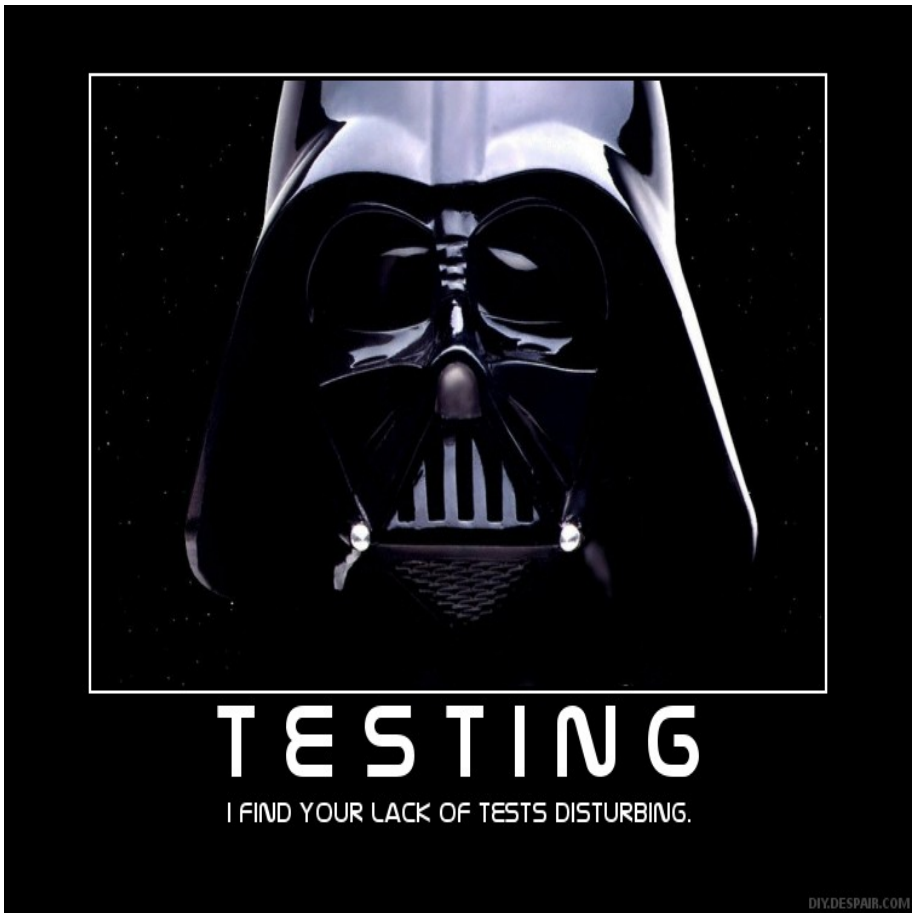
Automated GUI Testing in Industry

Version of January 2, 2012



Leon van Delft

Automated GUI Testing in Industry

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Leon van Delft
born in Vlaardingen, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

DSW Zorgverzekeraar
s-Gravenlandseweg 555, 3119XT
Schiedam, the Netherlands
www.dsw.nl

Automated GUI Testing in Industry

Author: Leon van Delft
Student id: 1185624
Email: leonvandelft@hotmail.com

Abstract

Every change to an application should be followed by a regression test if we want to assure that the application keeps functioning in conformance with its requirements. Executing such a test manually can be a time consuming and boring task. Time pressure often causes the regression test to be skipped, or only partially executed. Executing these regression tests automatically can be a solution to this problem. However, large costs, especially for maintenance, are involved in automating a big regression test. Also, many potential pitfalls can cause test automation projects to fail.

A Dutch health insurance company called DSW has tried to incorporate automated GUI testing in their test process before. A testing tool called WinRunner was used to automate the regression test of a mainframe application called ISIS. We identified the causes for failure of this project to be high maintenance costs due to GUI object recognition problems, ignoring possibilities for reusing overlapping parts in test cases, and using hard-coded values in test scripts.

These findings alongside with the pitfalls and guidelines identified by other researchers were used to avoid project failure and create a maintainable automated GUI testing solution. This automated GUI testing solution is called DARTH VADER.

Key points in the design of DARTH VADER are separation of test case data and test case logic, an easily maintainable GUI mapping, the use of a tool to maintain the test cases, and restoring a fixed set of test data prior to each run. We concluded that the improved bug detection and work satisfaction, and the fact that no changes to the application under test are taken into production without being tested, outweighed the costs of developing and maintaining DARTH VADER.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dipl-ing. M.S.Greiler, Software Engineering Research Group, TU Delft
Company supervisor:	Bauke Aukema, DSW Zorgverzekeraar
Committee Member:	Phd. H. -G. Gross, Faculty EEMCS, TU Delft
	Dr. Ir. W.-P. Brinkman, Faculty EEMCS, TU Delft

Preface

I would like to thank everyone who supported me during my thesis. Special thanks goes out to Jeroen de Haan and Bauke Aukema, from DSW Zorgverzekeraar, who trusted me enough to support me in this project. Arie van Deursen and Michaela Greiler helped me out in everyday problems during this research, and I would also like to thank them for doing so.

Not restricting myself to mentioning only people who have contributed to my research, I would also like to thank my parents for providing a house for me to live in during my study, giving me a place to fall back on. I would also like to thank God for having all the resources available it takes to follow education on this level.

Leon van Delft
Schiedam, the Netherlands
January 2, 2012

Contents

Preface iii

Contents v

List of Figures vii

List of Tables viii

1 Introduction 1
- 1.1 Testing by using the GUI . . . 1
- 1.2 DSW company profile . . . 2
- 1.3 Prior experience with automated GUI testing at DSW . . . 2
- 1.4 Goal of the research . . . 2
- 1.5 Structure of the paper . . . 3

2 Related work 5
- 2.1 Pitfalls in automated GUI testing . . . 5
- 2.2 Lessons learned from previous application of automated GUI testing . . . 13
- 2.3 Guidelines for test automation . . . 17
- 2.4 Tackling the maintainability problem. . . . 20
- 2.5 Conclusion . . . 25

3 Research Design 27
- 3.1 Research Roadmap . . . 27
- 3.2 General Hypotheses . . . 28
- 3.3 Choosing a test system . . . 29
- 3.4 Refining the general hypotheses for our setting . . . 35
- 3.5 Qualitative considerations . . . 36
- 3.6 Experimental setup . . . 37
- 3.7 Conclusion . . . 39

4 Choosing a GUI testing tool **41**
4.1 Choice for testing tool - custom creation or use of existing solution 41
4.2 Tool selection 42
4.3 Selecting the most suitable tool 44

5 Designing the testware **47**
5.1 Requirements for DARTH VADER 47
5.2 Choice for a layer model 48
5.3 Test set representation 49
5.4 Implementing DARTH VADER 57
5.5 R2D2 maintenance tool 62
5.6 Conclusion 65

6 Experimental results **67**
6.1 Bug detection 67
6.2 Maintenance needs 73
6.3 Test run setup 74
6.4 Test run analysis and feedback 75
6.5 Work satisfaction 76
6.6 Manual testing effort 77
6.7 Automated GUI testing development time 78
6.8 Summary 79

7 Lessons learned **81**
7.1 Automating manual test scripts 81
7.2 Fixing test data 82
7.3 Custom GUI elements 83
7.4 Compliance with guidelines 84
7.5 Understandability of test cases 85
7.6 Using new testing libraries 85
7.7 Putting the experimental results and lessons into perspective 86

8 Conclusions and recommendations **89**
8.1 Refuting or confirming the hypotheses 89
8.2 Answering the research question 94
8.3 Recommendations for extending automated GUI testing at DSW 94
8.4 Recommendations for future research 95
8.5 Conclusion 97

Bibliography **99**

A Appendix: Schematic view of thread of control **103**

B Appendix: Requirements Definition Document **107**

List of Figures

2.1 ATF Architecture 23

3.1 Timeline of the case study 39

5.1 Initial layer overview without the custom extension's layers. 48
5.2 Schematic view of the test case logic file with one blueprint highlighted. 52
5.3 Schematic view depicting how blueprints can be nested. 53
5.4 Schematic view of the structure of the test case data file. 54
5.5 Schematic overview of the input architecture depicting how an example test case data and blueprint are merged. 56
5.6 Complete architecture. 58
5.7 R2D2's user interface. 63

6.1 Well formed error message 70
6.2 Malformed error message 70

A.1 Thread of control within one class containing a method labeled with [TestMethod] 103
A.2 Thread of control between multiple classes that contain methods labeled with [TestMethod] 104
A.3 Thread of control for one test case (starting in a method labeled with [TestMethod]) 105

B.1 Requirements Definition Document - page 1 108
B.2 Requirements Definition Document - page 2 109
B.3 Requirements Definition Document - page 3 110
B.4 Requirements Definition Document - page 4 111
B.5 Requirements Definition Document - page 5 112

B.6 Requirements Definition Document - page 6 113

List of Tables

2.1 Project stage(s) referenced to by the ”Stage(s) of occurrence” column in Table 2.2. 8
2.2 Table of 'common pitfalls' taken from the report of Christer Person and Nur Yilmazturk. 9

3.1 Considerations for selecting a test system, based one the interests of DSW. . . . 31
3.2 Test system selection requirements. 32
3.3 Obtained research data for each project stage 39

4.1 Differences between Squish and MTM/CUT. 45

5.1 Contents of one line of the test case logic file. 53
5.2 Contents of one line of the test case data file. 54

6.1 Test results for different test runs in the June release. 68
6.2 The causes of failures being missed by the human test run. 68
6.3 The causes of failures being missed by the automated test run. 71
6.4 Test results for different test runs in the September release. 71
6.5 The causes of failures being missed by the human test run. 72
6.6 The causes of failures being missed by the automated test run. 73
6.7 Number of test cases executed/failures reported for both releases. 79
6.8 The causes of failures being missed by either the human or automated test run for both releases. 79
6.9 Estimated time investments required for applying automated GUI testing or manual testing. 80

8.1 The estimated costs and benefits connected to the choice of whether to apply automated GUI testing or not. 91
8.2 Activities needed to repair a bug that is detected by either the manual or the automated test run. 93

Chapter 1

Introduction

Software systems fulfill a crucial role in most modern companies. The requirements these systems have to fulfill arise out of the specific task these systems have within the company. Companies' software systems are constantly changing to meet new customer needs, to stay ahead of competitors, or to comply with new legislation. Since these software systems are critical, it is essential to assure they work in conformance with their requirements. To assure this, the software has to be tested every time it is changed. In this way, software testing is a critical part of quality management within the software development cycle.

The need to automate testing is constantly rising for a number of reasons:

- Since the degree to which companies are automating their processes is constantly growing, the size of their software systems increases over time. Naturally, the size of the test set needed to test this software grows as well. A bigger test set takes more time to execute.
- Testing is a very precise task. Errors in software may be triggered only by applying a very specific sequence of actions on the Application Under Test (AUT).
- The pace at which software is changed, is very high. Companies have to adept quickly to changes. This means the frequency of having to execute the test set is high.

1.1 Testing by using the GUI

Within the spectrum of automated testing, unit testing aims to test different components in isolation, instead of testing a software system as a whole. Since GUI components cannot be tested in isolation, it is not suitable for testing the GUI [21]. Automated GUI testing solves this by interacting with the GUI in the same way a human does. This means the mouse and keyboard are controlled to simulate a human user. However, many pitfalls are known which can cause automated GUI testing to fail.

The growing need for test automation, combined with the presence of these pitfalls, makes automated GUI testing an interesting subject for investigation. Substantial payback can be gained by companies when it is successfully applied. This means companies are

willing to invest time and money on automated GUI testing in order to be able to apply it in a successful manner.

1.2 DSW company profile

An example of a company that might benefit from automated GUI testing is DSW. DSW is a health insurance company with almost 375,000 insurees. They are in close cooperation with a partner, Stad Holland (which has over 75,000 insurees), sharing the same building, customer desk, and Information Technology (IT) back end. A large, innovative IT department is one of the principles upon which DSW bases its success. This is underlined by the fact that 70 out of the 400 employees are working for the IT department.

Previous work by Kishen Bhaggan [3] has shown that DSW has a mature software development process (a necessity for applying automated testing [10]). DSW has to handle a large amount of incoming and outgoing data: insurance claims, approvals of claims, customer registration and communication, and so on. Administrative IT systems handle the bulk of this data flow. these systems are critical and have high quality requirements. This make testing an important process.

Most test activities at DSW are manual, especially for legacy systems. However, new projects incorporate some degree of automated testing: unit testing, load testing and performance testing are often used. Apart from these forms of testing, DSW employs mostly functional testing, which is exclusively done by humans at this point. With innovation being one of the cornerstones of the IT department, DSW is willing to invest in this area if it leads to saving time or improving product quality and work satisfaction.

1.3 Prior experience with automated GUI testing at DSW

DSW has had prior experience with applying automated GUI testing for functional testing. A tool called WinRunner was used to capture and replay test cases. The test suite proved to be very fragile. Every new version of the AUT resulted in large parts of the test suite becoming unusable, and in need of maintenance. Maintenance took so much time that the test period often had to be extended because the automated test was not yet adapted. Eventually, the project was abandoned. This left DSW in a situation where there was still demand for automated GUI testing, but the costs for applying it in this form were too high.

1.4 Goal of the research

At this moment, there is still demand for automated GUI testing, but prior experiences with maintainability problems have discouraged its application. This discouragement is aggravated by the fact that maintainability has also proven to be a problem for other companies that seek to apply automated GUI testing [17, 19, 20]. However, since the demand for automated GUI testing still exist, we are interested in answering the question:

How can automated GUI testing be applied in a way that benefits DSW?

The last attempt to automate GUI testing failed because of maintainability problems. Solving these problems might result in a solution that is beneficial. We will try this by creating a maintainable automated GUI testing solution. We do so by applying and improving on ideas created by fellow researchers, and by identifying and avoiding the problems which led to the failure of the WinRunner project. We choose a test system to do a case study in order to analyze the effort needed to create and maintain our solution. The results are monitored as well to give an indication of how successful our solution is.

1.5 Structure of the paper

This Section explains the layout of the paper. Academic work that is related to our thesis is listed and discussed in Chapter 2. The design of the research, and selection of the test system, is presented in Chapter 3. Chapter 4 discusses our choice for a testing library to build our automated GUI testing solution upon. Chapter 5 discusses the design and implementation of the testing tool that we made on top of the chosen library. Chapter 6 presents the experimental results obtained from our case study. Chapter 7 explains the lessons we have learned, and lists threats to the validity of these results. Finally, conclusions and recommendations are given in Chapter 8.

Chapter 2

Related work

In this Chapter we will analyze academic work related to automated GUI testing. We do so to learn techniques and guidelines that can help us in creating a successful automated GUI testing application.

We first analyze the failure of the WinRunner project. To this end, possible reasons for automated GUI testing project's failure that were identified by other authors are discussed in Section 2.1. In Section 2.2 we analyze the failure of the WinRunner project to identify which of these pitfalls caused its failure. To avoid falling into one of the pitfalls, the guidelines composed by various authors, about how to apply automated GUI testing, are presented in Section 2.3. Since maintainability was recognized as the biggest cause for failure of the WinRunner project, we discuss techniques aimed specifically at dealing with maintainability issues in Section 2.4.

2.1 Pitfalls in automated GUI testing

We first want to get an idea at which point the WinRunner project failed. To this end, we read some experience and observation reports of various authors. These authors give an overview of guidelines and pitfalls to take into regard when one wants to apply test automation. An overview of these pitfalls along with identifiers to enable referencing, is given in this Section.

2.1.1 Lessons in test automation

Elfriede Dustin is the author of "Lessons in test automation". She has worked at several companies, where she reviewed the attempts to introduce automated GUI testing. The article is written with the intention that others can avoid the false starts and roadblocks that she has witnessed [9]. The lessons she has learned are grouped into a few categories, and discussed.

Use of various tools to support testing - various tools might be used in order to support the automated testing effort, which can potentially lead to several problems:

PF_1 For metrics which depend on more than one tool's data or the consistency between input data, the integration of tools might be necessary. Especially when tools are from different vendors, this can be a problem.

PF_2 Information that is needed in more than one tool, must be stored in multiple places. This makes maintenance harder because updates to data in once place, need to be propagated to the other replicas as well.

Effort on test automation is focused at the wrong area - Effort can be directed at the wrong area. This can be dangerous in the following manners:

PF_3 Automating test cases might become the goal of test automation, without any regard of whether a given test case is suitable for automation.

PF_4 Automating test cases might consume so much time that other activities are neglected.

PF_5 Automating the "wrong" test cases can result in a test script that is just as complicated as the AUT itself.

PF_6 Different test engineers might use different styles for creating test scripts, without complying to guidelines and standards.

Wrong expectations of test automation/test automation tools - Wrong expectations can result in setbacks in the following ways:

PF_7 Test case creation is often more complicated than tool-vendors claim. More time or better qualified personnel might be needed to automate testing.

PF_8 Training in the usage of tools is neglected in the beginning, making it take even longer before their usage pays back.

PF_9 Testing tools are sometimes introduced to save a project that is behind schedule. This is bound to fail, because getting used to a tool takes time, delaying the project even more.

PF_10 Testers who do not see immediate payback after using a tool, might resist using it.

PF_11 Payback is expected on the short term. However, as with all automation, payback of the initial investment takes more time.

Technical shortcomings - A testing tool might come short in various technical aspects:

PF_12 Reports produced by the tools might not contain data that is useful for monitoring the test progress or test status

PF_13 Third party controls (widgets) are sometimes not recognized by the testing tool. This makes it hard to test parts of the AUT containing those widgets.

Choice for testing tool is not well considered - this can result in project failure for a number of reasons:

PF_14 Some testing tools are intrusive. Meta data has to be inserted into the AUT's code in order for the testing tool to work. Software developers might be reluctant to do this, especially if they did not expect intrusiveness. Choice for a tool that does not require intrusiveness might be a better option in that case.

PF_15 Some testing tools might not be well-scalable (because they are backed up by an Access database for example). If scalability is required, this can lead to problems.

PF_16 If a testing tool is purchased before the architecture of the AUT is known, it might restrict AUT's architecture. For example, a testing tool might not work on programs created with a certain graphical toolkit, because it cannot recognize its GUI elements. This restricts the AUT from being constructed using that toolkit.

Versioning problems with testing tools - Version increments of test tools might lead to problems in the following ways:

PF_17 More than one version of the tool might be in use at the same time at one company. This can lead to compatibility issues.

PF_18 A newer version of a tool might have other environment requirements than an older one. A testing tool might require MS Exchange to function correctly in a new version of the software, while it did not need that in the previous version, for example. This might force a company into buying software they might not want to use themselves.

2.1.2 Pitfalls and Strategies in Automated Testing

Cem Kaner has noted that many myths are created around automated GUI testing. This often leads to failed attempts to apply it. In this article, Kaner reports on a meeting with 13 experienced testers. Together, they discuss the pattern of success and failure in automated GUI regression testing. He reports on several drawbacks, and strategies to overcome them [15].

Fundamental drawbacks - A number of drawbacks exist, which are fundamental to automated GUI testing. It is important to keep those in mind when choosing for automated GUI testing:

PF_19 Automating a test is more expensive than just running it. Automation being a tenfold more expensive than running it, is not uncommon.

PF_20 Because automating a test case takes time, there is a delay between wanting to run the test case, and having the test case available.

PF_21 Using automated GUI regression testing as the only method of testing does not yield a high enough probability to find errors. This is aggravated when only simple test cases are automated.

Apart from these fundamental issues that one has to take into regard, some approaches towards the implementation of automated GUI regression testing are also potentially dangerous.

Wrong expectation management - Expectations of automated testing can be misplaced in the following manners:

PF_22 Management expectations might be that payback for applying automated GUI testing occurs soon. This usually is not the case. Resetting this expectation is advised, as well as trying to create some early payback to keep a positive attitude among management.

PF_23 Good testers do not necessarily have to be good test automaters, and good programmers do not have to be good testers. Neither of these groups can be expected to function without the help of the other group.

PF_24 *Hard-coded values* - Using hard-coded values in captured scripts while this is not necessary makes scripts vulnerable to changes in the AUT.

2.1.3 Establishment of Automated Regression Testing at ABB: Industrial Experience Report on Avoiding the Pitfalls

The ABB company has had two projects to establish automated GUI testing. Christer Persson and Nur Yilmazturk, who have noted the high failure rate of automated testing projects, report on these two attempts. They created an overview of all the so called ”common pitfalls”, and recommend to incorporate the possible occurrence of those into formal risk management [19]. Each of these pitfalls is most likely to occur in one or more of the Automated Testing Life-Cycle Methodology (ATLM) stages. The stages of the ATLM are depicted in Table 2.1.

Table 2.1: Project stage(s) referenced to by the ”Stage(s) of occurrence” column in Table 2.2.

Stage ID	Stage name	Stage description
1	Decision to automate test	In this phase, the benefits and drawbacks of automated testing are evaluated and management support is acquired.
2	Test tool acquisition	Testing tools are selected and evaluated in this step. Organizational work should be done before this is started.

Continued on next page

Table 2.1 – continued from previous page

Stage ID	Description	Stage(s) of occurrence
3	Automated testing introduction process	Project teams doing software development are introduced with the process. Normal workflows for developing and deploying software are evaluated to find out were to place automated testing, and then the test process is adapted accordingly. Test tools are verified to work in the development environment.
4	Test planning, design, and development	This phase consists of test planning design. Test approaches are evaluated, effort is estimated and roles and responsibilities are set.
5	Execution and management of tests	Here the actual testing begins. Results of code execution have to be analyzed and defects in the AUT have to be tracked down.
6	Test program review and assessment	Collected metrics are reviewed, and suggestions for improvements to the AUT are given.

Table 2.2 presents a description of the pitfalls that have been identified, as well as a reference to the ATLM stage(s) in which this pitfall is most likely to occur.

Table 2.2: Table of 'common pitfalls' taken from the report of Christer Person and Nur Yilmazturk.

Pitfall ID	Description	Stage(s) of occurrence
PF_25	Uncontrolled introduction of test automation in immature organizations - "automated chaos yields faster chaos"	1
PF_26	Using automated test for sheer defect testing is expensive and inefficient	4
PF_27	Automating test for unstable test objects. Changes to requirements entail changes to test cases	4
PF_28	Introducing too many test cases at once - defect overflow is extremely dangerous.	4
PF_29	Test specification does not exist. Scripting without a previously prepared test specification is time consuming.	4
PF_30	Insufficient configuration management for test environment and testware	5
PF_31	Insufficient defect tracking - tracking of test object, test environment, and testware defects is not supported by a tool.	5

Continued on next page

Table 2.2 – continued from previous page

Pitfall ID	Description	Stage(s) of occurrence
PF_32	Defensive attitudes towards automated testing. Management is not committed to initiate introduction and establishment of automated testing.	1, 2, 3, 4, 5
PF_33	Poor or unknown test coverage of automated regression tests.	4
PF_34	Too much work, too little regression - a lot of work with test cases that are not repeated for a sufficient number of times	1
PF_35	The testware is not handled with the same care and professionalism as the shipped code - ignoring that test automation is software development.	5
PF_36	Capture and Replay functionality of test tools encourage bad testware architecture by default. Test scripts are easily ”hard-coded” due to wrong use of ”capture & play” features. This makes the test scripts sensitive to changes in user interface.	4
PF_37	Equating automated testing with manual testing instead of considering it as a complement to manual testing.	5
PF_38	A lot of maintenance of GUI regression tests due to an uncontrolled changing of the user interface.	5
PF_39	Establishment without any clear test strategy hence, generation of test code that no one really understands.	4
PF_40	Automated tests need more human intervention than expected. For example, test results should be analyzed, and test scripts should be maintained.	5
PF_41	Believing in automated tests reduce the test personnel. In fact, setting up an automated test bench requires more resources than setting up a manual test, and also, resources are needed for analysis and maintenance.	4
PF_42	The organization does not understand the concept of automated regression testing. There sometimes is a belief in miracles.	1, 5
PF_43	The used test tool does not support the needed functionality.	2
PF_44	The organization mandates 100 percent automation. However, automated testing is not always equal to better testing.	4
PF_45	Considering automation as a way to ”significant labor cost savings”	1
PF_46	Requirements of the test team skills are set too low and too narrow. It is common to believe that the auto test team does not need any technical skills in programming, testing, or project management.	3

Continued on next page

Table 2.2 – continued from previous page

Pitfall ID	Description	Stage(s) of occurrence
PF_47	Reports produced by the tool might be useless, i.e. do not present the information needed by the organization.	5
PF_48	Too late tool training in the process despite the fact that often, large amount of training is needed with automation of testing.	3
PF_49	Exotic target with poor tool supply. For example, problems with the tool recognizing third-party controls.	4
PF_50	Lack of test development guidelines thus, endangering reuseability, repeatability and maintainability of testware.	4
PF_51	Early automation often brings forth problems related to unstable test objects. Late automation finds bugs too late.	5
PF_52	Iterative software development models are misunderstood. A lot of changes are made within an iteration and this probably will cause problems to the Automated Testing	1
PF_53	Lack of controlling the decisions to automate or not. To automate wrong can create costs without any benefits.	4
PF_54	Lack of test oriented design. This causes the test object not to support an efficient and stable testing.	4
PF_55	Bad or none modular test block design and poor reuseability.	4
PF_56	Separation of test and development in combination with none or bad communication.	5
PF_57	No or weak definition of the project terminology early in the project	3
PF_58	Lack of adjusted overall test strategy covering both manual and automated testing	5

2.1.4 Observations and Lessons Learned from Automated Testing

This paper is written by Stefan Berner, Roland Weber and Rudolf Keller. They report on the observations they made in about a dozen projects which involved automated testing. They were involved in these projects in various roles: test manager, software engineer, tester, and so on. The paper presents lessons learned and best practices which are composed based on their experience in these projects [22].

Test automation strategy is often inappropriate - a number of often made mistakes which could potentially lead to a project's failure, fall under this category:

PF_59 Using the wrong level of testing (unit, integration, system, and so on) or wrong test type (functional, performance, and so on) to test a certain part of a system. Testing program logic is best done using unit testing, and system testing is more suitable to

GUI based testing, for instance. Picking a wrong testing level or wrong test type can considerably increase the effort needed for testing.

PF_60 Return on investment (ROI) is often expected very early, and mainly in the form of savings on human labor. However, ROI usually takes more time. Also, ROI calculations should include shortened release cycles and improved testing depth. Testing depth is improved because manual testers are freed from boring, repetitive tasks, and are able to invest more time in deeper, more complicated test cases.

PF_61 A good testing strategy should include multiple types and levels of testing. Since different types and levels of testing are suited for different kinds of testing, a mixture is the best for effective testing.

PF_62 Tools should be used wherever they could simplify the testing process. Execution of test cases is straightforward, but configuration and installation procedures of testing environments are often not taken into consideration as candidates for automation.

PF_63 *Tests are far more often repeated than one would expect* - The authors observed that in nearly all projects, the automated tests were run far more than expected. This influences the payoff in automating a test case. Each time it is run makes the initial investment more cost-effective.

PF_64 *The capability to run automated tests diminishes if not used* - The authors noted that in a changing test environment, the effort needed to run and maintain an automated test set increases disproportionally fast when not run frequently. This because the integrity and understandability of the test set decreases fast as time progresses and the AUT changes.

PF_65 *Automated testing cannot replace manual testing* - In the experience of the authors, automated testing re-validates certain parts of the AUT. It does not look for errors by trying to make the AUT crash, as is sometimes done by human testers when applying destructive testing. It does not fulfill the same role, and should not be used as a replacement.

PF_66 *Testability is a usually forgotten non-functional requirement* - Testing an application can be much harder when testability is not taken into regard while it was being designed and implemented. An application can come short on (among others) the following aspects when it comes to testability:

1. Design for testability of the AUT:
    a) It might not be designed in a layered or modular design. This makes it very hard to mock parts of the system to be able to test certain parts in isolation.
    b) It might give incomprehensible error messages, or have no error handling at all.
    c) It might contain too little checked assertions to allow for self-diagnosis.
2. Design of the test environment:

a) Non-transparent environment makes it hard to verify system state in order to find out whether test cases failed or not.

b) Lack of access to infrastructure like configuration management of the AUT makes it hard to configure the test environment.

Testware maintenance is hard - Maintenance of testware is often problematic because of a number of reasons:

PF_67 The design of testware is often not taken as seriously as that of ”normal” software. Documentation is often lacking, which results in a poorly understandable application.

PF_68 Test cases often share many similar parts, but are automated without exploiting reuse to implement these similar parts. This is essentially code-duplication. Code changes that involve duplicated parts, have to be carried out in more than one place. This makes maintenance more costly and error prone.

PF_69 The testware itself is often not tested thoroughly. This makes it hard to distinguish errors in the AUT from errors in the testware.

2.2 Lessons learned from previous application of automated GUI testing

In this Section, we analyze which of the pitfalls mentioned in 2.1 caused the WinRunner project to fail. To this end, the WinRunner project is described in 2.2.1. The pitfalls that contribute to the WinRunner project's failure are listed in 2.2.2. Finally, the aspects in which the WinRunner project was successful are discussed in 2.2.3.

2.2.1 Previous application of automated GUI testing

In September 2001, a mainframe application called ISIS (Integraal Systeem Inkomensverzekeringen Stadrotterdam, or Integral System Income insurances Stadrotterdam) was in use at the partner of DSW (which was called *Stad Rotterdam* back then, now it is called *Stad Holland*). It was a program used to store and manipulate insurance policy data for all kinds of insurances (health, car, cancellation, and so on). A terminal emulator was used to connect and interact with the system. Testing the system was done manually in an ad-hoc fashion. There was no testing script for ISIS. While the system was still being developed, automated GUI testing was chosen to be incorporated into the testing process. WinRunner was chosen as the application for running the test cases.

Test cases were recorded by a capture facility, and could be played back in a number of variations by using different sets of parameter values, which could be stored in a Comma Separated Value (CSV) file. Two external developers were hired to do the development and maintenance of the test set. The WinRunner test set was run in parallel with the human testing effort. Test reports were passed to the ISIS developers, which could then repair the ISIS system in case errors were found. The initial goal was to do a daily execution of the test set. However, when the GUI of ISIS was changed, it often took quite some time to

repair the test set again. The delay caused by this repair sometimes resulted in the manual testers having finished their testing before the automated test set had run once.

In September 2002, about one year after the project had taken off, it was abandoned because management did not expect return on investment anymore.

2.2.2 Identifying causes for failure

Three co-workers who were involved in the WinRunner project were asked to describe the WinRunner project and its failure. Using these descriptions, we have identified which pitfalls caused the WinRunner project to fail. For each of these pitfalls we explain why we have concluded this pitfall to be contributing to the failure of the WinRunner project. We also explain why the cause was labeled as either a major or minor contribution to the project failure. Some of the pitfalls that are very similar, are grouped and described together.

- *The goal is to automated test cases, regardless of their suitability for automation* (**PF_3**) - Some of the test cases for testing ISIS involved complicated logic. Branches in the use cases for the AUT had to be represented in test cases. Some very exotic test cases were also created. The total testing effort might be less when these test cases were executed manually. We labeled this cause as only a minor contribution. When describing what caused the WinRunner project to fail, the time taken to create test cases was not mentioned by anyone. So although the project did fell victim to this pitfall, it did not contribute greatly to its failure.

- *Test scripts can become just as complicated as the AUT itself* (**PF_5**) - The reason for falling into this pitfall is similar to **PF_3**. WinRunner has to be able to handle complex test cases. This made the input format much more complicated than desired. Some test cases were stored in multiple CSV files. Rows from the first file had to be combined with rows in the second file to handle branching in the test cases. This was described as a cause for many maintainability issues, since it was very hard to keep an overview of how test cases worked. Since it was mentioned by all of the three people involved in the WinRunner project as being a problem, this pitfall is regarded as a major cause for the project's failure.

- *Test case creation or maintenance is often more complicated than tool vendors claim* (**PF_7**) - Although the concept of WinRunner is simple, the practical application of WinRunner was harder than expected. WinRunner's user guide, which is over a thousand pages long, can be seen as an indication of its complexity [1]. Initially, one specialized WinRunner developer was hired to create and maintain the test set. Later on, this was even expanded to two full time hired developers. Although test case creation was harder than expected, hiring the developers solved this problem. Hiring external developers is usually more costly though. Also, after these temporary employees leave, most of their knowledge about the project and application is lost for DSW. However, they did not leave DSW before the project was canceled, so this did not contribute to the problem. Because of this, this pitfall contributed little to the project failure.

- *Testing tools do not support the needed functionality, for example, GUI element recognition* (**PF_13** and **PF_43**) - WinRunner has two modes of capturing a playback session: context-sensitive and analog [1]. The context-sensitive mode stores GUI elements in such a way that minor changes to the GUI elements do not break the test script. This is because the GUI elements can still be located by applying a smart search algorithm that searches for an element that matches the stored GUI element's properties in the object repository. However, the screen capturing capabilities of WinRunner on mainframe terminal emulator screens are problematic, since not all objects are recognized correctly. Analog mode had to be used instead of context sensitive mode. Using analog mode, the script just captures the exact user actions, and replays them when the test case is run. This greatly limits the capabilities of the testing tool, and is labeled as a major contributor to the failure of the WinRunner project.

- *Testers and test automators cannot be expected to function without the help of the other group* (**PF_23**) - The developers that worked on the WinRunner project, did not cooperate closely with the testers of ISIS. They worked in parallel. A test set was created, run, and the results were presented to the ISIS developers. This might have had influence on the test set's quality. However, this pitfall is only marked as a minor contributor. Although the developers were no full time testers, and did not cooperate with the manual testers, they were specialized in test automation. This makes it reasonable to believe they have a solid understanding of testing, making the lack of cooperation with testers less important.

- *Making use of capture & replay functionality or including hard-coded values in the test script in other ways, makes these scripts vulnerable to changes in the AUT* (**PF_24** and **PF_36**) - Because of the limitations mentioned in **PF_13**, the analog mode of recording was used. Two options are possible for selecting fields and navigating them: clicking the mouse on a fixed coordinate (relative to the applications window's position), or using the tab-key to navigate. Both methods are very fragile. Moving a GUI element to another location, or switching the order of the fields, is enough to break the test script. Developers chose to use the tab-key method. Still, the number of tab presses needed to navigate to a certain field is a hard-coded value. Inserting a field or swapping places with another field, is enough to break the test script. A large amount of maintenance was mentioned to arise from this issue, so we label it as a major contributor to project failure.

- *Unstable test systems in general, and unstable GUI's in particular, can cause high maintenance needs for test scripts* (**PF_27** and **PF_38**) - In 2001, while the WinRunner test set was created, the ISIS application was still under development. Much new functionality was added, and existing functionality was changed frequently. Especially considering the problems regarding hard-coded values (mentioned for **PF_24** and **PF_36**), this is considered as a major cause for project failure.

- *Creation of an automated test script is time consuming when no test specification exists* (**PF_29**) - During the time of the WinRunner project, at DSW, testing was

approached less professionally compared to the current approach. No test scripts existed, and the WinRunner developers used their best judgment in creating test cases. Since the time taken to create test cases from scratch was never mentioned by anyone as a problem, and we judge a WinRunner specialist experienced enough to create test cases in reasonable time, this pitfall is regarded as a minor one.

- *Establishment without any clear test strategy* (**PF_39**) - There was no clear testing strategy. Strategic decisions were left to the good judgment of the involved developers. But even though the strategy was lacking, the described problems of the involved people do not suggest this to be a major problem, so we label it as a minor one.

- *Bad or none modular block design, failing to reuse overlapping parts of test cases* (**PF_55** and **PF_68**) - There was some reuse in the recorded test cases (by supplying them with numerous parameter sets through CSV files), but this was only an option for filling in field values. This meant that sets of test cases could be created when all of the test cases consisted of the same sequence of actions, with the only difference being the desired field values. When sequences of actions were shared by more than one set of test cases, no reuse of this sequence of actions was possible. Furthermore, there were many interdependencies among test cases, making it very hard to move or remove test cases without influencing the outcome of other test cases. Since the involved project members reported large problems regarding bad reuse causing maintenance, this issue is marked as a major contributor to project failure.

- *Lack of an overall testing strategy covering both manual and automated testing* (**PF_58**) - No strategy for combining human and automated testing existed. Manual testers knew what was tested automatically, but there was no coordinated plan to categorize test cases on certain conditions, in order to divide them between human and automated testing. However, this was never mentioned as a big problem by the involved team members, so we label it as a minor cause.

- *Testability not taken into account while developing the AUT* (**PF_66**) - Testability was not considered while developing the ISIS system. The continuous change of the system was noted as a problem for applying automated testing, especially in combination with the poor performance of the testing tool. So we mark this as a major cause.

We made a summary that gives an overview of the major contributors (with the duplicates removed) to the project's failure. The pitfalls these items are based on, are also presented. Two types of issues are distinguished, technical, and non-technical.

Technical:

1. GUI controls are not recognized by the testing tool. - **PF_13**, **PF_43**

2. Using hard-coded values in captured scripts. - **PF_24**, **PF_36**

3. Bad design of the test set, ignoring possibilities for re-use. - **PF_55**, **PF_68**

Non-technical

1. Automating complex test cases, making the test script complicated as well. - **PF_5**
2. Automating test cases for a system that is
   a) Continuously undergoing change in requirements or GUI. - **PF_27**, **PF_38**
   b) Not designed for testability. - **PF_66**

2.2.3 Successful aspects of the WinRunner project

Even though the WinRunner project failed, several aspects were successful. We consider these to get a more balanced overview of the WinRunner project. Also, we seek to incorporate these positive aspects in our new solution. The most prominent advantages are those inherent to automated testing: speed of execution, precision and repeatability. Application specific advantages were also mentioned: the application of data-driven testing and good error reporting facilities.

2.3 Guidelines for test automation

The authors who described the pitfalls also gave guidelines and recommendations for applying automated GUI testing. We created an overview which will be used to minimize the chance of project failure due to the pitfalls. The guidelines and recommendations are preceded by an identifier, which can be used for referencing a certain guideline.

Lessons in test automation [9] - Dustin describes a test improvement process to improve the quality of the testing process. The process focuses on:

G_1 Raising corrective action proposals immediately when the test program's performance is at stake.

G_2 Collecting metrics to pinpoint problems during test execution and review the test quality.

G_3 Document the lessons learned and metrics collected.

G_4 There should be a focus on opportunities for improvement in the next iteration, instead of focusing on who is responsible for a certain failure.

Pitfalls and Strategies in Automated Testing [15] - Strategies advised by Kaner focus on expectation management (to prevent too high expectations), and some technical issues:

G_5 Use data-driven testing in case there are large numbers of repetitions of almost identical test cases.

G_6 Create or use a framework that acts as a library for utility functions, and enables reuse of often-used unified tasks.

G_7 Kaner strongly advises to regard test automation as normal software development when it comes to planning, setting requirements and using coding standards.

Establishment of Automated Regression Testing at ABB: Industrial Experience Report on Avoiding the Pitfalls [19] - The authors do not present their guidelines for avoiding each pitfall separately. Instead, guidelines are given for each of the ATLM stages. Complying to these recommendations should prevent (or lower the chance of) the involved pitfalls from making the project fail.

1. Decision to automate test

   G_8 Having a mature organization when it comes to handling process improvements. This greatly improves the chance of successfully overcoming the challenges involved in the introduction of automated GUI testing.

   G_9 Guarantee management commitment to ensure the availability of human and economical resources.

   G_10 Being well aware of the advantages and disadvantages of automated testing to prevent it from being applied in a wrong way.

   G_11 Provide cost-benefit analysis to be able to motivate the need for the project.

2. Test Tool Acquisition

   G_12 Testing tools should be evaluated to find out whether all of the required functionality is included. It is important shortcomings are noted early in the project.

3. Automated Testing Introduction Process

   G_13 Much terminology is involved in automated testing, and many different components need to be referenced in a clear manner. Therefore, it is important there is consensus on the terminology used in the project. It pays off to establish terminology early on in the project to avoid confusion.

   G_14 Expertise should be available in the development team. Coaches can be used to help team members with applying specialized knowledge. Care should be taken when using external resources. After they leave, enough knowledge should remain in the development team. Teams should be composed of project leaders, developers and testing specialists.

4. Test Planning, Design, and Development

   G_15 A test plan should be written to prevent an ad-hoc approach in test automation.

   G_16 Test specification should be available to prevent scripting from becoming costly. Existing test scripts can also be used.

   G_17 Test coverage information should be used whenever possible to prevent wasting effort on unnecessary or duplicate tests.

G_18 Testware reuseability, maintainability and modularity should be incorporated in the test script's design whenever possible.

G_19 Capture and replay functionality should be used with care as it can lead to high maintenance later on.

G_20 GUI element recognition by the test tool should be taken into mind when making changes to the AUT.

G_21 In case of an unstable test object, delaying creation of the test script might be worthwhile. Creating awareness of maintenance costs among the AUT's developers is also recommended.

G_22 The AUT should be designed for testability whenever possible.

G_23 Project planning should include resources to be reserved for applying automated testing.

5. Execution and Management of Tests

G_24 Test environment and testware should be put under strict configuration management.

G_25 Defect tracking should be automated to enforce a clear structure in bug-fixing.

G_26 Test reports should provide data that satisfies the information need of all stakeholders. Generation of test reports should be automated as much as possible.

G_27 When introducing automated regression testing, the test strategy has to be reevaluated to optimize the testing effort again.

6. Test Program and Assessment

G_28 Process improvement should be used in a formal, structured way to improve the test process.

Observations and Lessons Learned from Automated Testing [22] - Recommendations are given to evade the pitfalls which are most strongly observed in all of the projects. The recommendations are listed below:

G_29 Define a test strategy and maintenance activities.

G_30 Define goals for the test automation.

G_31 Choose a diversified test automation approach involving different test types and levels.

G_32 Evaluate the automation approach constantly.

G_33 Consider testability of the AUT right from the beginning.

G_34 Daily run and maintain the test set. Immediately repair any test case that does not pass.

G_35 Apply good engineering practices as with any other software development project.

2.4 Tackling the maintainability problem.

Since maintainability was the main issue, we look further for solutions on this issue than just the guidelines mentioned earlier. This Section presents techniques that are specifically aimed at dealing with maintainability issues. This is done to give us an overview of the options we have in the design of our solution.

2.4.1 Research assignment

In the research assignment that was carried out prior to the thesis project, we have also created an overview of techniques that deal with test case repair/maintenance. This Section can be seen in Section 4.3 of the research assignment [24]. The subsections in that Section deal with:

- *Capture/replay - maintenance mode* - Running a test script automatically, and interrupting the run for every error encountered. A user can then interactively fix the script at that point, by recording it again.
- *Mapping GUI elements across different versions of the AUT* - When a test case fails due to a GUI element that is not found anymore in the new version of the AUT, a heuristic is used to create a mapping from GUI elements in the old version of the AUT, to GUI elements in the new version of the AUT. When the mapping is done correctly, this makes the testing tool able to find the missing element again.
- *Event Sequence Repair* - When a test case is no longer executable, parts of the test case are removed until it is executable again.
- *Manual repair* - Manually inspecting test cases which do not run anymore, and repairing them by hand.
- *Actionable knowledge representation* - Storing test cases in an abstract way, and reusing any part of a test case that is shared by more than one test case. This reduces the number of places where code needs to be changed when the AUT changes.

The enumerated methods are discussed in more detail in the research assignment itself.

2.4.2 GUI Testing Made Easy

In this paper Alex Ruiz and Yvonne Wang Price describe what a testing tool should offer to simplify GUI testing [20]. Robustness and maintainability are taken as the two main focus areas for a library of functions to be developed. This library is to be used for automated GUI testing in Java Swing GUI applications.

Although DSW does not do any developing in Java, the main focus of the paper is on principle issues, which are the same in any graphical toolkit.

First, the authors identify some critical requirements that a GUI testing library should provide. They are preceded by an identifier again, to enable referencing:

TT1 GUI tests need to tolerate changes in the position, size and layout of any GUI component. Changes in the appearance of an application must not break any GUI tests.

TT2 GUI tests should have a reliable way to find GUI components. For example, a test should always be able to locate a specific button or text field in the GUI to test, regardless of how many times such a test is executed.

TT3 We do not need to check default component behavior. For example, we should not check that a button behaves like a button. This is the GUI toolkit provider's job. Our job is to test the program's logic.

The authors give some recommendations on how to create such a library, and how to create an environment in which it can be put to good use. These recommendations are discussed below.

TT4 The Model-View-Controller design pattern is recommended to be followed to separate business logic from user interface. This way, a better separation is possible between unit tests to test program logic, and GUI tests to test the GUI.

TT5 Provide test case-writers with a tool that makes it easy to write test cases.

TT6 The API of the library should be concise and easy to read.

TT7 The library should provide enough information about the GUI under test to help developers troubleshoot test failures.

With these requirements and recommendations taken into mind, the authors made a GUI testing library, FEST (Fixtures for Easy Software Testing).

They applied the principle of so called "Fluent interfaces". This concept is first described by Martin Fowler [12]. The API is said to be created as a fluent interface by making it easy to read and understand. Short names for fields and buttons, and intuitive identifiers for certain actions are used to achieve this.

Their tool provides component (GUI element) lookup in 3 ways: Lookup by name, type, or a user-defined search criteria. Another important GUI behavior that is taken into mind is the duration of some tasks. After a certain event (the button click that completes a login for example), the user might have to wait for a certain amount of time, before the login is completed. The action following the login has to be delayed until the login is completed. FEST provides the ability to provide time-out values to its actions to set an upper limit to the maximum time before an action is completed.

The most common types of errors for GUI test failure, are considered by the authors:

- Unexpected environmental conditions.
- A GUI component could not be found.
- More than one GUI component matched the given search criteria.

- A programming error

FEST provides features that help the developer with quickly fixing errors for all of these categories, except programming errors:

- *Environmental condition* - Environmental influences can sometimes cause a test case to fail. An example is given in a virus scanner that starts a scheduled run in the middle of a test case. The virus scanner's pop up might overlap part of the GUI, and prevent button clicks to be done in the overlapped area. To be able to more easily detect these errors, a screen shot is taken if the test case fails. In this screen shot, problems in the environment which cause a test case to fail, are usually visible.
- *A GUI component could not be found* - FEST generates ComponentLookupException exceptions to designate that a test case failed because it was unable to find a certain GUI element.
- *More than one GUI component matched the given search criteria* - The ComponentLookupException is used again. The multiple matching elements are displayed in the error message to indicate which elements matched the search criteria.

2.4.3 Design and Implementation of GUI Automated Testing Framework Based on XML

The authors of this paper describe how to design an automated GUI testing framework around an API[2]. Their primary focus is to design a framework that satisfies the following requirements (preceded by an identifier for referencing):

F_1 Test cases should be structured in a way that separates business logic of the AUT, and test case data.

F_2 An easy to use and maintain GUI mapping should be available. Short, descriptive names should be enough to identify GUI components for the test case designer. Maintenance to the mapping should be done exclusively in the GUI map.

F_3 Testware logic and test data should be clearly separated. Ideally, testers can create test sets without interference of the testware developers.

F_4 Options for (hierarchical) re-use of business logic should be provided, to improve maintainability.

They create a solution called Automated Testing Framework (ATF) to apply their ideas. The architecture of the ATF is depicted in Figure 2.1. The workflow within the ATF starts with the data file, parameter file and GUI mapping file being loaded into the system. Then the parsing engine translates the input to an executable script, which is executed with the help of the support libraries.

The main benefit the authors mention for this approach is the division of labor. Testers can create the input files, while developers work on the test operation module. This way,

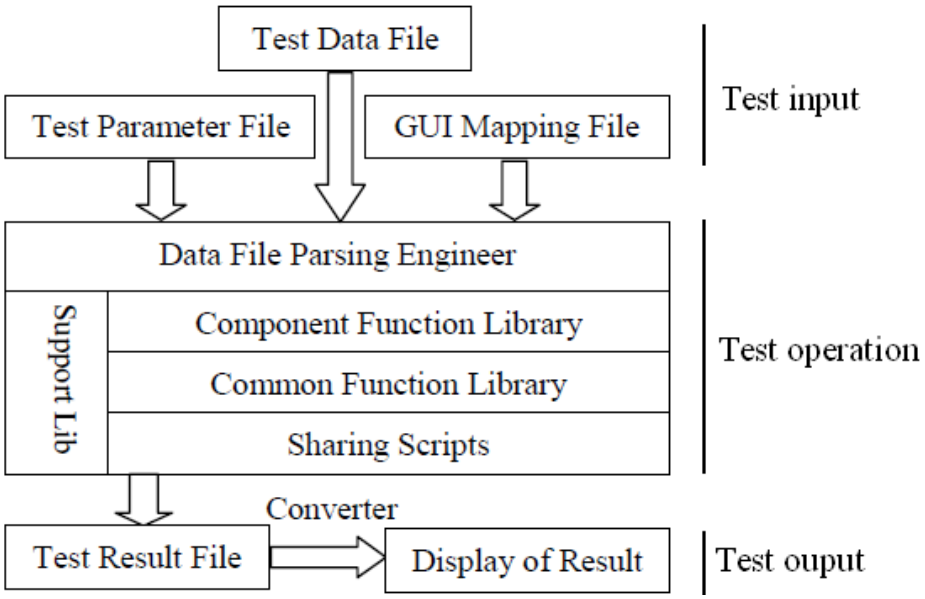


Figure 2.1: ATF Architecture

the testers can focus on test quality, and need less technical insight in how the tool works.

In order to get more insight in the ideas behind the ATF, the different input files, layers and modules that are mentioned, are described in more detail below. First, the three different input files:

Test data file - The data file consists of a series of keywords that express business logic in a structured way. Keywords are stored in a node structure, in XML format. The keywords can be divided into three layers. Each of these layers are used to describe events on a different level of granularity:

- Action node - Describes direct GUI interactions, like clicking a button.
- TestCase node - Contains a group of action nodes to describe more complex GUI interactions. For example, the sequence of actions to log in to a system.
- WorkFlow node - Contains multiple TestCase nodes to provide the highest level of abstraction.

All the possible actions on the GUI have to be stored in this hierarchical structure in the data file. Maintainability is improved by reusing lower level action definitions in multiple higher level (WorkFlow, TestCase) definitions.

GUI mapping file - The names that are used in the data file to describe GUI components, are short, logical names. These names are descriptive enough for humans to be able to identify

the GUI elements. However, these names cannot be used to identify GUI components automatically. The GUI mapping file maps these logical names to identifier strings (strings that contain values of properties of the GUI components). When a GUI component is mentioned in the data file, the test operation module retrieves the identifier string belonging to the GUI component's logical name, and is able to find the GUI component. This structure has three advantages:

- The data file is more readable because of the usage of logical names instead of complicated identifier strings.
- When a GUI component changes and its identifier string needs to be updated, this has to be done only in the GUI mapping file. The data file remains unchanged. This is because the references to the GUI components are done using logical names, which remain the same.
- A multi-language GUI can easily be coped with. One GUI mapping file for each of the languages is enough to keep the test set compatible in each language.

Test parameter file - While the test data file contains the business logic of the test set, the test parameter file contains the test input. The greatest advantage of this separation is that reuse of the same logic contained in one WorkFlow node is possible by using different input data which reference the same WorkFlow node.

The test operation module consists of two parts: data file parsing engine, and support library:

Data file parsing engine - The parsing engine translates nodes in the data file to a test script. Each of the levels of abstraction in the data file (WorkFlow, TestCase and Action) is broken down one by one. WorkFlow nodes are replaced by a string of TestCase nodes, and in turn, TestCase nodes are replaced by a sequence of Action nodes. Finally, Action nodes are translated to calls in the support libraries. Depending on the type of Action node, a support library is chosen. The parsing engine also replaces the logical values in the data file with the concrete values in the GUI mapping file, and parameter file.

Support libraries - The support libraries contain functions that are available to the specifier of the test through Action nodes. Three libraries are used in the ATF framework: component function, common function and sharing scripts. The component function library specifies functions that involve interactions with GUI components: clicking a button, typing in a field and so on. The common function library houses utility functions, like accessing databases, starting applications, handling wait events and so on. The sharing scripts are not explained. The advantage of using separate libraries is not really clear to us. People creating the test cases have to have more knowledge about the implementation of the test operation module this way, since they have to specify which library to use in the action node.

The output module is not described in much detail. What is said is that the output file

is formatted in XML, which in then converted to a graphical output by using Extensible Stylesheet Language Transformations (XSLT [11])

2.5 Conclusion

In this Section we have investigated possible reasons for project failure when trying to introduce automated testing. An overview of reasons observed by various authors is given. This overview was compared to observations done by three involved project members of the WinRunner project. From these observations, a list of possible causes for project failure in the WinRunner project was composed. After further analysis, this list has been summarized to the list below:

Technical:

1. GUI controls are not recognized by the testing tool.
2. Using hard-coded values in captured scripts.
3. Bad design of the test set, ignoring possibilities for re-use.

Non-technical

1. Automating complex test cases, making the test script complicated as well.
2. Automating test cases for a system that is
    a) Continuously undergoing change in requirements or GUI.
    b) Not designed for testability.

An overview of guidelines for applying automated GUI testing is created based on the work of other authors. These authors all have experience in numerous automated GUI testing projects. These guidelines are all possibilities to help us create a solution that does not fall victim to pitfalls mentioned in this Chapter. Since we judged maintainability to be the most prominent problem in the WinRunner project, related work that focused exclusively on maintainability was also analyzed. We looked back at our research assignment, and analyzed and discussed the work of two other groups of authors. These were focused on creating a maintainable and robust GUI testing API, and a GUI testing framework that enable robust and maintainable test sets from being created. Taking their techniques and experiences into mind, will help us avoid the maintainability problem during the design phase of our solution.

Chapter 3

Research Design

This Chapter outlines the process of designing our research. First, the research question is formulated in Section 3.1. This Section also discusses the steps we need to complete before we can formulate an answer to this research question. Section 3.2 presents the general hypotheses, and explains the way these are composed. The requirements for choosing a test system are presented in Section 3.3. The degree to which several systems meet these requirements, and the final choice for one of them, is discussed in this Section as well.

The general hypotheses are refined to be applicable to the chosen test system in Section 3.4. After refining the hypotheses, we realized a quantitative approach to be too narrow. We explain in Section 3.5 how we can improve our research by shifting to a qualitative approach. To formulate an answer to our research question in a qualitative manner, we need data. Section 3.6 describes which data we need, and how we can obtain this data within the course of our thesis project. Finally, Section 3.7 presents the conclusion of this Chapter.

3.1 Research Roadmap

The last attempt at automated GUI testing at DSW failed. This raises the question if it was possible at all to apply automated GUI testing at DSW in a successful manner. A research question is formulated to conclude on this issue:

How can automated GUI testing be applied in a way that benefits DSW?

Certain data is needed to answer this research question. The process to collect the needed data and answer the research question, consists of a number of steps. These steps form the roadmap of our research.

In an attempt to apply automated GUI testing in a way that benefits DSW, we will create an automated GUI testing solution according to the guidelines listed earlier. By following the guidelines, we seek to avoid project failure. Special care is taken to tackle the maintainability issue which caused the WinRunner project to fail. The techniques we analyzed earlier will be applied in our new solution to improve its maintainability. The new automated GUI testing solution should:

- combine existing guidelines and techniques for automated GUI testing, and possibly improve on them.
- be easily maintainable.
- be applicable in a realistic corporate environment.

To answer our research question, we need to weigh the pros of our solution against its cons. We must analyze how much effort it takes to create our solution, and how big the payback is. We need to apply our newly created automated GUI testing solution in a realistic environment in order to do this analysis. To do this analysis, and to answer our research question, we need to complete the following steps:

1. Compose general hypotheses in a way that refuting or confirming them supports a conclusion on the research question.
2. Choose a test system on which we can do a case study to extract the data needed to answer the hypotheses.
3. Refine the hypotheses for the setting of our research (the DSW company, and the chosen test system.
4. Determine which data is needed to answer the refined hypotheses, and devise a plan to gather that data from our test system within the time span of the thesis.
5. Design and implement the testware and test set to carry out the automated GUI testing, and gather the needed data to answer the hypotheses.
6. Formulate an answer to the research question, with the help of the answered hypotheses.

Together, these steps form the roadmap of our research.

3.2 General Hypotheses

To be able to answer the research question, we formulate a number of hypotheses. The hypotheses are set in such a way that refuting or confirming them helps us in concluding on our research question. Our hypotheses will be composed in their most general form in this Section without regarding the setting of the research. We do so because it is important to have general conditions on which we can judge whether an automated GUI testing solution is successful. These principle hypotheses should be applicable to all situations where one might want to analyze whether an automated GUI testing solution is beneficial or not. After a test system is chosen, we will refine our hypotheses.

The main hypothesis we composed is the following:

H1: *Using cost-benefit analysis, we can show that automated GUI testing benefits companies more than it costs.*

This hypothesis narrows down the space for interpretation by specifying on which ground to decide whether an automated GUI testing solution is beneficial or not to cost-benefit analysis. Many aspects to include in the cost-benefit analysis are trivial: amount of work to create and maintain the automated GUI testing solution, number of bugs found, license cost of used tools, and so on.

Some of the aspects are less straightforward though. The amount of human work involved in testing might be decreased by applying automated GUI testing. However, human effort might be saved in many ways, depending very much on how automated GUI testing is applied. To be able to use this aspect in the cost-benefit analysis, we try to answer a hypothesis that summarizes this question first:

H2: *Using automated GUI testing, we can reduce the human effort needed to test an application.*

Apart from the effort needed to test an application, the quality of the test is also important. A test that is easy to write, but uncovers fewer bugs than a more thorough test, is not necessarily better. The number of bugs found by our new solution is easy to measure. However, the number of bugs found, is not the only facet that determines the effort needed to repair bugs. The ease with which one can reproduce a bug, the time it takes to locate the fault in the source code that triggers the bug, the stage of the development cycle the bugs are found in, and so on, all influence the effort needed to repair found bugs. To get a clear picture of whether this part of the cost-benefit analysis changes with the introduction of automated GUI testing, we state another hypothesis:

H3: *Using automated GUI testing can lower the cost to repair bugs.*

3.3 Choosing a test system

Certain characteristics of the test system can have a large influence on the outcome of an automated GUI testing project. This makes the choice of our test system a very important matter. This Section explains this choice for a test system. First, the guidelines describing important characteristics for a test system are presented in 3.3.1. Several constraints arising from the nature of our thesis project put limitations to the choice for a test system. These constraints are discussed in 3.3.2. The interests of DSW that are related to the choice for a test system, are taken into consideration as well in 3.3.3.

A overview of test system selection requirements is composed by combining these guidelines, constraints and interests. 3.3.4 explains the process of combining these guidelines, constraints and techniques, and presents the resulting overview of test system selection requirements. After analysis, the DSW website and Eureka system failed to meet these requirements. This is discussed in 3.3.5. Finally, 3.3.6 discusses the choice for Polis Administratie Systeem (PAS) as our test system.

3.3.1 Considerations based on guidelines and techniques

Several guidelines and pitfalls are applicable to the choice for a test system.

First, the test system should be in a stable state. Especially its GUI should not change too much (**PF_27**, **PF_34** and **PF_38**). When this is not the case, we should consider delaying test case creation until the test system is stable (**G_21**). Second, test specification should exist for the test system (**G_16** and **PF_29**). Third, the test system should be designed for testability (**G_22**, **G_33**, **PF_54** and **PF_66**). Fourth, configuration management should be possible (**PF_30**). Finally, automated defect tracking has to be present (**PF_31**).

3.3.2 Considerations based on constraints and restrictions arising from our thesis project's nature.

Since this is a thesis project, there are some constraints to our research. The main constraint is the recommended time limit for a thesis project, which is 9 months. Since we want to finish within this time period, this puts a limit to the amount of work we can do. It also limits the amount of data we can gather to base our conclusions on.

We cannot conclude on the feasibility of automated GUI testing without knowing how robust the test set is in the face of changes to the AUT. This means we need to select a test system that is scheduled to be updated at least once within the time span of our thesis project.

Since no other people are allowed to cooperate on the thesis project (apart from supervision, and perhaps providing facilities to allow the thesis project to continue), the scope of the project is also limited.

The constraints discussed in this subsection are listed below. They are preceded by identifiers to enable referencing.

C_1 The amount of work we can do is limited.

C_2 The time span to gather data is limited.

C_3 The prospect test system has to be updated within the time span of our project.

C_4 No substantial work can be done by outsiders.

3.3.3 Considerations based on the interests of DSW.

It is in DSW's interest not to make automated GUI testing a critical part of the testing strategy for the chosen test system. This is because we do not yet know whether our solution will be successful and will be continued after the thesis project.

Since the initial investment of creating a automated GUI testing solution is expected to be substantial, it is profitable to use it more than once. Since most of the development is done in .NET, the potential for re-use is the biggest when a solution to test .NET applications is created.

The amount of manual testing that is potentially replaced by automated testing, partly determines how much benefit can possibly be achieved by test automation. It is in DSW's

interest to have the highest possible payback. This means we have to choose a test system that currently needs a large amount of human effort to test.

The higher the number of executed test runs, the higher the potential payback of having the test set automated. This means that the longer the AUT remains in use, the higher the payback is for DSW.

The constraints discussed in this subsection are listed below. They are preceded by identifiers to enable referencing.

I_1 The automated GUI testing solution cannot fulfill a critical part of a system's testing strategy.

I_2 An automated GUI testing solution for .NET systems is preferable over other systems.

I_3 A system that is currently tested using a large amount of human effort is preferred.

I_4 The AUT should still have a long lifetime left.

3.3.4 Combining the different considerations

When taking a close look at the considerations listed in subsections 3.3.2 and 3.3.3, we can see that they take on the form of constraints. Since the pitfalls and guidelines guard us from making common mistakes that can lead to failure, we can regard them in combination with the constraints, as an optimization problem. We have to select a test system that fits within the boundaries of these constraints, while keeping as much pitfalls and guidelines as possible into consideration.

In Table 3.1 we list all of the guidelines and pitfalls. Possible conflicts with constraints are explained.

Table 3.1: Considerations for selecting a test system, based one the interests of DSW.

Guidelines / constraints	**Conflict and resolution**
Test case creation can be delayed if the test system is not stable yet (**G_21**).	Since the time span of our project is limited (**C_2**), this is not an option. We have to choose a test system that is already in stable state.
The test system (especially its GUI) should be in stable state (**PF_27**, **PF_34**, **PF_38**).	As already stated for the previous conflict, we have to choose a test system that is already stable at this moment.
A test script has to be available for the test system (**G_16**, **PF_29**).	The amount of work we can do is limited (**C_1**). This forces us into choosing a test system that has an existing test script.
The test system should be designed for testability (**G_22**, **G_33**, **PF_54**, **PF_66**).	Since the amount of work we can do is limited (**C_1**), and no substantial work can be done by outsiders (**C_4**), we are forced to choose an existing system that is already suited for testability.

Continued on next page

Table 3.1 – continued from previous page

Guidelines / constraints	Conflict and resolution
Configuration management should be possible for the test system and its environment (**PF_30**)	Again, **C_1** and **C_4** force us into choosing a test system that already has options for configuration management.
Automated defect tracking should be available for the test system (**PF_31**).	Again, **C_1** and **C_4** force us into choosing a test system that is already backed up by an automated defect tracking system.

These guidelines and techniques have been adjusted to fit within the constraints imposed by the setting of our project. We can turn them into requirements for the choice of a test system.

Apart from the guidelines and techniques, the chosen test system has to fit within the boundaries of the constraints mentioned in 3.3.2 and 3.3.3. The total set of requirements for selection of a test system is thus formed by the union of these constraints, and the guidelines and techniques in Table 3.1. This union is depicted in Table 3.2.

Table 3.2: Test system selection requirements.

Test system selection requirement ID	Test system selection requirement description
TSS_RQ1	An existing test script has to be present.
TSS_RQ2	The test system has to be in a stable state.
TSS_RQ3	The test system has to be suited for testability.
TSS_RQ4	Configuration management should be possible.
TSS_RQ5	Automated defect tracking has to be present.
TSS_RQ6	The test system has to have a long lifetime left.
TSS_RQ7	Test system has to be updated within the time span of the thesis project at least once.
TSS_RQ8	The automated test set has to be an addition to an already good test plan, the test system should not rely on it.
TSS_RQ9	The test system has to be a .NET system.
TSS_RQ10	A large amount of human effort has to be involved in testing the test system.

3.3.5 Selecting a system to fit the requirements

There were a couple of systems that passed our attention. The most important ones were the website of DSW, the Eureka system, and the PAS system. The final choice was to pick PAS as our testing project. Why the other ones were judged to be less suited, is explained below.

Website - The website is used for four reasons mainly:

1. Present information regarding the DSW company and its insurance policies to the general public.
2. Enable a visitor to take out an insurance policy online.
3. Enable insurees to consult the status of their claims, correspondence, and so on.
4. Enable insurees to carry out mutations to their insurance policies: change the preferred method of correspondence, inform DSW about a change of address, and so on.

A large amount of human testing effort is involved in the testing phase of the website. Furthermore, the quality standards for the website are high since it is the most prominent portal towards the customers, and both cosmetic and functional requirements are critical.

However, a critical drawback is that the website is updated frequently. This in itself is not a problem, but many of the updates are cosmetic in nature. DSW wants to keep up to date with regard to standards of web design, and wants to exploit new technologies (Microsoft Silverlight for example). This continuous adaptation to new technology means the GUI of the website is not stable at all. A new version of the website featuring a complete turnover for the front end of the website is not uncommon. This makes us reject the website as a potential test system because it is in conflict with the requirement for a stable test system (**TSS_RQ2**).

Eureka - The customer service has to have access to many different applications to properly answer questions by insurees. Frequently, multiple systems have to be consulted to be able to get the needed information to answer customer question. This is an elaborate task, which in turn results in longer waiting time for customers, and a higher workload for customer service. The Eureka system is meant to replace all of these separate applications. All of the information that might be needed by customer service, is housed inside Eureka.

Eureka was a starting project, making the life expectancy of the application very long, and enabling testability to be incorporated in its design. This made it a candidate for applying automated GUI testing.

In the end, Eureka was not chosen because it was in conflict with a number of requirements for test system selection:

- A testing script should be available for the test system (**TSS_RQ1**).
- The test system should be in stable state(**TSS_RQ2**).

- The test system has to be updated (i.e., have a testing period) at least once within the time span of the thesis (**TSS_RQ7**), which is unknown at this moment.

3.3.6 The PAS system

The remaining candidate for applying automated GUI testing is PAS. PAS is the system that stores all of the insurance policy data. Searching for insurance policy date and applying changes to this data (registering for a policy, changing an existing policy) are all done through PAS. PAS can be accessed in two ways: through a GUI application, and an interface of webservices.

For each of the requirements in Table 3.2, we discuss whether or not the PAS system satisfies this requirement.

TSS_RQ1 - A regression test script for PAS is available. The script contains a description for executing 600 test cases. The test cases are not specified in much detail. They specify what is to be tested, and what the expected result is. It does not specify exact numbers to use for the test case in most cases.

TSS_RQ2 - The bulk of the work on the PAS project has been done. New functionality is added, existing functionality is improved, and bugs are fixed, but the main workflow in the application remains the same.

TSS_RQ3 - The system is well suited for testability. It has error reporting facilities, logging, and can easily be tested in isolation by mocking external system on which PAS depends.

TSS_RQ4 - The PAS project's development is integrated with Team Foundation Server (TFS, a platform that facilitates collaborated software development [7]). The configuration management is handled entirely by TFS.

TSS_RQ5 - As with TSS_RQ3, TFS handles automated defect tracking.

TSS_RQ6 - The PAS project was first used in the production environment in 2009. This means it is a relatively new system, with years of usage left before the end of life has been reached.

TSS_RQ7 - The PAS system will be updated in March, June and September 2011. So there will be three possibilities for our solution to be put to the test.

TSS_RQ8 - The testing of PAS is all done by hand now for the last couple of testing runs. No new test cases are planned to be introduced during the time span of the thesis project. Since we will be comparing the automated testing with human testing, the human testing is not skipped. Rather, it is run in parallel with the automated test set. This ensures the testing strategy of PAS will not become dependent on the automated GUI testing before the thesis project is finished. Afterwards it might become dependent on our automated GUI

testing solution, but at that time the feasibility question will be answered and the project's continuity guaranteed.

TSS_RQ9 - PAS is developed entirely in .NET.

TSS_RQ10 - The total human testing effort for PAS is big. As mentioned earlier, the test script for PAS contains 600 test cases. The total numbers of letters to be generated in each complete test run, is over 600. Over 130 hours of work are needed to run the entire test set. This means there are lots of possibilities for automated GUI testing to reduce the total testing effort.

As can be seen, all of the requirements for selecting a test system are fulfilled by PAS. This indicates that selecting PAS as a test system gives us a system that can benefit greatly by introduction of automated GUI testing. Also, selecting the PAS system gives us an AUT that will not easily fall victim to the most common pitfalls for project failure with regard to weaknesses in the AUT. Finally, selection of the PAS system as test system fits well within the constraints of our thesis project.

This means we set PAS as our test system, meaning PAS will be the AUT in our case study.

3.4 Refining the general hypotheses for our setting

Now that we have chosen a test system, we can refine our general hypotheses into specific ones. The refinement process of each of the general hypotheses, is described in a separate subsection.

3.4.1 Main hypothesis

We refine hypothesis **H1** by including the company and test system:

H1: *Using cost-benefit analysis, automated GUI testing on PAS benefits DSW more than it costs.*

3.4.2 Large manual regression set

The regression testing script for PAS is very large. Around 600 test cases have to be executed, and over 600 separate letters have to be generated. Manual execution of this testing script takes a long time, making it a very expensive activity. Bugs found during the testing period will result in a new deployment of PAS to the testing environment. Formally, this results in a new version of the software, which would need another testing run. This means that every time a bug is found and solved, another test run is needed, until no more bugs are found.

Running the automated testing tool (at almost zero cost) until we cannot find any more bugs, will reduce the remaining number of bugs in the PAS system. If manual testing is

done after this first automated testing sweep, there is a smaller chance to find additional bugs. This results in fewer deployments, and subsequently, less retesting.

Thus, we can refine the general hypothesis **H2** into one specified for the PAS project:

H2: *Having an automatic regression test set for the PAS project reduces the number of executions of the manual regression test.*

3.4.3 Catching bugs early

The automated GUI test set is run right after the developers have finished developing. Contrary to that situation, human testers only start their testing effort in the testing stage, which is one stage later in the software development cycle.

Since numerous studies have indicated a lower cost for bugs found earlier in the development/testing cycle [4, 5, 25], we expect the average costs of bugs to decrease because of automated GUI testing. Therefore, we come to the refined hypothesis:

H3: *Having an automatic regression test set for the PAS project test lowers the average relative cost of a bug.*

3.5 Qualitative considerations

During the course of our project, we realized a qualitative approach would be more appropriate than a quantitative one. This Section explains why we choose for a qualitative approach. The consequences for choosing this new approach are also explained.

3.5.1 Shifting focus towards a qualitative approach

During the research phase of the thesis, we wanted to focus mostly on verifying whether or not the benefits of our solution out-weighted the costs. The hypotheses we stated were aimed at quantifying the effort needed and the results obtained. We wanted to do this to be able to compare them using cost-benefit analysis.

Gradually, we came to realize that a large number of non-tangible aspects were involved in valuating testing and automated testing. These aspects are hard to include in a quantitative approach. Also, the amount of data we can gather during the course of our research, would not be enough to give a well grounded conclusion.

These considerations made us decide we could improve our thesis by taking a qualitative approach. After consulting our supervisors, this decision was finalized.

We will create a new automated GUI testing solution with a focus on improving maintainability. We will apply this automated GUI testing solution it in a case study, and weigh the pros against the cons in a qualitative way. We will also supply quantitative data to give an indication of the benefits and costs, but the conclusion will be drawn based on qualitative analysis.

3.5.2 Answering the hypotheses in the new approach

From the hypotheses we defined earlier in this Section, especially hypothesis **H1** is very much aimed at quantifying all (or many) aspects of automated GUI testing. Cost-benefit analysis implies this quantification. We will not answer this hypothesis in this manner anymore, but instead give a more qualitative comparison. An analysis is given on the pros and cons, and on which grounds we decided whether automated GUI testing was feasible or not.

For hypotheses **H2** and **H3** we will provide the answers, but instead of quantitative arguments, provide qualitative ones. This because we cannot provide solid quantitative conclusions based on limited experimental data.

3.6 Experimental setup

We need to gather data to formulate an answer to our research question in a qualitative manner. This Section describes which data we will collect during our research, and how we plan to do that. An overview of the data we need is given in 3.6.1. 3.6.2 lays out the roadmap, and 3.6.3 explains how we can obtain our needed data within that roadmap.

3.6.1 Needed data

An overview of the needed data is given below. Every data item is preceded by an identifier to enable us to reference them later on.

The time needed to setup an automated test run (**ND_1**), or a manual test run (**ND_2**).

The time needed to analyze the results of an automated test run (**ND_3**), or a manual test run (**ND_4**).

The number and types of bugs that are missed by:

ND_5 Automated GUI testing but found by manual testing.

ND_6 Manual testing but found by automated GUI testing.

The work satisfaction of the:

ND_7 Human testers with regard to their testing activities.

ND_8 Developers with regard to automated GUI testing.

ND_9 The time needed to maintain the automated test set.

ND_10 The frequency with which maintenance has to be done to the automated test set

ND_11 The time needed to create all of PAS's letters manually.

ND_12 The time needed to run the manual test set.

ND_13 The time needed to create the automated GUI testing solution.

3.6.2 Project roadmap

The PAS project's development is in pace with the so-called release cycle. The release cycle is a 3-monthly cycle consisting of a number of steps which together form the software development cycle. Every release, a number of so called "requests for change" are assigned to be developed. The requests for change are implemented, code-reviewed, tested, acceptance tested, and taken into production in the specified release. A number of project stages has been identified based on this release cycle. The stages are listed below, and a timeline is depicted in Figure 3.1.

1. *Development stage* - This is the stage where our automated GUI testing tool is designed and implemented, and its test set is created.
    - Start of this period - 01-02-2011, the beginning of the practical phase of our thesis project.
    - End of this period - 16-05-2011, the beginning of the June release's testing period.
2. *June release's testing period* - This is the stage in which our automated GUI testing tool is used to test PAS for the first time in an automated way.
    - Start of this period - 16-05-2011, the beginning of the June release's testing period.
    - End of this period - 03-06-2011, the end of the June release's testing period.
3. *Analysis and optimization stage* - In this stage, the results of the June release's test period are analyzed, and possible improvements to our automated GUI testing tool are implemented. The test set is expanded if enough time is left.
    - Start of this period - 03-06-2011, the end of the June release's testing period.
    - End of this period - 15-08-2011, the beginning of the September release's testing period.
4. *September release's testing period* - During this stage, the improved and enlarged test set is used to test PAS automatically for the September release.
    - Start of this period - 15-08-2011, the beginning of the September release's testing period.
    - End of this period - 02-09-2011, the end of the September release's testing period.

3.6.3 Gathering our data within the PAS roadmap

Most of the experimental data that we need to obtain cannot be obtained in an arbitrary stage of the case study. For example, we cannot determine the time needed to create an automated

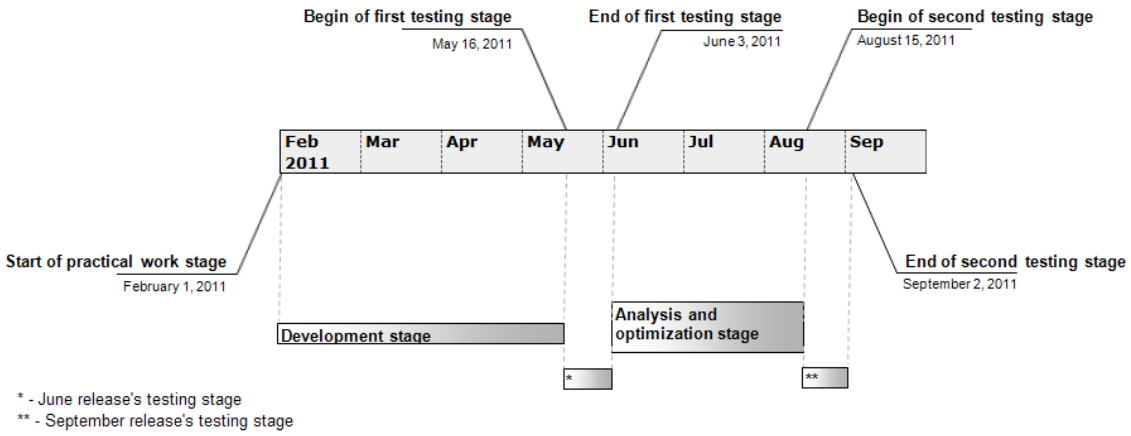


Figure 3.1: Timeline of the case study

GUI testing solution in one of the releases' testing periods. Table 3.3 presents an overview of all of the data we will obtain for each project stage. The choice for these stages is mostly based on the idea of equally spreading the workload over the duration of the case study.

Table 3.3: Obtained research data for each project stage

Stage	**Needed data obtained in this stage**
Development stage	**ND_7**, **ND_8**, **ND_11**, **ND_12**, **ND_13**
First testing stage	**ND_1**, **ND_2**, **ND_3**, **ND_4**, **ND_5**, **ND_6**, **ND_7**, **ND_8**
Optimization stage	**ND_9**, **ND_10**
Second testing stage	**ND_5**, **ND_6**, **ND_8**, **ND_9**, **ND_10**

3.7 Conclusion

In this Chapter we set out a roadmap for our research. The first step is composing a research question. We composed the following question:

How can automated GUI testing be applied in a way that benefits DSW?

Then we distinguished the different steps that we need to take in order to formulate an answer to this research question.

The next step is to compose general hypotheses. Three hypotheses are composed in such a way that either refuting or confirming them helps us in concluding on our research question.

To do a case study, we need a test system. Requirements that a test system should fulfill are set, and the DSW website, Eureka and PAS are analyzed to check if they fulfill these

requirements. After PAS was chosen as the most suited test subject, the general hypotheses are refined to this choice. The following hypotheses are used in our research:

H1: Using cost-benefit analysis, automated GUI testing on PAS benefits DSW more than it costs.

H2: Having an automatic regression test set for the PAS project reduces the number of executions of the manual regression test.

H3: Having an automatic regression for the PAS project test lowers the average relative cost of a bug.

After refining our hypotheses, our grounds for following a qualitative approach are explained. To refute or confirm our hypotheses and to formulate an answer to the research question in a qualitative way, we need experimental data to back up these claims. An overview of the experimental data we need is given. Finally, the project is divided into multiple stages. A schedule depicting which experimental data to obtain in which project stage, is given.

Chapter 4

Choosing a GUI testing tool

In this Chapter, the considerations regarding the choice for a testing tool are explained. First, Section 4.1 discusses why we choose to use an existing tool instead of creating our own solution. In Section 4.2, several requirements for testing tool selection are given. The degree to which several existing tools and libraries fulfill these requirements is also evaluated. As we will see, the good GUI object recognition capabilities of CUT lead to the choice of CUT as testing tool, which is explained in Section 4.3.

4.1 Choice for testing tool - custom creation or use of existing solution

The testing tool is the core element of an automated GUI testing solution. The first decision to make, is whether to built a tool of our own or to use an existing solution. The advantages of both options are listed below.

Custom creation is preferable over choosing an existing solution because:

1. We can choose exactly how the GUI testing tool works, contrary to choosing an existing solution, where we might not be able to fit everything to our needs.
2. No money is spent on possible license costs of a testing tool.
3. The GUI testing tool can be extended in case new GUI elements (which are not yet supported by the tool) are introduced in the AUT.
4. There is no risk of the tool's vendor abandoning their product, in which case maintenance of the product might become impossible.

On the other hand, choosing an existing solution is preferable over creating a custom one because:

1. An existing solution can be chosen off-the-shelf:
    a) Little to no development time is needed before it can be used.

b) Less human labor (cost) is needed to come to a usable GUI testing tool.

2. Updates on the testing tool are done by an external party. The only maintenance we need to do ourselves is on the test set.

3. No need to tackle complicated issues regarding accessing the GUI ourselves.

Both options have their strengths and weaknesses. The choice can be influenced by the company's nature. Some companies are strong supporters of keeping the entire process of testing in their own hands. Others companies gladly narrow down their test effort by applying third party tools.

In our case, the most important consideration is time. We can save much time by choosing an existing solution. The aspect of time is even more important since we want to complete our thesis within the recommended time constraints. Therefore, our preference is to use an existing solution.

4.2 Tool selection

The testing tool selection depends on four main requirements: first, it must be possible to use data-driven testing (**G_5**), second, a framework should be created that provides utility functions and enables the reuse of parts of test cases (**G_6**), third, all GUI object within PAS should be recognized (**G_12**), fourth, test report should contain useful data (**G_26**), and finally, the tool should be able to cope with cosmetic changes to the GUI (**TT_1**).

We investigated several off-the-shelf tools and libraries by automating a few test cases (three for each tool) to evaluate the degree to which these tools fulfill the requirements.

4.2.1 TestComplete

TestComplete is an automated testing tool made by SmartBear software [23]. We used version 8.0.290.7 in our analysis. Their product offers a wide range of possibilities for testing: load testing, unit testing, keyword/data-driven testing, and so on. Two editions are available: the enterprise edition and the standard edition. Two types of licenses are available for both versions: a node-locked license (for use on just one computer, costs $1999 for the enterprise edition and $999 for the standard edition), and a floating license ($4499 for the enterprise edition and $2999 for the standard edition). The tool is intuitive to use, and offers possibilities to create test cases in various ways. It is possible to create test cases visually, which does not require any programming skills. Scripts can be used to create more advanced test cases though. TestComplete supports test case creation in various programming languages: JScript, C#, C++, and so on. Possibility to re-use parts of test cases is also possible when using these scripting facilities. The output is customizable, meaning it can be tailored to suite our needs.

The tool satisfies all of our requirements with one exception. TestComplete did not recognize all of the GUI elements used in PAS. TestComplete failed to recognize the following GUI elements:

- Individual rows, columns and cells within a table.

- An individual tab within a row of tabs.
- Buttons contained in certain kinds of container objects (some containers allowed individual button recognition, but some did not).

4.2.2 AppPerfect App Test

AppPerfect App Test is a testing tool to create and run functional and regression tests. Version 12.0 is used for our evaluation. App Test is created by AppPerfect Corporation [6]. AppPerfect develops software to manage the quality of software development projects. They offer a wide range of products suited for different kinds of testing. App Test is the product to use for automated GUI testing. The costs for App Test are $299 for a desktop license and $399 for a floating license. Both licenses require an annual maintenance fee of $99 for prolongation. Use of scripting (in java script) to customize test cases is an option, as is data-driven testing. Good GUI element recognition is promised by using the GUI hierarchy to locate GUI elements, rather then using screen coordinates.

When we put this tool to the test however, the results are disappointing. GUI element recognition failed on exactly the same GUI elements as mentioned for TestComplete. On top of that, when executing a recorded test, the playback was very fragile. Errors occurred continuously due to problems with keeping focus on the AUT:

- Focus is not set automatically to the right window when the playback is started. Playback fails immediately if other windows overlay the PAS application's window.
- Clicking buttons that caused a pop up to appear was also problematic. In case a pop up was expected to appear after a button click, it did not always end up on top, and button clicks were applied to any overlaying window.

The biggest positive aspect of this tool is its speed. The speed of test case execution is very high, and the tool is lightweight to use.

4.2.3 Microsoft Visual Studio Testing Tools

Visual Studio 2010 Ultimate (VS2010U) edition is Microsoft's development studio. It includes tools and libraries that can be used for automated GUI testing. A single license for VS2010U costs $15.299.

One of the tools included in VS2010U, is Microsoft Test Manager (MTM). MTM enables users to record and playback test cases. These test cases can be parametrized to enable data-driven testing, and re-use of parts of test cases is possible by defining so called "shared steps"[8].

When we put MTM to the test, we found it to operate in an intuitive way. The tool is quite cumbersome though. It suffers from long waiting times to create test cases, store recordings, and so on. This is just a minor drawback however, the biggest problem is again found in GUI object recognition. The tool had problems coping with so called "text fields with a mask". They could be recognized and read, but MTM could not write text to them. Text fields with a mask are text boxes with certain restrictions on the possible input. For

example, a text box that should contain a phone number, cannot be filled with letters, and a text field that should contain a zip code, accepts input only in a predefined format.

The testing tools in Visual Studio also include a lower level testing facility, the Coded UI Testing (CUT) Application Programming Interface (API). CUT can be used as an extension to MTM by loading an action recording created by MTM. The recorded actions are translated into code, which can be edited to create more advanced test cases. CUT can also be used independently by using an action recorder, which immediately stores the different actions in code. By editing this code, re-use is possible. Facilities to use data-driven testing are also present. A drawback is the low-level nature of CUT. Much functionality has to be created ourselves, contrary to other tools, where this is already implemented.

4.2.4 Squish

Squish is a testing tool created by Froglogic [14]. Version 4.1 is used in our evaluation. Squish can do all sorts of testing on different platforms (even mobile platforms such as cell phones are supported). Data-driven testing is among the options, log output can be customized and re-use of parts of test cases is possible when scripting is applied (Squish supports scripting in Python, JavaScript, Perl and Tcl). Two kind of licenses are available: Tester licenses and Runner licenses. A Tester license comes with software to create and modify test cases. A Runner license only allows test cases to be run, not created or modified. The Tester license costs $2400 for the first year, with an annual support and update subscription of $800. The Runner license costs $450 with an annual update cost of $150.

Squish is a very easy to use tool. It took only a couple of minutes to automated a test case. It runs very fast and is robust to changes in position of GUI element. Object recognition was good in comparison with AppPerfect and TestComplete: individual tabs within a row of tabs could be identified, and rows, columns and cells within tables were also identified correctly. However, Squish also had problems identifying buttons within specific types of containers.

4.3 Selecting the most suitable tool

Based on the requirements listed in the beginning of this Section, two tools are immediately discarded from the selection. TestComplete and App Test fall short in a severe way when it comes to object recognition. Object recognition in Squish is not perfect either, but the elements Squish cannot successfully recognize are used only sparsely in PAS. This is contrary to the tab and table cell recognition, which is often used. From the tools incorporated in VS2010U, MTM does not fulfill our requirements either. Given the frequency of occurrence of masked text fields in PAS, failing to operate on those also is a severe shortcoming.

The final choice is between Squish and CUT. To make a choice between these two, the differences are depicted in Table 4.1.

Table 4.1: Differences between Squish and CUT.

Squish	**CUT**
Cannot recognize individual buttons when these are grouped in certain containers.	Recognizes all GUI objects.
Supports scripting in Python, JavaScript, Perl and Tcl.	Supports scripting in .Net languages (Visual Basic .Net, C#, Visual C++, F#).
Full version license cost is $2400, with an annual support and update subscription of $800.	A license for VS2010U costs $15.299. Every few years upgrade licenses are required, with prices ranging from $2000-$3000.
Is not used yet at DSW.	Is already in use at DSW.
Much functionality comes off-the-shelf.	Some functionality has to be custom made on top of the CUT library.

License costs do not matter since DSW already uses CS2010U. Implementing functionality ourselves that is available in Squish requires time investment, but so would the introduction of another programming language at DSW. What is left is slightly better object recognition on CUT's side. Taking all of this into consideration, we choose CUT as our testing tool.

Chapter 5

Designing the testware

In this chapter, the design of our testware is outlined. The requirements this testware has to fulfill are given in Section 5.1. CUT does not fulfill all of these requirements, which means it has to be enhanced with a custom extension. CUT together with this custom extension makes up our automated GUI testing solution, which is called DARTH VADER (DSW's Automated Regression Testers Helps Validating And Does Error Reporting). The creation of this custom extension, and the process of creating an initial layered model for DARTH VADER is described in Section 5.2. For the test set, we can avoid unnecessary maintenance by avoiding the use of hard-coded values, and by reusing overlapping parts in test cases. How we used keywords in the test set that are replaced at runtime to avoid hard-coded values, and how we used blueprints for test case execution to reuse overlapping parts, is explained in Section 5.3. In Section 5.4, for each of the layers in DARTH VADER's model, the level of abstraction, the tasks it has to fulfill, a global description of its design, the implementational details, and the division between CUT and custom functionality is described. Maintaining the test set can be hard, because it is hard for a human to create a mental overview of what a test case does, and because typing errors can easily be made. Section 5.5 discusses the creation of a tool called R2D2, that is designed to to solve these problems. We conclude in Section 5.6.

5.1 Requirements for DARTH VADER

A number of requirements have been identified from the guidelines and techniques listed earlier. A framework has to be created to provides options for reusing overlapping parts of test cases, and enable utility functions to be used (**G_6**). The framework should separate test case data from business logic of the AUT (**F_1**). It should also separate test case data from logic of the testware (**F_3**). An easy to use and maintain GUI mapping should be provided (**F_2**), as well as options for hierarchical reuse of business logic of the AUT (**F_4**). Finally, test reports should contain all of the data needed by all stakeholders (**G_26**).

The guidelines also stress to regard an automated GUI testing project as any other software development project, and apply good engineering practices whenever possible (**G_7** and **35**). In order to comply with this requirement, we have taken the so called *desirable*

characteristics of a design (minimal complexity, loose coupling, and so on) from Code Complete [16]. We use those as an extra set of requirements.

5.2 Choice for a layer model

In order to fulfill all of the requirements, we have to enhance CUT. We do so by creating an additional layer on top of CUT. This layer is used to create the test reports, issue GUI interaction calls to CUT, read and parse input data, error handling, and so on. We will call this layer the custom extension for now.

Within the layer model, the custom extension represents the outer layers, visible to the end user. CUT is hidden within the custom extension, and contains internal layers. The functionality of our GUI testing solution is presented in each of these layers at a different level of abstraction. Since the CUT and custom extension are conceptually related to each other as layers, we create a model that leaves this relation intact.

The first step in creating this model is the identification of CUT's layers. CUT can be called to interact with PAS's GUI, which it does by controlling the mouse and keyboard. This in turn is done by issuing system calls to the operating system. Two initial layers can thus be recognized within CUT: the *GUI interaction layer*, on which we can issue commands, and a lower level *Operating system interaction layer*. This initial overview is depicted in Figure 5.1.

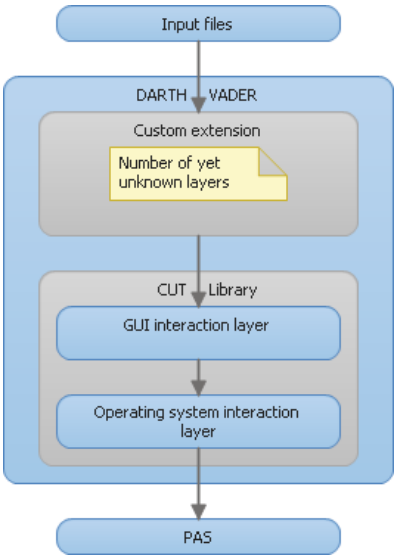


Figure 5.1: Initial layer overview without the custom extension's layers.

5.3 Test set representation

When designing the test set, the avoidance of hard-coded values, reuse of overlapping parts of test cases, and limited complexity of the resulting format, have to be taken into consideration.

How hard-coding mutation dates that are dependent on the current date causes maintenance, and how we use keywords that are replaced at run time to avoid this, is described in Subsection 5.3.1.

One possibility to reuse overlapping parts is to describe test cases as a path through a graph containing all possible GUI actions. Another possibility is based on separating test data and test logic, and reuse the logic. These possibilities are described in Subsection 5.3.2.

In our test set, the GUI interactions that make up a test case are stored separately from the parameters with which to execute these interactions. The details of this format are explained in Subsection 5.3.3.

Finally, in Subsection 5.3.4, we explain how restricting DARTH VADER's input to two files, and using named references instead of line numbers, prevents us from falling into the same pitfall as WinRunner did.

5.3.1 Avoiding the use of hard-coded values in scripts

The guidelines and pitfalls mention two ways of using hard-coded values that cause a frequent need for maintenance. How we are going to avoid both is explained below.

Using a capture and replay tool that uses fixed (hard-coded) coordinates to locate GUI elements - CUT locates objects by using a so called object repository. The object repository contains values of properties of GUI elements: labels, IDs, control type, position in the GUI tree hierarchy, and so on. This information is still hard-coded, but is much less volatile than fixed coordinates.

Bypassing the use of data-driven testing. This means test cases have to be hard-coded within the source code of the system - We will apply data-driven testing in our solution.

We identified another possibility to lower the need for maintenance by removing hard-coded values that is not mentioned in the guidelines. We can do so by avoiding the use of volatile hard-coded dates as test case parameters. Actions like registering for an insurance policy and changing the method of payment can all be executed at a certain date (the mutation date). There are a number of categories for this mutation-date. The category that the date belongs to, determines the way the mutation is executed. For example, a mutation can be executed with retroactive effect, or be scheduled for the future.

The boundaries for some of these date categories are dependent on the current date, which is volatile. Dates within these categories cannot be hard-coded. However, other date categories can be hard-coded without necessitating frequent maintenance though. An overview of categories and possibilities for hard-coding them, is given below.

DC_1 *Any date before the year 2006* - This category does not depend on the current date, and can be hard-coded as 12-31-2005.

DC_2 *Any date after the year 2006, but before the earliest possible retro-activity date* - This category does not depend on the current date, and can be hard-coded as 01-01-2006.

DC_3 *Any date between the earliest possible retro-activity date and now* - This date is dependent on the current date.

DC_4 *The current date* - This date is dependent on the current date.

DC_5 *Any date between the current date and the furthest possible mutation date in the future* - This date is dependent on the current date.

DC_6 *Any date after the furthest possible mutation date in the future* - This date is dependent on the current date, but can be hard-coded by using 01-01-2100. This is very probably longer than the lifespan of PAS or DARTH VADER.

Hard-coding the values for **DC_3**, **DC_4** and **DC_5** cannot be done without accepting extra maintenance. We avoid this by replacing each occurrence of one of these dates in the test set, by a keyword. At run time, these keywords are replaced by a dynamically calculated date that satisfies the conditions for that date category. The date categories, the keywords and corresponding calculated dates follow easily:

DC_3 *yesterday* - This keyword is replaced by the current date minus one day.

DC_4 *today* - This keyword is replaced by the current date.

DC_5 *tomorrow* - This keyword is replaced by the current date plus one day.

5.3.2 Reuse of similar parts in test cases

We need to identify similarities between different test cases to be able to reuse them. Each of the test cases in PAS's regression test script contains a description of the actions that need to be executed, and an expected result. Many test cases execute the same sequence of actions, with the only difference being the parameter values for these actions. Validations for these test cases also differ for each test case. We analyze techniques developed by other researchers to find possibilities for reuse of similar actions in different test cases.

Actionable Knowledge Representation (AKR) - This technique is described in *Actionable knowledge model for GUI regression testing* [26].

A test case consists of a string of actions to be applied at the AUT in a certain order. The idea behind AKR is to create a graph that represents all of the test cases within a test set. In this graph, actions are represented as vertices and the order between actions is represented by an edge between two vertices. The graph that is created is the union of all of the the actions contained in all the test cases, and all the possible orders of actions. This graph is much smaller than all of the separate test cases added up because actions that occur in more than one test case, are represented in the AKR graph by just one vertex.

Instead of describing a test case as a string of actions, it is described as a path through the AKR graph. A change to one of the AUT's actions results in a change to just one vertex of the AKR graph. All of the test cases whose path include the changed node, are implicitly repaired by this change.

The fact that every possibility for re-use of actions is used, is a big advantage of AKR.

However, a disadvantage exist. Every test case contains a description of the path it has to take through the AKR. When test cases are similar to each other with regard to the order in which actions are executed, they describe almost the same path through the AKR. AKR does not re-use the overlap in the order of actions. This data replication makes AKR very inefficient in dealing with changes to the order in which actions are executed. One change in the order of two actions means all replicas describing this order, have to be updated.

The Automated Testing Framework (ATF) - We analyze the strengths and weaknesses of the ATF (described earlier in Subsection 2.4.3) with regard to reuse of similar parts in test cases.

Contrary to AKR, ATF enables re-use of overlap in the order in which actions are executed. This is because of the hierarchical structure. A higher-level node is defined by stringing multiple lower-level nodes together. By stringing these nodes together, an order is defined and re-used. Changing this definition changes the order for every reference to the specific node.

A disadvantage of the ATF is that it uses a triple-layered hierarchical node-structure to store test case logic. This is overkill for representing PAS's test cases, which typically consists of about 10 actions. A triple-layered hierarchical structure might be more suited for testing programs that have a very complicated GUI (think of applications like Adobe Photoshop, with hundreds of menu entries divided over different sub-menus), or when one wants to test the behavior of an application when executing a number of different test cases subsequently in a specific order. In our case, trying to re-use overlap in a three layered hierarchical structure would be overkill. It would make the test data file (and the facilities needed to parse it) complex, and harder to maintain. Furthermore, the ATF is more focused towards GUI interaction testing, than regression testing.

5.3.3 Test set format

By analyzing the advantages and disadvantages of the techniques that facilitate re-use of parts of test cases, we have come up with some requirements for our test set's design. First, re-use of actions and order of actions has to be possible. Second, the structure of re-use should not be overly complex. Third, parameters and validations have to be stored separately from business logic. Finally, there should be an option to use keywords as parameters, which can be replaced at run time by calculated values.

We have come up with a test set format that satisfies all of these requirements. Just as in the ATF, the input is stored in two separate files. The file format for both of the input files is CSV. We choose this format because it is enough to store the needed input, and easy to parse. Also, as the guidelines describe, it pays off to create a tool to edit and create test

cases. A simple file format to store the test cases makes the creation of such a tool easier. The two files are discussed below.

Test case logic - This file contains the logic of the PAS application, and facilitates re-use of overlapping parts in test cases. Each separate line in the test case logic file contains one test case blueprint (in short, blueprint).

A blueprint describes the sequence of actions for one test case. A test case's action sometimes requires parameters to fully specify which interactions should be carried out on the GUI. When every test case that references one specific blueprint has the same parameter value for a certain action, this parameter value can be stored in a hard-coded way in the blueprint. If this is not the case, a keyword is stored in the position where the parameter should be placed.

The format for the test case logic file is depicted in Figure 5.2.

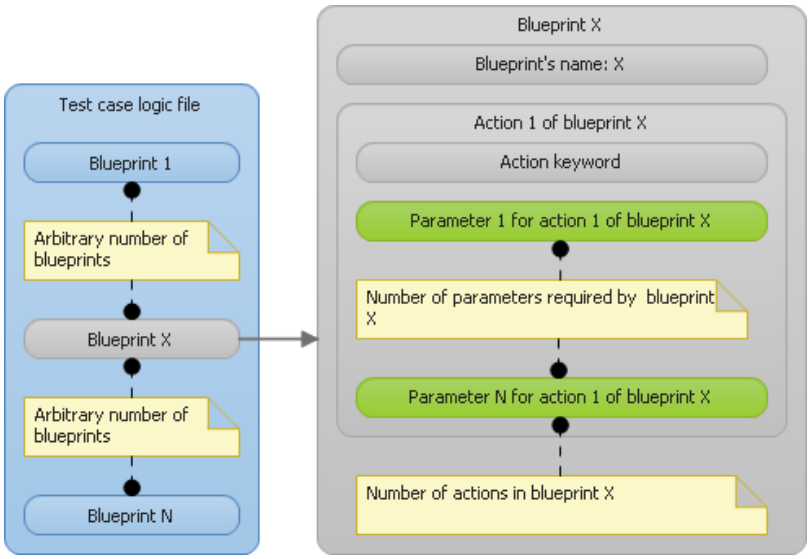


Figure 5.2: Schematic view of the test case logic file with one blueprint highlighted.

We also want to facilitate re-use of business logic within blueprints . This is done by allowing a blueprint to be included as a sub part within another blueprint. In case two blueprints A and B share a common part C, they can be defined as depicted in Figure 5.3 to re-use common part C. This eases maintenance in case a part of C has to be changed, since only one shared blueprint has to be changed, instead of every blueprint that includes this shared blueprint.

To give a concrete example of how a blueprint is represented in the test case logic file, we have taken the contents of one row from the file, and put it in Table 5.1. In this table,

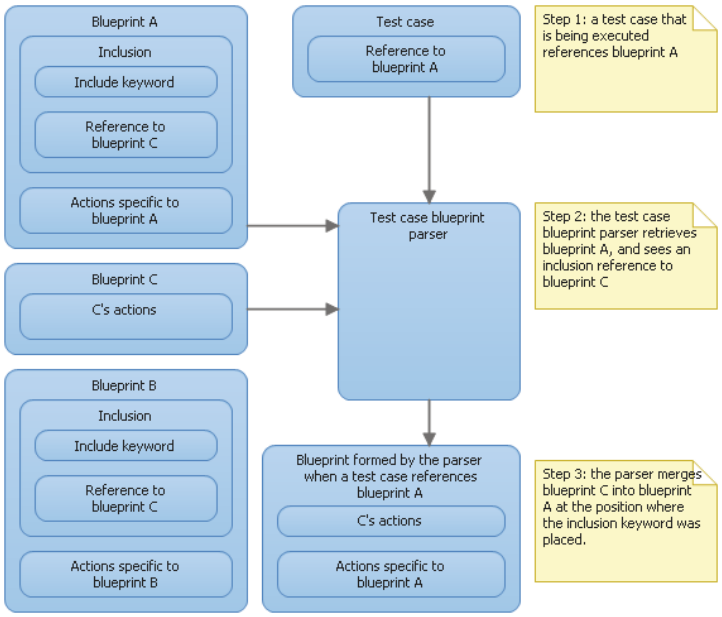


Figure 5.3: Schematic view depicting how blueprints can be nested.

we explain the meaning of each value. Each value (normally separated by semicolons) is represented in one row.

Fields	**Field's explanation**
BF_zoekRelatie_basis	The name of the blueprint. Can be referenced by test cases.
ClickButton	The first action of this blueprint is the click of a button.
ZoekRelatie_Screen	This parameter specifies which button should be clicked.
FillField	The second action of this blueprint is to fill a field with a value.
PARAM	This keyword indicates that the value of this parameter is supplied in the test case data file. This parameter contains the name of the field to be set (possibly more than one).
PARAM	This parameter is also supplied in the test case data file. This parameter contains the value the field should be set to (possibly more than one).
ClickButton	The third action of the blueprint is a button click.
ZoekRelatie_Zoekbutton	The name of the button to be clicked.
END	The keyword that indicates the end of the blueprint.

Table 5.1: Contents of one line of the test case logic file.

Test case data - Each separate line in the test case data file contains the parameters and validations needed to execute one test case. In order to couple a test case with its blueprint,

a reference to one blueprint is stored in every test case. The file format for the test case data file is depicted in Figure 5.4. When a test case is executed, the playback engine merges the parameters from the test case with the referenced blueprint. The colored parameters of the test case correspond to the colors found in the depicted blueprint. These matching colors indicate which parameter value from the test case corresponds to which parameter of the blueprint's actions.

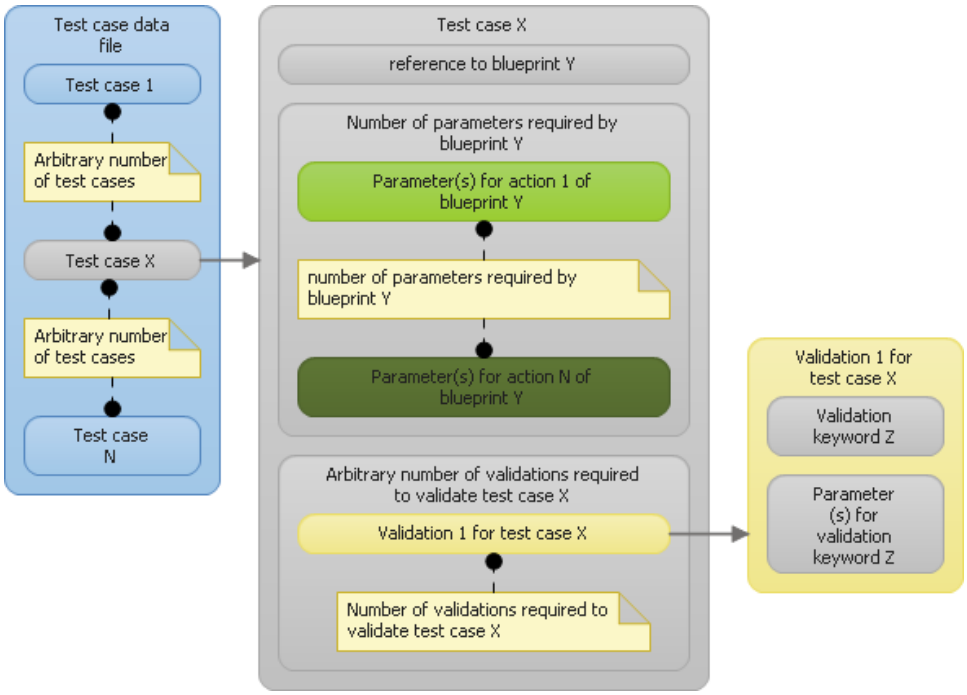


Figure 5.4: Schematic view of the structure of the test case data file.

To give a concrete example of how a test case is represented in the test case data file, we have taken the contents of one row from the file, and put it in Table 5.2. The meaning of each value is explained. Because this example includes all meta data and keywords, it contains more fields than the (abstract) example depicted in Figure 5.4.

Table 5.2: Contents of one line of the test case data file.

Field	**Field's explanation**
BEGIN	The keyword that indicates that the test case contained in this line should be executed. Any other value at the beginning of a line means that the test case should not be executed.

Continued on next page

Table 5.2 – continued from previous page

Field	Field's explanation
9111	The ID of the test case to be used within DARTH VADER.
9_1_1_1	The External ID. This matches the ID of a test case in the regression testing document.
Zoeken op bestaand relatienummer	Short description of the test case.
Moet goed gaan	The expected result of the test case.
BF_zoekRelatie_basis	A reference to a blueprint for test case execution.
DEPENDENCIES	When this test case's execution is dependent on the successful execution of certain other test cases, this keyword should be followed by their internal ID's.
PARAM	The keyword indicating that the following fields (up to the next keyword) contain values for the first parameter.
ZoekRelatie_Label	The first value of the first parameter.
ZoekRelatie_RelatieNr	The second value of the first parameter.
PARAM	The keyword that indicates that the following fields contain values for the second parameter.
DSW	The first value of the second parameter.
203099591	The second value of the second parameter.
VALIDATE	The keyword that indicates that the following actions are validations for this test case.
CheckField	This validation involves the checking of an observed value of a certain field, against an expected value.
PolisBlad_BSN	The name of the fields which value should be checked.
127450002	The expected value for the indicated field.
END	The keyword indicating this is the last field of the test case.

Figure 5.5 depicts the position of the input files within the architecture. It also schematically shows the path the test case data takes through the system, and how it is combined with a blueprint to be transformed into an executable sequence of actions.

5.3.4 Comparing test set format complexity - DARTH VADER vs. WinRunner

Earlier we have reported that the WinRunner solution became hard to maintain partly because of a complex file format. We have to take care not to fall victim to the same problem again.

In the WinRunner solution, the number of input files was equal to the number of different GUI screens. Every input file contained the input for all of the test cases for one screen.

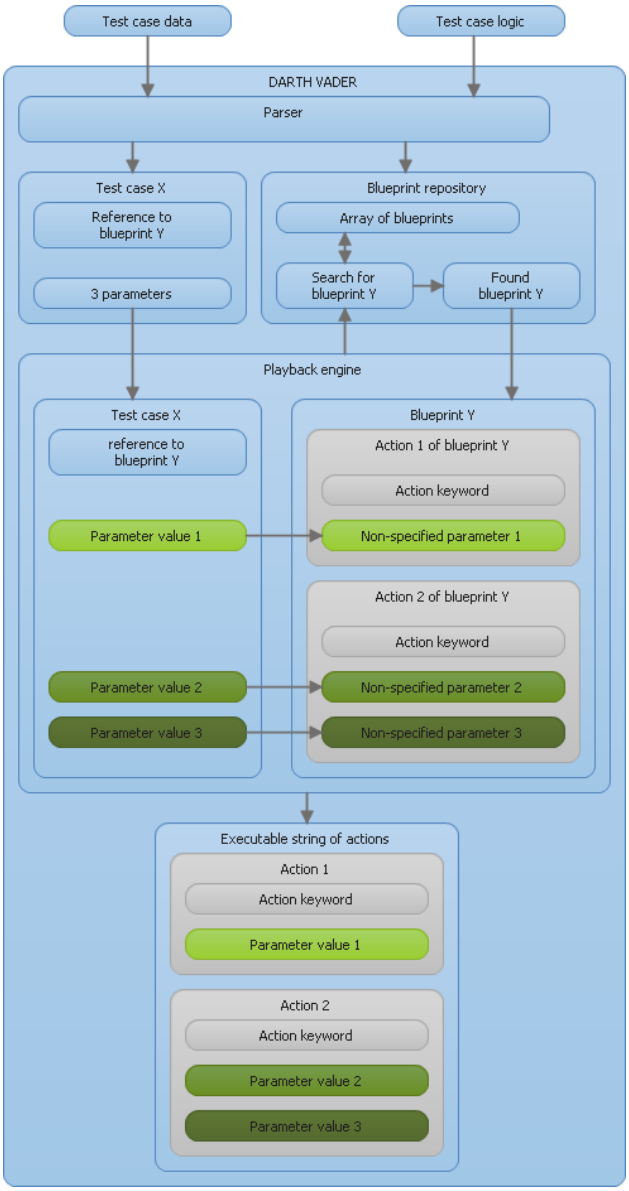


Figure 5.5: Schematic overview of the input architecture depicting how an example test case data and blueprint are merged.

This means when a test case uses multiple different GUI screens, it was spread over an equal number of input files, with each file containing a part of the test case. File identifiers combined with line numbers were used string the subsequent parts together.

Using line numbers is very volatile. When a test case is removed or added, the line numbers of subsequent test cases are also influenced. In a worst case scenario, every file has to be rebuild to update the line number references. By adding meta data to the input files, maintenance could be supported by tools. Maintenance remained problematic nonetheless.

However, our solution is easier to maintain than WinRunner. In our current solution, a test case is always spread over just two files, and test cases use names instead of line numbers to reference a blueprint.

5.4 Implementing DARTH VADER

DARTH VADER's complete layer model is shown in Figure 5.6. It contains, from top to bottom: test case execution layer, action and validation layer, GUI interaction layer and operating system interaction layer. We describe for each of these layers what the level of abstraction is, which tasks it executes, how it is designed, which classes implement the layer, and which parts are custom made and which parts are taken from CUT.

5.4.1 Test case execution layer

Level of abstraction - The test case execution layer carries out automated GUI testing on the abstraction level of executing test cases.

Tasks - Regarding automated GUI testing at the level of test case execution (while taking into account the way these test cases are inserted in the system), the following sequence of actions has to be carried out for every test case:

1. Read a line from the test case data file.
2. Tokenize the line.
3. Parse the tokenized line.
4. Execute the test case.
5. Log the results of the test case's execution.

Global design description - The tasks that we have identified can be seen as steps that are needed to complete a test case. For most of these steps, the output can be used as input for the next step. The line of the test case data file that has been read, is fed into a tokenizer. The tokenized line is then fed into the parser to be transformed into a more complex data-structure to represent the test case. The parsed data structure then determines the execution of the test case. Finally, the results of the execution are used to create a log.

This process can be implemented as a pipeline (or *pipes and filters architecture* [13]), where the output of a given step, is used as input for the next one. By implementing it as

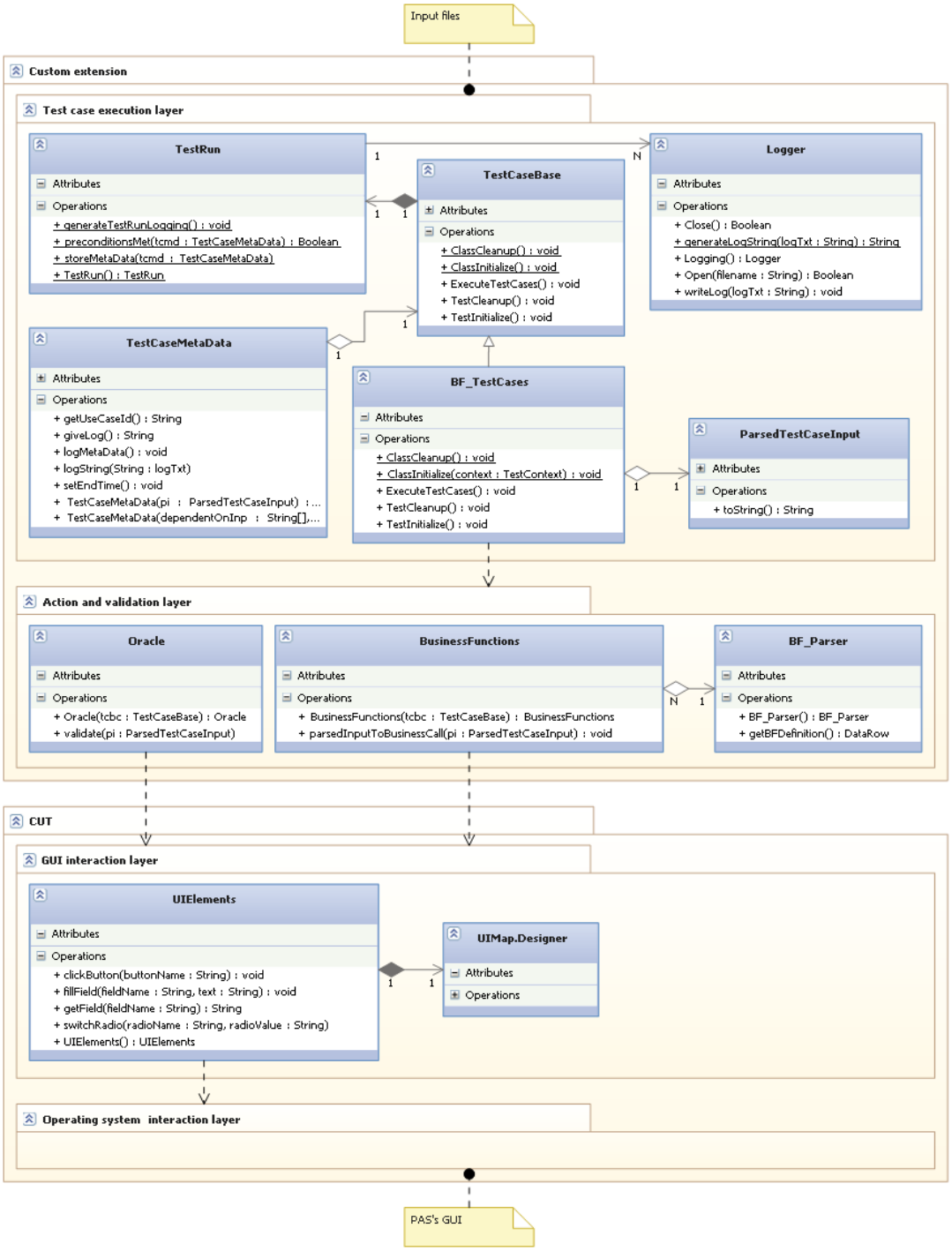


Figure 5.6: Complete architecture.

a pipeline, the steps needed to execute a test case stay very clearly visible. Also, when a certain requirements changes, the impact on the implementation is isolated to one stage of the pipeline. For example, if the desired input format for test cases changes, or the reports created by DARTH VADER have to be formatted differently, only one stage of the pipelines is affected.

Implementational details - The *BF_TestCases* class is the ”driver” for the pipeline. In this class, test case data lines for the business functions are entered and tokenized in the system. After this, calls are dispatched to complete the other steps of the pipeline. The driver is contained in function *ExecuteTestCases*. *TestCaseMetaData* is a helper class, which stores meta data for every test case that is being executed. *ParsedTestCaseInput* is the parser for the tokenized input lines. The *TestRun* class stores all of the meta data of the test cases executed in the current run. At the end of a run, or in case of errors, the *TestRun* class calls the the *Logger* class to log details regarding the test case execution.

Automated testing library functionality versus custom made functionality - On this abstraction level, the following parts of the functionality are not custom made:

- By using a *DataSource* tag, an input file is specified. Then Visual studio processes this file by reading the first unprocessed line, tokenizes it, and stores the tokenized result in a *TestContext* object, which can be referenced during the entire execution of the specific test case. This is done recursively until the entire file is read.
- The control flow needed for setting up preconditions or guaranteeing post conditions, is guided using *ClassInitialize*, *ClassCleanup*, *TestInitialize*, *TestCleanup* and *TestMethod* tags. For clarity, this control flow is depicted in appendix A.
- Using an *ApplicationUnderTest* Object to create a handle to the AUT. Using this handle, Visual Studio can start and stop the application, and can guarantee that the AUT always keeps focus during the execution of the test run.

The custom made functionality within this layer consists of: the pipeline-driver that guides the test case’s execution, the parsing of the input, logging the results and exception handling.

5.4.2 Action and validation layer

Level of abstraction - The action and validation layer carries out automated GUI testing on the abstraction level of applying a sequence of actions to the AUT, and expecting a certain response which should match the validation(s).

Tasks - Test case execution at the abstraction level of executing actions and validation can be decomposed it into a number of tasks:

1. Merging the test case with the blueprint it references.
2. Translating the action-keywords and their parameters into calls to the GUI interaction layer.

3. Translating the validation-keywords and their parameters into calls to the GUI interaction layer (when a test case has no validations, this step is skipped).

Global design description - As in the test case execution layer, step 1 and 2 can be seen as a pipeline. However, calling the design of this layer a pipeline architecture design would be overkill for a two staged pipeline.

Implementational details - The *BusinessFunctions* class presents the *parsedInputToBusinessCall* function to the outside world. This function is responsible for step 1 and 2 of this layer. The *BF_Parser* is used for retrieving the blueprint. This class reads the test case logic file once, and can be used as a repository for retrieving blueprints. The *Oracle* is responsible for executing step 3. To do so, it presents the *Validate* function to the outside world.

Automated testing library functionality versus custom made functionality - In this layer, no CUT or other library functionality is used. Every part of this layer's functionality is custom made.

5.4.3 GUI interaction layer

Level of abstraction - The GUI interaction layer carries out automated GUI testing on the abstraction level of direct interactions with the GUI.

Tasks - Carrying out automated GUI testing at the abstraction level of this layer involves, among others, the following tasks:

1. Clicking and double clicking buttons
2. Reading from and writing to text fields
3. Searching within tables
4. Checking presence of elements in the GUI

Global design description - There are many tasks this layer has to be able to execute to carry out automated GUI testing at this abstraction level. However, conceptually, most of these tasks are very easy. This layer is a big library-like class with one function for every task mentioned for this layer. Most of these functions are not much more than a call to a CUT API function (contained in the operating system interaction layer). However, some of them, like searching a table, cannot be translated to one CUT API call. Functions like these are more complicated.

Implementational details - The *UIElements* class contains the functions for all of the tasks of this layer, while the *UIMap.Designer* class contains the variables that reference to GUI elements.

Automated testing library functionality versus custom made functionality - In this layer, the CUT functionality handles two primary functions:

- Maintain a GUI map. This is a Object Repository which maps GUI element properties (which are used as keys to identify GUI elements) to program variables.
- Provide classes which can be used as handles to mouse, keyboard and WinControls (WinComboBox, WinText, WinTable, and so on).

The custom made parts of this layer:

- A custom made GUI map containing short and concise variable names is provided. This custom made GUI map acts like a wrapper around the automatically generated GUI map. This wrapper is made to comply with guideline to create an easy to use and maintain GUI mapping (**F_2**). The GUI mapping generated by CUT contains long, complicated, GUI-tree hierarchy dependent variable names. By creating a wrapper, we simplify the references to GUI elements in the blueprints and test cases.
- Every possible GUI interaction on the AUT is represented in this layer by a function which handles the execution of this interaction. Some of these functions are complex (such as functions for searching tables), while some of them are more like wrappers. These wrappers only use the GUI map to retrieve a handle to the the GUI element that is referenced in the specific GUI interaction, and the GUI interaction is carried out on this handle.

5.4.4 Operating system interaction layer

Level of abstraction - The operating system interaction layer carries out automated GUI testing on the abstraction level of issuing system calls to the operating system to control the mouse and keyboard.

Tasks - To carry out automated GUI testing, this layer has to be able to carry out, among others, the following tasks:

1. Control the mouse pointer, mouse-wheel, mouse buttons and keyboard.
2. Read properties of GUI elements.
3. Detect the presence of GUI element.

Of course, the tasks that concern controlling the mouse and keyboard are not about moving the mouse physically, or physically applying keystrokes to the keyboard. These actions have to be mimicked by the operating system interaction layer.

Global design description - This layer is completely implemented by CUT. Even the interface to use this layer is all predefined by CUT's API.

Implementational details - None of our classes implements any of the tasks for this layer. The only class that references the API of CUT directly is the *UIElements* class.

Automated testing library functionality versus custom made functionality - All of the functionality of this layer, apart from some customizable configuration settings for CUT, is implemented by CUT. Certain algorithms and heuristics are used by CUT to locate GUI elements. These algorithms and heuristics use parameters which can be set manually. The default values of the following variables are explicitly overridden in DARTH VADER:

- *DelayBetweenActions* - This value determines the waiting time between two successive interactions on the GUI. Too little delay between actions can cause the playback to stall because the GUI is not yet ready to receive input after the previous action. Too much delay causes slow playback since much time is wasted because of long waiting times. The default value was 100 milliseconds, which we changed to 250 milliseconds. Playback was smoother with this setting, as we concluded from experience.

- *ShouldSearchFailFast* - Sometimes a referenced GUI element cannot be found. To avoid waiting for this element for a long time, the fast fail option can be enabled. When the playback engine is confident the GUI element cannot be found (and prolonging the waiting time will not solve this), the search is aborted and the test case fails. The default value for this setting is true, which we changed to false. We concluded by experience that this improves robustness in playback.

- *SearchTimeOut* - This value indicates the maximum time that the playback engine should attempt to locate a GUI element. According to Atif Memon 30 seconds is a safe limit for this value [18]. The default setting is two minutes. Since a shorter time is still safe, and improves playback speed, we used 30 seconds for this value.

- *MatchExactHierarchy* - When searching for a GUI element, it is matched to an object in the Object Repository. A match is established when all of the properties of the GUI element match those of an object in the Object Repository. When MatchExactHierarchy is set to false (as is the default), the position of the GUI element in the GUI tree-hierarchy is not considered in this matching algorithm. This improves the search algorithm's ability to deal with a changing GUI, but can sometimes result in a false positive (when two different GUI elements are equal, but located in different screens for example). We changed this setting to true, since it improves the robustness of the test execution.

5.5 R2D2 maintenance tool

It is recommended to provide test case writers with a tool that makes it easy to write test cases. Subsection 5.5.1 discusses the requirements of such a tool. Subsection 5.5.2 explains how we have implemented R2D2 to fulfill these requirements.

5.5.1 Identifying weak aspects of test case creation and maintenance

When looking at the format of the test set, we find there are a number of weaknesses when it comes to maintenance. We create our tool in such a way that we can avoid these weaknesses:

1. A test case is stored in two separate files. When we want to analyze a test case, we only see a reference to a blueprint, and a set of parameters. Looking at the test case input alone is not enough to get an overview of what a test case does. We need to look for the blueprint, and try to see which parameters values in the test case correspond to which parameter positions in the blueprint. When multiple parameters exist (which is usually the case), this can be hard.

2. Blueprints can be defined recursively. Although recursively retrieving these blueprints is not hard to do automatically, it can be hard to visualize what the complete blueprint looks like by hand.

3. The input files are stored in CSV format. This format is very fragile. One misplaced semicolon can make a test case invalid. Also, no input checking is done while creating or editing the CSV files. When a typing error is made in an action keyword, or a test case specifies too few or too much parameters, the test case is invalid.

5.5.2 Avoiding weak aspects of test case creation and maintenance

We have created a tool to edit the test case logic and test case data files. The GUI of this tool is depicted in Figure 5.7.

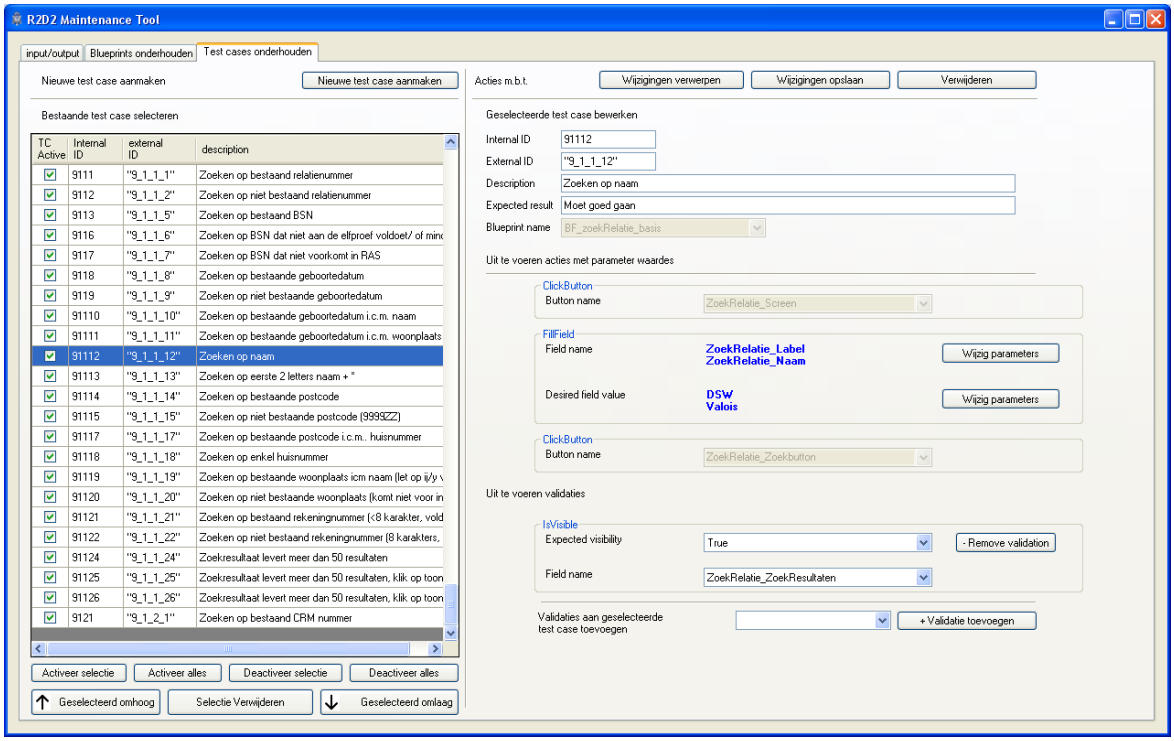


Figure 5.7: R2D2's user interface.

The tool helps in keeping overview while creating or maintaining test cases. It also guards the user from making several sloppy errors. The steps in which this is done, are the following:

1. *Read test case logic and test case data files* - The path to the location of the input files can be given, after which they can be read and parsed by clicking a button. After this is done, the tab-pages to edit the blueprints and test cases are activated.

2. Edit blueprints and test cases

   - *Blueprints* - We can create, edit and delete a blueprint:
     - *Create a new blueprint* - When a new blueprint is made, it must be given a name. Also, an arbitrary number of actions with corresponding parameters can be added to the blueprint. This can all be done by clicking buttons, and selecting actions from a predefined list. This greatly reduces the chance of making typos or making errors in the number of required parameters of a certain action.
     - *Edit an existing blueprint* - A table lists all the existing blueprints. Double clicking a row in this table loads the blueprint. After loading this blueprint, editing can be done in the same way as editing a newly created blueprint.
     - *Delete an existing blueprint* - In the same table of all existing blueprints, a blueprints can be selected and be deleted.
   - *Test cases* - We can create, edit and delete a test case:
     - *Create a new test case* - When creating a new test case, an ID for the test case has to be entered, and a blueprint for its execution has to be selected. The actions belonging to the selected blueprint are shown, with possibilities to fill in parameters for each position where the blueprint requires parameters to be filled in.
       R2D2 prevents errors in the input for the required parameter values by limiting the options for parameter input to the correct type of parameter. For example, when a blueprint specifies that a button has to be clicked, a button name has to be specified by the test case. When creating such a test case, R2D2 then presents the user with one drop down list that lists all of the buttons contained in the GUI map. This prevents two types of errors: specifying a button name that is not contained in the GUI map (by making a typing error for example), or specifying a parameter that is not of the button type.
       Test case validations can be added and removed as well. When a validation is added, the type of validation is selectable from a drop down list. After selecting the type of validation, the parameters have to be filled in. R2D2 uses the same method as described for the action parameters, to guard the user from making typing errors, and specifying the wrong type of parameter.

  - *Edit an existing test case* - An existing test case can be loaded from a table by double clicking it. After this, editing the test case is equal to editing a newly made one.
  - *Delete an existing test case* - From the overview of test cases, a test case can be selected and be deleted.
- *Storing the files* - After the files have been edited, they can be saved. The path where to write the output is specified, and writing the new test case logic and test case data files can then be done by clicking a button.

By enabling the creation and editing of blueprints and test cases in this way, we have solved the problem of keeping overview of what a test case does, and the vulnerability of CSV files to wrong parameter types and typing errors. The problem of visualizing recursively defined blueprints has not been solved by this tool yet. However, numerous ideas exist to enhance the tool, and this one can be included. This is further discussed in Chapter 8.

5.6 Conclusion

After the requirements for DARTH VADER have been identified, we choose to use a layer model for its architecture. An initial architectural overview describing the relation between CUT, the input files and the AUT, has been created.

The test set representation has to provide options for reusing overlapping parts of test cases, and avoid the usage of hard-coded values. A test set representation has been designed that makes use of two CSV input files. One of them contains blueprints for the actions that form a use case. The other file contains the test case's validations and parameters with which to execute the blueprint's actions. Reuse is facilitated by allowing blueprints to be reused for multiple test cases. Maintenance in the form of frequently having to update volatile dates is prevented by using keywords that are replaced by a calculated date at runtime.

The architectural picture is then expanded into a complete layer model. Four layers are identified: test case execution layer, the action and validation layer, the GUI interaction layer and the operating system interaction layer. Each of these layers deals with automated GUI testing at a different abstraction level and has a certain internal design. The internal design for the test case execution layer and the action and validation layer, is a pipeline model. The GUI interaction layer is a wrapping library. The operating system interaction layer consists almost exclusively out of CUT made functionality, so we do not have to consider its internal design. The division between custom functionality, and functionality taken from CUT or another automated testing library, is described for each of these layers as well.

Finally, R2D2 is a tool that assists testers or developers in creating test cases. It strongly reduces the likelihood of making errors in test case creation, and makes it easier to get an overview of how a specific test case's parameters are contained in a blueprint.

Chapter 6

Experimental results

This Chapter presents the experimental results from our case study on PAS. The experimental results can be grouped into several categories, each discussed in a separate Section. Section 6.1 gives an overview of the differences in detecting failures between the automated and manual test run. The time that is needed for maintaining the test set and the regression testing document, is presented in Section 6.2. The steps needed to setup an automated or manual test run and the time these steps take, are presented in Section 6.3. Section 6.4 explains how much time is needed to analyze the results of automated and manual test runs. The work satisfaction for DSW's employees with and without applying automated GUI testing are compared in Section 6.5. The effort needed to execute the regression test script manually is determined in Section 6.6, and the effort needed to automated this regression test script is presented in Section 6.7. Finally, the results are summarized in Section 6.8.

6.1 Bug detection

The differences between the failure detection of automated and manual testing have been logged during the June and September release. Several remarkable differences have been identified. Subsection 6.1.1 presents and explains these differences for the June release. For the September release, they are presented in Subsection 6.1.2.

6.1.1 Results for the June release

The test results of the June release's manual and automated test run are compared in this Subsection. The results for the automatic generation of letters are described separately from the results of the regression test set.

Regression test

Table 6.1 gives an overview of the results of the test runs in the June release. We mark a test case's execution as failed when there is a nonconformance between the observed and expected result.

	Number of test cases executed	**Number of failed test cases**
DARTH VADER test environment	158/582	12/158
DARTH VADER acceptance environment	158/582	13/158
Manual run test environment	433/582	8/433
Manual run acceptance environment	-	-

Table 6.1: Test results for different test runs in the June release.

There are some side notes to the table. Just one of the failures has been reported by both the automated and manual test set. This means the automated test run reported 11 failures that were not reported as failures by the human testers. Table 6.2 gives an overview of the reasons for this big difference.

ID	**Description of cause**	**Number of missed failures**
HMCJ_1	Human testers are less strict in their validations. No failures are reported when: 1. The expected result is not the same as the observed result, but the observed result is considered correct. 2. The observed result is similar to the expected result, but not exactly the same.	5
HMCJ_2	The test script's specification is not precise enough. Not all of the conditions necessary to reach a certain expected result are always mentioned. Human testers know by experience which these conditions are, and implicitly include activities that are not mentioned in the regression testing script, in the testing process.	3
HMCJ_3	DARTH VADER reports a false failure because the test case's parameters are filled with the wrong values.	1
HMCJ_4	Human testers executed their test run in a later deployment. Some failures which were present in the initial version on the test environment, might have been repaired in these later deployments.	1
HMCJ_5	Human testers skipped a test case that results in failure.	1

Table 6.2: The causes of failures being missed by the human test run.

Causes **HMCJ_1** and **HMCJ_2** are very abstract. Examples of failures being missed by the human testers as a result of these causes, are given below:

HMCJ_1

- PAS enables users to search for insurees on certain criteria. When no insurees are found, the expected result is an error message. However, the observed response is that of a message to inform the user that no insurees match the supplied search criteria. This is *not* an error message. Human testers do not report a failure because they judge PAS's behavior as being adequate.
- When carrying out transactions, all of the required input has to be entered first. After this is completed, a confirmation window appears. Any possible error in the input should be reported and corrected before this confirmation window is shown. In some cases, this did not happen. Input validation was not done properly, and the error message was delayed until after trying to confirm the transaction. DARTH VADER is strict when it comes to the expected moment the error message is supposed to show up, while the user is not. This results in DARTH VADER reporting a failure that was missed by the human testers.
- An error message should be clear and concise, as depicted in Figure 6.1. In some cases, an error message contained a technical description as is in Figure 6.2. The human testers accepted just any error message as sufficient, since they knew what they did wrong. However, DARTH VADER checks for a specific error message and reports a failure when another error message is shown.

HMCJ_2 An insurance policy can consist of two separate parts: the basic (obligated) insurance policy, and an (optional) addition called *aanvullende verzekering* (AV). When an AV is changed, an administrative fee can be charged. When an insuree changes his AV two times in the same month, and the administrative fee is charged for the second time in the same month, PAS should issue a warning. However, a technical restriction exists. There should be at least 10 minutes between the first and second attempt to change the AV, else the warning is not issued. This ten minute waiting period is not specified in the test case. This means the preconditions of the test case were not properly described, and the expected result was not produced, which resulted in DARTH VADER reporting a failure. The human testers knew about this extra precondition. After implicitly including this in their testing activities, the system produced the expected result, so no failure was reported.

DARTH VADER detected three failures on the acceptance test environment that it did not detect on the test environment. These failures were not found by the human test run either since there was no human test run in the acceptance test environment. All three failures were the result of one error in PAS. The error was related to the so called *eigen risico*. This error was introduced by a bug fix resulting from the failures reported in the test environment.

Some failures reported by the human testers, were not reported by DARTH VADER. Table 6.3 gives an overview of the causes for these differences.

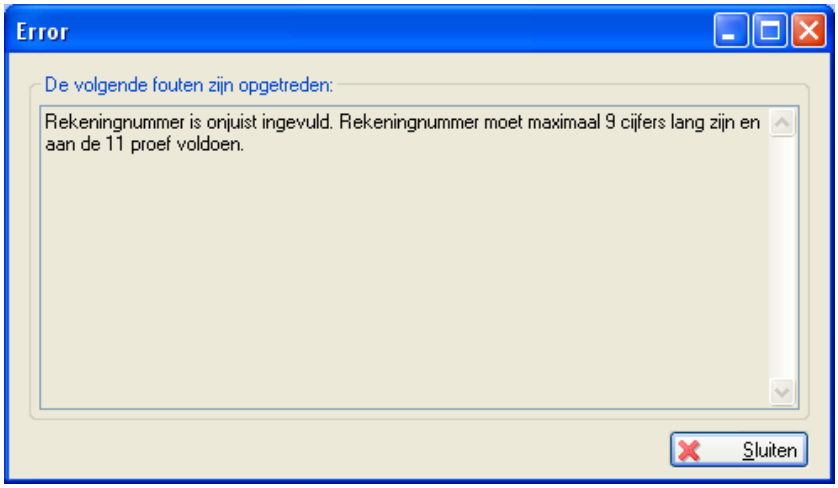


Figure 6.1: Well formed error message

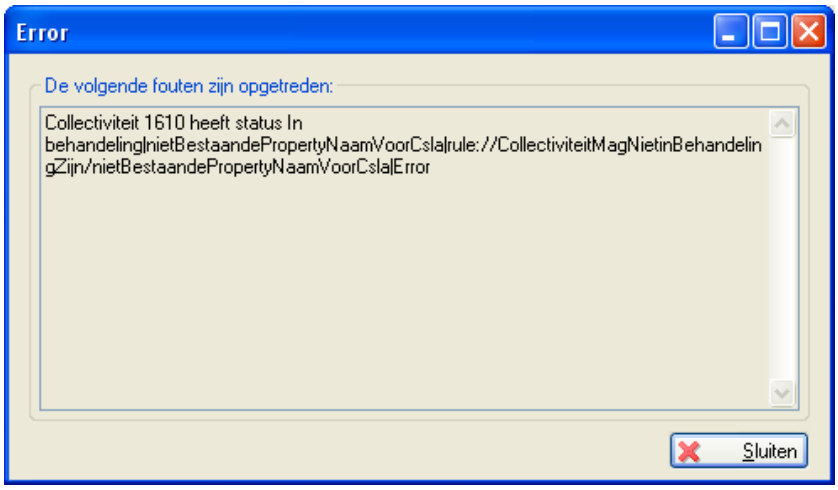


Figure 6.2: Malformed error message

ID	Description of cause	Number of missed failures
AMCJ_1	Not all of the test cases that are executed in the manual test run are yet included in the automated test run.	6
AMCJ_2	Errors in the validations of the automated test set that caused failures to be missed.	1

Table 6.3: The causes of failures being missed by the automated test run.

Automatically generated letters

In total, 627 letters have to be generated, printed and checked. Since the test set was not completed at the beginning of the June release's testing period, not all of the letters were automatically generated. The letters that had not been automated, were generated by hand.

In the acceptance environment, 583 letters were automatically generated. Three failures were reported while generating these letters. All of these had not been discovered during the manual test run on the test environment. Even though the letters were generated automatically in the acceptance environment, checking whether the content was correct was still done by hand.

6.1.2 Results for the September release

The results for the September release are presented in the same way as we did for the June release. First we present the results of the regression test, then we present the results of the generation of PAS's letters.

Regression test

The number of test cases that were run, and the number of them that failed, is shown in Table 6.4.

	Number of test cases executed	Number of failed test cases
DARTH VADER test environment	256/582	33/256
DARTH VADER acceptance environment	256/582	8/256
Manual run test environment	445/582	25/445
Manual run acceptance environment	-	-

Table 6.4: Test results for different test runs in the September release.

Although the number of test cases in the human test run is much bigger than that of the automated test run, the number of failures reported by these runs is almost the same. This is a remarkable observation. After analyzing these test runs, we have created an overview explaining why these results are so different.

The first observation is that there is very little overlap between failures found by the automated and human test run. Three failures were reported by both of the test runs. Apart from that, the results have no overlap. This means 30 out of the total of 33 failed test cases, were not reported by the human testers. Table 6.5 depicts the causes that we have identified for these results. The other way around, out of the 25 failures reported by the human testers, 22 were not reported by the automated test run. Table 6.6 depicts the causes we have identified for these differences.

ID	**Description of cause**	**Number of missed failures**
HMCS_1	The automated test set was run immediately after the test environment was signaled to be stable. This initial version still contained bugs, which were gradually removed with several small updates to the test environment. Since the human test run takes much time, it is inevitable that some of the test cases are executed in a version that already has some bugs removed.	25
HMCS_2	Human testers overlooked failures that were present in the system's non-critical functionality: • Auto-completion on fields that did not work. • Default field values were not calculated correctly. • Combobox fields that should be restricted to one choice under certain conditions allowed multiple options. • When a warning is expected by the test script, but it is not observed in the test run, this is not always reported as a failure.	4
HMCS_3	Some test cases were labeled as "not important" by the human testers, and therefore, were not executed.	1

Table 6.5: The causes of failures being missed by the human test run.

Automatically generated letters

Contrary to the June release, the script to automate the generation of 583 letters was al-

ID	Description of cause	Number of missed failures
AMCS_1	Not all of the test cases that are executed in the manual test run are yet included in the automated test run.	15
AMCS_2	Errors in the validations of the automated test set that caused failures to be missed.	3
AMCS_3	Human testers have a wider scope for signaling failures. Not only the expected response of a test case is checked, but accidentally spotted malfunctions (that are not mentioned specifically in the test case's expected response) while executing the test case are reported as failures as well.	2
AMCS_4	The human test run is executed in a later version of PAS. Although in general, less failures are present in newer versions, a failure can also be introduced by a bugfix.	1

Table 6.6: The causes of failures being missed by the automated test run.

ready available during the testing stage of the September release. When this script was run for the first time on the test environment, generation of any arbitrary letter failed. This meant the run could not be completed. A soon as the error that caused this initial problem was solved, a complete test run could be executed. No failures were reported during this run. Another complete test run was executed on the acceptance test environment. This run also did not discover any failures.

6.2 Maintenance needs

This Section discusses the two main causes for maintenance needs:

1. During the September release, maintenance caused by changes made to the data that PAS uses, has been analyzed. The results are discussed in Subsection 6.2.1.

2. Changes made to PAS's GUI or logic during the September release, also caused maintenance. This is discussed in Subsection 6.2.2.

6.2.1 Maintenance due to data refresh

In every release, the databases at the back end of the PAS application are refreshed. They are overwritten by a copy of the production environment's data just before the test period begins. During the September release, the test set contained 256 test cases. 8 of these failed after the data was refreshed.

Including the time needed for diagnosis, it took 15 minutes on average to repair such a failure. In total, 2 hours were spent to repair the test set. By applying these experienced

values to the total number of test cases (583), we estimate a maintenance need of 4 hours and 30 minutes to repair the test set with regard to refreshed data.

6.2.2 Maintenance needs due to changes in PAS

At the beginning of the September release,about one third of all test cases failed, as well as the generation of all of the letters. This was caused by two small changes to PAS's GUI:

1. One button in the root menu of PAS had changed, and could not be identified by CUT anymore. All of the test cases that included an interaction with this button failed.

2. After a letter was generated, an Adobe Reader plugin object was used to display the generated PDF. Checking whether this plugin object was loaded in the GUI was used as a validation for the generated letters. Along with the new version of PAS, a new version of Adobe Reader was used to display the generated PDF. This object was not recognized correctly by CUT, and the validation for all of the letters failed.

The diagnosis of these problems took about one hour each. Repairing them was fairly simple though. Capturing these GUI objects again to refresh the Object Repository was enough to repair all the test cases again. This took another 30 minutes. This results in a total of 2 hours and 30 minutes.

We want to estimate the maintenance need when the entire regression test script is automated. To make this estimation, we need to consider that the maintenance need due to changes in PAS is dependent on the number of repairs we have to carry out. This in turn is dependent on the number of GUI elements that are used in our test set that have changed in the newer version of PAS. Only the number of GUI elements used in our test set changes when test set size is increased, the number of GUI elements that are changed in a newer version of PAS is not dependent on test set size. This means that in order to make an estimation for the maintenance need for the entire regression test script, we have to estimate the increase in the number of GUI elements used in our test set.

In general, different blueprints are used to test different parts of the GUI, and thus, use different GUI elements. This means we can use the increase in the number of blueprints to calculate the increase in the number of different GUI elements used. The measurements of 2 hours and 30 minutes is based on a test set containing 21 blueprints. The total test set would include 32 blueprints. This results in an estimated GUI element usage increase of 52% (33/21 * 100%) when the entire test script would be automated. The maintenance need is anticipated to increase with the same percentage. This means 3 hours and 45 minutes of needed maintenance time due to changes in PAS are estimated when the entire regression test script is automated.

6.3 Test run setup

Certain tasks have to be completed before a test run can be executed. During the June release, these tasks have been identified and analyzed for both the automated test run, and the

human test run. The results are given below.

Automated test run - Before running an automated test run, these tasks have to be completed:

1. We need to reconcile with PAS's developers to:
    a) Determine whether the environment we are testing on is in a stable state yet.
    b) Determine which of the letters have to be created. This is because it is not always preferable to generate all of them in each test run, since checking those letters is still done manually and takes quite some time.
    c) Empty the queue of letters that is batched for printing. The print batch should only contain letters generated during the latest test run.
2. The test cases that are responsible for the creation of the letters have to be included in the test case data file.

We recorded the time needed to complete these actions. About one hour is needed to setup an automated test run.

Manual test run - Compared to the setup activities needed before running an automated test run, only 1a and 1c have to be done by the manual testers before a manual test run. When we asked the manual testers, they estimated to spent 30 minutes on these activities.

6.4 Test run analysis and feedback

After a test run is completed, its results have to be analyzed. Feedback to the developers about possible failures has to be given. Again, we can distinguish between the time this takes for an automated test run, and a manual test run. We recorded analysis times during the June release.

Automated test run - After the execution of the automated test run, all of the test cases that failed have to be analyzed. The cause for a failure can often only be determined by reproducing the test case manually. This includes looking at the ID of the test case that failed, looking up this test case in the input files, and re-running it manually with the same parameters. By logging start and end times, we measured 1 hour to be needed to reproduce all of the test set's failures.

After the analysis is done, the results have to be passed on to the developers. In order to be able to log the results, and present them in a clear way, they have to be put into a document. This is also measured to take one hour for the entire test set.

To come to a time estimation for analyzing the entire regression test script, we apply these recorded times (1 hour for test case reproduction and 1 hour for creating a document) for 158 test cases to the total number of 583 test cases. The estimation for reproducing all failed test cases is then calculated as 1 * (582/158) = 3 hours and 40 minutes. The calculation is exactly the same for the document's creation. The total time needed for analysis and

feedback is then estimated by adding up these two values, which results in 7 hours and 20 minutes.

Manual test run - Reproducing the failures is not necessary when applying manual regression testing. The moment the application crashes, the performed actions can immediately be logged. The results of a test run still have to be passed on to the developers though. Since the document for passing the results to the developers is equal to the one the developers have to make, we assume the manual testers to need the same amount of time for creating it, which is 3 hours and 40 minutes.

6.5 Work satisfaction

The testing activities of both testers and developers change when introducing automated GUI testing. During the project, we have interviewed and observed testers and developers during their testing work. We want to determine how the work satisfaction changes for automated GUI testing compared to the work satisfaction for traditional, manual testing. The work satisfaction is included in the experimental data because possible deterioration or improvement in work satisfaction also has to be taken into account in the decision of whether (and how) to apply automated GUI testing.

6.5.1 Work satisfaction for test automaters

The work satisfaction for developing, maintaining and executing DARTH VADER has been analyzed. The only one involved in test automation was myself. This means that we do not know whether the work satisfaction I observed is shared by other developers as well.

DARTH VADER's development - The work satisfaction experienced during design and implementation of DARTH VADER was not different from any other software development project. This personal experience is confirmed by the guidelines that advise us to approach test automation as any other software development.

DARTH VADER and test set maintenance - Maintenance consisted mostly of tracing back errors that caused test cases to fail, which is debugging. The work satisfaction experienced during this debugging was very similar to the work satisfaction experienced while debugging any other piece of software. This was confirmed by the fact that debugging failed test cases became easier and more pleasant over time when the quality of the errors messages and logging was improved.

Execution of the test set - The execution of the test set was the only part of test automation where the work satisfaction was different from the work satisfaction involved in normal software development. Analysis and feedback of the executed test runs was regarded as boring. This was mostly because of the simple and time consuming task of creating the document for feedback to PAS's developers. Reproducing the test cases from the two CSV

input files manually can be hard. Up until the creation of R2D2, this was experienced as a tedious task.

6.5.2 Work satisfaction of executing the regression test script by hand

When interviewing the testers about their testing activities, they indicated that a number of aspects about regression testing were experienced as irritating or boring.

1. In their opinion, the regression test script contained many unnecessary test cases. They regarded some functionality as very simple and superficial, and not worthy of being tested every release. Having to execute these test cases twice for every release caused irritation.

2. Manual regression testing was regarded as boring due to the amount of test cases that had to be executed.

3. Some failures were reported to occur very frequently. The manual tester's attitude towards this recurrence was that it was of no use to report such an failure, because the same failure would probably occur again the next release.

4. The execution of some test cases requires insurees that match very specific conditions to be looked up. A large percentage of these specific insurees have to be looked up frequently because the data on the test environment is refreshed before every release. Testers consider this continuous search for insurees as time consuming and irritating.

5. The generation of letters was described as a very repetitive, boring process.

The way we use DARTH VADER determines how much the work satisfaction of the manual testers can be improved. When it is used in the same way as during our experiment, it addresses only issue 3 and 5. When it is used to replace the manual execution of a part of the regression test script by automated execution, it could address all issues.

6.6 Manual testing effort

Human testing consists of manually executing the regression testing script, and generating all of PAS's letters by hand. The time it takes to complete these activities is discussed below.

Executing the regression testing script - About 60 hours of human labor are needed to execute the full regression test for just one environment. Since the regression test has to be executed in both the test- and acceptance test environment, 120 hours are needed in every release to conduct a full regression test.

Generating and checking PAS's letters manually - Three working days (24 hours in total) are spent by one tester to the test letters in one environment. During these days, only a small number of the letters is tested. If no failures are detected for these letters, this is assumed to be a good indication of quality for all of the letters. Out of the three testing days, 13 hours

are spent on creating the letters. The remaining 11 hours are spent on checking whether the content and layout are correct.

For every release, the letters have to be created in both the test-, and acceptance test environment. This means the time spent on the generation of the letters is 26 hours for each release. Also, for every release, 22 hours are needed to check the letters.

6.7 Automated GUI testing development time

The time needed to create our automated GUI testing solution consists of the creation of the custom extension to CUT, the automation of the regression test set, and the automation of the letter's generation. We have measured the time it takes to automate a limited set of test cases and letters during the development stage. We have extrapolated these measurements to estimate the time needed if we were to automate the entire regression set, and automate the generation of all of the letters. The values we observed and the calculation we did are explained in detail below.

Creating the custom extension - We kept track of the number of hours we spent on designing and implementing our custom extension. The total time was 128 hours.

Automating the regression test set - We recorded the time it takes to automate a blueprint (and one test case that references it). This is 90 minutes. Creating one additional test case that reference this blueprint took 15 minutes on average. If we were to automate the entire regression test, we would need to implement 32 blueprints, and another 550 test cases (582 total test cases, minus the 32 that have been created while creating the blueprints). This would take an estimated (32*90) + (550 * 15) = 11,130 minutes (185 hours and 30 minutes).

Automating the generation of the letters - Their are about 60 different letters, that have to be generated for 10 insurees (each insuree having a slightly different customer profile). The time needed to automate the generation of the first group of letters took 20 hours (1,200 minutes). By making efficient use of gained knowledge, the next group of letters took much less time to automate. We measured 30 minutes for each group. In total, we spent 1,200 + (9*30) = 1,470 minutes for automating the generation of the letters. This set does not include all of the letters, since only 583 out of the 627 letters had to be generated automatically. These letters were excluded because the way these were generated was dependent on data in external systems. To automate the generation of all of the letters, we estimate to spent 1,470 * (627/583) = 1,581 minutes (26 hours and 21 minutes).

The total time needed to automate all human test activity, can be estimated by adding up the total time needed to automate the entire regression test set, the automatic generation of all of the letters, and the creation of DARTH VADER. The estimation for this total is 26 hours and 21 minutes + 185 hours and 30 minutes + 128 hours. This adds up to 339 hours and 51 minutes.

6.8 Summary

A summary of our experimental results is presented. First, the number of test cases executed and the number of failures detected for each test run of DARTH VADER and the manual testers, are presented in Table 6.7.

Test method	**Test environment**	**June release**		**September release**	
		Test cases executed	**Failures reported**	**Test cases executed**	**Failures reported**
Automatic	Test	158	12	256	33
Automatic	Acceptance test	158	13	256	8
Manual	Test	433	8	445	25
Manual	Acceptance test	-	-	-	-

Table 6.7: Number of test cases executed/failures reported for both releases.

Multiple failures were reported by all test runs. The human test run resulted in the detection of other failures than those detected by DARTH VADER's run. The same applies the other way around. Table 6.8 summarizes the causes for failures being missed by either the human or automated test run.

Description of cause	**Missed failures June release**	**Missed failures September release**
Failures missed by the human test run		
Human testers are less strict in their validations.	5	4
The test script's specification is not precise enough.	3	0
DARTH VADER reports a false failure.	1	0
Human testers executed their test run in a later deployment.	1	25
Human testers skipped a test case that results in failure.	1	1
Failures missed by the automated test run		
Not all of the test cases that are executed in the manual test run are yet included in the automated test run.	6	15
Errors in the validations of the automated test set that caused failures to be missed.	1	3
Human testers have a wider scope for signaling failures.	0	2
The human test run is executed in a later version of PAS.	0	1

Table 6.8: The causes of failures being missed by either the human or automated test run for both releases.

Estimates for the time needed to develop and maintain an automated GUI testing solution has been included in the experimental results. The time needed to setup, analyze and execute both the manual and automated test runs, has been recorded as well. A summary of these estimates is given in Table 6.9.

Activity	Automated testing	Manual testing
Maintenance		
Maintenance need due to data refresh in PAS.	4 hours and 30 minutes.	-
Maintenance need due to changes to PAS	3 hours and 45 minutes	-
Setup and analysis		
Test run setup.	1 hour	30 minutes
Test run analysis and feedback.	7 hours and 20 minutes	3 hours and 40 minutes
Execution		
Executing the regression test.	-	60 hours
Generating the letters.	-	13 hours
Checking the letters.	-	11 hours
Development		
Design & implementation of DARTH VADER	128 hours	-
Automating the test set	185 hours and 30 minutes.	-
Automating the letter generation	26 hours and 20 minutes	-

Table 6.9: Estimated time investments required for applying automated GUI testing or manual testing.

The work satisfaction of software developers working on a GUI test automation project, is the same as the work satisfaction of those working on conventional software development projects. Only the reproduction of test cases and the creation of large documents for providing feedback to the AUT's developers, has been experienced as tedious and boring.

The manual execution of the test script is judged as boring due to the large number of test cases and letters. Irritating aspects have also been identified:

- Having to execute test cases that were judged to be unnecessary.
- Testers felt that testing was useless because identical failures re-occurred in almost every release.
- Searching for test subjects that satisfied specific conditions took a large amount of time.

Chapter 7

Lessons learned

This Chapter discusses the lessons we have learned during the course of the thesis. All of these lessons are first described in general. After this, the way they occurred within our thesis project at DSW, is discussed. The following lessons are discussed:

- Section 7.1 discusses how the automation of a test script that is designed for manual execution, can cause additional initial work.
- Section 7.2 discusses how changes to test data can invalidate test cases and cause a need for maintenance. How test data changes, and how it can be avoided is also explained.
- Section 7.3 discusses how using custom GUI elements can harm a testing tool's object recognition.
- Section 7.4 discusses how complying with guidelines right from the start of a project can help in setting goals and limitations, and prevent projects from failing due to pitfalls.
- Section 7.5 discusses how test cases can be made more easily understandable by adding meta data.
- Section 7.6 explains why the maturity of a testing tool should be included in testing tool selection criteria.

Some of these lessons have to be put into perspective because of the limited scope of the thesis project. The project setting, duration and assets included in the experimental data, are limited. How the validity of specific lessons or conclusions is threatened by these limitations, is discussed in Section 7.7.

7.1 Automating manual test scripts

7.1.1 General lesson

The availability of an existing test script is an important criterion for test system selection. However, a test script designed for human execution is often informal in nature. The human

ability to interpret these informal test cases enables them to execute these scripts. However, to execute a test case automatically, it needs to be described as an exact sequence of actions.

As we can see, the human ability to interpret actions is helpful in executing less formally specified test cases. However, because human testers sometimes falsely judge expected and observed results to be equal (and thus, not report a failure while there is one), errors in the regression test script or the AUT can remain hidden

These considerations results in the following lessons:

LL_1 *When automating a testing script that is created to be executed by human testers, a substantial amount of additional work should initially be expected in the form of:*

1. *Adjusting the test cases by describing its actions and preconditions more formally.*
2. *Correcting errors in the regression testing script and AUT because of the strict comparison of observed and expected result.*

7.1.2 Applicability to DSW

PAS's regression testing script does not mention all of the button click or preconditions for every test case. Instead, it describes test cases in an informal way. For the automated testers, some failures occurred due to this informal description. Also, many of the failures that were never reported by human testers, were reported by DARTH VADER because of its strict validations.

7.2 Fixing test data

7.2.1 General lesson

Changing test data can cause test cases to fail. We learned that by storing the test data at a specific moment in time, and restoring it prior to each test run, we have a fixed set of test data. This brings us to the following lesson:

LL_2 *To prevent test cases to become invalid due to data changes in the test data, a test data backup can be restored prior to each test run to be able to run the test set on a fixed data set.*

7.2.2 Applicability to DSW

At DSW, two reasons have been identified that can cause test data to change:

1. *Data refreshment policy* - Keeping an up to date data set to test on, is regarded as an important asset at DSW. Refreshing the test and acceptance environment with new test data every few months has been done for years. However, refreshing test data causes additional work for testers. For manual testers, a data refresh can invalidate the selection of certain insurees as test subject, for example. For automated GUI

testing, this problem is even aggravated because every detail of every test case is completely defined in a script.

2. *Data-changing mutations* - Some test cases contain mutations that change the test data. We tried to undo these changes by executing a test case containing the opposite transaction of the data-changing mutation. However, we experienced many drawbacks when applying this solution:

   a) Some specific mutations cannot be undone.

   b) One additional test case has to be created for every test case that involves a data-changing mutation.

   c) When the test case containing the data-changing mutation fails or is not executed, it does not have to be rolled back. This means the additional test case containing the undo-mutations should not be executed. This creates interdependencies between different test cases. These interdependencies make the test set more complex.

   d) When the test case containing the undo-mutation is not executed, the test case containing the data-changing mutation cannot execute properly in the next test run (since its preconditions are not restored to what they are expected to be). This creates interdependencies between different test runs.

   Another option is to use a series of test cases to create or alter data before executing the test run. However, this requires additional test cases to be implemented. On top of that, this solution also suffers from the problem of test cases being interdependent.

By applying the lesson, storing the test data at a specific moment in time, and restoring it prior to each test run, we have a fixed set of test data. This solves both the problem of undoing data-changing mutations, and test data refreshes.

7.3 Custom GUI elements

7.3.1 General lesson

The fact that third party controls (widgets) are sometimes not recognized by the testing tool is a known pitfall (**PF_13**). However, the definition of this pitfall should be extended. Custom created GUI elements (even when created as a composition of multiple GUI elements that are all recognized by the testing tool), might also cause recognition problems. The AUT's developers should keep this in mind while making changes to the AUT. This results in the following lessons:

LL_3 *Developers have to realize that making use of custom made GUI elements can result in recognition problems with the testing tool, and thereby reduce testability of the AUT.*

7.3.2 Applicability to DSW

Visual Studio contains a large toolkit with commonly used GUI elements. All of these elements are supported by CUT, which means DARTH VADER can recognize and interact with them.

Sometimes, custom GUI elements are created (or existing ones are extended) when the default toolkit does not suffice. These custom created GUI elements are not always fully recognized by CUT. This can often be solved by using CUT in a creative way, or by extending the GUI capturing capabilities. However, this is an error prone, time consuming activity.

7.4 Compliance with guidelines

7.4.1 General lesson

Many guidelines exist addressing choices that have to be made early in the software development life cycle (design, choosing a testing tool, and so on). Since changing a design or testing tool later on in a project is very costly, this means the penalty of learning these lessons by experience is very high. The line between a successful test automation project, and a failure, is very thin when errors are made in these choices. Using guidelines is vital to keep the change for failure as small as possible.

Taking the guidelines into consideration is also useful for setting certain limitations and goals for the design and implementation of an automated GUI testing solution. This brings us to the following lesson:

LL_4 *Compliance with guidelines right from the start of a project, helps in setting goals and limitations, while minimizing the chance for project failure.*

7.4.2 Applicability to DSW

Guidelines and pitfalls were used for a number of stages in our project in the following way:

1. *Selection of a test system* - Several requirements that a test system should fulfill have been composed. We could discard Eureka and DSW's website from our selection since these did not fulfill these requirements. PAS fulfilled all of the requirements and was selected as test system.

2. *Testing tool selection* - Requirements for testing tool selection have been derived from the guidelines. TestComplete, AppTest and Squish failed to meet these requirements (especially GUI element recognition), and were discarded from the selection. CUT fulfilled all of the requirements and was chosen as our testing library.

3. *DARTH VADER's design* - The following aspects of DARTH VADER's design are based on guidelines:

    - The creation of a framework.

- The separation of business logic from test case data by separating the test case data file and the test case logic file.
- The creation of the UIElements class that contains a more easy to use and concise GUI map compared to the GUI map made by CUT.
- Allowing blueprints to include each other's definition to facilitate reuse of business logic.

7.5 Understandability of test cases

7.5.1 General lesson

The understandability of test cases is important for their maintainability. However, the requirements to reuse test case logic and separate test case data and test case logic, often lead to the use of a complex file format. This can make the understandability of test cases hard. This leads to the following lesson:

LL_5 *Meta data or tool support should be used to help users of the testware with understanding a complex file format.*

7.5.2 Applicability to DSW

Parsing the test cases is no problem for DARTH VADER. For a human however, it can be hard to translate an identifier, a link to a test execution blueprint and a set of parameters to a clear mental picture of what the test case does. To make it easier to create a mental picture of what a test case does, the test case description from the regression testing document is included in the test case data file.

However, even though this description helps, there is still much room for improvement. R2D2 can be extended in a number of ways to make it even easier to understand test cases.

7.6 Using new testing libraries

7.6.1 General lesson

Existing guidelines and pitfalls stress the importance of checking whether a testing tool includes all of the required functionality. However, the value of choosing for proven technology is not stated for these requirements. Given the substantial investment involved in introducing automated GUI testing, this is almost always advisable. A testing tool or library that has not matured yet might be unstable, contain many bugs, have little documentation or have a small user base. The lesson learned can be stated as follows:

LL_6 *The value of choosing for proven technology, should be considered in testing tool selection.*

7.6.2 Applicability to DSW

The CUT library is still fairly new. Some problems resulting from this currently short lifespan are experienced during our thesis project:

- At this moment, very little official documentation existed for CUT. This can be problematic when it is unclear what certain functionality does, or what its side effects are.
- When encountering difficulties during implementing, searching on blogs and forums for other developers who have experienced the same problems, sometimes leads to an answer. However, the user base of CUT is not yet very large. This means the chances of finding blogs or forum threads about similar problems are slim.

7.7 Putting the experimental results and lessons into perspective

This Section discusses the threats to the validity of our lessons and experimental results. Our results are based on one project undertaken exclusively at DSW with just one full time project member. Subsection 7.7.1 explains how this puts a limitation to the relevance of our results. How increasing experience, initial adaptation of the test script, long term rise in maintenance cost, and the out dating of test data, influences the relevance of our lessons and conclusions, is explained in Subsection 7.7.2. In Subsection 7.7.3 we discusses how validity is threatened by not including ad-hoc work, supporting work by other employees and the flexibility offered by automated GUI testing, from our experiment.

7.7.1 Limited scope regarding the setting of the project

A number of aspects in which the thesis project's scope is limited, and the consequences this has on the validity of our experimental results and lessons learned, are discussed below.

Results are based on one project - Every project is different with respect to the AUT, testing strategy, time constraints, and so on. This means lessons learned on the current project do not necessarily have to apply to other projects. For example, the lesson to use fixed test data for a long time might not be applicable to projects which are meant to use up to date data.

Data gathered at one company - Some considerations made during the thesis are largely based on the situation at DSW. This means that care has to be taken when applying them at other companies. For example, manual testing at DSW is done by the same people who are everyday users of the AUT, while other companies use dedicated testers. It is important to keep this in mind when analyzing the results of the comparison between automated and human test runs.

Performance in test automation based on one person - Most of the work done on this project is done by just one person. The only involvement of outsiders consists of supporting tasks,

and reviewing of the design and important decisions. Working alone, little overhead time is needed for communication with co workers. This is not a very realistic or desirable situation for a real life application of automated GUI testing. We should keep in mind that in a scenario where a team is working on an automated GUI testing solution, some extra overhead time is needed.

Testing is done on a internal application - PAS is an application that is not accessible to external customers. Since quality requirements for systems that are accessible to customers are different, some conclusions and lessons might not be applicable when testing such a system. For example, our conclusions might not be applicable to testing a website (which often has strict cosmetic requirements).

Also, all of the experimental results regarding time measurements for development, maintenance, analysis, and so on, are all based on one person. Other persons might work slower or faster. The time measured for completing certain activities should not be regarded as a precisely calculated average, but more a coarse indication with a large margin.

7.7.2 Measurements based on limited duration of the project

The project lasted for little more than one year. This also results in threats to validity of our results. An overview of these threats is given below.

No prior experience regarding automated GUI testing - Both DSW and the developer have very limited prior experience when it comes to automated GUI testing. Development of such a solution for the second time likely takes less time than the first time. Experience also influences the time needed for analyzing test runs and maintaining the test set. For later test runs, the time needed for those activities is also likely to decrease. These two observations mean we have to put the recorded time for development, analysis and maintenance in perspective.

Regression test script adjusted after initial stage - As explained in the lessons learned, much initial work can be expected to adjust the regression testing script to the script execution and validation of the automated test run. After two testing periods, many of these issues have been solved. Maintenance and analysis times will likely decrease in later releases.

Maintenance cost can rise on the long term - Because the requirements for software systems (both the AUT and DARTH VADER) are expected to change over time, maintenance costs usually increases later on in the software lifecycle. Within the time period of this thesis, we cannot measure how substantially this influences the maintenance costs. When drawing conclusions, we need to take possibly rising maintenance costs into account.

Test data becomes outdated - Freezing the test data at a certain moment in time did not influence the test runs in a negative way so far. However, the effects on the long run are

unknown. Several people opposed the idea of using fixed test data because over time, the situation that is tested, is not up to date and realistic anymore. The possibility that this might become a problem needs to be taken into consideration.

Limited changes to the AUT - The two releases in which DARTH VADER was executed, featured only limited changes to PAS. Only bug fixes and changes to existing functionality were implemented. Conclusions on the ability of DARTH VADER to cope with changes in PAS do not take into account the ability of DARTH VADER to adapt to new functionality.

7.7.3 Activities excluded from the experiment

Experimental data has been gathered about the most prominent assets of automated GUI testing. Some less prominent assets have not been included, but could still influence our conclusion. They are discussed below.

Ad-hoc work - On a number of occasions, the developers of PAS requested a test run to smoke test some minor changes that were applied to PAS outside of the release cycle. Once the awareness rises that the facilities are in place to execute such cheap test runs, the demand will probably grow. This realization puts the potential payback of DARTH VADER into perspective, since our experiment did not consider test runs that are not part of a release.

Supporting work by other employees - Some overhead in coordinating test runs of DARTH VADER is done by test coordinators. Adjusting the test script to suit the needs of DARTH VADER also takes time from co workers. The time these activities take is not measured in our experiment. When drawing our conclusions, we have to take into account that certain small investments of time are left out of the equation.

Flexibility of automated GUI testing - It can be hard to reserve personal for testing activities during the testing periods. Holiday, illness and competing activities all have to be taken into consideration when working with human personnel. Although running DARTH VADER also requires human involvement, this is much less substantial than human testing. Also, analysis and maintenance required for automated testing do not have to be done at some exact moment in time. Instead, it can be chosen (within certain constraints) freely. Another example of flexibility of automated GUI testing, is the possibility to execute nightly test runs. These advantages are not included in the experimental data.

Skill level required for operating an automated testing tools - Executing a test script manually requires some background knowledge of the test system. However, when given a little time, almost anyone can execute a regression test script. On the contrary, operating an automated testing tool is more complex, and much more learning time is required to use it efficiently.

Chapter 8

Conclusions and recommendations

In Section 8.1, we discuss why hypothesis 1 is confirmed, hypotheses 2 is refuted and hypothesis 3 remains unresolved. Since hypothesis 1 is confirmed, we know automated GUI testing can be applied in a way that benefits DSW. To answer our research question, we will explain *how* automated GUI testing should be applied to be beneficial. This is discussed in Section 8.2. Even though the benefits of automated GUI testing outweigh the costs, improvements are still possible. Section 8.3 explains how these can be achieved: reduction of the number of the false failures, running DARTH VADER overnight on the build server, extending R2D2 to reproduce test cases, and assigning test cases to either the automated *or* the manual test set.

In Section 8.4, future research that should be conducted to improve the applicability of automated GUI testing is proposed. Guidelines for time requirements regarding GUI test automation should be composed, as well as models for estimating the long term loss of effectiveness of the automated GUI test set. Also, research towards finding possibilities to group test cases by preferred method of execution should be undertaken. Furthermore, research is needed towards the cost of repairing specific kinds of bugs that are detected in a specific environment.

We conclude our thesis in Section 8.5.

8.1 Refuting or confirming the hypotheses

Subsection 8.1.1 explains that higher bug detection, improved work satisfaction and preventing untested changes to PAS from being taken into production, leads us to confirming the first hypothesis. DARTH VADER does not bring down the required number of test runs. It also cannot act as a replacement for a necessary human test run. This leads us to refuting the second hypothesis, as is explained in Subsection 8.1.2. How the inability to quantize the improved bug detection due to lacking data leads to hypothesis 3 remaining unresolved, is explained in Subsection 8.1.3.

8.1.1 Hypotheses 1 - Using cost-benefit analysis, automated GUI testing on PAS benefits DSW more than it costs.

The first step in cost-benefit analysis, is the identification of the different costs and benefits that have been observed during the course of our research. We restrict ourselves to the costs and benefits connected to the choice whether to apply automated GUI testing or not. These costs and benefits are depicted in Table 8.1.

We can simplify the comparison of costs and benefits by removing common factors that are present in both costs and benefits.

On the cost side, we recognize that two automated test runs are executed for each release. This means we can estimate the average maintenance time for each test run by dividing 3 hours and 45 minutes by two. This results in about 1 hour and 50 minutes. By adding the cost for each test run, 8 hours and 30 minutes, to this value, we come to a total estimate of 10 hours and 20 minutes.

On the benefit side, the estimated 26 hours saved by letter creation are not realistic. This value is based on the assumption that all letters are generated by the human testers, which is not the case (only a cross section is created). The actual time saved is less. We also take into regard the fact that test automation personnel (software developers) are more expensive than manual testers.

When comparing the 10 hours and 20 minutes of expensive developer time, with a fraction (of unknown size) of the 26 hours of cheaper human test time, we assume them to be in the same order of magnitude, and leave them out of the equation.

The cost-benefit comparison is now simplified to a large initial time investment on one side, versus higher bug detection, prevention of boring and repetitive work, and lowering the amount of changes that go into production untested, on the other side. We judge the benefits to outweigh these costs, which means we confirm the hypothesis. The arguments for this decision are given below.

1. PAS is a critical system.
    a) Preventing changes to be taken into production without being tested (being tested at all, or being tested on the acceptance environment) is regarded as very important.
    b) Higher bug detection, although judged as less severe because of the nature of most of the bugs being detected, is also valued very high.
2. Work satisfaction is an important DSW company value. A substantial part of several employee's work consists of testing activities. Raising the work satisfaction for these employees is regarded as important.
3. All variables left on the cost side are one-time investments, while the variables on the benefit side are applicable to every test run. PAS has a long lifetime left, so plenty of test runs will be executed. Each test run will bring down the relative cost of the initial investment. Over time, the benefits will outweigh the costs.

Costs	Benefits
Cost/benefits for each test run	
1. Setup time needed for an automated test run is measured to be 1 hour. 2. Analysis and feedback time needed for an automated test run is measured to be 7 hours and 30 minutes. In total, one automated test run costs 8 hours and 30 minutes.	1. Bug detection has improved. 2. Work satisfaction has improved by preventing the boring, repetitive generation of letters to be done by human. Possibilities for improving work satisfaction by removing trivial test cases from the test script and having DARTH VADER execute those automatically. 3. Executing the letter generation automatically prevents 26 hours of human labor. 4. No untested changes are taken into production when time pressure causes a human regression test to be (partially) skipped.
Cost/benefits for each release	
Maintenance to testware and test set takes 3 hours and 45 minutes.	
One-time cost/benefits	
1. DARTH VADER's design and implementation - 128 hours. 2. Automating the regression test script - 185 hours and 30 minutes. 3. Automate the generation of the letters - 26 hours and 30 minutes. Total one-time costs: 340 hours.	

Table 8.1: The estimated costs and benefits connected to the choice of whether to apply automated GUI testing or not.

8.1.2 Hypotheses 2 - Having an automatic regression test set for the PAS project reduces the number of executions of the manual regression test.

Ideally, every bugfix should be manually retested until no more bugs are found. This means multiple human test runs should be executed on each environment in every release. During the course of the research, we found out that this ideal was unrealistic. Even executing one full manual test run was often problematic. Often, parts of test runs were skipped, letter generation was only done for a cross-section of the total, and acceptance testing was restricted to a minimum. Since human testing is already at a minimal level, the idea of minimizing human test effort by retesting the application automatically until no more failures are reported, does not apply. We cannot confirm the hypothesis on this ground.

All of the automated test runs missed several failures that were detected by the human testers. The limited size of the automated test set can be accounted for most of these missed failures, but several failures were also missed due to the limited scope of an automated GUI testing solution, and due to errors in the validation rules. This means that replacing the human test run by an automated test run, causes bugs to be missed. This lowers the quality of testing.

Summarizing this, we refute this hypothesis on the following grounds:

1. Lowering the necessary number of human test runs is not possible, since the number of human test runs is already at a minimum.

2. Replacing a necessary human test run by an automatic test run is not possible, since this would compromise the quality of the regression test.

8.1.3 Hypotheses 3 - Having an automatic regression test set for the PAS project lowers the average relative cost of a bug.

As research indicates, the earlier in the software development cycle a bug is found, the cheaper it is to fix [4, 5, 25]. These studies state that the effort needed to repair bugs when they are found in the test environment, is less than the effort needed when bugs are found in the development environment. The studies stated base their results on the fact that when a bug is found in the test environment, the following activities need to be completed before the bug is removed:

Report Reporting the bug back to the developers

Repair Repairing the bug.

Update Bringing a new version of the AUT to the test environment.

This contrary to bugs found in the development environment, where only the **repair** step is needed to repair a bug.

Since we are able to execute our test run before the human testers, we could report bugs earlier. However, when analyzing which activities need to be completed before a bug is

Automated test	**manual test**
Report	
The time needed to analyze an automated test run is higher than for a manual test run. This is because a failure in the automated test run, has to be reproduced before it can be reported back to the developers. Also, some failures are based on misinterpreted expected functionality. When there is doubt whether a failure really is a bug, this has to be reconciled with the manual testers.	When a test case fails during manual execution, reproducing the test case is not much easier. Also, there is no need to reconcile anyone about failed test cases.
Repair	
Solving an error is done by PAS's developers. This activity is completely the same for bugs resulting from the manual, or automated test run.	
Update	
PAS is run locally. All of the services PAS depends on also run locally. Since no other users can access this version of PAS, updating this copy is easier than updating the test environment.	The test environment has to be updated. Multiple simultaneous users accessing this server at the same time make this harder than a local update.

Table 8.2: Activities needed to repair a bug that is detected by either the manual or the automated test run.

solved after being found by DARTH VADER, we cannot conclude the average cost to be lower than for bugs found by human tester. In Table 8.2, we compare the activities needed for repairing bugs found by either DARTH VADER, or the manual test run.

The differences between automated GUI testing and manual testing, are found in the **report** and **update** step. Automated GUI testing has a slight advantage in the **update** step. However, especially due to the need to reproduce a test case from two CSV files, this advantage is easily canceled out in the **report** step.

On the contrary, automated GUI testing improves bug detection. Some of the bugs that were found by DARTH VADER, were not found by the human testers. If these bugs slip through to the production environment and are found their, the cost to repair them is substantially higher.

To come to a conclusion, we have to compare cheaper bug reporting, by higher bug detection. However, we do not have sufficient data concerning the number of bugs that would slip through to the production environment, and the cost of repairing these bugs. This means we are not able to make this comparison, which in turn means this hypothesis remains unresolved.

8.2 Answering the research question

The research question we want to answer is the following.

How can automated GUI testing be applied in a way that benefits DSW?

The confirmation of hypothesis 1 shows that automated GUI testing can be applied at DSW in such a way that the benefits outweigh the costs. The primary factor that made our solution successful, is compliance with the guidelines. Considering these guidelines in an automated GUI testing project is vital. Apart from following guidelines, three approaches that added greatly to our solution have been identified. They are listed below:

1. *Using fixed test-data* - Maintenance due to data refresh has been measured to take 4 hours and 30 minutes for each release. The usage of fixed test data completely removed the need for this maintenance. Creation time for creating data changing test cases also dropped after using fixed test data.

2. *Automating repetitive tasks completely* - An important contribution to the benefits of automated GUI testing, is the automatic generation of PAS's letters. Possibilities to automate boring manual labor should be embraced.

3. *Debugging regression test script and test set* - DARTH VADER executes the test script very precisely. By debugging the test script and test set, we could gradually improve the quality of DARTH VADER.

8.3 Recommendations for extending automated GUI testing at DSW

Even though our approach was concluded to be beneficial to DSW, there is still room for improvement. Subsection 8.3.1 explain how we can improve our solution by reducing the number of false failures. Subsection 8.3.2 explains that bugs could be detected earlier and quality control could be improved by running the automated test set every night on the build server. Subsection 8.3.3 explains why R2D2 should be extended to reproduce failed test cases. Finally, Subsection 8.3.4 discusses how assigning test cases to either the automated or manual test set, improves work satisfaction and saves time.

8.3.1 Reduce the number of false failures

In the past two releases, many failures detected by DARTH VADER could be accounted to errors in the regression testing script. Expected results were not defined correctly, preconditions were not included, and so on. Many changes have been made to the regression testing script to reduce the number of these failures. However, some errors still exist. Removing these errors results in less false failures, and would bring down the test run analysis and feedback time.

8.3.2 Run DARTH VADER overnight on the build server

When the setup of a test run is automated, and DARTH VADER can be run daily on the build server (where the newest version of PAS is built every night), the bugs would be detected earlier in the software development process. This would lower the average cost of removing a bug. A large part of DARTH VADER's costs consists of the initial development. This is a one-time cost, contrary to DARTH VADER's benefits, which occur in each test run. This means the relative costs for each test run drops when DARTH VADER is executed more often.

Also, a daily run would provide continuous feedback about the quality of PAS and the regression test script. This helps in guarding quality standards.

8.3.3 Extend R2D2 to reproduce failed test cases

A large part of the analysis time for an automated test run consists of reproducing failed test cases. To ease this problem, R2D2 should be extended to support stepwise execution of a test case. The executed commands should be displayed to the user, and executed subsequently. In this way, it becomes easier for the user to see why, and at which point, a test case fails.

8.3.4 Assigning test cases to either the manual, or automated test run

DARTH VADER is better at detecting failures than the human testers when it comes to checking for default values, restrictions on input field values, and error messages. Removing these test cases from the manual test run would not compromise the regression test quality. On the other hand, since human testers are much better at validating cosmetic functionality, there is no added value to including such test cases in the automated test set.

The following improvements could be made by labeling test cases with a preferred method of execution:

- Work satisfaction is improved:
  - There is no need for human testers to execute trivial, repetitive test cases.
  - There is no need for test automation personnel to implement complicated validations that could easily be done by humans in a fraction of the time.
- Time is saved because not all test cases have be to implemented in an automated test set *and* executed by hand in a human test run.

8.4 Recommendations for future research

This Section gives some pointers for possible subjects for future research. In Subsection 8.4.1, the need for research to establish guidelines on the time taken to introduce automated GUI testing, is explained. How research into the loss of effectiveness of a test set due to changing test data and GUI of the AUT can help to improve decision making and planning, is explained in Subsection 8.4.2. Subsection 8.4.3 discusses why research towards

the possibilities of labeling test cases with a preferred execution method is needed. Finally, Subsection 8.4.4 explains the need for future research towards the cost of repairing specific kind of bugs in specific environments.

8.4.1 Developing guidelines for time requirements regarding GUI test automation

One of the most important criteria for companies to decide whether automated GUI testing should be applied, is the amount of time that has to be invested. However, very few guidelines exist to make an estimation of the amount of time that is required. This can lead to large losses in two ways. First, companies that could benefit from automated GUI testing, might be discouraged to use it because they overestimate the cost involved in its introduction. Second, companies can abandon automated GUI testing projects after being disappointed by unexpected high costs. Developing a set of guidelines to make estimations can prevent these problems.

8.4.2 Long term loss of effectiveness of the automated GUI test set

Over time, changing of the test data and the AUT's GUI reduces the effectiveness of an automated GUI test set. Preventing this drop in effectiveness by adapting the test set, is the main cause for maintenance. The maintenance cost is an important part of the cost-benefit analysis to judge whether automated GUI testing is beneficial to a company. This means research aimed at estimating the speed in which a test set loses its effectiveness helps in making the decision whether to apply automated GUI testing or not. When companies are able to make a good estimation of the time needed for maintenance, reserving resources for maintaining the automated GUI testing solution can also be improved.

8.4.3 Categorizing test cases for preferred method of execution

The execution of test cases to validate, among others, default field values and error messages, can be left to DARTH VADER. However, the relevance of this conclusion is limited because of the scope of our project. To increase the relevance, and to know whether this conclusion is valid in a general context, more research is needed. In general, a categorization of test cases to be able to label them with a preferred method of execution (manual or automatic) would be helpful. Work satisfaction, bug detection and work needed to execute such test cases, should all be considered in this research.

8.4.4 Repair costs for specific kinds of bugs

The relative cost to repair bugs that are found in either the development, test or production environment has been investigated by a number of researchers [4, 5, 25]. However, the type of a bug (bugs in critical functionality versus bugs in non-critical functionality for example) is not taken into regard. Also, the setup for the automated GUI testing environment does not match any of the environments described by these researchers. Different relative costs to repair bugs are involved when environments are not setup in the same way. Hypothesis 3

remains unresolved because of this lacking data. More research about cost to repair specific kind of bugs in a specific environment would be needed to come to a conclusion on this hypothesis.

8.5 Conclusion

Hypothesis 1 is confirmed because using automated GUI testing improves bug detection and work satisfaction, and reduces the amount of untested changed that are taken into production, at acceptable costs. Having an automated regression test does not reduce the number of executions of the manual regression test, which means we refute hypothesis 2. The reasons for this are the inability of automated GUI testing to bring down the number of necessary human test runs, and the inability to replace a human run by an automated one without lowering the regression testing quality. Hypothesis 3 remains unresolved. The time saved by humans being faster at reproducing and reporting bugs could not be compared with potentially high costs due to undetected bugs slipping through to the production environment.

We can apply automated GUI testing in a way that is beneficial to DSW, by following the guidelines and lessons learned. On top of that, using fixed test data, automating repetitive tasks completely, and debugging the regression test document and test set, are helpful as well. This answers our research question.

Future work at DSW should be aimed at reducing the number of false failures, incorporating a test run of DARTH VADER on the build server, extending R2D2 to be able to reproduce failed test cases easily, and assigning test cases to either the manual, or the automated test set. This would all improve the payback of applying automated GUI testing.

Finally, future research should be aimed at:

1. Developing guidelines for time requirements regarding GUI test automation
2. Long term loss of effectiveness of the automated GUI test set
3. Categorizing test cases for preferred method of execution
4. Repair costs for specific kinds of bugs in a specific environment.

Bibliography

[1] WinRunner User's Guide. http://www.cbueche.de/WinRunner%20User%20Guide.pdf, January 2003.

[2] M. Zhan B. Mu and L. Hu. Design and Implementation of GUI Automated Testing Framework Based on XML. In *Proceedings of the 2009 WRI World Congress on Software Engineering - Volume 04*, WCSE '09, pages 194–199, Washington, DC, USA, 2009. IEEE Computer Society.

[3] K. Bhaggen. Test Automation in Practice. http://swerl.tudelft.nl/twiki/pub/Main/PastAndCurrentMScProjects/Thesis_Kishenkumar_Bhaggan.pdf, September 2009.

[4] B. Boehm. Software Engineering Keypoints - Revisiting Software Engineering Economics. http://www.cs.stevens.edu/~lbernste/cs552spr07/Lectures/CS%20552%20SER.pdf, March 2007.

[5] M.R. Chin-Yu Huang; Lyu. Optimal Release Time for Software Systems Considering Cost, Testing-Effort, and Test Efficiency. *IEEE Transactions on Reliability*, 54(4):583 – 591, 2005.

[6] AppPerfect Corporation. GUI/ .NET Functional Testing. http://www.appperfect.com/products/app-test.html, January 2010.

[7] Microsoft Corporation. Visual Studio Team Foundation Server 2010. http://www.microsoft.com/visualstudio/en-gb/products/2010-editions/team-foundation-server, January 2010.

[8] Microsoft Corporation. What's New for Testing. http://msdn.microsoft.com/en-us/library/bb385901.aspx, January 2011.

[9] E. Dustin. Lessons in Test Automation. *Software Testing & Quality Engineering magazine*, pages 16 – 21, September/October 1999.

[10] J. Rashka Dustin, E. and J. Paul. *Automated Software Testing: Introduction, Management, and Performance.* Addison-Wesley, 1999.

[11] Wiki editors. Extensible Stylesheet Language Transformations. http://en.wikipedia.org/wiki/XSLT, September 2011.

[12] M. Fowler. Fluent Interface. http://www.martinfowler.com/bliki/FluentInterface.html, December 2005.

[13] David Garlan and Mary Shaw. An Introduction to Software Architecture. Technical report, Pittsburgh, PA, USA, 1994.

[14] Froglogic GmbH. Froglogic - Squish. http://www.froglogic.com/products/editions.php, January 2011.

[15] C. Kaner. Pitfalls and Strategies in Automated Testing. *Computer*, 30:114–116, April 1997.

[16] Steve McConnell. *Code Complete*. Microsoft Press, 2004.

[17] Scott McMaster and Atif M. Memon. An Extensible Heuristic-Based Framework for GUI Test Case Maintenance. In *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, ICSTW '09, TESTBEDS workshop GUI-Based Applications, pages 251–254, Washington, DC, USA, 2009. IEEE Computer Society.

[18] Atif Memon. Automatically Repairing Event Sequence-Based GUI Test Suites for Regression Testing. In *ACM transactions on Software Engineering and Methodology*, volume 18, pages 4:1–4:36, New York, NY, USA, November 2008. ACM Press.

[19] Christer Persson and Nur Yilmazturk. Establishment of Automated Regression Testing at ABB: Industrial Experience Report on Avoiding the Pitfalls. In *Proceedings of the 19th International Conference on Automated Software Engineering (ASE)*, ASE, Vasteras, Sweden, 2004.

[20] Alex Ruiz and Yvonne Wang Price. GUI testing made easy. In *Practice and Research Techniques. Testing: Academic & Industrial Conference*, PART TAIC, pages 99–103, 2008.

[21] Per Runeson. A Survey of Unit Testing Practices. *IEEE Computer Society*, July/August 2006.

[22] R. Weber S. Berner and R.K. Keller. Observations and Lessons Learned from Automated Testing. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 571–579, New York, NY, USA, 2005. ACM.

[23] SmartBear Software. Automated Testing Tools - TestComplete. http://smartbear.com/products/qa-tools/automated-testing/, January 2011.

[24] L. van Delft. Introducing Automated GUI Testing to DSW, February 2011.

[25] R.L. Vienneau. The Cost of Testing Software. In *Proceedings of the Annual Reliability and Maintainability Symposium*, pages 423 – 427, Orlando , Florida, USA, 1991. IEEE Computer Society.

[26] Y. Shen Z. Nanyang Zunliang Yin Miao, C. Miao. Actionable Knowledge Model for GUI Regression Testing. In *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, IAT '05, pages 165–168, Washington, DC, USA, 2005. IEEE Computer Society.

Appendix A

Appendix: Schematic view of thread of control

In this appendix, the sequence diagrams depicting the the thread on control from a number of viewpoints.

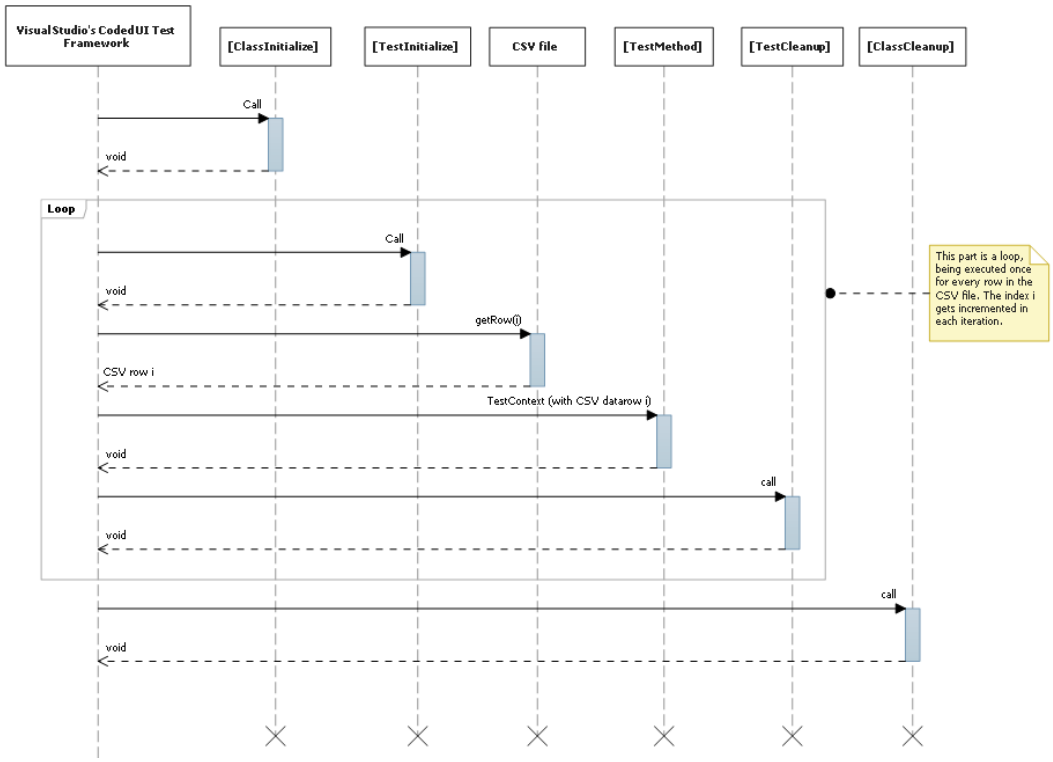


Figure A.1: Thread of control within one class containing a method labeled with [TestMethod]

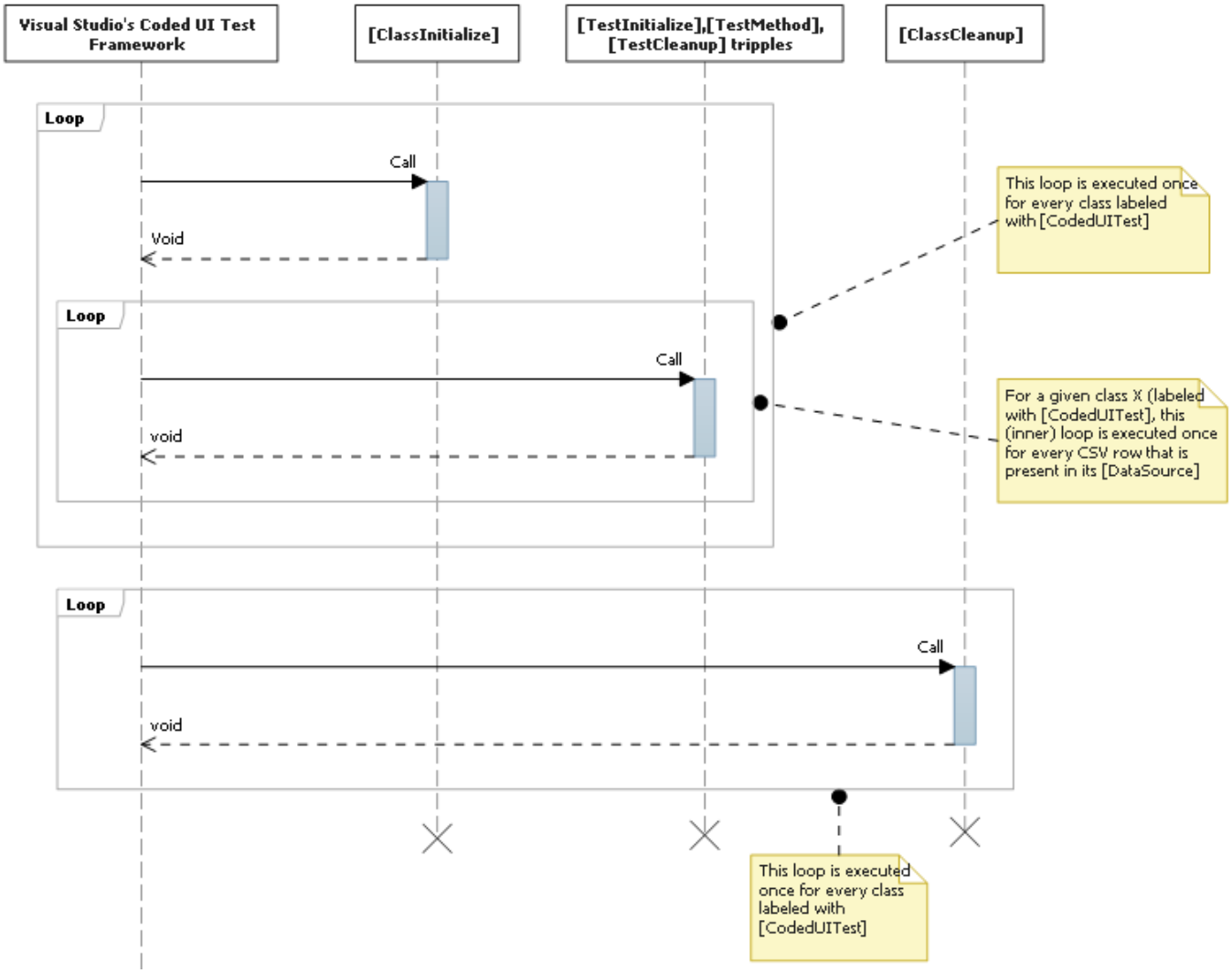


Figure A.2: Thread of control between multiple classes that contain methods labeled with [TestMethod]

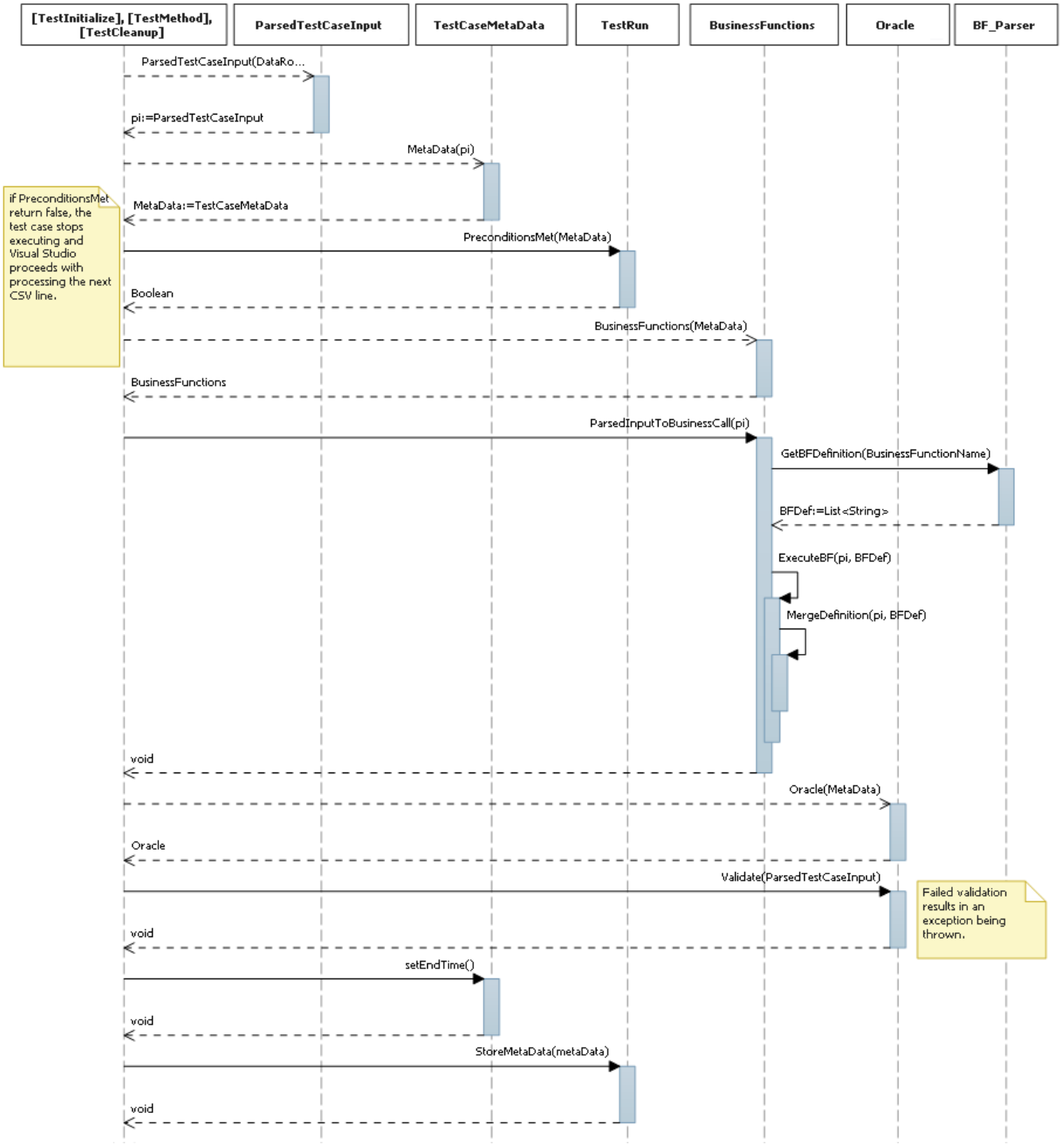


Figure A.3: Thread of control for one test case (starting in a method labeled with [TestMethod])

Appendix B

Appendix: Requirements Definition Document

This appendix displays the Requirements Definition document. Version 1.3 (the final version) is depicted below.

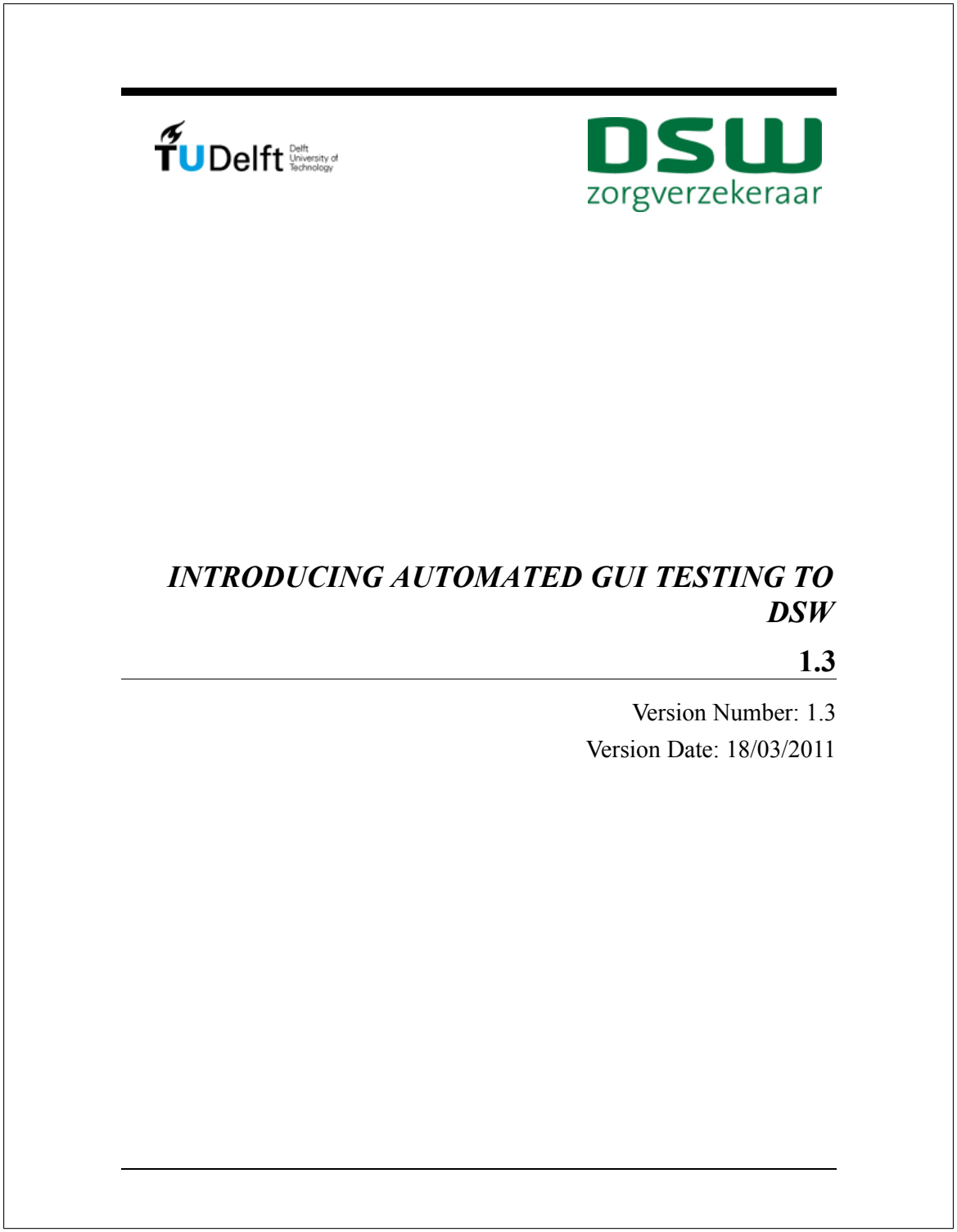


Figure B.1: Requirements Definition Document - page 1

TABLE OF CONTENTS

1 INTRODUCTION.....3

2 FUNCTIONAL REQUIREMENTS.....3

3 NON-FUNCTIONAL REQUIREMENTS.....4

4 MOSCOW MODEL.....5

Figure B.2: Requirements Definition Document - page 2

1 INTRODUCTION

1.1 PURPOSE OF THE REQUIREMENTS DEFINITION DOCUMENT

The requirements definition document defines the functional and non-functional requirements for the application we will build in this thesis project. The requirements definition document is created during the requirements definition phase of the project. It is intended to clearly present which requirements the automated GUI tester will have to satisfy.

The project's final deliverable, the automated GUI tester, will be called DSW's Automated Regression Tester Helps Validating And Does Error Reporting (DARTH VADER). From now on, when we want to refer to the automated GUI tester, we will use this abbreviation.

2 FUNCTIONAL REQUIREMENTS

This chapter will specify which functional requirements DARTH VADER will fulfill. Each of the requirements will be labeled. In order to make the labels more meaningful, different categories for the requirements will be made. Labels will include this categorization.

The final section of this chapter will explain why and how the requirements will be prioritized.

2.1 GENERAL

FR_GEN_1: DARTH VADER will be able to execute test cases

FR_GEN_2: DARTH VADER will be executable by developers

FR_GEN_3: DARTH VADER will be executable by testers

2.2 INPUT/OUTPUT

FR_IO_1: DARTH VADER will provide the developers with debugging messages to monitor the correct functioning of the DARTH VADER application itself.

FR_IO_2: DARTH VADER will log general data regarding a test suite's execution on the PAS application: the number of failed tests, identifiers for failed tests, and test execution time.

FR_IO_3: DARTH VADER will log detailed feedback about the execution of each test case on the PAS application. The reason for failure (in case of failure), error line numbers, and so on, are all provided.

FR_IO_4: DARTH VADER will be able to do data driven testing (this would imply test case content can change without necessitating changes to the DARTH VADER implementation)

FR_IO_5: Test cases will be incorporated in the codebase of DARTH VADER, with reuse of code taken into account. Because of this, only limited change to the implementation of DARTH VADER will be needed, in case the regression test script changes.

FR_IO_6: Test cases will be incorporated in the codebase of DARTH VADER in such a way that, in case the regression test script changes, a proportionally big change to the implementation would be needed.

FR_IO_7: The DARTH VADER application will be able to use so called "Fields with a

Figure B.3: Requirements Definition Document - page 3

mask".

2.3 CODE COVERAGE

FR_CC_1: Visual studio will be configured in such a way it measures the code covered by DARTH VADER. This is not really a requirement of the DARTH VADER application itself, but it is a project requirement.

FR_CC_2: DARTH VADER will cover 70% of PAS's code by executing the test suite.

2.4 TEST SUITE

FR_TS_1: All test cases in the manual regression testing script, except for those that need access to other systems to be validated, will be translated into automated test cases.

FR_TS_2: A subset of the manual regression test script will be translated into automated test cases. Test cases concerning use cases that form the basic of functionality PAS provides, will be picked first.

2.5 TEST EXECUTION

FR_TE_1: A test run will not stop after failure of one test case. Succeeding test cases will still be executed.

FR_TE_2: This requirement is an addition to FR_TE_1. Dependencies between test cases will be defined. The failure of a test case that has dependent test cases, will prevent those dependent test cases from being executed. This make it easier to distinguish a true test case failure from one that did not have its preconditions met, and failed because of that.

2.6 PRIORITIZING THE REQUIREMENTS

When implementing these requirements, we have to cope with a time limit. Because of this, not all of the requirements are likely to be satisfied. A MoSCoW model prioritization will be given later in the document to describe which requirements take priority.

3 NON-FUNCTIONAL REQUIREMENTS

Apart from business and functional requirements, some non-functional requirements have also been defined.

3.1 HARDWARE/SOFTWARE REQUIREMENTS

Since DARTH VADER will be made using Visual Studio 2010 ultimate, licensing for this application is a requirement. A computer capable of meeting the system requirements of Visual Studio 2010 ultimate is also needed. Visual Studio and PAS need a Windows platform (with .NET framework 3.5 installed on it) to run on. The server PAS runs on, has some additional software requirements: ISS 6.0, BizTalk and SQL server 2008.

3.2 PERFORMANCE REQUIREMENTS

There are different goals with regard to test set size. If the entire manual regression test script is translated to automated GUI tests, more time will be needed than if just a subset is translated. The entire test suite's execution should be completed in less than a working day. Six hours of total execution time is set as the maximum time required that we will aim for. In case a subset of the entire suite is implemented, the maximum required time is less. The maximum required time is then proportional to the size of this subset compared to the entire suite.

Figure B.4: Requirements Definition Document - page 4

3.3 SUPPORTABILITY REQUIREMENTS

A design document will be made to improve supportability. Part of this design document, is a class diagram, displaying the different classes, their attributes and functions, and how the classes are interconnected. Sequence diagrams will also be provided. To improve maintainability, a report describing how every part of the code is meant to be used or changed, is also given. Different changes to the PAS application, and needed adaptations on DARTH VADER to cope with these changes, are described.

3.4 USER DOCUMENTATION REQUIREMENTS

The actual users of DARTH VADER will primarily be developers. The documentation we need from them (the functionality they want to be tested in the regression test) is listed in supportability requirements. Another group of users that is involved in this project, are the PAS users. The PAS users control what the content of the regression test script is. We always need to have an up to date version of the manual regression test script document available in order to implement a proper test suite.

4 MOSCOW MODEL

The MoSCoW model will be used to set priorities for the different requirements. The four different priorities are:

1. Must have (Mo)
2. Should have (S)
3. Could have (Co)
4. Would like to have (W)

Number one (must have) is the highest priority; requirements labeled as “must have”, are essential for the project’s success. Number four is the lowest; requirements labeled with “would like to have”, would be a nice addition to an application, but are not at all critical to the project’s success. The other priorities are moderately important, with “should have” being more important than “could have”.

Requirement	Must have	Should have	Could have	Would like to have
FR_GEN_1	**x**			
FR_GEN_2	**x**			
FR_GEN_3		**x**		
FR_IO_1	**x**			
FR_IO_2	**x**			
FR_IO_3	**x**			
FR_IO_4		**x**		
FR_IO_5	**x**			

Figure B.5: Requirements Definition Document - page 5

FR_IO_6	**x**			
FR_IO_7	**x**			
FR_CC_1		**x**		
FR_CC_2			**x**	
FR_TS_1				**x**
FR_TS_2	**x**			
FR_TE_1	**x**			
FR_TE_2			**x**	

Figure B.6: Requirements Definition Document - page 6