framework also. A reason was provided by system developer that "it's just one method which takes a string and you can do whatever you want." The senior system developer stated:

> *"It was very clean, the interface for new executors was very simple and easy to extend and that's very good point."* — Senior System Developer

Upon the justification of SeAT's extensibility, a new adaptor to Microsoft Unit Testing Framework was added to SeAT by the authors. The process was timed and it took around 4 hours to add such a new adaptor. It was estimated by a system developer that it would take him a couple of hours to add a new technology to SeAT as he found it effortless.

— *Flexibility*

During the assessment, SeAT was used to create and execute test cases in another project. This project is a website built in Java with RESTful services. 2 executors were used, i.e. `SikuliExecutor` and `ManualExecutor`. The framework worked as intended with this new project. If there is a need of testing the RESTful services, a new executor can be created to interact with the website on the component level. The result shows the flexibility of the framework when applied to different projects.

6.3.2 Tidiness

With a centralized Orchestrator, where all test scripts of different technologies are co-located, SeAT successfully solves the biggest problem encountered by the practitioners — the disorganization and distribution of simulators and test scenarios. Consequently, it encourages the practitioners to do it right from the beginning which in turn reduces the possibility of troublesome searching process for the corresponding test scripts when needed.

The industrial respondents showed positivity towards the solution:

> *"We would have everything (simulators and scenarios) in one place which makes it easy to find."* — System Developer

> *"If you use this, you have to put everything in one place otherwise it won't run, you will encourage people to do it right. The testers don't need to hunt for the files."* — System Developer

6.3.3 Test reporting aid

With the support from the system, test reporting is made easier for the testers. Such improvement is much appreciated since reporting and tracking test results in the current situation is time consuming and error prone. There is room for enhancement which is to send these results to the test management system. To express the simplicity of reporting test results, a senior system developer said:

> *"Instead of reading the paper and doing the notes on a paper just click click click and read what's on the screen."* — Senior System Developer

6.3.4 Simplified test cases

As defined in the framework, a test step is a single action performed on a specific target machine. Thus, a step has to be straightforward and unambiguous. On the one hand, such a step can be easily automated without breaking the whole structure of a test cases. Therefore, the automation process can be done incrementally which is suitable for a legacy system. On the other hand, such convention motivates a better way of writing test cases where test steps are granulated, sequential, clear and simple as a system developer put it:

> *"It could encourage better way to write them. Now it can be that someone writes a step in a test case that is too huge. This encourages them to write clearer test cases."* — System Developer

6.3.5 Knowledge gap resolution

As mentioned earlier in the challenges of the current test activities, experience with the system and the simulators is crucial for the testers today to do their job with the ATC system. Testers need to understand what is going on, which makes not anyone can do the testing, especially since test specifications and other documentations are not guaranteed to be up-to-date.

By introducing SeAT — which improves productivity of the testing process, requires test cases to be written in a clean, clear and simple manner and organizes every technologies in one place — prior knowledge about the system needs not be extensive.

> *"You don't need to be an expert of the system. You can get anyone to do the testing."* — System Developer

6.3.6 Productivity enhancement

The interviewees agreed that the system boosts the productivity of the testers in the testing activities. One of the reasons is that they no longer need to refer to a lengthy test specification when executing the test cases step by step. In combination with the encouragement to write clear and simple test cases, the risks of misinterpretation are mitigated. Thereupon, testers are more focused on the testing itself. Even though pure manual test cases are performed using the framework, a sequence of pop-up windows instructing how to execute the test steps gives testers a better medium than referring to a stack of papers. The product development manager said:

> *"It goes much faster. There's much less room for misinterpretation, misreading and misunderstanding of the test cases. It's also when you are reading test cases, you need to jump back and forth between the system under test and the things you are reading and you forget where you were. Here, it takes step by step and it's much easier to see where you are. The quality of the testing process itself could be improved."* — Product Development Manager

In this particular project, setting up the system to the correct state before each test case is time-consuming. These setups most of the time repeat the same pattern and are a combination of different technologies. With the framework, such setups can be fully-automated and it definitely speeds up the activities. It was confirmed by a senior system developer:

> *"A lot of time you spend on setting up the simulator and finding files. So that could be very large improvement."* — Senior System Developer

6.3.7 Test process reformulation

To employ SeAT into the current projects at the company, changes in the process of writing tests are inevitable. At the moment, the practitioners use DOORS — a requirement management application developed by IBM to organize requirements and test cases specifications. Therefore, integration to such system is needed and approval from the high level managers is required.

> *"The way we should do it is to work with DOORS and enter the test cases there. But DOORS is really cumbersome. I think that we would prefer to do it with this tool and probably make some import into DOORS."* — Product Development Manager

However, as the senior system developer declared, this problem is more or less out of scope of the framework since it is more or less their issue of using SeAT in an efficient way. Nevertheless, the product development manager was confident about the potential of SeAT and believed that other engineers would prefer using a unified framework to create and execute their tests.

> *"People would like to write test cases using this tool."* — Product Development Manager

6.3.8 Test automation encouragement

Having had a platform to unify test technologies, testers are encouraged to write fully-automated test cases. Otherwise, there is still a need of manual interventions and testers have to be present during the testing process. One system developer claimed:

> *"It is a big difference between everything automatic and having one manual step in the middle."* — System Developer

The product development manager affirmed: *"Writing automation is more fun than testing itself."* Such motivation leads to the fact that: *"People would be more inclined to write good automated tests, which improves the quality, and (they) would feel confident that the things written is going to work."* — he added. Understandably, most developers have a mindset of not doing things manually. If they are provided a useful tool to assist their job, they are willing to change their way of working.

6.3.9 Towards continuous integration

SeAT provides a good platform on which test technologies can be unified and automated or at least semi-automated. In all cases, it is a step toward continuous integration — which is now performed mostly on low levels of abstractions, namely unit level and integration level. It opens up a possibility of increasing high level testing in projects which adopt continuous integration ideology. A test specialist endorsed the capability of SeAT:

> *"It's good for smoke tests, which are boring to run. And you have to run them repetitively."* — Test Specialist

7

Discussion

This chapter discusses the results presented in chapter 6 and connects them to earlier research literature. In the begining (section 7.1) the focus falls on the challenges of using different test technologies and practices together. Later (section 7.2) the method of unifying different test technologies and practices is discussed. The last section (section 7.3) points out the effects of the unified test framework on the validation and verification activities.

7.1 RQ1 — Challenges of using different test technologies and practices together

All software companies today apply software testing techniques in order to increase the quality of their products and to ensure that the client requirements are met. Test automation [10] is essential and can have positive impact in many areas [50], it could potentially save money and solve many testing problems. In spite of its benefits, automation is not a *silver bullet* [50], meaning that it experiences a lack of generic applicability. Some of the results of this research also show that there are cases where automation cannot be applied or its cost, in terms of time and effort, is too high. The researchers observed a case where automation is inapplicable: the tester had to go to a server room to disconnect a network cable to test the redundancy.

Testers, who want to harvest the benefits from automation, but stumble upon cases where full automation is impossible, are left with no choice but to combine manual practices with automation tools in their test cases. Although, manual and automated testing are seemingly contrasting, they could be complementary [10]. On the other hand, their common use introduces new problems to the testers. Leither's research [10] has identified some challenges in using manual and automated testing, however it is limited to unit testing only. There are other works, which focus on comparison of automation tools [5], problems with automation [50], or challenges in software testing as a whole [11],

but none of them seems to refer to the problem of unification. This thesis has identified four categories of challenges in combining automation tools on different abstraction levels and manual testing.

In a company which has vast amounts of automated scripts, written in different languages and supported by various tools, it could be a nightmare for the testers to find the right script, the right tool and start a scenario. Serious amount of time is wasted in performing these actions over and over again, especially during regression testing [51] where tests are run after each and every test. All problems which are related to (a) the connection between a test step and an executable script and (b) running such a script with the help of a proper tool have been classified as **Untidiness**.

In addition, having a wide range of automation tools leads to a large amount of cognitive information that has to be kept in the testers' minds. Specifically, the testers need to be familiar with the exact steps to run a test script, and they have to know the features of the tools and details about the scripts. This problem concerns mostly inexperienced testers. However, in many companies the testing process is performed once in a few months, therefore even skilled testers have to reminisce such information needed to perform test cases. Moreover, familiarity of the system is needed in many occasions, because the test cases are not descriptive enough. Challenges related to these problems have been classified as **Insufficient knowledge**.

As the researchers observed the current practices in the case company, reporting the outcome of a test case is performed manually via paper form. Then the information is input electronically into an archive system. Many companies use this approach, which is time consuming and it leads to unnecessary effort from the testers. On top of that, this method is error-prone because the tester might omit writing down the result of an action or connect the output to the wrong step. Moreover, if any unexpected output — which is not specified in the test case — occurs, it requires additional effort from the tester to document. Similar problems have been categorized as **Test reporting problems**.

One of the reasons to automate test execution is to allow the tester to proceed with other tasks while the tests are being executed. Without test orchestration this is highly unlikely. In order to observe the output from a test script the tester has to wait for its execution, especially in the case of smaller scripts. Moreover, in case of distributed environments different scripts have to be executed on different machines, which leads to interacting with several workstations and possible waste of time. This research has concluded that even in case of automation of the test scripts the testers are all the time involved in the process of testing and are not able to refer to other tasks or to perform many test cases at the same time. This problem is specifically severe when most of the test steps in a test case are automated but few of them or even one is manual. Then the presence of the tester is required all the time because he is unaware at what time exactly he has to perform this manual step. Problems referring to unnecessary waste of time are classified as **Inefficient time exploitation**.

In conclusion of this section, there are advantages and disadvantages related to all of the testing techniques and different combinations amongst them is crucial for the successful testing of a system. Consequently, there are many issues bound to the mixed

testing activities. All of the identified problems lead to inefficiency in the process of software testing. The time taken to perform a test suite is prolonged and testers' effort is being wasted.

7.2 RQ2 — Method for unifying different test technologies and practices

Challenges found in the prior exploration were derived into functional and quality requirements of the developed framework. This section provides a mapping between the challenges and the supported features of SeAT.

— **Untidiness** was solved by having a centralized Orchestrator to store all test scripts in an organized manner.

— To assist testers who have **insufficient knowledge** about the system and simulator, Orchestrator was built with a descriptive user interface. Every test case is decomposed into test steps, each of which is a single interaction with a particular machine. Such approach ensures the simplicity of the tests.

— Regarding **test reporting problem**, SeAT provides a way for the tester to trace for exceptions and responses from the automation tools and also a place to write down their exploration while performing the tests.

— Pertaining to the **inefficient time exploitation** problem, SeAT accelerates the test process by encouraging automation and running all consecutive automated steps at once.

Although the study was conducted in a legacy system within a company, it is applicable for other systems as well since the framework was also tested against a web project. Besides, with the well-defined interfaces and the employment of dependency injection technology, SeAT is highly extensible. The results of such quality attributes are mentioned in the framework assessment (subsection 6.3.1) Therefore, the framework is generalizable.

In the course of investigating an appropriate way to unify different test technologies and practices, some patterns were detected.

— *Granularity of the steps:* a way to unify them is to make the test steps as small as possible. There should not be any mixture of several test technologies or complicated interactions with several machines within one test step. Instead, each step consists strictly of interaction(s) on a particular level against a particular target machine. Such an approach guarantees the automatability of the test steps. It also enables the reusability of the test scripts since many test cases repeat the same patterns or have mutual subsets of test steps which are ideally reused throughout a test suite.

— *Simplicity of the structure:* test cases should be written in a clear and simple manner with sequential steps. The needs of complicated structures such as loops and conditions can be compensated by highly reusable test scripts. Such high level logic can, as well, be delegated to the extensible executors.

— *Amongst adjacent abstraction levels:* unification amongst different levels of abstraction is one of the core ideas of the framework. However, it is unrealistic to unify all of them or any arbitrary combination of the levels in one test case. For instance, it is irrational to have a step which runs on the unit level in the middle of other acceptance level test steps. Therefore, combining adjacent levels — such as acceptance level and system level or system level and integration level — is considered more appropriate. When it comes towards the unit test level, the combination is more restricted since ideally unit tests are independent. Nevertheless, looking at another angle of the unification, it provides a good way to systematise and arrange all test cases in a centralized system which serves as a living documentation of the application status.

— *Semi-automation on the high levels:* Semi-automation achieves its best value when performed on acceptance and system levels as it accelerates manual work. On the unit level, semi-automation is possible but it does not add value to the process. Preferably, code analyses, inspections and reviews are conducted separately from automated unit tests in order to minimize the running time of such tests.

In addition, one of the encountered problems in the studied company was the length of the test cases. It is due to the fact that one test case usually covers many requirements. Such approach of verifying a set of requirements with one test case is not academically appreciated but it helps the testers to save time. This is due to the fact that setting up the system into a correct state before any test cases is time consuming, thus, combining test cases omits the repetition of setting up the system multiple times. Nevertheless, the bulky test cases result in a situation where some sequentially written test steps are independent of one another. SeAT supports independent steps by allowing the user to specify if a step is critical or not. If two steps are dependent the first one is defined as critical and if it fails the test execution does not continue to the next step. In the test cases more complex control statements (such as loops and conditional statements) are not recommended by Google [52] and there was none similar found in the case company test specification. However, although SeAT does not support control statements, their usage could be delegated in the automation scripts if needed.

7.3 RQ3 — Effects of the unified test framework on testing activities

Testing is the most widespread validation and verification technique for assuring adequate software behavior and it involves a lot of activities, thus, it faces a number of challenges [11]. SeAT affects the testing process by focusing on these challenges. As all

of the interviewees agreed, it makes the testing process less time consuming and simplified by organizing the resources needed for execution of test cases. As the research of Cervantes [23] implies, the spare time could be used for more testing activities. This leads to an increase in the test coverage, hence, the quality of the software is improved [23].

As previously concluded, there are some cases where manual testing in inevitable, moreover, Leither has shown in his research [10] that unification of manual and automated testing produces positive results. This fact is dependent on the disadvantage of manual testing, that is, it allows testers to perform only sequential test runs. This leads to an increase in the time needed to complete the tests [23]. However, manual testing is indeed needed so that developers can use their reasoning to develop useful test cases and construct complicated input data. Therefore, SeAT incorporates the use of manual testing and automation test tools together and it is a useful **Productivity enhancement** tool as several of the experts in the case company have pointed out. The pop-up window in SeAT indicates the time for manual action and allows the testers to multitask during the process of execution of test scripts because they do not need to wait for the result. SeAT simplifies the whole process and bypasses the need for the testers to refer to the test cases written in paper form. This feature results in less errors during testing because jumping back and forth between the system under test and the test specification is a potential place for mistakes as the product development manager said during the interviews. It was even suggested by one system developer that SeAT could be used even if there is absolutely no automation in the test cases and the framework still be helpful.

On the other hand are the cases where tests are fully automated, meaning that there are scripts covering all the steps in a test case. This results in extensive amount of test scripts, and as observed, the scripts are spread in different locations and run with the help of various tools. However, many papers have concluded that it can potentially save time. Cervantes [23] states that automation provides the testers the capability to perform parallel testing and that automated tests are not dependent on testers' working hours and they could run constantly. The problem that a large amount of scattered scripts and tools brings is being mitigated with the use of SeAT. All of the interviewees responded that the framework solves the problem with multiple-automation-technology integration. Moreover, the centralized Orchestrator (section 5.2) stores all scripts in one place and executes them on the correct client machine. The testers no longer need to keep different scripts on different machines. Before this practice was hampering the execution of scripts as well as their maintenance. One system developer referred to this problem as *"file hunting"* and several developers were quite positive about the **Tidiness** that SeAT brings to the testing process. A similar tool has been developed during Leitner research [10], although it only supported unit testing and lacked the distributed capabilities of SeAT.

In addition, SeAT has proved to be a good reporting tool. The testers had difficulties to input the result from a manual or automated test step into an appropriate software. Currently they are documenting the results in paper form which is time consuming and requires effort, particularly in the case of a distributed environment. SeAT successfully

solves this problem and introduces a structured, organized and simple way of documenting the results. The results are gathered and stored in the Orchestrator and could be easily extracted. However, additional research and implementation about the supported formats is needed.

Furthermore, the testing process suffers from insufficient knowledge from the testers' side because the test automation implementation requires available experts [13]. In addition, the test case specification is ambiguous and requires prior knowledge of the system, which new testers usually lack. SeAT could be used as a helping tool where all tests are created by experienced developers and then run and controlled by employees with less knowledge of the system. Several senior testers were anxious about this capability of SeAT since they can use the tool to mitigate some of their responsibility to the junior testers or developers. A possible feature is that a screen-shot of a successfully executed manual step could be taken during the creation of a test case and shown to the tester who runs the test case. This screen-shot will be complementary to the step description. This approach could help the inexperienced testers to easily recognize when a step fails or passes without any knowledge of the system under test.

SeAT affects the process of creation of the test cases and it encourages automation. This encouragement is due to the fact that many of the problems that are faced during unification of test activities are solved by SeAT and the users would feel confident that the written test cases are going to work. After the evaluation interviews it was discovered that SeAT is perfectly suited for testing during continuous integration process [53] which requires tests to be run after every build of the system since companies want to shorten the feedback loop to detect errors early on in the development process. This approach requires repetitive run of the test cases and any excessive time is multiplied by the number of runs. Vitally, automation is a must to achieve continuous integration and most of the automated tests, particularly in the projects at the studied company, are in the low levels of abstraction, namely unit level and integration level. However, the validity of a single component or a combination of components do not justify the correctness on the system level or acceptance level. During regression acceptance test, many of the test scenarios are repeated throughout the project lifetime to ensure a minimum level of acceptance. If such tests are automated, they can also be integrated to continuous integration and run all the time. Because SeAT strives to reduce the excessive time which is wasted in putting the system under test in the correct state, "file hunting" and reporting the results of test runs its benefits are harvested best in continuous integration environment. This observation is supported by the opinion of a test specialist in the case company who also suggested that SeAT could be also beneficial while performing smoke tests [54], whose main characteristic is to be able to be performed fast. Consequently, testers only need to perform exploratory testing, which is believed to be a good way of detecting defects in the system. The repetitive tedious job is automated.

8

Conclusion

This research identifies the problems from unification of different test practices, provides a solution and studies effects of the latter on the verification and validation process in a software company. Through a design science research a set of unification challenges (section 6.1) are identified, an IT artifact is developed (chapter 5) and effects of the implementation of this artifact in the case company are observed (section 6.3).

In relation to RQ1 — challenges of using different test technologies and practices — four categories have been defined: 1) **Untidiness** is a category consisting of all problems arising from the missing connection between a test case step and an executable script, and disorganization of the supporting automation tools. These problems include excessive time and effort. Regression testing seems to suffer the most from untidy testing process. 2) Due to **Insufficient knowledge**, which concerns mostly inexperienced testers, the testing process is hindered and errors are often made. 3) The testers are often obliged to waste efforts on writing the results of test cases on paper and then transferring this information to electronic systems, especially in the case of manual testing. These efforts require time, they are error-prone, hence, this study categorize them as **Test reporting problems**. 4) There are other cases where time is being wasted during the testing process. Good example is the case where automated and manual testing are interleaved. Although automation is supposed to save time it actually requires the presence of a tester during execution. Therefore **Inefficient time exploration** is defined as a category of problems related to unification.

Based on the requirements elicited from the challenges a method for effectively unifying different test technologies and practices has been developed and implemented as a framework — SeAT. This framework has been applied in the appropriate environment and its attributes and functions have been assessed by experts and students. Additionally, the effect that SeAT introduces to the testing process of a company has been studied. SeAT has proved that it confronts to the business needs by solving the identified

challenges of unification. It is a good tool for reporting and it keeps the testing process more organized. Furthermore, SeAT assists the testers when they experience lack of knowledge, thus, reducing the errors during testing. The framework helps its users to save time and allows them to effectively multitask since it does not require their presence at all time. By supporting the testers in different ways, SeAT affects the whole process of testing and most importantly encourages automation.

The challenges identified by this thesis can be used as a checklist by companies which struggle with the use of many automation tools and also use manual testing. The method for unification allows practitioners to develop similar frameworks and tools and it helps them understand which are the key concepts of developing such framework. SeAT is potentially a very useful tool for productivity enhancement, although some additions and changes might be required. This thesis contributes to the existing body of knowledge within the field of software engineering. It compromises with the lack of studies related to the unification of software testing methods and practices.

Although students were involved in the research, the case company was only one and it would be beneficial to extend this study to other companies as well. More challenges of unification could be identified in other domains and the effects of the method for unification can be further investigated with experiments on other systems and testing processes. Out of scope was the complexity of the test cases and how companies write their test suites. Further studies in this area will improve the applicability and efficiency of SeAT. Finally, in order to make SeAT a finished tool usable by experts in industry, some changes are required on its user interface and functionality such as: 1) connecting to test management tools, reading test cases from them or reporting back; 2) changing SeAT's clients so that they connect to the Orchestrator automatically; 3) SeAT is built on Microsoft .NET framework it requires .NET framework to run. Thus, supports for operation systems different than Windows will be much appreciated.

Bibliography

[1] A. Hevner, S. Chatterjee, Design research in information systems, Integrated Series 9 in Information Systems 11, (2010).

[2] P. Bourque, Guide to the Software Engineering Body of Knowledge (SWEBOK), The IEEE Computer Society, Angela Burgess, 2004.

[3] J. Ryser, M. Glinz, A practical approach to validating and testing software systems using scenarios, QWE '99, 3 rd International Software Quality Week Europe, 1999.

[4] C. Kaner, J. Falk, H. Q. Nguyen, Testing Computer Software, 2nd Edition, Van Nostrand Reinhold, New York, 1993.

[5] E. Borjesson, R. Feldt, Automated system testing using visual gui testing tools: A comparative study in industry, Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference, IEEE, Montreal, QC, 2012, pp. 350–359.

[6] C. Kaner, Exploratory testing, Quality Assurance Institute Worldwide Annual Software Testing Conference, Orlando, FL, 2006, pp. 1–9.

[7] W. W. Royce, Managing the development of large software systems, Proceedings of IEEE WESCON 26 (August), 1970, pp. 1–9.

[8] N. Dhingra, Mayank, Contingent study of black box and white box testing techniques, International Journal of Current Engineering and Technology 4.

[9] Ieee standard for software unit testing, 1008-1987 (1986).

[10] A. Leitner, I. Ciupa, B. Meyer, M. Howard, Reconciling manual and automated testing: The autotest experience, System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference, IEEE, 2007, p. 261a.

[11] A. Bertolino, Software testing research: achievements, challenges, dreams, Future of Software Engineering, 2007. FOSE '07, IEEE, Minneapolis, MN, 2007, pp. 85–103.

[12] M. E. Khan, F. K., A comparative study of white box, black box and grey box testing techniques, International Journal of Advanced Computer Science and Applications, (2012).

[13] I. Sommerville, Software engineering, 8th Edition, Pearson Education Limited, 2006.

[14] G. Adzic, Specification by example, 2011.

[15] I. Singh, A mapping study of automation support tools for unit testing, Mälardalen University School of Innovation, Design and Engineering, 2012.

[16] A. Adamoli, D. Zaparanuks, M. Jovic, M. Hauswirth, Automated gui performance testing, Software Quality Journal (2011) 1–39.

[17] J. Andersson, G. Bache, The video store revisited yet again: Adventures in gui acceptance testing, Extreme Programming and Agile Processes in Software Engineering (2004) 1–10.

[18] A. M. Memon, Gui testing: Pitfalls and process, IEEE Computer 35 (2002) 87–88.

[19] E. Alégroth, F. R., H. Olsson, Transitioning manual system test suites to automated testing: An industrial case study, Software Testing, Verification and Validation (ICST), IEEE, 2013.

[20] M. Fagan, Advances in software inspections, IEEE Transactions on Software Engineering (1986) 744–751.

[21] R. W. Selby, V. R. Basili, F. T. Baker, Cleanroom software development: an empirical evaluation, IEEE Trans. on Software Engineering, (1987).

[22] T. Gilb, D. Graham, Software Inspection, Wokingham: Addison-Wesley, (1993).

[23] A. Cervantes, Exploring the use of a test automation framework, Aerospace conference, 2009 IEEE, IEEE, Big Sky, MT, 2009, pp. 350–359.

[24] J. Racino, Policy, program evaluation and research in disability: Community support for all, 1999.

[25] E. Stake, The Art of Case Study Research, SAGE Publications, 1995.

[26] S. Lauesen, Software Requirements Styles and Techniques, Biddles Ltd of Guildford and King's Lynn, Edinburgh Gate, Harlow, 2002.

[27] Iso standard for software engineering – product quality, parts 1, 2 and 3, 9126-9126 (2003).

[28] A. Cockburn, Agile Software Development, 2nd Edition, Highsmith Series Editors, 2009.

[29] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empirical Software Engineering 2 (2009) 131–164.

[30] P. Runeson, M. Höst, A. Rainer, B. Regnell, Case study research in software engineering, Wiley, 2012.

[31] Yin, R. K., Case study research: Design and methods, Sage, 1994.

[32] C. Robson, Real World Research: A Resource for Social Scientists and Practitioner-Researchers, Wiley, 2002.

[33] R. K. Yin, Case Study Research: Design and Methods, 3rd edition, SAGE Publications, 2003.

[34] A. Shenton, Strategies for ensuring trustworthiness in qualitative research projects., Education for Information 2 (2004) 63–75.

[35] I. Boxill, C. Chambers, E. Wint, Introduction to social research with applications to the caribbean, University of The West Indies Press, (1997).

[36] R. Feldt, A. Magazinius, Validity threats in empirical software engineering research-an initial survey, SEKE (2010) 374–379.

[37] S. A. McLeod, Observation methods (2015).
URL `http://www.simplypsychology.org/observation.html`

[38] P. Kruchten, Architectural blueprints — the "4+1" view model of software architecture, IEEE Software 12 (6) (1995) 42–50.

[39] M. B. Juric, A hands-on introduction to bpel (2009).
URL `http://www.oracle.com/technology/pub/articles/matjaz_bpel1.html`

[40] A. Karande, M. Karande, B.B.Meshram, Choreography and orchestration using business process execution language for soa with web services, IJCSI International Journal of Computer Science Issues, 2011.

[41] M. J., K. Pant, Business Process Driven SOA using BPMN and BPEL: From Business Process Modeling to Orchestration and Service Oriented Architecture, Packt Publishing, 2008.

[42] R. Perrey, M. Lycett, Service-oriented architecture, Applications and the Internet Workshops (2003) 116–119.

[43] T. Andrews, F. Curbera, H. Dholakia, Business process execution language for web services, version 1.1, bea, ibm, microsoft, siebel systems, 2003.

[44] Microsoft, Iis 7.0 and your hardware (2015).
URL `https://msdn.microsoft.com/en-us/library/cc268240.aspx`

[45] Oracle, Ofm system requirements (2015).
URL http://docs.oracle.com/html/E38687_01/12c_fusion_requirements.htm#SYSRS2998

[46] H.-P. D. Company, Unified functional testing (uft) (2015).
URL http://www8.hp.com/se/sv/software-solutions/unified-functional-automated-testing/

[47] EtestingHub, Software testing tools-test director (2012).
URL http://www.etestinghub.com/testdirector.php

[48] M. Fowler, Inversion of control containers and the dependency injection pattern (2004).
URL http://martinfowler.com/articles/injection.html

[49] R. C. Martin, The dependency inversion principle, C++ Report 8 (1996) 61–66.

[50] D. Hoffman, Cost benefits analysis of test automation, Software Quality Methods, 1999.

[51] J.-M. Kim, A. Porter, G. Rothermel, An empirical study of regression test application frequency, The Journal of Software Testing, Verification & Reliability 15 (2005) 257–279.

[52] Google, How to write good test cases (2015).
URL https://code.google.com/p/robotframework/wiki/HowToWriteGoodTestCases

[53] M. Fowler, Continues integration (2006).
URL http://martinfowler.com/articles/continuousIntegration.html

[54] Standard glossary of terms used in software testing, version 2.4, International Software Testing Qualifications Board.

[55] H. Y. Yang, E. Tempero, H. Melton, An empirical study into use of dependency injection in java, Software Engineering, 19th Australian Conference, IEEE, Perth, Australia, 2008, pp. 239–247.

[56] H. Olsson, H. Alahyari, J. Bosch, Climbing the "stairway to heaven" a multiple-case study exploring barriers, Transition from agile development towards continuous deployment of software (2012) 392–399.

[57] A. Albreshne, P. Fuhrer, J. Pasquire, Web services orchestration and composition — case study of web services composition (2009).

A

Interview Questions

Interview questions	Categories
1. What is your role in the organization? 2. How many years of IT industrial experience do you have? 3. Have you done automatic or manual testing before? 4. Are you familiar with the following types of testing: • Unit • Integration • System • Acceptance	Background
RQ1 — What challenges are experienced in practice when different test technologies and practices are used together for system verification and validation?	
5. Do you think that automatic testing could help you in your work? • Make it less time consuming • Make it possible to multitask • Raise the quality • Other?	Challenges and Potentials

6. Have you used tools for automatic testing? Which ones? If YES, can you think of any problems? Examples? 7. Have you used manual testing and automation testing tools together in one test case? If YES, what are the benefits? Any problems? 8. Have you used different tools for automatic testing together in one test case? Which ones? If YES, what are the benefits? Any problems?	
RQ3 — How does a unified test framework affect the verification and validation activities in industrial practice?	
9. Do you think that this framework could fit into the testing process of your company? Where? • During the creation of the tests • During the repetitive usage of the tests • Other?	Applicability
10. Do you think that the framework solves the problems with the usage of manual testing and automation testing tools in one test case? Which ones and to what extend? 11. Do you think that the framework solves the problems with the usage of different tools for automatic testing in one test case? Which ones and to what extend?	Validity
12. From 1 (easy) to 10 (difficult), how difficult is it to understand the purpose of the framework? 13. From 1 (easy) to 10 (difficult), how difficult is it to create test cases and related test steps? 14. From 1 (easy) to 10 (difficult), how difficult is it to get the result from the execution of a test case?	Usability
15. From 1 (easy) to 10 (difficult), do you think that a new test technology could be easily added to this framework? Example?	Extensibility
16. In your opinion could this framework save time and effort for the testers? Why do you think so?	Productivity
17. How do you think this framework could be improved? 18. Other observations? Anything we have not covered?	Open-ended questions