DSW ZORGVERZEKERAAR

Test Automation in Practice

Master's Thesis Report

Version 1.07

Kishenkumar Bhaggan BSc

Test Automation in Practice

THESIS REPORT

submitted in partial fullfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Kishenkumar Bhaggan BSc
born in Paramaribo, Suriname

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

DSW Zorgverzekeraar
Department of ICT
's-Gravenlandseweg 555, 3119XT
Schiedam, the Netherlands
http://www.dsw.nl

DEDICATION

I dedicate this master thesis report to
my father Dewbar Bhaggan and
my mother Ramkoemarie Bhaggan
who constantly stood by and supported me during
my study in The Netherlands

Test Automation in Practice

Author: Kishenkumar Bhaggan BSc
Student id: 1217380
Email: kish_bhagg@hotmail.com

ABSTRACT

As software systems evolve, manual software testing becomes more and more difficult, time-consuming, and costly. Therefore tests should be automated to reduce testing efforts. But the introduction of automated testing also introduces new costs, such as maintenance costs for the automated test suites. In order to make automated testing effective and efficient, so that it can reduce overall costs, new techniques and methodologies are required. During the thesis project, data-driven testing and unit testing was applied. The costs before test automation and costs after test automation were compared. This comparison revealed that test automation reduces test effort and overall costs of software development and leads to the production of more reliable and maintainable software.

Preface

During my last year at the secondary school, Mr. Dr. J.C. de Miranda Lyceum I in Suriname, I worked as a part-time network and system administrator for the small internet café of my brother. There I got programming experience, after which I came to like programming. I chose to study Technical Informatics at Delft University of Technology. As my first three years of my BSc study passed, I started to look for a job, doing so I came in touch with DSW Zorgverzekeraar, where I work as a web developer. During my work I also studied the Master Computer Science at Delft University of Technology. During the fourth year I realized that DSW Zorgverzekeraar does not perform automated tests, I decided that my thesis subject should be test automation. This report is about my work during the thesis project. I am very grateful for the support and feedback I received during my study from friends, relatives, and instructors.

In particular, I want to thank my daily supervisor at Delft University of Technology, ir. H.J.A.M. Geers, for his very valuable comments on former versions of this report and his feedback and support during the last two years, I also want to thank my chair supervisor, Prof. Dr. Arie van Deursen, for his helpful comments and tips for my graduation project. In addition I would like to thank my supervisor at DSW Zorgverzekeraar, Job Scheffers, for his helpful comments on former versions of this report and tips during my graduation project. Furthermore I would like to thank my manager at DSW Zorgverzekeraar, Jeroen de Haan, my team leader, Samira el Haddioui, and my project leader, Marielle Terlien, for their support for the start of my graduation project and their support during the graduation project. Finally I want to thank all my friends, relatives, and colleagues who stood by me and helped me endure the most tough times during my MSc study period.

Enjoy reading this thesis!

Kishenkumar Bhaggan BSc.
September 2009

Delft, The Netherlands

Contents

List of Figures

List of Tables

1 Introduction

As software systems evolve, software testing becomes a difficult and time-consuming task when performed manually. New techniques must therefore be applied to improve the test process and reduce test effort of testers. Test automation is one of the techniques which can be applied. Test automation is an improvement of manual testing.

1.1 Problem Statement

Test automation can reduce project duration and test effort [1]. It can improve software quality, i.e. with a good automated test suite code can be refactored at any time to improve maintainability and software reliability. However if a test process is chaotic, test automation will only increase costs. Therefore an analysis of the current test process is required before introducing test automation.

Test automation is an enhancement to manual testing, but it is not feasible to automate all tests of a software project. Limiting factors are mainly GUI tools that are incompatible with custom GUI controls. Some tests are impossible or very difficult to perform automatically, for example verifying whether an e-mail sent by a custom application is displayed correctly in various mail clients. Another limitation of test automation is the cost of test automation and maintenance costs of the automated test suite. This document will therefore answer the question:

> ***How much time or test effort can be saved with test automation and what are the costs of test automation?***

1.2 Scope and Methodologies

During the master thesis project test automation is introduced at DSW Zorgverzekeraar. The methodologies and techniques used to introduce test automation are Test-Driven Development and Data-Driven Testing. The costs and benefits of this test automation have been described and estimated. DSW

Zorgverzekeraar can decide whether or not to automate tests based on these estimations.

Because business logic in the web application at DSW Zorgverzekeraar is implemented in the UI layer, it is not possible to detect bugs in the implementation of the business logic by unit tests. Therefore automated UI testing was introduced. But automated UI testing has some drawbacks, it requires functionality to be built first. And automated UI testing is difficult to maintain. In order to perform automated tests beneficially, code was refactored (business logic was moved from the UI layer to another layer) in order to perform elaborate unit testing. The costs and benefits are then estimated.

1.3 Limitations of Software Testing

Any nontrivial software will have bugs. It is clear that passing a single test is not enough to show that a system is bug-free. There are several reasons why it is impossible to proof that a system is bug-free. Consider an example program which draws a quadrangle and accepts 4 lines as input where point coordinates are limited between 1 and 10. In this case 10^4 (10 x 10 ways to draw a point and thus $(10 \times 10)^2$) ways to draw a line). There are four lines thus $(10^4)^4 = 10^{16}$ possible inputs of four lines are possible. Assuming a tester tests 1000 input combinations a second and works 24 hours a day and 365 days a year. The tester could test all possible input combinations in $10^{16}/10^3 = 10^{13}$ seconds. At 3.14×10^7 seconds a year, it would take $10^{13}/(3.154 \times 10^7) = 3.171 \times 10^5$ years are needed to perform all possible tests. This makes it clear that the output of the program for all possible input combinations cannot be verified within reasonable time.

Another limitation is that code can hide faults. Suppose an expression is noted as x + x, where it should be x * x, by using the input of 2, the correct result is produced and the tester could assume that the system contains no faults. Consider the following code fragment [2]:

```
int scale(int j)
{
	j = j – 1; //should be j = j + 1
	j = j / 30000;
	return j;
}
```

From the 65,536 possible values for j only 40 values will produce an incorrect result: -30000, -29999, -27000, -26999, -24000, -23999, -21000, -20999, -18000, -17999, -15000, -14999, -12000, -11999, -9000, -8999, -6000, -5999, -3000,-2999, 2999, 3000, 5999, 6000, 8999, 9000, 11999, 12000, 14999, 15000, 17999, 18000, 20999, 21000, 23999, 24000, 26999, 27000, 29999, 30000. None of these values is a boundary of j. If all possible values except these 40 were used, then even a nearly exhaustive test would not reveal the bug.

Other reasons are that testing use requirements as a point of reference. When requirements are incorrect or incomplete, incorrect tests are produced. Omissions are also not revealed if implementation-based testing is used, as missing code cannot be tested. Bugs in tests can also result in incorrect test results.

1.4 Chapter Overview

Chapter 2 gives a brief description of the software development process at DSW Zorgverzekeraar. Chapter 3 describes the analysis of the current test process, it presents the key areas that should be improved and how to improve the key areas. Chapter 4 describes the introduction of automated UI testing at DSW Zorgverzekeraar and presents the techniques used for introducing automated UI testing. Chapter 5 describes the Test-Driven development methodology and the differences between automated UI testing and unit testing. Chapter 6 presents the case studies performed using UI testing and the combination of UI testing and unit testing and their results. Chapter 7 describes a cost-benefit analysis of automated testing. Finally, Chapter 8 presents the conclusion of this document.

2 DSW Zorgverzekeraar

Before to start analyzing and suggesting improvements it is important to have a look at the organization's software development and software testing processes. In this chapter the software development process and software testing process of DSW Zorgverzekeraar is described.

Section 2.1 describes the current software development process, section 2.2 the current software testing process. In Chapter 3 the maturity and costs of the current testing process will be analyzed.

2.1 The Software Development Process

The software development department of DSW Zorgverzekeraar consists of approximately 60 men/women. Software development is performed iteratively using RUP. For most of the .NET systems change requests or bug reports are submitted through an online portal called Teamplain, for some other systems Lotus Notes is used (Aanpassingsverzoek database). When a change request or bug report is submitted to the system, the project leader of the corresponding project assigns priorities to the change request or bug report. Developers start implementing the features with high priorities. When the implementation period is elapsed, the system is deployed in a test environment. There the system is tested by the end-users of the system. After the high priority bugs, revealed during the testing period, have been fixed, the system is deployed in an acceptance testing environment. In this environment testing is performed only to detect configuration errors (conforming RUP), but in practice, the whole regression test set used for the test environment is performed again. Hereafter the system is put in the production environment. Only high priority bugs found in the acceptance and production environment will be fixed. An overview of the process is given in Figure 1.

Only limited testing is performed in the production environment to detect configuration errors. Web services developed during this period are only tested by developers manually, some limited automated testing is performed on the web services.

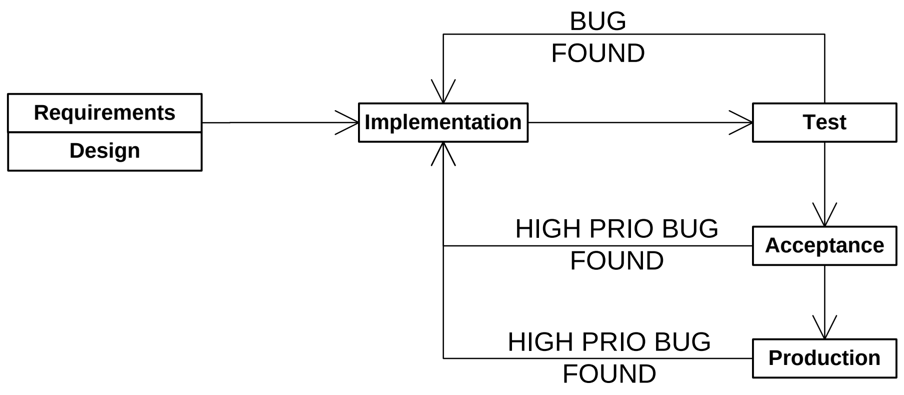


Figure 1 - Development process at DSW Zorgverzekeraar

2.2 The Current Software Testing Process

Software testers at DSW Zorgverzekeraar are end-users of the systems, they are not full-time available to test systems; most of the testers are also involved in other organizational activities. Software testers of systems are not dedicated to one system only. For example a tester could be testing the policy administration system and be testing the web application too. During the testing period no or limited test planning (i.e. distribution of tasks, when, what and how testing should be performed) is performed, which may result in outdated test plans. Some of the important test cases are documented, but many are outdated. There are many reasons why test case documentation are not maintained. Testers think that the documentation is not read by other testers or that testers are so involved with other activities that there is no time to maintain documentation.

The test coordinator is responsible for the testing activities in a particular project, he/she will distribute the testing activities among the testers. The test coordinator prioritizes bugs, and gives input to the project leader whether the high priority bugs are resolved.

During regression testing, when new functionality is implemented or when existing functionality is modified, only modifications are tested, not the complete system. It is however necessary to test the entire system, because a change in some functionality can ‘break’ another functionality. Consider the following example. See Figure 2, suppose the Aanmelding object is used by the two use cases: AanmeldFlow and VerzekerdeToevoegen. Suppose that some new requirement requires a change in the AanmeldFlow and there are no modifications in the requirements of the VerzekerdeToevoegen functionality. When the developer begins with the development, he might modify the Aanmelding object. This can result in an (unintentional) modification of the VerzekerdeToevoegen functionality. Because testers perform black-box testing, they do not have knowledge about this structure and they are are not aware that the VerzekerdeToevoegen functionality might have changed too. Because only modifications are tested and since there was no modification in the requirements of the VerzekerdeToevoegen functionality, the (unintentional) modification of the functionality would not be detected and a possible bug that might have been introduced in that functionality, which will not be detected during the test period and might be detected by the end-users of the system when the system is taken into production.

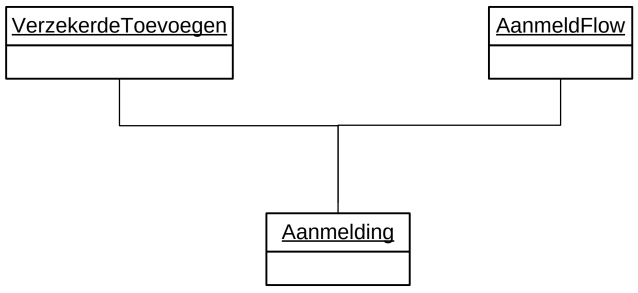


Figure 2 - Example associations

Due to limited time and resources it is not feasible to perform a regression testing at full scale. Test automation offers the possibility to perform a regression testing at full scale, i.e. testing the complete system during regression testing and therefore reduce bugs found by end-users.

Another problem with manual testing is that when use cases, test specifications, or design documents are not available, the functionality can not be tested thoroughly, because testers will not be able to determine how to test the functionalities, and they would not be aware of the scenario's to be tested. This is because the test scenario's are only in the mind of the tester who was involved during the development of the functionality (when the functionality/use case was introduced in the system). Another reason is that manual tests might be different every time they are executed. If automated tests were created during the development of a functionality and maintained, then the functionality would be tested thoroughly and consistently even if use cases, test specifications, or design documents are not available.

2.3 Concluding Remarks

Software is developed iteratively at DSW Zorgverzekeraar, a good regression test suite is therefore required. As only modifications are tested and other parts of the system(which could also be affected by the modifications) are not, a risk is introduced.

Test automation may therefore reduce these kinds of risks and increase system reliability.

3 Test Process Improvements at DSW Zorgverzekeraar

As mentioned earlier, test automation can increase costs if current test process is not structured, as is the case at DSW Zorgverzekeraar. Certain improvements are therefore required in the current test process. To improve the test process, the Test Process Improvement (TPI) model will be used. This chapter presents an analysis of the current test process at DSW Zorgverzekeraar and presents the test process improvement suggestions. Section 3.1 outlines the test process improvement model. Section 3.2 describes the analysis of the current test process. Section 3.3 describes the test process improvement suggestions.

3.1 Test Process Improvement Model

The test process improvement (TPI) model offers a reference framework to determine the strengths and weaknesses of an organization’s test process [3]. It also provides support for the formulation of specific, feasible proposals for the improvement of the test process in terms of lead time, expenses, and quality.

The basis of the TPI model is TMap, but it can also be applied in organizations that use methodologies other than TMap [3].

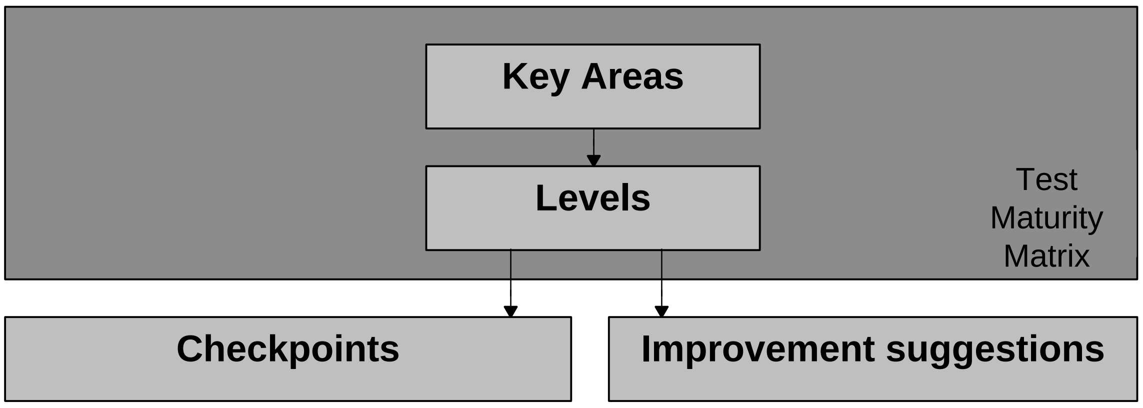


Figure 3 - The TPI Model [3]

Figure 3 shows an overview of the TPI model. All these aspects will be described in the following subsections.

3.1.1 Key Areas

For each test process that requires specific attention, key areas can be identified. The key areas form the basis for improving and structuring the test process. Twenty key areas are identified by the TPI model. Separate key areas are included for low-level tests, such as unit and integration testing, to give enough attention to low-level tests in improving more “mature” test processes. The most relevant key areas are described in Table 1.

Key area	Description
Lifecycle model	Within the test process, a number of phases can be defined, such as planning, preparation, specification, execution, and completion. In each phase, several activities are performed. For each activity, the following aspects should be defined: purpose, input, process, output, dependencies, applicable techniques and tools, required facilities, documentation, etc. Using a lifecycle model implies an improved predictability and manageability of the test process because the different activities can be planned and monitored in mutual exclusion.
Estimating and planning	Test planning and estimating indicate which activities have to be carried out when, and what the necessary resources (people) are. Good estimation and planning are very important because this is the basis of, for example, allocating resources for a certain timeframe.
Test-specification techniques	A test-specification technique is a standardized way of deriving test cases from source information. Applying these techniques gives insight into the quality and depth of the tests, and increases the reusability of the test.
Metrics	Metrics are quantified observations of the characteristics of a product or process. For the test process, metrics of the progress of the process and the quality of the tested system are very important. They are used to control the test process, to substantiate the test advices, and to make it possible to compare systems or processes. Why does one system have far fewer failures in operation than the other system? Or why is one test process faster and more thorough than the other? Specifically for improving the test process, metrics are important for evaluating consequences of certain improvement actions by comparing data before and after performing the action.
Test automation	Automation within the test process can take place in many ways, and has in general one or more of the following aims: • Fewer hours needed

	• Shorter lead time • More test depth • Increased test flexibility • More and/or faster insight in test process status • Better motivation of the testers
Commitment and motivation	The commitment and motivation of the people involved in testing are important conditions of a smooth-running test process. The people involved include not only the testers but also, for example, the project management and line management. The latter are important mainly in sense of creating good conditions. The test process thus receives enough time, money, and resources (quantitatively and qualitatively) to perform a good test, in which cooperation and good communication with the rest of the project results in a total process with optimum efficiency.
Test functions and training	For each test process in the organization, a certain methodology or working method is used, comprising activities, procedures, regulations, techniques, etc. When these methodologies are different each time, or when the methodology is so generic that many parts have to be drawn up every time, it has a negative effect on the test process efficiency. The aim is that the organization uses a methodology that is sufficiently generic to be applicable in every situation, but that contains enough detail so that it is not necessary to rethink the same items each time.
Reporting	Testing is not so much defect detection as giving insight into the quality level of the product. Reporting should be aimed at giving well-founded advice to the customer concerning the product and even the system development process.
Defect management	Although managing defects is, a project matter and not specifically for the testers, the testers are mainly involved in it. Good management should be able to track lifecycle of a defect, and to support the analysis of quality trends in the detected defects. Such analysis is used, for example, to give well-founded quality advice.
Testware management	Products of testing should be maintainable and reusable, so they must be managed. As well as the products of the testing, such as test plans, specifications, databases and files, it is important that the products of previous processes as design and code are managed well, because the test process can be disrupted if the wrong program versions, etc. are delivered. If testers make demands on version management of these products, a positive influence is exerted and the testability of the product is increased.
Test process management	For managing each process and activity, four steps are essential: plan, do, check, and act. Process management is of vital importance for the implementation of an optimal test in an often turbulent test process.
Lower-level testing	Low-level tests are carried out almost exclusively by the

	developers. Well-known low-level tests are the unit test and integration test. Just as in evaluation, the tests find defects in an earlier stage of the system development path than the high-level tests. Low-level testing is efficient because it requires little communication, and often the finder is both the error producer and the one who solves the defect.

Table 1 - Key areas of TPI [3]

3.1.2 Levels in the TPI Model

Key areas are provided with a number of maturity levels in the TPI model [3]. The number of levels may differ for each key area. By using the levels the current situation of the test process and the targets for improvement can be determined. In order to be classified at a level, the appropriate checkpoints/requirements must be met. The checkpoints of a level includes the checkpoints of the lower levels. For example if a key area is classified at level B, it meets the checkpoints of level A and the checkpoints of level B. The key areas, which are relevant for this thesis together with the checkpoints/requirements for each level are provided in Table 2.

Key area	Level A	B	C	D
Lifecycle model	Planning, specification, execution	Planning, preparation, specification, execution and completion		
Estimating and planning	Substantiated estimating and planning	Statistically substantiated estimating and planning		
Test-specification techniques	Informal techniques	Formal techniques		
Metrics	Project metrics (product)	Project metrics (process)	System metrics	Organization metrics (≥ 1 system)
Test automation	Use of tools	Managed test automation	Optimal test automation	
Commitment and motivation	Assignment of budget and time	Testing integrated in project organization	Test engineering	
Test functions	Test manager and testers	(Formal) methodical, technical and functional support,	Formal internal quality assurance	

		management		
Reporting	Defects	Progress (status of tests and products), activities (cost and time, milestones), defects with priorities	Risks and recommendations, substantiated with metrics	Recommendations have a software process improvement character
Defect management	Internal defect management	Extensive defect management with flexible reporting facilities	Project defect management	
Testware management	Internal testware management	External management of test basis and test object	Reusable testware	Traceability system requirements to test cases
Test process management	Planning and execution	Planning, execution, monitoring, and adjusting	Monitoring and adjusting in organization	
Low-level testing	Low-level test lifecycle (planning, specification, and execution)	White-box techniques	Low-level test strategy	

Table 2 - Levels in the TPI model [3]

3.1.3 Test Maturity Matrix

Improvements of the test process can be derived from the levels determined for each key area. Between the key areas and levels several dependencies exist. For example level A of key area 'metrics' states that metrics are kept per project. This means that for each unit of time, hours accounting data and defect data have to be recorded. Monitoring these data means that the test process for key areas 'reporting' and 'defect management' are at least at level B and level A, respectively. This implies that level A of 'metrics' is dependent on level B of 'reporting' and level A of 'defect management'.

All key areas are mutually related in the test maturity matrix, which is given in Table 3, on basis of these dependencies. Key areas are indicated vertically and the test maturity scales are indicated horizontally. This results in a matrix with thirteen scales of test maturity.

The open boxes between the levels indicate that a higher degree of maturity for the given key area is related to the maturity level of other key areas. A test process at level B for some key area must meet all requirements of level B, otherwise it is classified at level A.

Key areas	Scale	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Lifecycle model			A			B									
Estimating and planning					A							B			
Test-specification techniques			A		B										
Metrics							A			B			C		D
Test automation					A				B			C			
Commitment and motivation			A				B						C		
Test functions and training					A			B				C			
Reporting			A			B		C					D		
Defect management			A				B		C						
Testware management				A			B				C				D
Test process management			A		B								C		
Low-level testing					A			B		C					
					Controlled					Efficient				Optimizing	

Table 3 - Test maturity matrix [3]

If a test process for a certain key area does not meet the requirements of level A, then the process is set to the zero scale. The various scales of test maturity can be divided into three categories [3]:

- **Controlled**. The test process is carried out in phases according to a test strategy defined in advance. Informal test-specification techniques are used for testing, and defects are recorded and reported. The testware and test environment are controlled.
- **Efficient.** The test process is efficient. The efficiency is achieved by test process automation, integration between the various test levels and with the other parties within the system development, and consolidation of the working method of the test process in the organization.
- **Optimizing.** Changing circumstances as new architecture and development techniques require an adjustment in the test process [3]. The aim is at continuous improvement of the generic test process.

After the analysis of a test process, the test maturity matrix is constructed. The matrix provides all stakeholders with a clear view of the level of the key areas of the test process. The matrix can then be used for the improvement of the test process. The key areas with a low test maturity must be improved first. Experience has shown that it is inefficient to have key areas with high scale of test maturity, while other key areas have a low or average scale).

Consider the following example (see Table 4 and Table 5). The test process does not meet the requirements of level A for key area ‘test strategy’. It does meet the requirements of level A for key area ‘lifecycle model’ and ‘moment of involvement’. As a reaction to this result it is decided to coordinate the test strategy with high-level tests (level B) and a full lifecycle model (level B). Attention to the moment of involvement is currently not considered relevant. The result is given in table Table 5.

Key areas	Scale	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Test strategy			A					B				C		D	
Lifecycle model			A			B									
Moment of involvement				A				B				C		D	

Table 4 - Test maturity matrix for a test process [3]

Key areas	Scale	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Test strategy			A	-				B				C		D	
Lifecycle model			A	-		B									
Moment of involvement				A				B				C		D	

Table 5 - Test maturity matrix showing improvements [3]

3.1.4 Checkpoints

Checkpoints are measuring instruments to objectively determine the level of a key area. Each level contains a set of checkpoints. These checkpoints must be reached to be able to classify a test process to that level. To be classified at a higher level, the checkpoint of that level and all lower levels must be met [3]. For example if a test process is classified at level B, it must reach the checkpoint of level A and the checkpoints of level B.

3.1.5 Improvement Suggestions

The checkpoints can be used for process improvement, but improvements suggestions per level can also be used to improve the test process. These suggestions are not mandatory, but are suggestions which are meant as hints and tips [3]. Each level has a number of these improvements in order to achieve the level concerned [3].

TMap also offers a large set of improvements collections with the various stages, techniques, checklists, and procedures.

3.2 Analysis of the Current Test Process

In order to improve the current test process at DSW Zorgverzekeraar, the current test process has to be analyzed. This section presents the analysis of the current test process. First the test process maturity will be analyzed, then the statistical metrics. These metrics are:

- Number of production bugs
- Time to analyze production bugs (i.e. determine which line of code triggers the fault)
- Time to solve production bugs
- Number of bugs found during test period
- Time to analyze bugs
- Time to solve bugs
- Test time per functionality

Production bugs are bugs not found during the test period, but in the production environment by end-users.

Section 3.2.1 describes the analysis of the test maturity. Section 3.2.2 describes the analysis of the costs.

3.2.1 Test Process Maturity

In order to analyze the test process, a questionnaire was created. The questions of this questionnaire was answered by the test coordinator. With the input of the test coordinator, the test maturity matrix was created for DSW Zorgverzekeraar. The questionnaire can be found in the appendix. This questionnaire is created to determine the level of a key area. Per key area that is relevant for test automation a question has been created. The questions are multiple choice. The key areas that are relevant are:

- Test Automation
- Test-Specification techniques
- Test functions and training
- Testware management

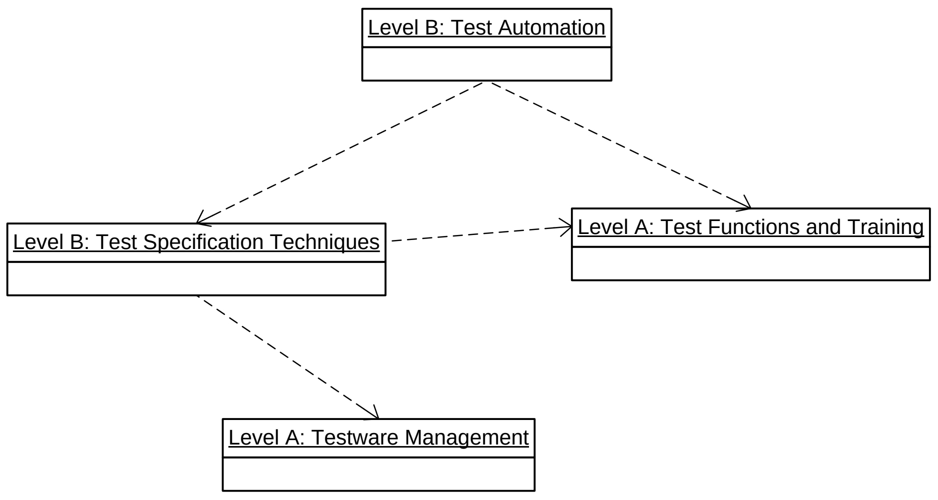

Level B: Test Automation
Level B: Test Specification Techniques
Level A: Test Functions and Training
Level A: Testware Management

Figure 4 - Dependency graph Test Automation

These key areas are relevant to improve the test process by test automation. According to the dependency overview of Sogeti level B of key area test automation is dependent on level B of test-specification techniques and level A of test functions and training [4]. Level B of the key area test-specification techniques is dependent on level A of test functions and training and level A of testware management. These are all dependencies of the key areas for test automation. Figure 4 gives an overview of this dependencies.

In order to perform test automation beneficially, test planning must also be performed. Test planning is important, because it will provide testers and other stakeholders with an overview of automated and manual tests and test cases. It will also make clear to testers which tests they will have to perform manually and therefore level A of test planning is required. Level A of key area estimating and planning is dependent on level A of key area lifecycle model, and level A of key area lifecycle model is dependent on level A of key area commitment and motivation. An overview of the dependencies is given in Figure 5.

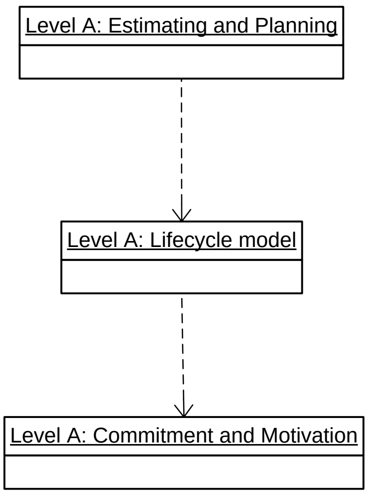

Level A: Estimating and Planning
Level A: Lifecycle model
Level A: Commitment and Motivation

Figure 5 - Dependency graph Test Planning

Another important aspect of introducing test automation is the reporting of progress of tests and time spent during an activity (metrics), in order to ensure that the costs of test automation are low and that not much test effort is required. This requires level B of reporting. Level B of reporting is dependent of level A of lifecycle model, level A of defect management and level B of test process management. To suggest improvement analysis will only be performed on these key areas and improvement suggestions will be provided for only these key areas. An overview of this dependency is given in Figure 6.

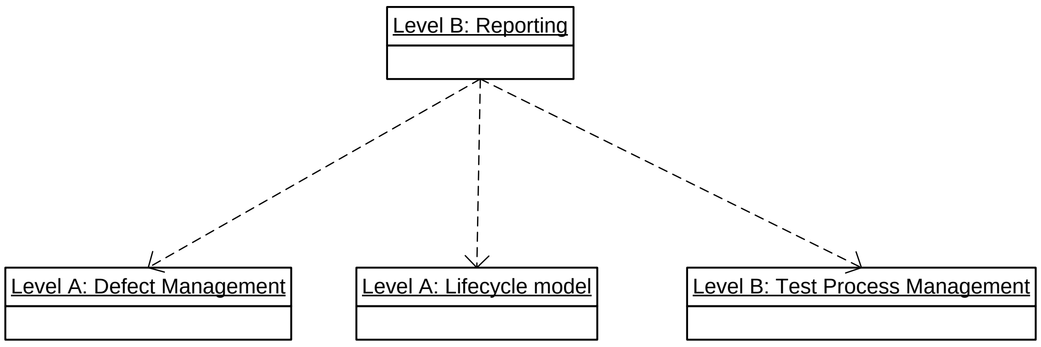


Figure 6 - Dependency graph Reporting

During the literature study, the test coordinator of the web project has been replaced. The new test coordinator could therefore not answer the questions for the past test process. The new test coordinator had a more structured approach, therefore the questions were answered for future testing processes. With his answers, the test maturity matrix was constructed (Table 6).

Key areas Scale	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Lifecycle model		A			B									
Estimating and planning				A							B			
Test-specification techniques		A		B										
Metrics						A			B			C		D
Test automation				A				B			C			
Commitment and motivation		A				B						C		
Test functions and training				A			B				C			
Reporting		A			B		C					D		
Defect management		A				B		C						
Testware management			A			B				C				D
Test process management		A		B								C		
Low-level testing				A			B		C					
Controlled								Efficient				Optimizing		

Table 6 - Maturity analysis

3.2.2 Cost Analysis

In order to prove that test automation is efficient and optimizes the current test process, some metrics have to be estimated. These are:

- The number of production bugs.
- Time to analyze production bugs (i.e. determine which line of code triggers the fault).
- Time to solve the production bugs.
- The number of bugs found by testers during test period.
- The time to analyze the bugs.
- The time to solve the bugs.
- The number of deployments per release

The metrics mentioned above are only estimated for the web application project of DSW Zorgverzekeraar. The web application project consists of approximately 90 use cases. That means that 90 use cases have to be tested every time a new version of the system is deployed in the testing environment.

The number of production bugs

The number of bugs detected in the production environment is estimated by the project coordinator to a number of five bugs per release.

Time to analyze production bugs

Time to analyze bugs is dependent of the kind of bug detected. Sometimes it is because a value is hard coded and only that value has to be changed, this is a matter of minutes. But sometimes it is more complicated, it can take a day to find the line of code that triggered the failure. The average time to analyze a production bug is therefore set to two hours per bug.

Time to solve production bugs

Once the line(s) of code that triggered the failure is found, it can just be a matter of minutes to solve the bug. But sometimes developers must arrange a meeting in order to discuss the bug and the solution. Therefore the average time to solve a production bug is set to two hours. Once the bug is solved, the application must be tested again (regression) through all the environments (test and acceptance). It can therefore take one to two days for a bug fix to be in production.

The number of bugs found by testers during test period

The number of bugs found by testers should be reduced by automated tests. The automated tests are run by developers before the application is ready to be tested by testers. They will therefore remove bugs early in the development cycle.

The number of bugs found during the test period of the previous release of the test site and their progress is given in Figure 7. The red line indicates the active bugs during the test period, the green line indicates the bugs that were resolved (solved by developer and not yet tested by testers). The yellow line indicates bugs that were reactivated, for example a bug was fixed in version 10.0, and are introduced again in version 11.0. This report was generated by the TFS TeamPlain, in which bugs are reported. Production bugs are not reported in TeamPlain, but directly to developers.

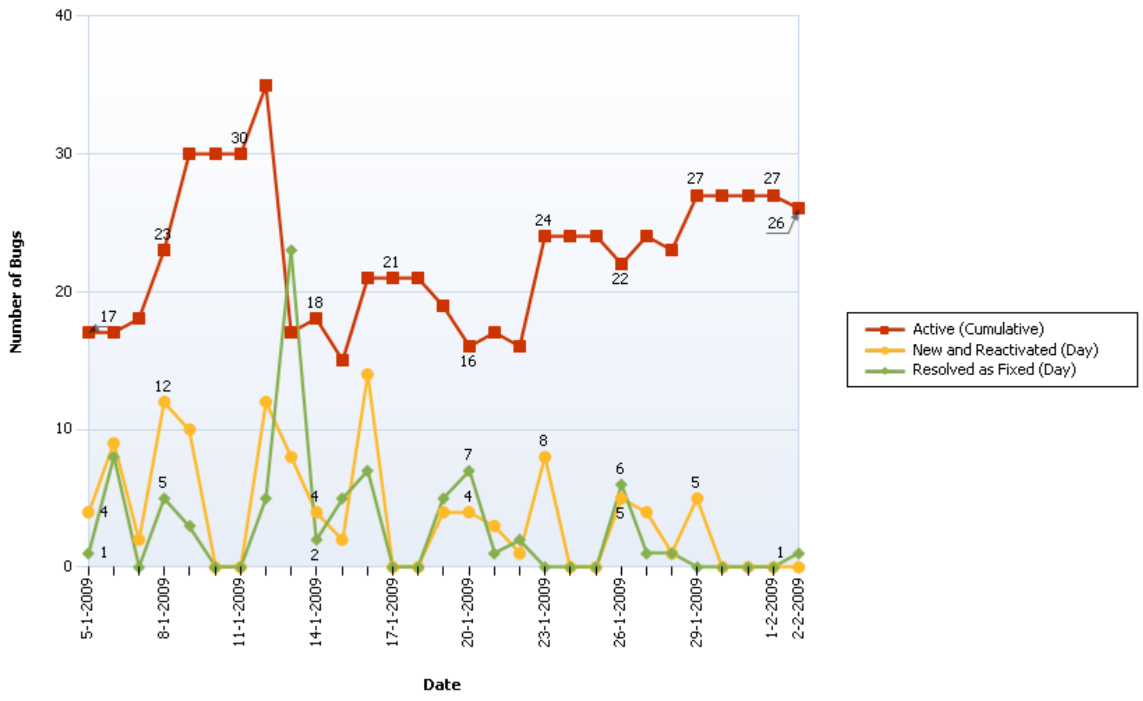


Figure 7 - The number of bugs found during the test period

Time to analyze bugs

During the test period many bugs are found. The time to analyze them is set to two hours per bug on average.

Time to solve bugs

Sometimes developers must also arrange meetings in order to discuss the bugs found during the test period and the solutions. Therefore the average time to solve a bug found during the test period is also two hours.

The number of deployments per release

The number of deployments differs per release. The number of deployments per release is dependent on the number of functionalities to be modified and the number of new functionalities. The average of the past five releases is taken. The average number of deployments per release is set to 26.

3.3 Test Process Improvement Suggestions

To provide suggestions for the improvement of the test process, the current test process was analyzed. The results of this analysis is summarized in Table 6. This table will be used to generate the improvement suggestions. In order to have a controlled test automation, the test automation key area has to be classified at level A. As can be seen from the table, automated and low-level testing are the only key areas which are classified at level 0.

At this moment no tools are used during the test period. To improve the key area Test Automation, testing tools have to be introduced. This improvement will be implemented during the thesis project.

Low-level testing is not performed at all for the web application project of DSW Zorgverzekeraar. Improve this, the test-driven development methodology needs to be followed. Again this improvement will be implemented during the thesis project.

3.4 Concluding Remarks

In order to introduce test automation, a structured test process has to be in place. The current test process has therefore been analyzed. After the analysis it was concluded that most of the key areas of the test process, that are necessary for test automation, are mature. The key areas test automation and low-level testing however, are not matured. These must and will be improved. A couple of metrics have been analyzed in order to measure the improvements with test automation.

4 Introduction of UI Testing

The business logic is implemented in the presentation (UI) layer in almost all places of the web development project at DSW Zorgverzekeraar. This will make elaborate user-interface (or UI or web) testing necessary. A tool was created to enable data-driven UI testing and an experiment has been performed to estimate the reduction of maintenance costs by the use of this tool. This chapter describes the techniques to perform UI testing, data-driven UI testing, the data-driven tool, which was created during the thesis, and finally the experiment and its results.

Section 4.1 presents a record and playback technique. Section 4.2 describes data-driven techniques to perform UI testing, section 4.3 describes the tools that will be used. In section 4.4 the tool, created during the thesis, to perform data-driven testing is introduced. Section 4.5 describes the differences between the data-driven and record and playback techniques. Section 4.6 describes an experiment performed to show that data-driven techniques reduce maintainability costs. Section 4.7 evaluates the results presented by the experiment performed.

4.1 Record and Playback

There are several tools that provide record and playback functionalities. These tools are used to record user inputs and store them in a test script. After the information is stored, it can be used to play the recorded script at any time. These tools can be very effective for performing regression testing. During the first test, the tool can be used to record the tests a user executes. Bugs found, are reported to developers, who will produce a new build of the system in the test environment. Now the tester only has to load the test script(s) he/she has recorded and run it. This saves time for the tester and increases the tester’s productivity.

Unfortunately small changes in the user interface can cause test scripts to fail. The scripts need to be recorded again. Recorded scripts contain hard-coded data. A small change in the user-interface may lead to many changes in the hard-coded data. Research [5] has shown that a simple change in the user-interface may cause 74% of the test scripts to become unusable. This means that a change in the user-interface may lead to additional hours of work for the maintenance of the generated script or for re-recording of the test. Therefore new/additional techniques are needed to organize test scripts in such a way that the maintenance costs of the test scripts can be reduced. Data-Driven techniques offer better organization of test scripts and lower maintenance costs of the test scripts. These techniques will be explained in the next section.

4.2 Data-Driven Techniques

Data-driven testing provides higher maintainability of automated user-interface tests. Two data-driven techniques are explained in [6] and will be outlined in this section:

1. The “Functional-Decomposition” Method
2. The “Keyword-Driven or Testplan-Driven” Method

These methods are outlined in the following subsections.

4.2.1 The Functional-Decomposition Method

The main idea of the functional-decomposition method is to reduce all test cases to most fundamental tasks. Generic scripts can be written to handle various actions. These scripts can be divided in:

- **Driver**. The driver initializes the test as necessary, then the test case scripts are run in the preferred order.
- **Test case.** The test case scripts perform the application logic using the business function scripts.
- **Business function.** These scripts perform the specific business logic within the application.
- **Subroutines.** These are used by business function scripts. If a subroutine is used by more than one business function script, a subroutine script is created in order to reuse the subroutine.
- **User-defined functions.** These can be generic, application specific, and screen access functions.

In an example situation the business function and subroutines scripts use the user-defined scripts for screen access, the test case script uses the business function and subroutines scripts, and the driver calls the test case script. If the user-interface changes, the data of the user-defined script change. So only four scripts are necessary to perform any number of tests. Whereas with record and playback tools a script must be created for each test. This method requires a text editor to create and update the data files.

The advantages of this method are:

- Redundancy and effort duplication reduction in automated test script creation by modular design and the use of files for input and data verification.
- Parallel development of scripts with software development. If the functionality changes, only the business function scripts have to be modified.
- The input/output and expected results are saved as easily maintainable records.

Disadvantages of this method are:

- Proficiency is required in the scripting language.
- Multiple data files are required for each test case. These data files must be kept in separate directories for each test case.
- Not only the detailed test plan should be maintained, but also the data in the various data files.

4.2.2 The Keyword-Driven or Testplan-Driven Method

This method eliminates most of the disadvantages of the previous method. Now the entire process in this method is data-driven. As the name states, this method is keyword based. Consider the following example:

Keyword	Field/Screen Name	Input/Verification Data	Command
Start_Test:	Screen	Main Menu	Verify Start Page
Enter:	Selection	2	Select 'Change Assurance'
Action:	Press_Key	F5	Access 'Change Assurance Screen'
Verify:	Screen	\`Change Assurance\`	Verify Change Assurance Screen
Enter:	New Assurance	6067	Enter new assurance
Action:	Press_Key	Enter	Process assurance change
Verify:	Screen	Confirmation Screen	Verify the confirmation screen
Verify:	New Assurance	6067	Verify the new assurance just entered
Action:	Click	Submit	Process the assurance change
Action:	Press_Key	F9	Return to main menu
Verify:	Screen	Main menu	Verify return to main menu

Table 7 - Example test case

Each of the keywords in the Keyword column causes a utility script to process the remaining column as input parameters to perform specific functions. The test engineer must document and develop the test case, so it is useful to create the automated test case early. If this test case will be used to create other test cases for the change assurance screen, then the data in red should be changed.

The execution is as follows. The above table is saved in Microsoft Excel®, a comma separated file, or an XML file. The file is then read by a Controller script of the application. When a keyword is encountered, a list is created using the values of the remaining columns. This continues until the data found in the field/screen name is null or empty. The controller script then calls an utility script associated with the keyword and passes the list as parameter. The utility script processes the list and returns to the Controller script which then continues to the next keyword till the end of the file.

This method inherits the advantages of the Functional Decomposition Method and adds the following advantages:

- The test case can be written in spreadsheet format. The tester only has to maintain the spreadsheet.
- The test plan does not necessarily have to be created in Excel. It can be done in any application which produces comma or tab separated files.

- If someone experienced in the scripting language of the test tool creates the utility script, then the tester can use the automated test tool immediately via the spreadsheet method, without learning the scripting language. The testers must only have knowledge of the keywords and the data format required, to use in the test plan. Testers can therefore work more productive with the test tool. It allows more extensive training of the test tool at a more convenient time.
- If the detailed test plan already exists, it will not be difficult to translate it to the spreadsheet format.
- After some utility scripts are written, they can be reused to test other applications. This will allow the organization to set up automated testing for most applications in just a few days, rather than weeks.

The disadvantages of this method are:

- When developing customized functions, proficiency is required in the scripting language. This holds for any method used including record/playback.
- If the application requires more than just a few customized utilities, testers will be required to learn a number of keywords and their formats.

4.3 Tools

The use of tools is necessary to perform efficient test automation. Two tools wil be used within the organization. These are:

- Microsoft® Visual Studio® Team Edition for Software Testers
- Selenium

As Microsoft® Visual Studio® is only available to developers and not to testers, testers will use Selenium to automate tests. Developers will create the automated tests which will be run in the .NET enviroment (Microsoft® Visual Studio®). In meantime, while developers create automated tests (not many tests are automated), testers can use Selenium to minimize their test effort.

The tools mention will be described in the following subsection. Section 4.3.1 describes the Microsoft® Visual Studio® Team Edition for Software Testers. Section 4.3.2 describes Selenium.

4.3.1 Microsoft® Visual Studio® Team Edition for Software Testers

Microsoft® Visual Studio® Team Edition for Software testers enables testers to perform web, load and unit tests which is fully integrated in the Visual Studio Environment. With this version of Visual Studio a large number of tests can be managed. It enables web testing for Web Services and ASP .NET applications. Automated record and playback options can be used to create and execute the tests. Web tests can be data bound to practice data-driven testing. Custom validation rules can be added through a simplified web test editor. A more important functionality of this web testing tool is that code can be generated easily in .NET languages(such as Visual C# or Visual Basic) from the web test, which can make maintenance even easier. This tool also enables performance testing. The web testing tool of Visual Studio 2005 however does not support AJAX, in order to support this functionality the tester can extend the tool. This feature is added in Visual Studio 2008, so web tests in Visual Studio 2008 support AJAX. The web test recorder of Visual Studio only records server calls which are created with or without AJAX.

To demonstrate how web applications are tested, a webtest was created. The following steps are required for creating a webtest (see Figure 8):

- Add a new webtest, this opens Internet Explorer and records every step of the user/developer.
- Then the webtest is saved.

After the webtests are created, some validation rules can be added. See Figure 9 for how a validation rule can be added. Webtests can also be run data-driven, a datasource can be attached to the webtest. When webtests are created, they can also be used in performance/load testing. The loadtest tool of Visual Studio simulates the test cases with a user-defined load and can also simulate different kinds of platforms.

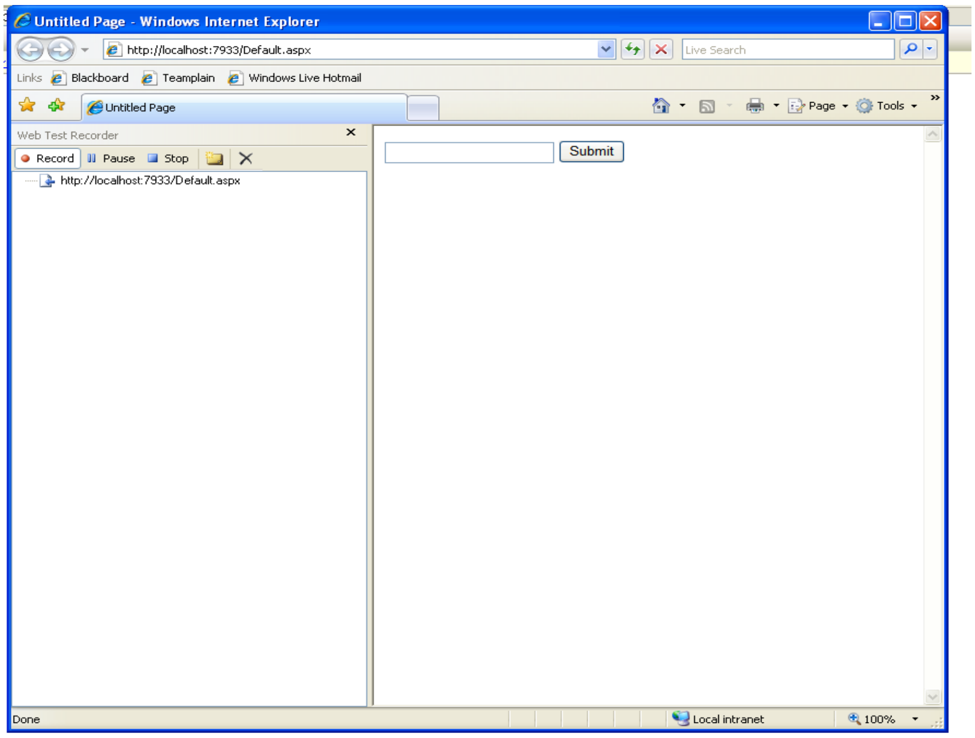
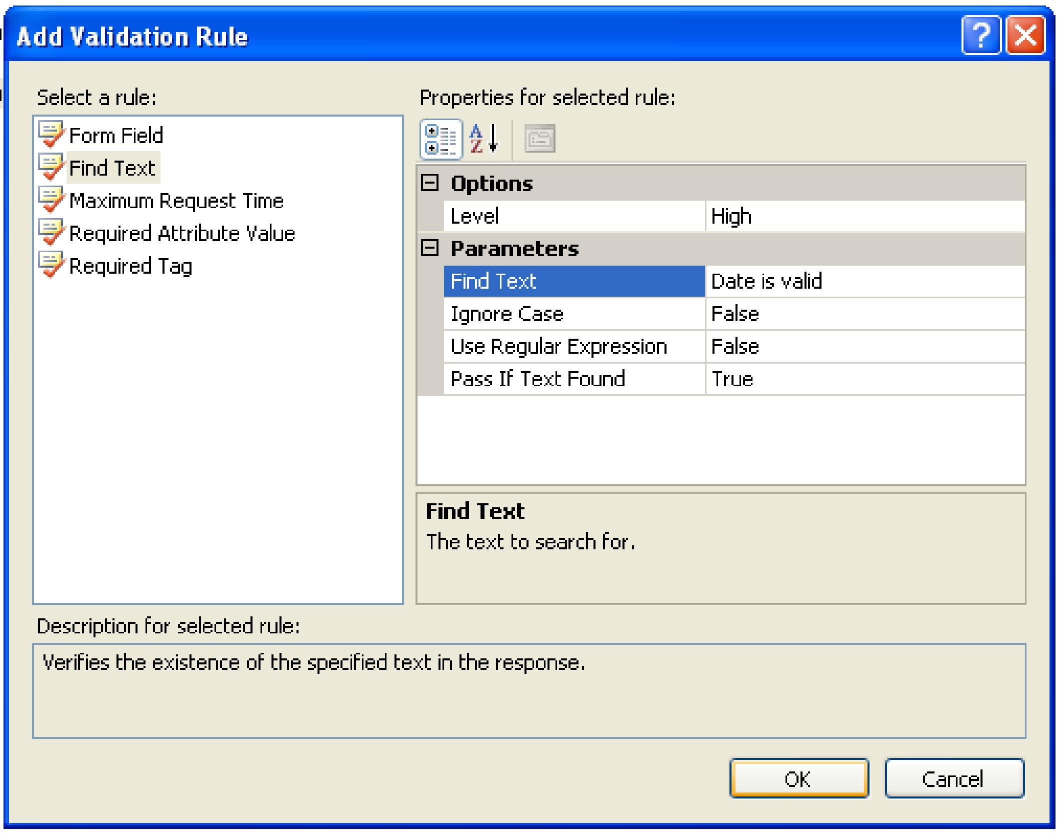

Untitled Page - Windows Internet Explorer
http://localhost:7933/Default.aspx
Web Test Recorder
Record
Pause
Stop
http://localhost:7933/Default.aspx
Submit
Done
Local intranet
100%

Figure 8 - Recording a webtest


Add Validation Rule

Select a rule:
- Form Field
- Find Text
- Maximum Request Time
- Required Attribute Value
- Required Tag

Properties for selected rule:

Options	
Level	High
Parameters	
Find Text	Date is valid
Ignore Case	False
Use Regular Expression	False
Pass If Text Found	True

Find Text
The text to search for.

Description for selected rule:
Verifies the existence of the specified text in the response.

Figure 9 - Add validation rule

4.3.2 Selenium

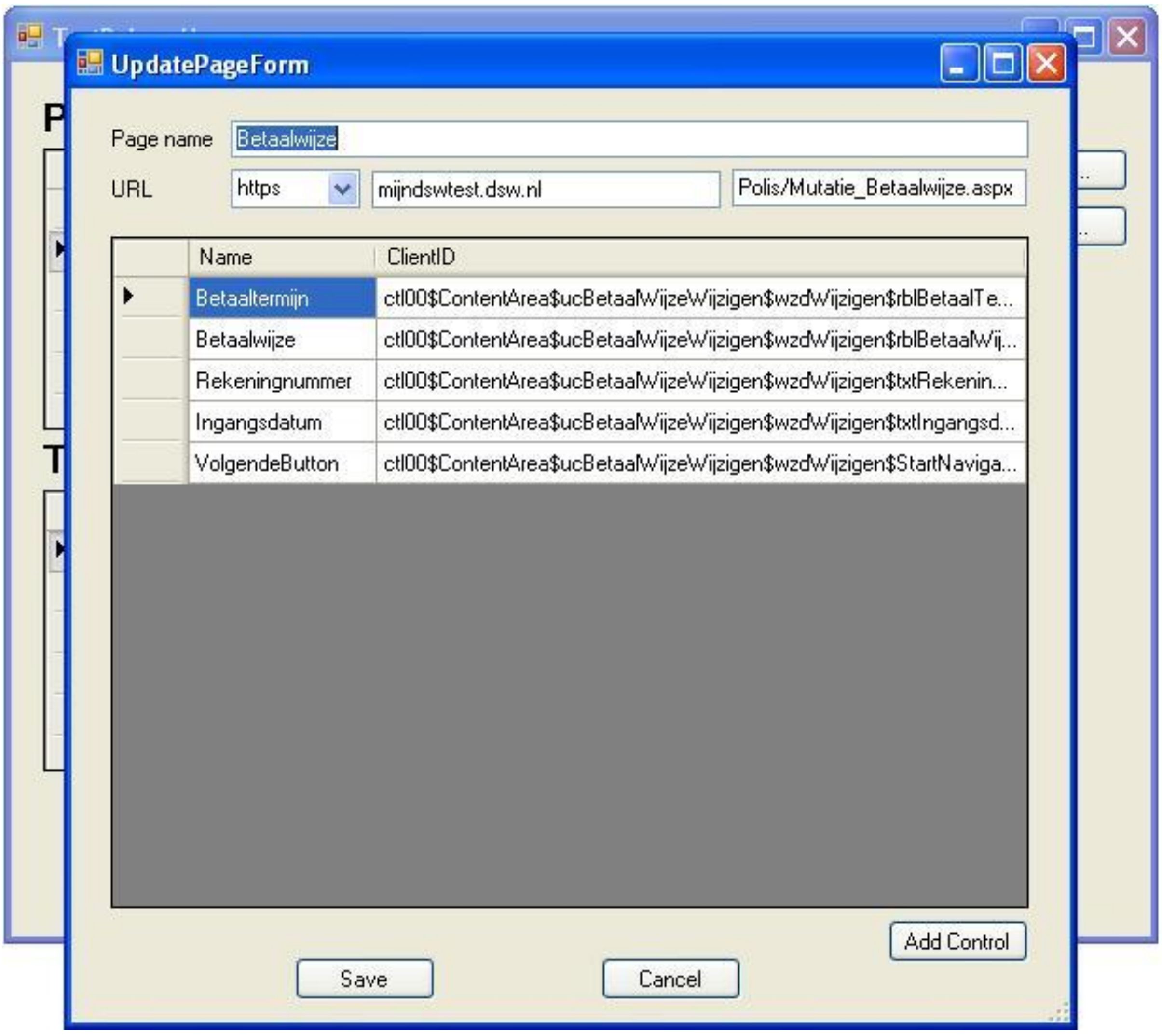
Selenium is an open source record-and-playback tool for web application. This tool is provided as a plug-in for the Mozilla Firefox web browser. Under windows it only supports Firefox. After recording web tests code can also be generated from this tool. The supported languages are: C#, Java, PHP, Perl, Python, and Ruby. A custom code generator can also be used to output in a language other than mentioned above. The NUnit testing framework can also be used in order to automate the tests. The difference with the Visual Studio web test is that Selenium records the user events (click for example), while the Visual Studio web test only records server calls.

4.4 Data-Driven Testing Tool

As can be seen from the data-driven techniques mentioned in section 4.2, a data file can be used to store screen attributes, such as the position of a textbox or a button. Microsoft® Visual Studio® offers data-driven techniques but not as extensive as needed. The data-driven techniques offered by Microsoft® Visual Studio® are as follows. Data-driven webtests can be created and data-driven, but only the input parameters are stored in a separate file. For instance consider the situation that a value must be entered in a textbox; with Microsoft® Visual Studio® only the value is stored in a separate file or database; if the name of the textbox changes, the script has to be re-created. To improve this situation a customized data-driven tool was created to generate data-driven webtests. A screenshot of the tool is given in Figure 10.

With this tool a webpage can be created. Each webpage has a name, URL, and collection of webpage controls. A webpage control has a name and a client id, this id is the generated ID of an ASP.NET control. Then test cases can be created with the tool. A test case has a name and a collection of test case rows or commands. A command consists of an action, webpage, webpage control, and value. Actions specify what action should be performed. For example Navigate is an action, it does not require the webpage and webpage control attributes to be entered, therefore these can attributes can be left null. Another example is the Input action. The test should be told to input a value in a webpage control which is placed on a webpage. After the test is created, the webtest code is generated by the tool. The code must be added to the test project of Microsoft® Visual Studio®. The data (entered by the user/developer) are stored in a database. At execution time the webtest is dynamically created by fetching the data from the database.

This tool enables the Keyword-driven data-driven technique in the Microsoft® Visual Studio® platform. An experiment has been performed to estimate the reduction of maintainability costs by using this tool. The experiment and its results are described in Section 4.6 and Section 4.7, respectively.



Page name: Betaalwijze
URL: https | mijndswtest.dsw.nl | Polis/Mutatie_Betaalwijze.aspx

Name	ClientID
Betaaltermijn	ctl00$ContentArea$ucBetaalWijzeWijzigen$wzdWijzigen$rblBetaalTe...
Betaalwijze	ctl00$ContentArea$ucBetaalWijzeWijzigen$wzdWijzigen$rblBetaalWij...
Rekeningnummer	ctl00$ContentArea$ucBetaalWijzeWijzigen$wzdWijzigen$txtRekenin...
Ingangsdatum	ctl00$ContentArea$ucBetaalWijzeWijzigen$wzdWijzigen$txtIngangsd...
VolgendeButton	ctl00$ContentArea$ucBetaalWijzeWijzigen$wzdWijzigen$StartNaviga...

Figure 10 - Screenshot TestDriver

4.5 Record/Playback Tools versus Data-Driven Testing Tools

As mentioned in section 4.3 a combination of record/playback and data-driven testing will be used. The use of record/playback is not recommended, but it can provide some benefits. Record/playback testing was criticized, because of the maintainability of the generated test scripts.

When no tools are used, testers have to perform all tests manually. If a bug is found, it is reported. Developers will fix the bug and deploy new version of the application in the test environment. Testers will have to perform the same tests to verify that the bug is fixed and that no new bugs introduced. When using a record/playback tool, testers can record their tests during the first deployment of the application and then run these tests if a newer version of the application has been deployed in the test environment. This will reduce the test effort of testers. As mentioned earlier record/playback tools are abandoned because the test scripts generated are difficult to maintain. Experience learns that the probability that a client id of a webpage control changes during minor releases is low. Therefore the probability that scripts have to be re-recorded and maintained is also low. A drawback of this solution is that after a major release of the application, test scripts will have to be rerecorded. But when no tools are used testers would have to retest the application, so not time is spent when recording the test. Data-driven testing on the other hand offers the benefit that tests will not have to be rerecorded, but maintained. The maintenance costs are lower than the costs of re-recording the tests. But in our case data-driven testing can only be performed by the developers with Microsoft® Visual Studio® that is only available to developers. Thus it is not possible to automate the tests of all functionalities. Therefore while developers create the data-driven tests, testers can use record/playback to perform the tests, which are not yet data-driven by developers.

4.6 Experiments

An experiment has been performed to determine the reduction of maintenance costs by using data-driven testing. These results are presented and evaluated in the following subsections. A simple application was created. Tests were written and used for both techniques (recorded and data-driven) to measure the maintenance costs. The application requires the user to log in. When the user is logged in, he can verify whether a date in some format is valid or not. Five simple tests were recorded and scripted. The same five tests were scripted data-driven (generated by the TestDriver). The instructions of the experiment, is given in the appendix B. During the experiment the following steps were performed:

- A name of a control was changed in the login page
- If recorded and data-driven test failed, both tests had to be modified accordingly

Four colleagues performed the experiment. Each of the colleagues reported the time spent and the number of files modified using the recorded script and the time spent and number of files modified using data-driven scripts. Table 8 and Table 9 show time spent and number of modifications for the recorded scripts and data-driven scripts, respectively. The results will be discussed in the next section.

Colleague	Time spent	Number of modifications
Andrea Walop	4 minutes	5
Richard Simpson	1 minute	5
Amit Oemraw	1 minute	5
Qurratulain Mubarak	4 minutes	5

Table 8 - Maintenance costs recorded scripts

Colleague	Time spent	Number of modifications
Andrea Walop	1minutes	1
Richard Simpson	30 seconds	1
Amit Oemraw	30 seconds	1
Qurratulain Mubarak	2 minutes	1

Table 9 - Maintenance costs data-driven scripts

4.7 Evaluation

The results of the experiment are given in the previous section. These results will be discussed. As can be seen from the results, the reduction of maintenance costs are 75%(1x) and 50%(3x). The number of modifications required for simple application and tests is reduced by 80%. It can be concluded from this example the maintenance costs of automated UI tests is significantly reduced by data-driven testing. These results are observed for even very small projects as the one used in the experiment. Because the fact that the effectiveness of data-driven testing can be observed on even small projects like the one of the experiment. The reduction of maintenance costs is set to 56%.

4.8 Concluding Remarks

As business logic is implemented in the presentation layer, more elaborate UI testing is required. Maintenance costs of UI testing is very high when using recorded scripts. A data-driven technique is therefore used. A data-driven tool has been created. The experiment performed during the thesis show that although the experiment has been performed on a very small scale, the tool can reduce maintenance costs by 56%. This means that on larger scale applications, such as the applications at DSW Zorgverzekeraar, the reduction will be higher than 56%.

5 Refactor using Unit Tests

When business logic is implemented in the presentation(UI) layer, elaborate webtesting is necessary. Webtests are run when functionality is complete. This implies that bugs will be detected late in the development life-cycle. In order to find bugs earlier, business logic should be removed from the presentation layer and be put in another layer (code refactoring). Then unit tests must be written or extended. This will be done by using the test-driven development methodology. This chapter therefore describes the Test-Driven Development methodology and the differences between unit testing and UI testing.

Section 5.1 describes the Test-Driven Development methodology. Section 5.2 describes the tools used during the thesis project. Section 5.3 describes the differences of unit testing and UI testing.

5.1 Test-Driven Development

Test-Driven development (TDD), sometimes referred to as Test-Driven Design or Test-First Programming, is a development practice where unit tests are written before code is written (test-then-code) [7, 8]. Running the tests results in a failure. Then code is written that is tested by these tests. If the code is unstructured, the next step is to refactor the code and execute the tests. If tests fail, the code has to be modified. Doing so ensures that the refactor step does not break validity of the code. These steps are outlined in Figure 11 and are performed as followed [9]:

- A number of automated test cases are written
- The new unit tests are run to ensure they fail (since the code is not written yet)
- The code is implemented
- The unit tests are re-run to ensure they pass with the new code
- The implementation is refactored when necessary
- All tests are run periodically (at least once a day) to ensure that new code or changes to the code do not introduce a failure

Doing so allows tests and code to be written incrementally and the system to be tested incrementally. It is similar to a bottom-up approach, because code is written in small pieces and then aggregated.

TDD is a programming technique which leads to the production of thoroughly tested code and the best possible implementation. Because of the existence of tests, programmers can safely modify code and design for maintainability. By this programmers are able to produce more reliable and maintainable code. TDD does not only lead to reliable, but also to maintainable code.

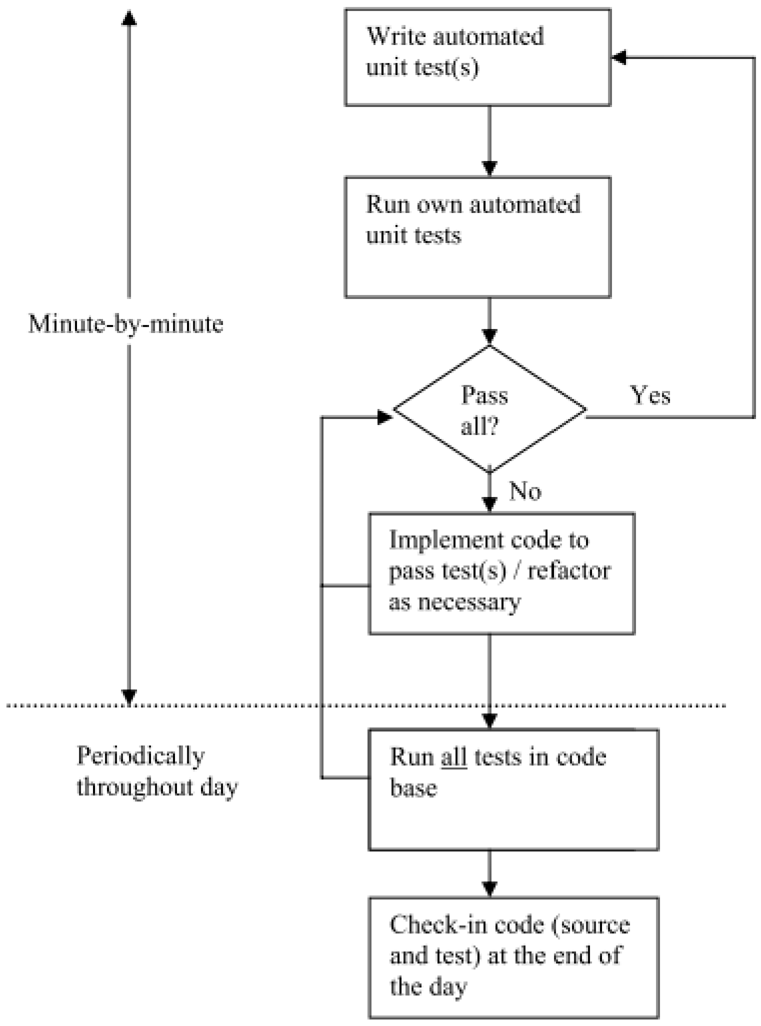

Write automated unit test(s)
Run own automated unit tests
Minute-by-minute
Pass all?
Yes
No
Implement code to pass test(s) / refactor as necessary
Periodically throughout day
Run all tests in code base
Check-in code (source and test) at the end of the day

Figure 11 - Test-Driven Development [9]

These are some of the benefits of TDD. More benefits observed [10, 11] are:

- **Efficiency.** Developers detect faults/failures early in the development cycle when new code is added to the system. This makes it easy to detect and remove bugs, and to reduce debugging time. The additional time spent is compensated by the time reduction gained [11]. Unit testing are actually performed with TDD and becomes an integral part of the project [10]. At IBM 50 percent of improvement was reached in the defect rate [10].

- **Quality improvement.** With TDD, software quality is improved [9-11]. The system is more reliable and maintainable as mentioned above.
- **Reduction of defect inject.** When new code is added to the implementation, it will immediately be tested and defect injection will be reduced. If a test fails after new code has been added, the code has to be changed in order to pass the test and all other tests.
- **Test frequency.** Tests are run more often with TDD. It is therefore straightforward that tests are automated. This makes regression testing easier and allows fast bug detection when new code is added.

Some disadvantages observed in [12] are:

- **Lack of test framework.** If no testing framework can be found for the programming language, then TDD can become a poor choice. However test frameworks are currently available for most programming languages.
- **Faults may not be found.** By the bottom-up approach (code is written in small pieces and aggregated) it becomes unlikely to find 'deeper' faults (faults which require more understanding of the code).

5.2 Tools

Tools are necessary to perform efficient unit testing. There are two tools that can be used:

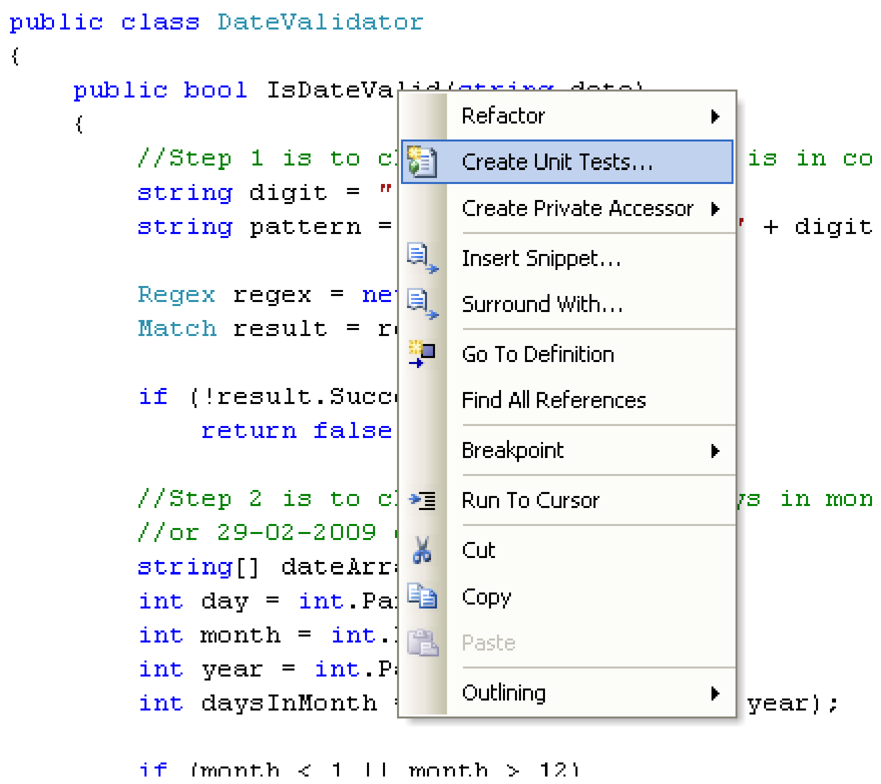
- Microsoft® Visual Studio® Team Edition for Software Testers
- NUnit

Because developers use Microsoft® Visual Studio® for development with the test tools integrated, only this tool will be used. Therefore only the Microsoft® Visual Studio® Team Edition for Software Testers will be described here.

5.2.1 Microsoft® Visual Studio® Team Edition for Software Testers

Microsoft® Visual Studio® Team Edition for Software testers enables testers to perform web, load and unit tests fully integrated in the Visual Studio Environment. More detail of the Microsoft® Visual Studio® Team Edition for Testers was provided in section 4.3.1.

In Figure 12 one can see how a unit test is created in Visual Studio.This can be done either by right-clicking in the source code and selecting “Create Unit Tests…” or by manually creating a unit test. If the developer clicked “Create Unit Tests…” in the screen of Figure 12, the developer gets a window from which he can choose the methods of the class for which an automated unit test should be created (Figure 14). After the selection is made, the unit test is then generated; a sample of generated code is given in Figure 13.



```
public class DateValidator
{
    public bool IsDateVa
    {
        //Step 1 is to c                      is in co
        string digit = "
        string pattern =                     ' + digit

        Regex regex = ne
        Match result = r

        if (!result.Succ
            return false

        //Step 2 is to c                     ys in mon
        //or 29-02-2009
        string[] dateArr
        int day = int.Pa
        int month = int.
        int year = int.P
        int daysInMonth                      year);

        if (month < 1 || month > 12)
```

Figure 12 - Create Unit Tests in Microsoft® Visual Studio® Team Edition

```
[TestMethod()]
public void IsLeapYearTest()
{
    DateValidator target = new DateValidator();

    int year = 0; // TODO: Initialize to an appropriate value

    bool expected = false;
    bool actual;

    actual = target.IsLeapYear(year);

    Assert.AreEqual(expected, actual, "Entities.DateValidator.IsLeapYear did not return the expected value.");
    Assert.Inconclusive("Verify the correctness of this test method.");
}
```

Figure 13 - Generated Unit Test

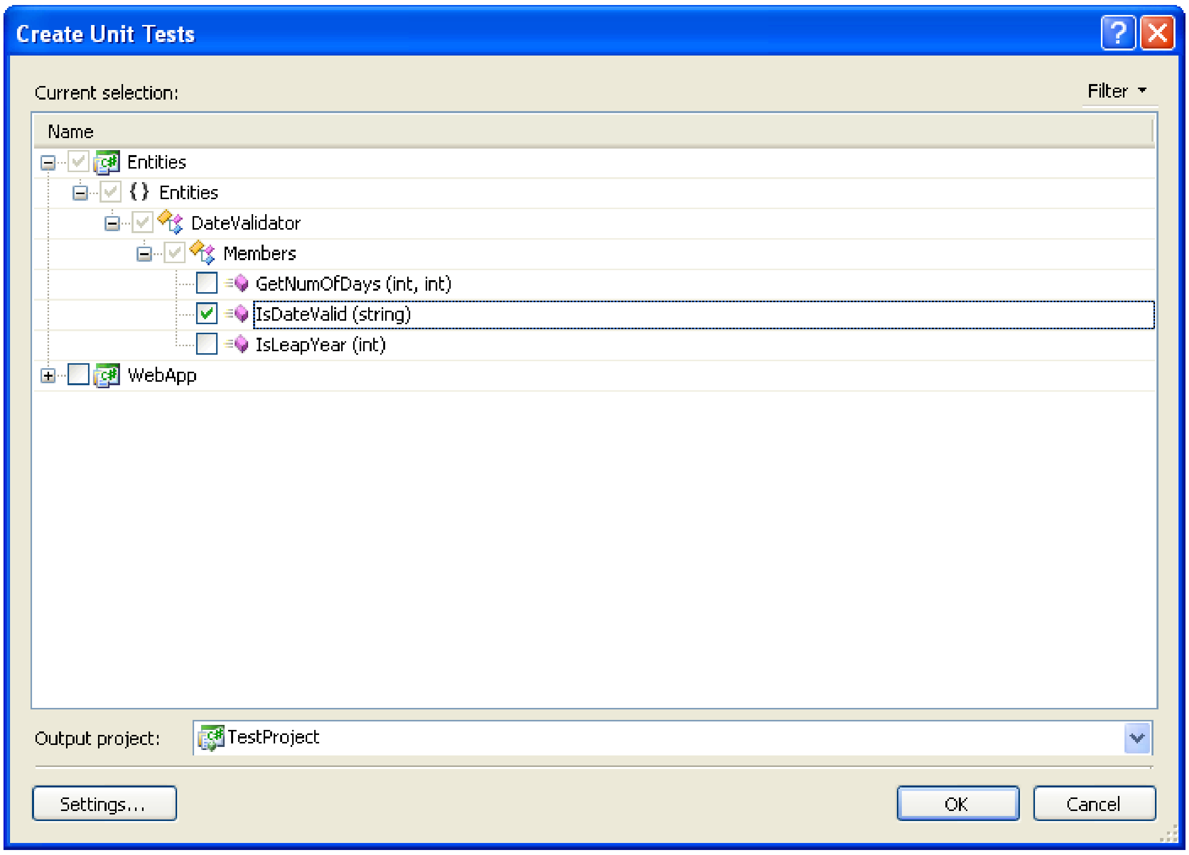


Figure 14 - Create Unit Tests Wizard

The generated test can then be modified to do the actual testing. These tests can also be run data-driven, by adding the following line of code above the test method:

```
[DataSource("Data Source=datasource;uid=username;password=password;",
"TestCaseTable")]
```

5.3 Unit versus UI Testing

UI testing requires the functionality to be built first, then UI tests are created and executed. Unit testing does not require the complete functionality to be built, but only the individual units to be constructed. Therefore unit tests can be executed earlier than the UI tests during the development lifecycle and bugs are detected early. This not only reduces test efforts, but implementation effort also.

Consider the following example. See Figure 15, ReisverzekeringControl is the UI component, where ReisverzekeringTask is the controller and is used by the ReisverzekeringControl to perform operations on the Reisverzekering object. Suppose that no unit testing is performed, but only automated UI testing is performed. A fault in the Reisverzekering object will be detected by the UI tests. The developer will have to inspect the code of all the three components in order to find the code which triggered the fault. If unit tests were created and executed for the Reisverzekering object, fault would have been detected earlier and the developer would not have to investigate the code of the ReisverzekeringControl and ReisverzekeringTask. Thus unit tests also reduce the time to investigate code and find the error that triggered a fault.

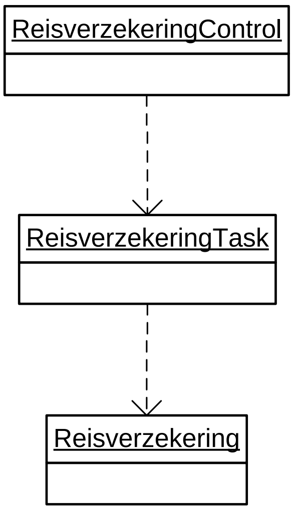


Figure 15 - Example usage diagram

Maintenance costs of UI tests are higher than the maintenance costs of the unit tests. Unit testing can reduce costs significantly and will find bugs early in the development cycle. When using both unit and UI testing, more benefits may be gained, which will be explained next.

Consider the situation that only unit testing is performed. Unit tests lead to early bug detection. Therefore the system will contain less bugs, which means that the system will be deployed only a few times. This means that test effort is reduced, because testers will not have to test the same functionality again and again. Another benefit of unit tests is that code can be refactored while the unit tests are in place. Modification in the units can then be validated by the unit tests.

Unit tests therefore reduce test effort and increase system quality (maintainability and reliability). On the other hand if automated UI testing is also performed, then test efforts will be reduced again, because some of the tasks of the testers are automated and testers will be able to focus on more difficult tasks. So the combination of automated UI tests and unit tests will reduce test effort and increase system quality. The optimal solution is to use both, unit tests and automated UI tests.

Another reason why automated UI testing is effective and necessary is that unit tests can only detect faults in dedicated business units. If some business logic is not implemented in a unit, but in a user-interface on a button click event, this is not covered by a unit test, what introduces a risk for the system. UI testing will cover these code and therefore faults in business logic implemented in the user-interface will also be detected.

Another important reason to perform automated UI testing is that UI tests can be executed in the several environments and platforms of the system, i.e. test environment, acceptance envrinoment, Windows XP, Windows 2003, Linux. Real world scenarios can me simulated by UI tests. Unit tests cannot be run on other environments, unless the test is also deployed with the code, which is not the case for DSW Zorgverzekeraar.

5.4 Concluding Remarks

Test-Driven Development is an effective methodology to detect faults/bugs in the system, while the system is under construction. Tests are written first. Then the implementation code is written. The code is then tested, if the code is unstructured, it is also refactored and retested. Following the Test-Driven Development systems with high reliability and maintainability is produced.

When business logic is implemented in the UI layer, unit tests are not able to detect faults in the business logic, therefore UI tests are used. UI tests reduce test effort significantly, but they introduce high maintenance costs; they require functionality to be built before the tests are created and executed. In order to decrease the maintenance costs, unit tests are created. But in order to detect the bugs with unit tests, the business logic must be moved to some other layer. And after the code is refactored, unit tests are created to test the code. Unit testing enables the testing of the individual units early in the development lifecycle, early bug detection and higher maintainability. In order to produce systems with high reliability and maintainability and keeping testing costs low, the combination of UI tests and unit tests should be applied.

6 Case Studies

Case studies where performed to determine the effectiveness of UI testing alone and of UI testing in combination with unit testing. These case studies are described and their results are presented. Section 6.1 describes the case study performed with UI testing, and presents the results of the case study. Section 6.2 describes the case study performed with the combination of unit and UI testing and presents the result of the case study.

6.1 UI Testing

During the case study of UI testing, UI tests were created for part of the functionalities of the DSW Zorgverzekeraar's web application. These tests were run every time the system was deployed for regression testing. The web tests discovered some bugs immediately, before testers could find them. Developers fixed these bugs and subsequently deployed the fix to the test environment. Unfortunately, it was not possible to measure the test effort improvement by test automation during the test period, because testers did not have much time and the deadline was strict. Therefore an estimation is given of how much time (in percentage) would have been saved by automated UI testing.

Section 6.1.1 presents the bug rate during the test period. Section 6.1.2 describes the test effort improvement which was achieved by automated UI testing.

6.1.1 Bug Rate

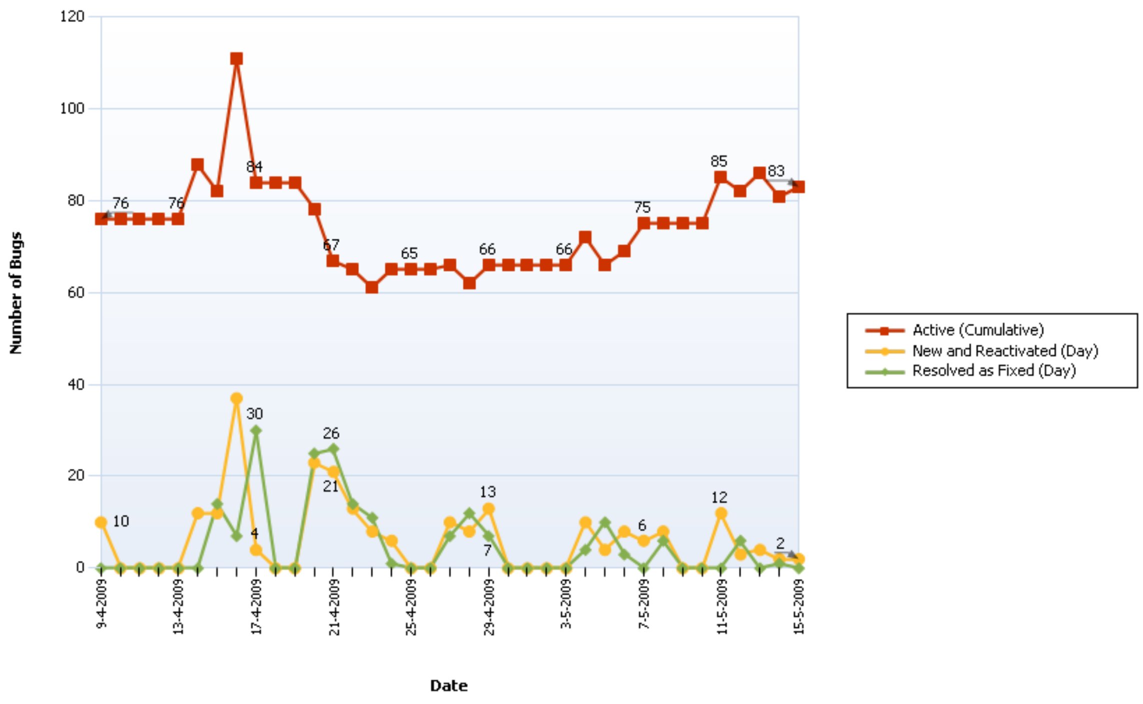


Figure 16 - Bug rates after test automation

After the testing period a report of the bug rate was generated by Teamplain. The bug rate during the test period is presented in Figure 16. The difference between the active bugs at the start of the testing process and at the end of the project is 7 bugs (red line). As can be seen from the figure also, 2 new bugs remain in the system. This means that 2 new bugs were introduced and 5 bugs of the previous versions of the system were not fixed.

6.1.2 Test Effort

Because of the strict timeline and pressure during the test period it was not possible to measure the test effort improvement, also many of the testers were not aware of the automated tests. The measurement is therefore performed by the author of this document. He performed the same steps as the automated tests and measured the time spent. These steps were performed for one functionality. The results of the time spent by the tester and by automated test are given in Table 10.

Test	Duration manual test (sec.)	Duration automated test (sec.)
Test case 1	120	11
Test case 2	120	11
Test case 3	60	11
Test case 4	60	11
Test case 5	60	11
Test case 6	120	11
Test case 7	120	16
Test case 8	120	11
Total	780	93

Table 10 - Manual versus Automated Test

As can be seen from Table 10, the total time spent during testing by the user is 780 seconds and that of the automated test is approximately 90 seconds. The test effort for this functionality has been reduced by 88%.

6.2 UI and Unit Testing

During the case study, unit tests were created for a part of the system, of which code was refactored. The existing UI tests were used to validate the modifications. However the refactoring performed was very broad, the main units (which are used almost everywhere in the system) were also refactored. Therefore the existing UI tests were not strong enough to cover the whole system and detect the faults. Only one developer was available during this period to write unit tests, which led to limited unit testing. Therefore measurements show that unit testing was ineffective and lead to higher costs, because the number of production bugs were higher and the number of deployments were increased. This results will be discussed in section 6.2.1 and some causes of increased costs are described in section 6.2.2.

6.2.1 Results Refactoring using Unit Tests

A great portion of code was refactored, while limited specifications were available. Only one developer created unit tests which were run during the build on the server (on every check in). Many bugs were detected by testers during the testing period, but not for the functionalities for which the automated UI tests were created. These were detected by the automated UI tests themselves. After production many major bugs were found (a total of 6 bugs). These bugs were detected in other functionalities, for which no automated UI tests existed. The number of deployments was also increased, because many bugs were detected, but the deployments in the testing environment started early (when a functionality was completed it was deployed in the test environment) . There was a total of 29 deployments.

6.2.2 Costs Increase

As can be seen from the results of the previous section, the code refactoring and unit tests may have increased costs. The number of deployments and the number of bugs detected by end-users were both increased. But as also can be seen from the results, the functionalities for which the automated UI tests were created, were ‘stable’, i.e. no bugs were detected for those functionalities by end-users. Because refactoring was broad, and little specification was available, and the design documents were not available, unit tests could not detect all faults.

Another reason why it seems that unit testing has increased costs is that limited unit testing has been performed. Only one developer with little specifications cannot create unit tests for all possible scenarios, because not all scenarios were specified and one developer does not have enough resources/time to determine the scenarios which were not specified.

6.3 Concluding Remarks

Case studies have been performed to determine the effectiveness of UI testing only and UI testing in combination with refactoring and unit testing. The case study performed with UI testing only, shows that automated UI testing can significantly reduce test effort, while introducing refactoring and unit testing can increase costs and test effort. This increase in costs and test efforts are caused by incomplete/insufficient specifications.

7 Cost-Benefit Analysis of Test Automation

The results of the case studies performed were presented in Chapter 6. These results are used in this chapter to measure the costs and benefits of test automation. At first, the measurement techniques are described. Then the actual costs and benefits are computed.

Section 7.1 describes the measurement techniques. Section 7.3 presents the costs and benefits of UI testing. Section 7.4 presents the costs and benefits from UI and unit testing.

7.1 Measurement Techniques

The software development process at DSW Zorgverzekeraar looks like the one given in Figure 17.

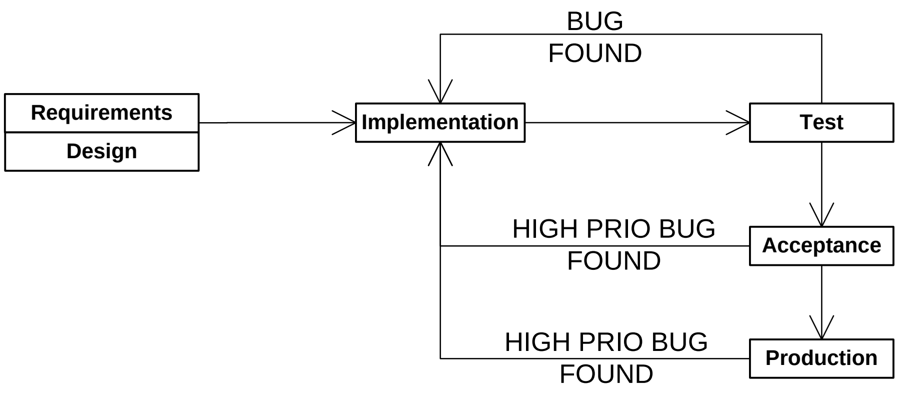


Figure 17 - Software Development process at DSW Zorgverzekeraar before Test Automation

The total costs of the software development process (TC_{SDP}) can be computed by computing the number of hours spent on requirements and design ($TC_{R\&D}$), the number of hours spent on the implementation (TC_{IMPL}), the number of hours spent on testing (test effort, TC_{TEST}, costs of testing in the test phase), and the number of hours spent on acceptance testing (TC_{ACCP}). As can be seen from the figure a number of cycles exists. If a bug is found during the testing period, acceptance period, or production period, the bug is fixed and the system is then deployed in the testing environment and retested. Total costs of testing is determined by the number of cycles in test phase and TC_{TEST}, the number of cycles in acceptance and TC_{ACCP} (Equation 7.1).

$$TC_{TEST\ EFFORT} = numcycles_{TEST} \times (TC_{TEST}) + numcycles_{ACCP} \times (TC_{ACCP})$$

Equation 7.1

The total costs of the software development process can be computed by the formula described in Equation 7.2. In this formula the average costs of to fix bugs (AC_{BUGFIX}) is also taken into account.

$$TC_{SDP} = TC_{R\&D} + TC_{IMPL} + TC_{TEST\ EFFORT} + (\text{avg. number of bugs solves} \times AC_{BUGFIX})$$

Equation 7.2

As can be seen from section 3.2.2 the average costs to fix a bug is set to two hours. If the total number of bugs fixed is estimated at 100, then the total time to fix bugs is 200 hours. Because a small part was automated during the thesis project, the number of bugs were to affected (by much). The factor avg. number of bugs solves x AC_{BUGFIX} is therefore left out from Equation 7.3. Equation 7.3 will be used in the further analysis.

$$TC_{SDP} = TC_{R\&D} + TC_{IMPL} + TC_{TEST\ EFFORT}$$

Equation 7.3

In order to detect bugs during the acceptance testing period, it is recommended to use the same test set that was used to test during the testing period. The total number of hours spent on acceptance testing would then be equal to the total number of hours spent on the testing, because the same amount of time would be needed to execute the same tests. This produces the formula presented in Equation 7.4.

$$TC_{TEST\ EFFORT} = (numcycles_{TEST} + numcycles_{ACCP}) \times TC_{TEST}$$

Equation 7.4

In order to compute TC_{TEST} (total costs of testing for 1 cycle) the number of hours required to execute the several test cases must be computed.

The web application project currently consists of 90 use cases, of which 80% are complex. For each complex use case 20 test cases are required on average, and for each simple use case 10 test cases are required. There are 72 complex use cases and 18 simple use cases, which leads to 72 * 20 + 18 * 10 = 1620 test cases. A tester spends a certain amount of time to test the system. In order to test a complex use case a tester needs 10 minutes and to test a simple use case, the tester will need 5 minutes. The total number of hours needed to test the entire system equals to (72 x 20 x 10 +

18 x 10 x 5)/ 60 = 255 hours. This is presented in Figure 18. As can be seen from the figure, the execution of a test cases costs 0.15 hours.

Now the costs of requirements, design, and implementation are computed. Currently there are 9 developers working on the web application, of which 4 work on other projects, what results in 6 full time developers working on the web application project. The implementation period including requirements analysis and design (which is also performed by the developers) of the web application project is set to 8 weeks = 320 hours. Thus:

$$TC_{IMPL} + TC_{R\&D} = 6 \text{ x } 320 = 1{,}920 \text{ hours}$$

Equation 7.5

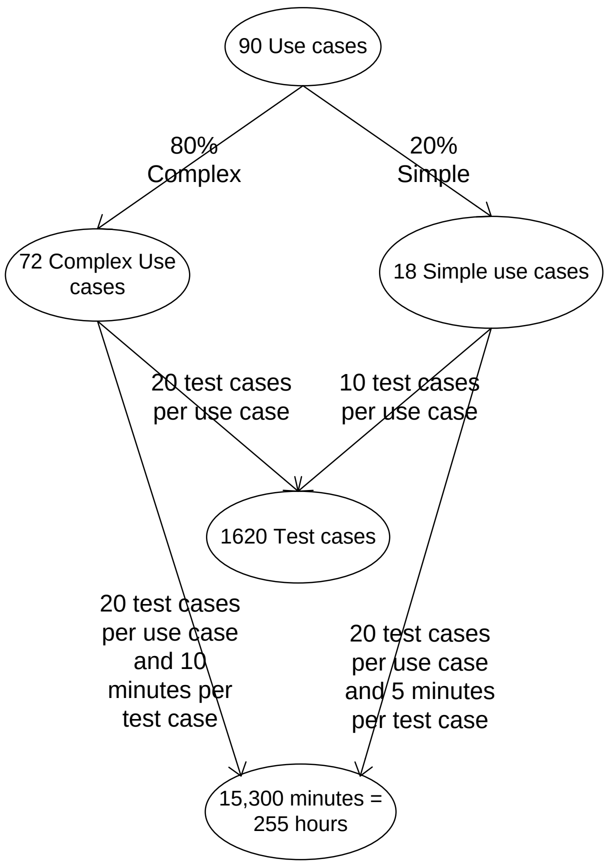

90 Use cases
80% Complex
20% Simple
72 Complex Use cases
18 Simple use cases
20 test cases per use case
10 test cases per use case
1620 Test cases
20 test cases per use case and 10 minutes per test case
20 test cases per use case and 5 minutes per test case
15,300 minutes = 255 hours

Figure 18 - Computing test effort from the number of use cases without test automation

7.2 Test Effort before Test Automation

As can be seen from Figure 18, the total number of hours required to test the entire system is set to 255. Thus TC_{TEST} = 255 hours. To be able to compute the total costs of testing, the number of cycles of testing, and acceptance testing.

From experience with acceptance testing the average number of cycles during acceptance testing period is set to 5, and the total number of cycles to 26. Using Equation 7.4:

$$TC_{\text{TEST EFFORT}} = (26 + 5) \text{ x } 255 = 7905 \text{ hours.}$$

Equation 7.6

This result shows that it is not possible to test the entire system, if a release is planned for three months. The costs of development is then:

$$TC_{\text{SDP WITHOUT TEST AUTOMATION}} = 1920 + 7905 = 9825 \text{ hours}$$

Equation 7.7

7.3 UI Testing

The formulas constructed in the previous section will be applied in this section to compute the test effort with test automation. The reduction of the test effort is, as stated in Section 6.1.2, set to 88%. When this reduction is applied to Equation 7.6, the total number of hours required to test per cycle is:

$$TC_{\text{TEST}} = (100\%\text{-}88\%) \text{ x } 255 = 30.6 \text{ hours.}$$

Equation 7.8

From Equation 7.8 $TC_{\text{TEST EFFORT}}$ can be computed:

$$TC_{\text{TEST EFFORT}} = (26 + 5) \text{ x } 30.6 = 948 \text{ hours}$$

Equation 7.9

But some additional effort is required to create the automated tests. To create a recorded webtest, the test has to be executed and recorded, which means that the creation time of a webtest equals the time to test the functionality. Hereafter the test has to be refactored. This process is given in Figure 19.

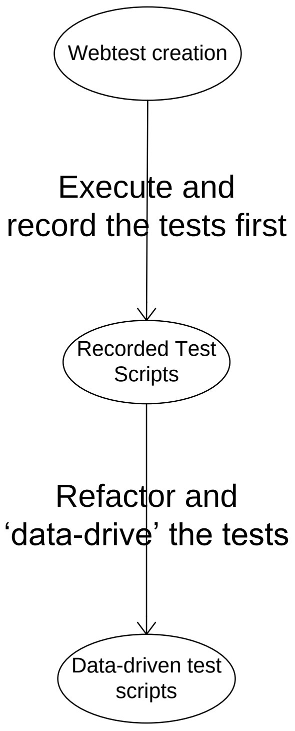

Webtest creation
Execute and record the tests first
Recorded Test Scripts
Refactor and 'data-drive' the tests
Data-driven test scripts

Figure 19 - Web test creation

The refactoring efforts are estimated at 50% of the creation effort. With data-driven scripts, test script maintainability is reduced by 56%. After test scripts are created, 56% of the time to create the script is spent to maintain the data-driven webtests. Thus, a component ($TC_{TEST\ MAINTENANCE}$) is introduced in Equation 7.3:

$$TC_{SDP} = TC_{R\&D} + TC_{IMPL} + TC_{TEST\ MAINTENANCE} + TC_{TEST\ EFFORT}$$

Equation 7.10

Test maintenance costs are set to 56% of $TC_{TEST\ WITHOUT\ AUTOMATION}$. But not all use cases are modified after each release, therefore test maintenance costs are lower than 56% of TC_{TEST}. It is estimated that on average 40% of the use cases cases are modified. Therefore $TC_{MAINTENANCE}$ can be set to:

$$TC_{TEST\ MAINTANCE} = 40\% \times 56\% \times TC_{TEST\ WITHOUT\ AUTOMATION} = 0.224 \times 255 = 58 \text{ hours}$$

Equation 7.11

Thus, the total number of hours spent during the software development cycle with UI test automation is:

$$TC_{SDP\ WITH\ UI\ TEST\ AUTOMATION} = 1920 + 58 + 948 = 2926 \text{ hours}$$

Equation 7.12

The test suite creation time is:

$$TC_{TEST\ CREATION} = 150\% \times TC_{TEST} = 11857 \text{ hours}$$

Equation 7.13

In order to create the automated test suite for the entire system, the total costs will be 11,857.5 hours. The total costs of the software development will then be increased by this amount. The reduction of automated testing is, if testst are already created: 9,825.0 – 2,926.6 = 6898.4 hours or 70%. Thus the test creation costs can be overcome by two releases.

Consider the case that it is not feasible to automate all of the tests of the system. Consider that it is only feasible to automate 40% of the tests. This means that 60% of the tests are performed manually and 40% of the tests are automated. The TC_{TEST} is then (assuming that tests are already automated):

$$TC_{TEST} = 0.6 \times 255 + 0.4 \times 255 \times 0.12 = 165.2 \text{ hours}$$

Equation 7.14

Thus:

$$TC_{TEST\ EFFORT} = (26 + 5) \times 165.24 = 5122 \text{ hours}$$

Equation 7.15

Thus:

$$TC_{SDP\ WITH\ LIMITED\ UI\ TEST\ AUTOMATION} = 1920 + 58 + 5122 = 7100 \text{ hours}$$

Equation 7.16

The reduction is then: 9825 – 7100 = 2724, and thus 27.7%. In this case the test creation costs would be 40% of the test creation costs given in Equation 7.13. This results in:

$$TC_{TEST\ CREATION(LIMITED)} = 2964 \text{ hours}$$

Equation 7.17

In this case the test creation costs can also be overcome in two releases.

7.4 UI and Unit Testing

With unit tests in place the number of hours spent on the implementation, requirements and design will be increased, because each unit's behavior has to be specified for unit testing. During the development period design documents and unit tests will have to be maintained. This can increase costs by 50% of the implementation costs (in the worst case). Therefore $TC_{R\&D}$ + TC_{IMPL} is set to 150% of the $TC_{R\&D}$ + TC_{IMPL} computed in Equation 7.5:

$$TC_{R\&D} + TC_{IMPL} = 1.5 \text{ x } 1920 = 2880 \text{ hours.}$$

When a unit test suite is in place, the number of cycles will be reduced, because faults will be detected early and no deployments will be needed. The number of cycles after introducing unit testing was 29 (section 6.2.1). This number is higher than the average number of cycles of 26, because the testing period started a week after the development period, which allowed developers to deploy completed components (after refactoring) to be tested by the testers. Deployments were performed in the testing environment each time a component or functionality was complete or a bug was fixed. Therefore the number of cycles during the implementation period is not counted and the number of cycles is set to 22. The TC_{TEST} remains unchanged (Equation 7.8). Thus the $TC_{TEST\ EFFORT}$ is:

$$TC_{TEST\ EFFORT} = 22 \text{ x } 30.6 = 673.2 \text{hours}$$

Equation 7.18

Thus:

$$TC_{SDP\ WITH\ UI\ AND\ UNIT\ TEST\ AUTOMATION} = 2880 + 58 + 673.2 = 3611.2 \text{ hours}$$

Equation 7.19

Comparing the results of Equation 7.19 with Equation 7.12, it can be said that the total developmnent costs have been increased by 3611.2 – 2926.6 = 684.6 hours or 23%. This increase is because at this moment unit testing is performed at a small scale and therefore the effect of it is not visible yet.

When specifications are created and maintained in order to create and maintain unit tests, unit tests will be able to detect faults/bugs early. The use of specifications also increases the maintainability of the system, developers know what a unit does (it's behavior is specified). Thus the system will be more reliable and maintainable on a long-term and test efforts will also be decreased, because unit tests will be able to detect bugs early.

Consider the case where specification and design documents are maintained. At the author's company (SSDC NV with just 10 kilo lines of code), design documents are available (and maintained), units are specified and a strong unit test suite is in place (almost 95% code coverage). In this case cycles are only caused by errors in the requirements (the requirements did not match the wish of the customer), usability issues (text color, or information missing on a screen). Because functionality is complete and stable (i.e. no unexpected errors occur when the submit button is clicked) testers/users can report most, if not all, usability issues at one time (which results in one cycle) and the requirement mismatches can also be determined and reported. Therefore the number of cycles can be reduced by 50%, that results in:

$$TC_{TEST\ EFFORT} = 11 \times 30.6 = 336 \text{ hours}$$

Equation 7.20

If code is written and tested thoroughly by unit tests and documentation is available, the total costs of requirements, design, and implementation will not be increased by 150%, but just slighlty (as also observed at SSDC NV). Thus the implementation costs remain the same. Because systems at SSDC are much smaller than the web application at DSW Zorgevrzekeraar, the increase of requirements, design and implementation costs is set to 125%. Thus:

$$TC_{R\&D} + TC_{IMPL} = 125\% \times 1920 = 2400 \text{ hours}$$

Equation 7.21

The total costs of software development is then:

$$TC_{SDP\ WITH\ UI\ AND\ UNIT\ TEST\ AUTOMATION} = 2400 + 58 + 336 = 2794 \text{ hours}$$

Equation 7.22

The reduction of unit and UI test automation compared to only UI test automation is then 2926 – 2794 = 132 hours or 4.5% and a more reliable and maintainable system is produced. Automated unit and UI testing does not only reduce testing costs, but also implementation costs in a long run, especially when code needs to be refactored. The reduction of test effort and total costs of the software development process is given Table 11.

	TC_{TEST}	$TC_{TEST\ EFFORT}$	$Reduction_{TEST\ EFFORT}$	TC_{SDP}	Reduction
No test automation	255	7905	-	9825	-
UI test automation	30.6	948	88%	2926	70%
Limited UI test automation	165.2	5122	35%	7100	27%
UI and Unit testing	30.6	336	95%	2794	71%

Table 11 – Reduction by test automation

7.5 Concluding Remarks

The results of the case studies performed in Chapter 6 were evaluated in this section. From the results can be concluded that the introduction of unit tests has increased the total costs slightly. An important cause of this increase is the lack of specifications. If no specifications exists for a unit, it becomes a difficult task to create unit tests, because developers may not know how to verify the unit's behavior. If specifications are maintained, unit tests will be able to detect bugs early and therefore the number of cycles will be reduced significantly over time. This will not only cause test efforts to decrease and so decreasing the total costs, but will also lead to the production of more reliable and maintainable systems.

8 Conclusion

This chapter presents the conclusion of this document. The activities of the thesis project are summarized at first. Then the conclusion is presented. Section 8.1 presents the brief summary of the thesis project. And section 8.2 presents the final conclusion.

8.1 Brief Summary

Test automation was introduced at DSW Zorgverzekeraar during the Master Thesis. In order to introduce test automation in an organization, the test maturity level of an organization must be at a certain level. The current testing process was therefore analyzed. Important key areas were improved in such a way that test automation would be effective. Then automated UI testing was introduced, because most of the business logic at DSW Zorgverzekeraar’s web application was implemented in the UI layer. But automated UI testing has some drawbacks, such as high maintainability costs, and functionality should be complete before creating the automated UI test. The business logic was therefore moved from the UI layer to another layer. This enabled unit testing to detect faults in the implementation of the business logic. Then the results are presented from the case studies performed. These results are then used to compute the costs of test automation and the costs reduction of the testing process and the entire software development process.

8.2 Conclusion

Efficient test automation requires a structured test process. The test process at DSW Zorgverzekeraar was therefore analyzed by applying the TPI model. From this analysis was concluded that only the key area test automation needed improvements.

After the analysis of the current test process, automated UI testing was introduced at DSW Zorgverzekeraar. There were two techniques to perform UI testing, record-and-playback and data-driven testing. In the literature survey was concluded that the data-driven technique was more efficient. Therefore a data-driven testing tool was created, because the data-driven technique mentioned in the theory of data-driven testing is not available in current testing tools.

UI testing has some drawbacks, such as high maintainability costs; it requires functionality to be built first in order to be executed. Unit testing can therefore be more efficient, because with unit testing bugs can be detected early, and the maintainability of the system is also increased. Therefore a combination of automated unit and UI testing should be used.

Case studies have been performed to estimate test effort reduction and reduction of costs of the software development process by automated unit and UI testing. The results of these case studies performed are used to compute the costs of test automation and the costs reduction by test automation. These computations are presented in chapter 7. As can be seen from the computations, even if 40% of the test cases were automated, the total costs of software development is reduced by 27.7% with only UI testing. While unit testing can reduce the number of cycles, this reduction will be higher, if performed well. With unit testing, not only software reliability is increased, but also maintainability; with a automated unit test suite, code can be refactored easily. Thus it can be concluded that test automation not only reduces costs of the software development process, but also increases maintainability and reliability.

Bibliography

1. Dustin, E., J. Rashka, and J. Paul, *Automated Software Testing: Introduction, Management, and Performance*. 1999: Addison-Wesley.
2. Binder, R.V., *Testing Object-Oriented Systems: Models, Patterns, and Tools (The Addison-Wesley Object Technology Series)*. 1999: Addison-Wesley Professional.
3. Pol, M., R. Teunissen, and E. van Veenendaal, *Software Testing: A Guide to the TMap Approach*. 2002: Addison-Wesley.
4. *Dependencies Key Areas*. 2004 [cited 23-03-2009]; Available from: http://www.iquip.nl/images/Dependencies_tcm6-32140.pdf.
5. Memon, A.M. and M.L. Soffa, *Regression testing of GUIs*, in *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*. 2003, ACM: New York, NY, USA. p. 118--127.
6. Zambelich, K., *Totally Data-Driven Automated Testing*.
7. Koskela, L., *Test driven: practical tdd and acceptance tdd for java developers*. 2007, Greenwich, CT, USA: Manning Publications Co.
8. Beck, K., *Test Driven Development: By Example*. 2002, Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.
9. Bhat, T. and N. Nagappan, *Evaluating the efficacy of test-driven development: industrial case studies*, in *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*. 2006, ACM: New York, NY, USA. p. 356--363.
10. Maximilien, E.M. and L. Williams, *Assessing test-driven development at IBM*, in *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*. 2003, IEEE Computer Society: Washington, DC, USA. p. 564--569.
11. Williams, L., E.M. Maximilien, and M. Vouk, *Test-Driven Development as a Defect-Reduction Practice*, in *ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering*. 2003, IEEE Computer Society: Washington, DC, USA. p. 34.
12. Jorgensen, P.C., *Software Testing: A Craftsman's Approach*. 1995: CRC Press, Inc.

Appendix A Test Process Analysis Questionnaire

For each of the following questions/statements, the answer can either be true/false or yes/no

Life-cycle model

The test process is minimally phased in: planning, specification, and execution.

For the planning phase, the test basis and test strategy are determined, the test organization is set up, test products are determined, the test infrastructure and tools are defined, planning is determined, and the test plan is set up.

For the specification phase, test specifications and test scripts are created, the test object and test infrastructure are specified, the test infrastructure is realized.

For the execution phase, tests are executed, and test results are documented.

Estimating and planning

The test estimation and planning are founded (i.e. it is not enough to say " We did it like this last time")

During the test process the test estimation and planning are monitored, and adjusted as necessary.

Specification techniques

The test cases are derived with some techniques.

This techniques leads to the description of test cases: begin, actions to be executed, and expected result.

Metrics

During the test process the following input-statistics are monitored:

- resources used(time)
- activities performed (execution time)
- size of the system to be tested (number of functionalities)

During the test process the following output-statistics are monitored:

- test products: specifications and test cases
- test progress: tests already executed, completed?
- Number of bugs and their status

These statics are used in the test report

Test Automation

Decision has been made to automate some activities in the test planning/execution. The test management/coordinator and the IT department are involved in this decision.

Automated tools are used to support test planning/execution

The test management/coordinator realize that the use of tools provide more benefits than disadvantages.

Decision has been made which tests to automate and which not.

A test tool is used.

If record and playback tools are used, then the maintenance of the tests is taken into account during development.

The test tools can be used in the next test process.

Commitment and motivation

Management drives testing with time and money.

The team is experiences and has enough knowledge about/with testing.

Most of the testers are full-time testers.

Test functions and training

The test personnel consists of minimally a test coordinator and a number of testers.

The tasks and responsibilities are determined.

Test personnel is trained or are sufficiently experienced.

Reporting

The bugs are periodically reported.

Defects are reported and prioritized.

The progress of each test activity is periodically reported. Aspects are: execution time, what has been tested, what went wrong during execution and what should be tested.

Defect management

The various stadiums of the lifecycle of bugs are reported. These are:

- Unique identifier
- The person who reported the bug
- Priority
- Problem description
- Status

Testware management

The testware is internally managed by the testers and follows a described process.

Test process management

The test process is only planned and executed.

The test process is planned, executed, monitored, and adjusted as necessary.

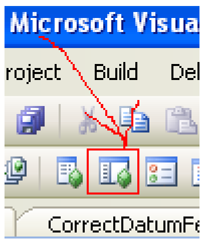
Appendix B Experiment Instructions

Instructions of experiment

This experiment is conducted to demonstrate that data-driven tests have lower maintenance costs than recorded webtests.

Extract the zip file (DataDriven.zip) to a directory. Open the solution TestEvaluation.sln in the sources directory. The TestEvaluation solution consists of two projects: Web, and WebTests. WebTests has two directories: Recorded and Data-Driven. In the recorded directory, the recorded tests are placed. And in the DataDriven directory the data-driven tests. A custom tool is used which is located in the Binaries directory (TestDriver.Win.exe).

Open the Test List Editor in Visual studio. See image below:



Run the tests in the test list 'Recorded'. All tests must pass.

1. Go to the Login.aspx.
2. Change the ID of the textbox txtUsername to txtUsername2. Make changes as necessary to build the solution.
3. Run the recorded webtests again.
4. Now all tests should fail. This is because the webtests POST a request to the server with a parameter txtUsername, which is unknown for the server. Therefore the txtUsername parameter should be changed to txtUsername2 in all webtests. Change the webtests as necessary to pass. Please record the time needed to perform the changes and the number of files changed.
5. Run the recorded tests again and now all tests should pass.
6. Run the data-driven tests. All tests should fail.

7. The data-driven test are not maintained with code, but with a custom tool created. This tool is located in the Binaries directory (TestDriver.Win.exe). Again these webtests fail, because they POST a wrong parameter to the server (txtUsername). This should be txtUsername2. Open the tool. Double-click on the Login page and change the Username clientid to txtUsername2. Run the data-driven tests again and these should pass. Please record the time needed to perform the changes and the number of files changed.

Thank you for participating in the experiment.