

Software testing: A case study of a small Norwegian software team

by
Lin Zhang

Thesis Supervisors:
Lars Line, Phillip Larsen
Asplan Viak AS

Jan Pettersen Nytnun, Andreas Prinz
University of Agder

This Master's Thesis is carried out as a part of the education at the University of Agder and is therefore approved as a part of this education. However, this does not imply that the University answers for the methods that are used or the conclusions that are drawn.

University of Agder
Faculty of Engineering and Science
Department of Information Communication Technology
Grimstad, May 2012

Keywords: software testing, automated testing, manual testing, case study

Abstract

As applications and software grow larger and more complex, software testing has been put in a core position to make sure that the defects or bugs will not break the software. However, it has been generally regarded that the traditional manual software testing simply cannot satisfy the requirement of testing more often and earlier also with better code coverage. Software quality might be poor with improper software testing which makes companies lose their money and even their reputation.

Bikube as a core software in Asplan Viak is being upgraded to a new version and requires being well tested during the development. Automated testing may offer some good features which we cannot gain from the current software testing strategy. So we need to figure out whether or not it is worthwhile to implement automated testing in the development of document management system Bikube 2011.

A case study has been conducted in this thesis to implement automated testing in all the architecture layers in Bikube 2011 and the results are verified. From the results evaluation, we found out that it is worthwhile to implement automated testing in Bikube 2011 because it can easily offer high code coverage and fast execution speed which give us more confidence on each stage of the product development life cycle. The results also shows that automated testing somehow can drive the design of code so as to improve the software quality. We can well control the cost of the automated testing by only automating the suitable tests and identify the breakeven point to maximize the benefits of the automated testing.

Preface

This thesis is the partial fulfilment of the two-year Master of Science program in Information and Communication Technology (ICT) at University of Agder, Norway. The period of conducting this thesis is from January to May 2012 and the workload equals to 30 ECTS. The thesis is given by the Norwegian company Asplan Viak AS.

First and foremost, I would like to thank my supervisors Lars Line and Phillip Larsen from Asplan Viak AS for giving me such an interesting task and all the help that they could offer. This thesis gave me the chance to work in a real company and work up-to-date. I would also like to thank Professor Andreas Prinz and Professor Jan Pettersen Nytnun at University of Agder. They have been highly supportive throughout the whole thesis period. Last but not least, I would like to thank Øystein Andreassen for giving me great support and ideas during the thesis period.

Contents

Abstract	2
Preface	3
1 Introduction.....	8
1.1 Background.....	8
1.2 Problem Definition.....	9
1.3 Methodology	10
1.4 Report Outline.....	11
2 Literature Review.....	12
2.1 Software Testing	12
2.1.1 Definition.....	12
2.1.2 The Quality of Software Testing	14
2.2 Manual Testing and Automated Testing	15
2.2.1 Definition.....	15
2.2.2 Comparison.....	16
2.2.3 The Observations of Automated Testing	17
2.2.4 A Pilot Study of Automated Testing.....	18
3 Analysing Bikube 2011.....	21
3.1 The Description of Existing Tests	21
3.2 The Problems of Existing Tests.....	23
4 Case Study.....	24
4.1 Hypotheses	24
4.2 Procedure Design	26
4.2.1 Preparation	26
4.2.2 Procedure	27
4.2.2.1 Training.....	28
4.2.2.2 Implementation	28
4.2.2.3 Maintenance	31
4.3 Implementing Automated Testing.....	32
4.3.1 Selection of Testing Objects.....	32
4.3.2 Selection of Testing Tools and Frameworks.....	33
4.3.2.1 Visual Studio 2010	33
4.3.2.2 Telerik Test Studio	33
4.3.2.3 Mspec	34
4.3.2.4 NDBUnit.....	35
4.3.2.5 Ncover	35
4.3.3 Experiment of Automated Testing	36
4.3.3.1 Automating User Interface Testing	36
4.3.3.2 Automating Service Layer Testing.....	43
4.3.3.3 Automating Data Access Layer Testing.....	45
4.4 Maintenance	48
4.5 Results Evaluation	49
4.5.1 Hard Features	50
4.5.2 Soft Features	50
5 Recommendations.....	57
6 Discussion	60
6.1 Verification of Hypotheses Results.....	60

6.2 Performance of Methodology	61
7 Conclusion and Future Work.....	62
Reference	63
Appendix A - Glossary	65
Appendix B - User Story	66
Appendix C - Sample Code.....	70
Appendix D - Log Trace	73
Appendix E - Gantt Chart.....	74

List of Figures

Figure 1.1: Relative cost to correct a defect in software development cycle [1].....	9
Figure 1.2: Methodology.....	10
Figure 2.1: Building quality to completion.....	13
Figure 2.2: Software testing process	13
Figure 3.1: Bikube 2011 web portal in Oppdrag domain.....	21
Figure 3.2: Seven parts in manual testing.....	22
Figure 3.3: Testing objects in Portal-visninger manual testing.....	22
Figure 3.4: Tests example in Medarbeidere	22
Figure 4.1: Structure of hypotheses.....	24
Figure 4.2: System environment.....	27
Figure 4.3: The procedure of implementing automated testing.....	28
Figure 4.4: Bikube 2011 architecture using Entity Framework 4.0.....	29
Figure 4.5: The way of organizing the tests.....	30
Figure 4.6: Setting up Ncover for the user interface.....	36
Figure 4.7: Outlook of the filters.....	36
Figure 4.8: Test cases in Telerik with requirement 1.....	38
Figure 4.9: Test cases in Telerik with requirement 2.....	39
Figure 4.10: Test cases in Telerik with requirement 3.....	40
Figure 4.11: Test Result of Requirement 1.....	41
Figure 4.12: Test Result of Requirement 2.....	41
Figure 4.13: Test Result of Requirement 3.....	42
Figure 4.14: Selecting geography in user interface.....	44
Figure 4.15: Test result of requirement 4.....	45
Figure 4.16: Tests with requirement 5.....	46
Figure 4.17: Test result of data access layer.....	47
Figure 4.18: Comparison of implementation and maintenance cost	52
Figure 4.19: Implementation of first test suite.....	52
Figure 4.20: The cost of implementing different test suites.....	53
Figure 4.21: Cost of first test suite with requirement 1.....	53
Figure 4.22: Cost of first test suite with requirement 2 (left) and 3 (right).....	54
Figure 4.23: Symbol coverage	55
Figure 4.24: Function coverage.....	55
Figure 4.25: Code coverage of test suite 5 “retrieving geography information”.....	55
Figure 4.26: Code coverage of test suite 6.....	56

List of Tables

Table 2.1: Critical metrics	15
Table 2.2: Description of features	16
Table 2.3: The comparison between manual testing and automated testing	17
Table 2.4: The necessities when investing automated testing	18
Table 2.5: The overall information of pilot project	19
Table 4.1: Test cases with requirement 1	37
Table 4.2: Test cases with requirement 2	38
Table 4.3: Test cases with requirement 3	39
Table 4.4: Bug/Problem in test cases	42
Table 4.5: Test cases with requirement 4	43
Table 4.6: Hypotheses and results	49

1 Introduction

With the newest release of Sharepoint by Microsoft, Asplan Viak AS decided to upgrade its Sharepoint-based document management platform Bikube from version 2007 to version 2011 in order to make use of the new features from Sharepoint.

When a software is under development, it requires testing to verify the quality of the software. During the development of Bikube 2007, only manual testing was used for locating the defects in the software. Asplan Viak would like to take a look at an opposite testing strategy-automated testing and to compare it with manual testing during the development of Bikube 2011 to find out whether automated testing is worthwhile to be implemented in Bikube 2011 or not .

However, since Asplan Viak have never used automated testing during the development before, this master thesis aims to offer both theoretical and practical information about the process of implementing automated testing in the development of Bikube 2011 as well as the performance results.

1.1 Background

Asplan Viak is an advisory company in the construction who runs projects for variety of customers from the whole Norway. Bikube 2011 is a document management platform designed for Asplan Viak which is used for sharing and storing project documents as well as customers' information among internal employees. Bikube 2011 offers a platform for employees to update, follow up and review different projects at any time and any place easily. Bikube 2011 consists of two parts, web portal and sharepoint room. All the projects can be found in the website portal and each project corresponds to a sharepoint room. The list of projects that have been visited and all the user-involved projects will be shown in Outlook.

One tiny mistake in Bikube may break the quality of the whole software. For instance, the previous stored documents are missing from Bikube, which will influence the process of related project. Not proper tested software may cause serious problems, for instance, the failure of health-related services [1], even the failure of a government system [2].

Poor software quality costs the United States' economy approximately \$59.5 billion every year (NIST 2002) [1]. Companies waste their investment and even lost reputation when software goes wrong. The function of the software is extremely important in all the industries to support the work.

So it's essential to test the software before it is released into the real life in order to make sure that there is no serious problem which may cause the software to fail. Due to the budget, time and different requirements, a suitable software testing method or strategy is necessary to find out the problems or defects in the software effectively; otherwise the software will be vulnerable after the installation.

Defects which are found and corrected earlier are less expensive than those are not found until released into production. The Figure 1.1 illustrates the relative cost to correct a defect during different development steps.

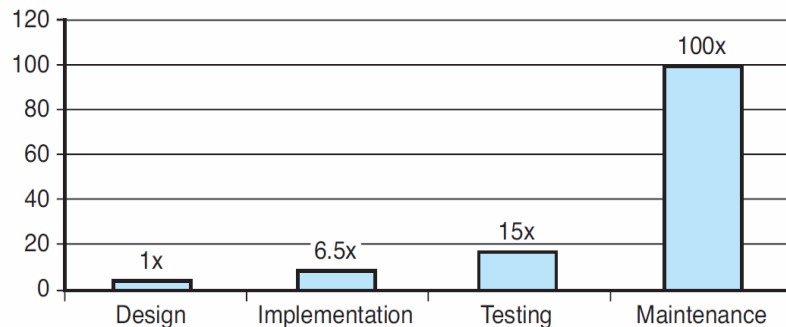


Figure 1.1: Relative cost to correct a defect in software development cycle [1]

Automated testing has been regarded as a useful strategy to detect defects in an earlier stage. As applications and software grow larger and more complex, manual testing simply cannot keep up with the speed of the requirement of testing more often and earlier. In fact, more and more develop teams start to implement automated testing in their development cycle. However, there existed quite a few problems of traditional test automation, for instance, knowledge barriers and management trouble. But with some mature technologies and tools such as Visual Studio, automated testing is moving beyond a luxury method and becoming a necessity for software.

Testing Bikube manually requires large amounts of repeated tests. In software development, a tiny change can often break “unrelated” code. Hence, developers are afraid to change important or complex code because of unpredicted consequences. It is time consuming to do tests manually in all the potential related parts. However, increased trust in the code when doing changes can be offered from automated testing. Once we set up automation scripts, we can run them just by clicking a button without any dedicated human resource.

1.2 Problem Definition

The quality of Bikube 2011 influences the business in Asplan Viak. The software touches hundreds of people directly or indirectly, either enabling them to finish their work effectively, or causing them frustration. Software testing is able to make sure that Bikube works properly. Automated testing as a testing strategy of test execution allows the tests being executed faster than manual testing. As some researches [16] point out, automated testing is feasible to be implemented in all the software. However, is automated testing

really suitable for Bikube 2011? This carried out the main problem in this thesis: whether or not it is worthwhile to implement automated testing in the development of the document management system Bikube 2011.

In order to solve the main problem, we need to implement the automated testing in Bikube 2011 and evaluate the performance of the automated testing by comparing it with its opposite test strategy - manual testing. The research questions are defined as follow.

RQ 1 How to implement automated testing in Bikube 2011?

RQ 2 How is the performance of automated testing comparing to manual testing?

RQ 3 Which benefits could automated testing give in the design and implementation of new functionality in Bikube 2011?

1.3 Methodology

We use the following methodology for solving the research questions and provide us an answer. The methodology is shown in the following figure.

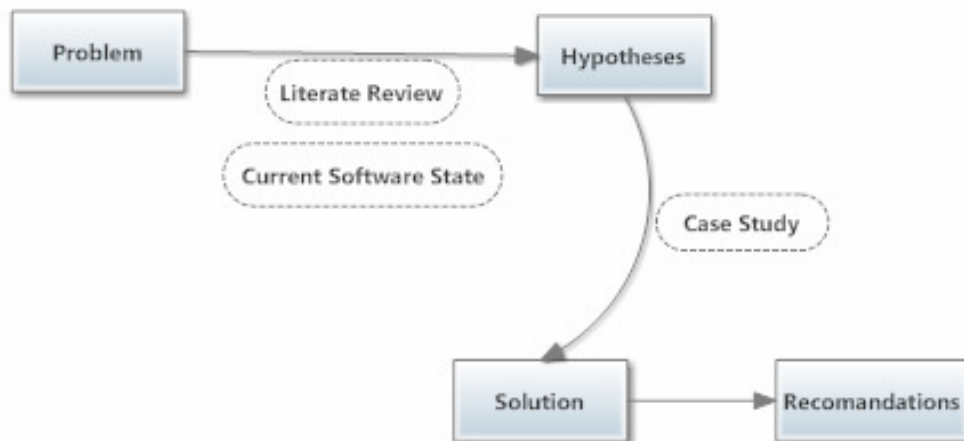


Figure 1.2: Methodology

Starting with the problem, we will come up with the hypotheses by checking the literature review and the current testing state of Bikube 2011. First of all, a literature study is essential to offer the definition of software automated testing and what has been done in that domain. The literature study should result in a synopsis of the methods, as well as a comparison of different methods as related to each other. We are interested in getting information regarding what kinds of software testing methods exist today. Especially, we will focus on the pros and cons of automated software testing and manual software testing. Secondly, we need to have an idea about the current state of Bikube 2011 in Asplan Viak. We will analyse the method that has already been used to implement software testing in the development of Bikube 2011.

After the hypotheses have been identified, we will verify them through a case study. Finally, we will analyse the results against the hypotheses based on the case study.

With the above steps, we are able to answer the main problem in this thesis and offer our recommendation.

1.4 Report Outline

In Chapter 1, the background and motivation of the master thesis are introduced. The problem and the methodology for solving the problem are also stated here.

Chapter 2 contains the definition of software testing, automated testing and manual testing. The comparison of automated testing and manual testing is also given here.

Chapter 3 analyses the current software testing state in Bikube 2011.

In Chapter 4, the case study is explained. The case study starts with identify the hypotheses. Then the procedure of the case study is designed. The implementation of automated testing will be carried out based on the procedure. Finally, the results of the case study will be evaluated.

In Chapter 5, we give some recommendations of implementing the automated testing based on the case study.

Chapter 6 discusses the whole thesis from hypotheses results and methodology performance.

Chapter 7 provides a brief summary of the problem, the solution and the results. Lastly, options for future work are given.

2 Literature Review

In this section, we look through the related work which has been done in the research and industry areas. When we develop a system, we usually follow the process of defining requirement specifications, implementing and verification. During the process, software testing plays a role to verify whether or not the implementation fulfils the requirement specification. Firstly, we explain which role software testing is playing in the software development life cycle as well as the methods and the strategies which are used to perform the software testing.

Secondly, we give the definition of the automated software testing and the manual software testing. We will list the pros and cons of these two testing types according to the previous studies. Some observations and pilot studies of automated testing will be made afterwards.

2.1 Software Testing

Software testing is an important and essential step in the software development life cycle (SDLC). Generally, software testing has different types which are used to fulfil specific requirements in SDLC. Software testing can be performed in several steps. Some of the steps are related to specific testing strategies.

2.1.1 Definition

Software testing determines whether or not a product works correctly and efficiently according to the requirements of end-users [4]. Software testing subjects a piece of code to both controlled and uncontrolled operating conditions in order to verify the output which should be in accordance with pre-defined conditions.

Through software testing, we are able to find problems in the system so that high quality software can be provided. It has been suggested that software testing should occupy 40% of a software development budget [4].

During the software development life cycle, software testing starts with unit testing, following by function testing/integration testing and ending with acceptance testing to build high quality software [1]. The different software testing types are defined below.

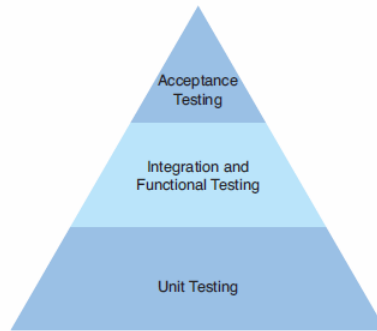


Figure 2.1: Building quality to completion

Unit Testing is a small piece of code which can be executed automatically to check the logical behaviour of methods or classes. A unit test can enable us to test just the code inside particular methods or functions and nothing else.

Integration Testing is the process to ensure that different components of the application work together. Integration testing fits between unit testing and acceptance testing. It also can be performed between targeted system and external applications that the system communicates with.

Functional Testing is testing performed to validate specific requirements and is appropriate for a continuous testing process in which each function is tested as it is completed and retested as new functions are integrated to discover regression bugs.

Acceptance Testing performs the final functional test before software is released to the market or delivery to the end user in order to check whether the software is fully functional and meets the customer's needs as well as expectations.

Now we look into the detail of software testing, the software testing consists of Analysing Requirement, Tests Designing, Tests Execution, Problems Identifying, for validating functionality (Figure 2.1). Problem fixing is the next step after a problem has been detected. The problem will be reported to developers. So this step is more like developers' job which is not focused in software testing.

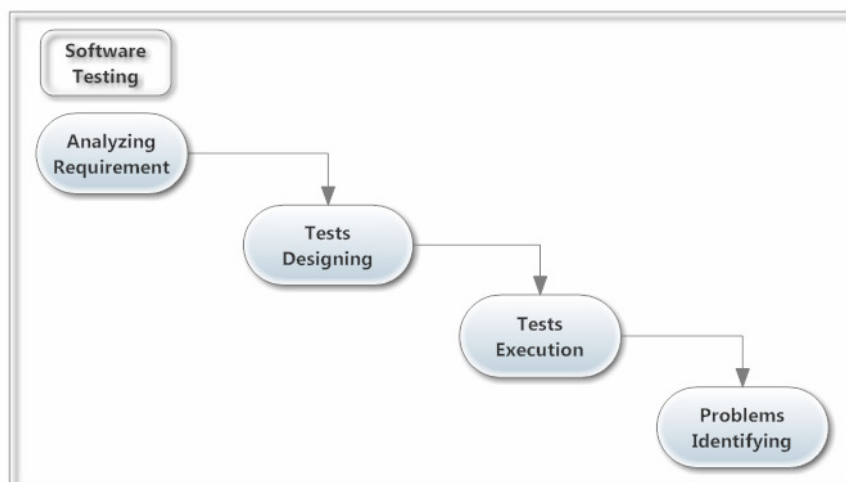


Figure 2.2: Software testing process

Detaching bugs and errors from the code relies on different testing strategies during software testing process. The sum collection of different strategies assures that the software meets the targeted quality.

➤ How tester executes the test during **Tests Execution**?

Basic methods of executing tests include Manual Testing and Automated Testing. The main difference between these two types is whether a testing tool is used or not under the Test Execution part. More details will be described in chapter 2.2.

➤ How tester treats the test object during **Tests Designing**?

White box testing and black box testing are two common strategies in tests designing. These two approaches are used to describe the point of view that a tester takes when designing tests. Black Box testing is carried out against the functional specifications in order to check any abnormal system behavior. White box testing deals with program's internal logic and code structure [11]. There also exists grey box which is a mix of the above two boxes.

Software testing is a complex process, a proper combining of different software test types, and strategies can perform a group of test activities that are aimed at testing a component or system focused on a specific test objective of finding defects in that particular component.

2.1.2 The Quality of Software Testing

The main purpose of conducting software testing is to improve software quality. Software metrics are helpful to measure the software testing either when it is under development or after the system is ready for use. The metrics helps in judging the quality of software testing in order to value how well the testing has been done.

The fact is that we can measure almost everything; metrics can become overwhelmed easily. It is important to measure the critical parts which actually are useful. We should select specific measurement according to our unique needs. We listed several useful measures [5] that are widely used in software testing.

<i>Quality Scope</i>	<i>Metric</i>	<i>Description</i>
Requirement	Function Test Coverage	The number of test requirements that are covered by test cases that were executed against the software divide by the total number of test requirements.
Tests Design	Code Coverage	It describes the percentage of which the source code of a program has been tested. The goal of every team may be different, but strive for approximately 70% to 80% code coverage.
Tests Execution	Execution time	The time it takes to run the tests.
Problem Identifying	Total bug count	This number simply puts all the other numbers in perspective, such as the percentage of bugs found in a given step or iteration

Table 2.1: Critical metrics

2.2 Manual Testing and Automated Testing

In this thesis, we focus on the testing strategies in the ‘test execution’ step. So, the definition of manual testing and automated testing is introduced. Then the comparison between these two testing strategies will be given based on the previous studies. Some observations and a pilot study of automated testing will be given to have the idea about how to implement the automated testing.

2.2.1 Definition

Manual testing is the oldest and most widely used software testing strategy which requires a tester to perform test execution manually on the target software without the help of any test automation tools. It is a laborious and slow activity that requires the testers to possess a certain set of actions by themselves, for example, information submission, user interfaces navigation or attempt to hack the software or database etc. [11]. Conducting good manual testing depends very much on the qualification of the testers since human can easily get tired and unfocused.

Automated testing uses test automation tools to control the test execution automatically. It is the process of writing a computer program to do testing that would otherwise need to be

done manually. The tests will pass when the actual outcomes equal to the predicted outcomes [6]. Automated testing helps us to eliminate the influence of human interruption and minimize the variability of results [11].

2.2.2 Comparison

Based on [7], the differences between manual testing and automated testing are compared in terms of test rerun ability, result visibility, test area, process fragility and human interruption. The description of these features is in table 2.2.

<i>Features</i>	<i>Description</i>
Test rerun ability	Whether the tests can be used more than once without any extra effort or not. For the same function/module, do we need extra effort to run the test after the first run?
Result visibility	Whether or not the test result (Pass or Fail) can be generated automatically after the test is finished?
Test area	The areas which are suitable for using manual testing or automated testing.
Process fragility	Whether or not the test process is easy to be interrupted by human or other elements?
Human interruption	How human can interrupt the testing process?

Table 2.2: Description of features

Manual testing and automated testing are compared in the following table in order to give us the idea about their pros and cons.

	<i>Manual testing</i>	<i>Automated testing</i>
Test rerun ability	-Low Tests can NOT be rerun automatically, every time running the test, the same effort is required.	-High Tests can be rerun without any additional efforts. After the test is written, it can be run automatically without any extra effort as long as the target component does not change.
Result visibility	-Low No result will be generated.	-High The test result will be returned with detailed.
Test area	-Suitable for integration test and usability testing -Complex scenarios/cases using risk based approach	-Suitable for testing a unit (class/method), a module, a system -Generic functions that will not change much for regression test -Regression testing
Process fragility	-Yes Easy to be interrupted by human, then test process will be stopped.	-No No interruption since the test is run by machine automatically
Human interruption	-The process might be stopped when human's attention is distracted	-The quality of the testing scripts is fully influenced by human. The test may not be passed due to the mistakes in scripts.

Table 2.3: The comparison between manual testing and automated testing

According to the above table, it would be nice to automate everything to gain the advantages offered by automatic testing, even hiring a fully qualified tester to do this job. However, it is difficult to achieve the full automation. For instance [8], some GUI tasks which are easy for people are hard for computers because it is very hard to automatically notice all categories of significant problems while ignoring the insignificant problems to verifying the test results.

2.2.3 The Observations of Automated Testing

Although automated testing is widely believed to be an efficient method to improve software quality and draws lots of attention in current testing research [9], automation is still not well-off in industry today [10].

Persson and Yilmaztürk [11] note that the establishment of automated testing is a high risk and a high investment project which makes companies shrink back at the sight of

implementing automated testing. The investment includes implementation costs, maintenance costs, and training costs [12].

Investment	Description
Implementation costs	Direct investment costs such as time and human resources.
Training costs	The cost that requires for tester to start performing automatic testing.
Maintenance costs	The cost that it takes to maintain the automatic test cases when some changes happen in the corresponding code.

Table 2.4: The necessities when investing automated testing

➤ **Implementation costs**

It is expensive to go through the whole implementation procedure of designing and executing test scripts. Fixed scripts may only be suitable for limited test scenarios. They can be reused, but only when the system has not big modification. As shown in a survey of Australia in 2004 [13], about 50% of the respondents reported that cost was the major barrier to use automation for software testing.

➤ **Training costs**

Training costs usually is huge when the team transfer from manual testing to automated testing because team members have to change their working practices. Instead of performing manual testing daily, testers' tasks may be completely transformed to maintaining automated test cases.

Testers usually do not have related programming skills such as a tool or a language which are required to develop scripts. Test automation requires that members must know not only programming skills but also test requirements. But there is always a risk that automation developers might misunderstand test requirements just like application developers often misunderstand business requirements [14].

➤ **Maintenance costs**

There is a connection between implementation costs and maintenance costs [15]. During the implementation stage, if the system is designed to only require minimum maintenance, then the implementation costs will increase while the maintenance costs decrease, and vice versa. The balance of these costs should be focused, otherwise if the maintenance is ignored, updating an entire automated tests can cost as much, or even more than the cost of performing all the tests manually [15]. Maintenance costs will be really expensive when big changes come in the software.

2.2.4 A Pilot Study of Automated Testing

Many case studies [16] have been conducted to explain how a non-automatic testing team transfer into an automatic testing team and the lessons that need to be learned during the process. Considering our project and team size, it is smart to look through some projects

with similar background. We will take a case study (Table 2.5) as pilot to know what others have learned, and our own automated testing is more likely to be successful.

Application domain	Insurance
Application size	1,000,000 LOC
Location	Denmark
Team size	3
Software layer	Business components
Project length	1 year
Automation Tool type	Commercial
Date	2007-2008
Success	Yes
Still breathing?	Yes
Requirement of specification	Yes

Table 2.5: The overall information of pilot project

There are four main reasons for the pilot company to do automated testing instead of manual testing.

1. The IT systems became more and more complex.
2. There were not enough resources to do an adequate regression test on all the releases and also hot fixes. This made the regression test very time consuming.
3. The team wanted to make sure that exactly the same tests were executed every time. The manual operation is not repeatable and associated with a high risk of mistakes.

Before applying automated testing in the system, there were only some small functional tests. It was actually just a sanity test of the application. This little regression was too risky for the business, so a real regression test was needed.

When the team decide to conduct automated testing instead of manual testing, they divided the project into two time periods. The first period was a learning process to understand the task and the tools. They needed to achieve the common knowledge to handle project properly later. They selected the capture/replayor BPT automation tool with keyword-driven automated components to conduct automated testing. The second period was the practical implementation; they tested for real and automated a set of real tests for the application. The functionality that they started with to automate is simple and stable so there they would not get stuck or experience too many changes.

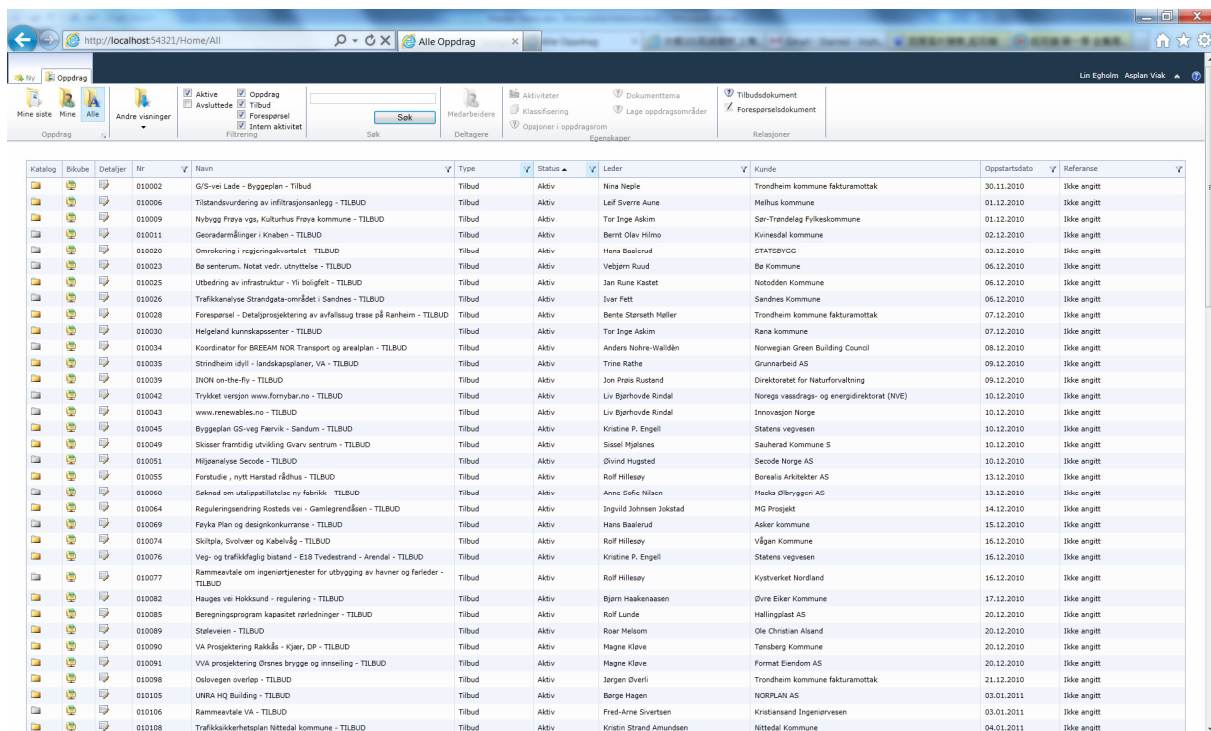
The calculation suggested the team would break even on the investment in a reasonable time. The breakeven point is estimated to be 1 year.

Some good points that have been made in this pilot study:

- Cooperate with developers, get known with the system through developers.
- Ensure common understanding and knowledge.
- Understand the whole business application to ensure the right structure and size of test cases.
- “Keep it simple.”

3 Analysing Bikube 2011

Bikube 2011 is a website portal for storing projects and each project has a corresponding sharepoint room site. In this thesis, we focus on testing the portal part automatically. The portal website can be divided into two main domains: *Ny* and *Oppdrag*. The following graph is the screenshot of the Bikube 2011 web portal part in *Oppdrag* domain. All the projects are specified with name, type, status, leader, customer and start date. More details can be found after we open the correspond sharepoint room.



Katalog	Bikube	Detaljer	Nr	Navn	Type	Status	Leder	Kunde	Oppstartsdato	Referanse
			010002	G/S-vei Lade - Byggesplan - TILBUD	Tilbud	Aktiv	Nina Tople	Trondheim kommune fakturamottak	30.11.2010	Ikke angitt
			010006	Tilstandsundersøking av infiltrasjonsanlegg - TILBUD	Tilbud	Aktiv	Leif Sverre Aune	Molde kommune	01.12.2010	Ikke angitt
			010009	Nybygg Frøya vgs, Kulturhus Frøya kommune - TILBUD	Tilbud	Aktiv	Tor Inge Askim	Sør-Trøndelag Fylkeskommune	01.12.2010	Ikke angitt
			010011	Geordamølinger i Klaben - TILBUD	Tilbud	Aktiv	Bernt Olav Hilmo	Kvinnesdal kommune	02.12.2010	Ikke angitt
			010020	Omstrukturering i regjeringskvartellet - TILBUD	Tilbud	Aktiv	Hans Baalerud	STATSBUD	03.12.2010	Ikke angitt
			010023	Ba serteron, Nettet vedr. utbytte - TILBUD	Tilbud	Aktiv	Vebjørn Roud	Bir kommune	06.12.2010	Ikke angitt
			010025	Utbedring av infrastruktur - Yt boligfelt - TILBUD	Tilbud	Aktiv	Jan Rune Kaaet	Notodden kommune	06.12.2010	Ikke angitt
			010026	Trafikkanalyse Strandgata-området i Sandnes - TILBUD	Tilbud	Aktiv	Ivar Fett	Sandnes kommune	06.12.2010	Ikke angitt
			010028	Forespørsel - Detaljprosjektering av avfallsog tross på Ranheim - TILBUD	Tilbud	Aktiv	Bente Størseth Møller	Trondheim kommune fakturamottak	07.12.2010	Ikke angitt
			010030	Heljeland kunnskapsenter - TILBUD	Tilbud	Aktiv	Tor Inge Askim	Rana kommune	07.12.2010	Ikke angitt
			010034	Koordinator for BREEM NOR Transport og arealplan - TILBUD	Tilbud	Aktiv	Anders Hohre-Walden	Norwegian Green Building Council	08.12.2010	Ikke angitt
			010035	Strindheim idyll - landskapsplaner, VA - TILBUD	Tilbud	Aktiv	Trine Rathe	Grunnarbeid AS	09.12.2010	Ikke angitt
			010039	BNOR on-the-fly - TILBUD	Tilbud	Aktiv	Jon Preis Rustand	Direktoratet for Naturforvaltning	09.12.2010	Ikke angitt
			010042	Trykket versjon www.fornybar.no - TILBUD	Tilbud	Aktiv	Liv Bjørthovde Rindal	Noregs vassdrags- og energidirektorat (NVE)	10.12.2010	Ikke angitt
			010043	www.renewables.no - TILBUD	Tilbud	Aktiv	Liv Bjørthovde Rindal	Innovasjon Norge	10.12.2010	Ikke angitt
			010045	Byggesplan GS-veg Færevik - Sandum - TILBUD	Tilbud	Aktiv	Kristine P. Engell	Statens vegvesen	10.12.2010	Ikke angitt
			010049	Skisser framtidig utvikling Gvarv sentrum - TILBUD	Tilbud	Aktiv	Sissel Mjøltnes	Sauherad Kommune S	10.12.2010	Ikke angitt
			010051	Miljøanalyse Secode - TILBUD	Tilbud	Aktiv	Olvind Hugstved	Secode Norge AS	10.12.2010	Ikke angitt
			010055	Forstudie, nytt Harstad rådhus - TILBUD	Tilbud	Aktiv	Rolf Hillsey	Borealis Arkitekter AS	13.12.2010	Ikke angitt
			010060	Sekundær utvipsattilrette ny fabrikk - TILBUD	Tilbud	Aktiv	Anne Sofie Nilan	Hekla Offbygger AS	13.12.2010	Ikke angitt
			010064	Reguleringsendring Rosteds vei - Gamlegrøndøsen - TILBUD	Tilbud	Aktiv	Ingvald Johnsen Jakstad	MG Prosjekt	14.12.2010	Ikke angitt
			010069	Fåvika Plan og designkonkurranse - TILBUD	Tilbud	Aktiv	Hans Baalerud	Asker kommune	15.12.2010	Ikke angitt
			010074	Skiltpa, Svolvær og Kabelvåg - TILBUD	Tilbud	Aktiv	Rolf Hillsey	Vågan kommune	16.12.2010	Ikke angitt
			010076	Veg- og trafikkfaglig bistand - E18 Trvedestrand - Arendal - TILBUD	Tilbud	Aktiv	Kristine P. Engell	Statens vegvesen	16.12.2010	Ikke angitt
			010077	Rammesvite om ingeniørtjenester for utbygging av havner og farleder - TILBUD	Tilbud	Aktiv	Rolf Hillsey	Kystverket Nordland	16.12.2010	Ikke angitt
			010082	Hauges vei Høkkund - regulering - TILBUD	Tilbud	Aktiv	Bjørn Haakenaasen	Dvre Elker kommune	17.12.2010	Ikke angitt
			010085	Beregningsprogram kapasitet rørdninger - TILBUD	Tilbud	Aktiv	Rolf Lund	Hallingdal AS	20.12.2010	Ikke angitt
			010089	Støveveien - TILBUD	Tilbud	Aktiv	Roar Melsom	Ole Christian Alsand	20.12.2010	Ikke angitt
			010090	VA Prosjektering Rakkås - Kjør, DP - TILBUD	Tilbud	Aktiv	Hagne Klave	Tonsberg kommune	20.12.2010	Ikke angitt
			010091	VVA prosjektering Østres brygge og innseiling - TILBUD	Tilbud	Aktiv	Hagne Klave	Format Eiendom AS	20.12.2010	Ikke angitt
			010098	Oslovegen overlop - TILBUD	Tilbud	Aktiv	Jørgen Overli	Trondheim kommune fakturamottak	21.12.2010	Ikke angitt
			010105	UNRA HQ Building - TILBUD	Tilbud	Aktiv	Børge Hagen	NORPLAN AS	03.01.2011	Ikke angitt
			010106	Rammeavtale VA - TILBUD	Tilbud	Aktiv	Fred-Arne Svertsen	Kristiansand Ingeniørvesen	03.01.2011	Ikke angitt
			010108	Trafikksikkerhetsplan Nittedal kommune - TILBUD	Tilbud	Aktiv	Kristin Strand Amundsen	Nittedal kommune	04.01.2011	Ikke angitt

Figure 3.1: Bikube 2011 web portal in *Oppdrag* domain

3.1 The Description of Existing Tests

Some manual testing has already been done in the Bikube 2011 portal part in user interface level. All the tests have been stored in the tables in OneNote. The manual testing has been divided into seven parts: *Portal-visninger*, *Medarbeidere*, *Egenskaper*, *Ny*, *Oppdragsrom*, *Tjenester* and *Add-in* (Figure 3.2). The corresponding English are Portal views, Employees, Properties, New, Task Room, Services and add-in. We are not going to introduce all the tests here but give two examples of the tests in the *Portal-visninger* and *Medarbeidere* parts which are related to the web portal part in order to show how the manual testing is conducted and organized.



Figure 3.2: Seven parts in manual testing

In the *Portal-visninger* part, the aim of testing is that the projects are well sorted and the filter work well when we search the projects by *Nr*, *Navn*, *Type*, *Status*, *Leder*, *Oppstartsdato* and *referanse* in *Mine siste*, *Mine* and *Alle* areas. The related panes, icons and areas that are involved in the tests are shown in the following screenshot.

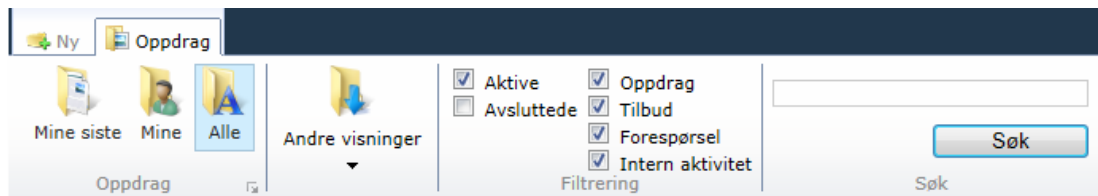


Figure 3.3: Testing objects in *Portal-visninger* manual testing

In the *Medarbeidere* part, the function of the right modification of project members (changing title, deleting members or adding new members) when a project has been created are tested.

The manual tests are designed in a way that consists of *Punkt* (testing area), *Beskrivelse* (description) and *Resultat* (result). An example can be seen in the following screenshot.

Punkt	Beskrivelse	Resultat	
M1 Tilgang/ Aktivering	Aktiveres ved Medarbeiderknappen på <u>ribbon</u> . Tilgjengelig når et oppdrag (ikke forespørsel) er valgt	Ok	
M1 Oppdragsleder	Kan skifte oppdragsleder, velge blant deltagere Kan melde inn alle medarbeidere Sjekk at ikke sluttede medarbeidere vises Kan slette alle eks seg selv Aktiviteter vises (de som er definert i oppdraget) Roller vises (Medarbeider, <u>Kvalitetssikrer</u>) Sjekk at medarbeidere synkes med oppdragsrom	Ok Ok Ok Ok Ok Ok Ok	
M2 Ansatt	Kan kun melde inn /slette seg selv	Ok	
M3 Varsling	Varsling blir sendt ved påmelding Innhold i varslings ok	Ok Link er feil	

Figure 3.4: Tests example in *Medarbeidere*

After the manual tests have been designed, the testers conduct the tests based on the description and marked the result as OK when the tests pass while specify the problem if the tests fail.

3.2 The Problems of Existing Tests

The manual tests seem to be thorough, but there exist several problems in the current manual tests.

1) The range of the tests

The manual testing which has already been conducted in Bikube 2011 is only at the user interface level. In other words, all the tests are acceptance testing which use black box strategy in order to check the final product, not the structure inside. The range of the current manual tests has limitation.

2) The organization of the tests

Figure 3.4 shows the way of organizing the manual tests. We can see that the description of each test is more like the expectation of test results rather than the actual process of conducting tests. There are no detailed steps about how to execute each test. Once the tester find defects, it is difficult for the developer to perform the exact same test for generating the same defects when they try to fix them.

The information in each test is also over-size. Once some features are changed in Bikube, we need to do regression testing to make sure that the changes will not introduce new defects. But with the current manual tests, we have to spend much effort to run the whole test while only some parts of that test are related to the regression testing. We have to waste our time to run irrelevant test because they are over-size.

3) The quality of the tests

When developer is designing a function, they would like to run the same test over and over until the function is totally fulfilled. However, we have no choice but run the exactly same manual test again and again with same efforts each time.

Furthermore, we do not know how many codes and which requirements that the manual tests have been covered. The lack of such information cannot give us the confidence that the quality of the manual tests is high.

4 Case Study

In the following chapter, firstly we will make the hypotheses based on the above literature review and the state of current software testing in Bikube. Then we design a reasonable case study procedure for conducting automated testing in Bikube and then perform it for real. The case study results will be evaluated in the end of this chapter.

4.1 Hypotheses

The hypotheses are divided into the hard features for evaluating the process of automated testing implementation and the soft features for evaluating how well the automated testing implementation is.

Hard features evaluate the physical parts in the case study which include organizing tests, implementation procedure and tools. Hard features focus on the process of setting up the automated testing in Bikube 2011. Soft features focus on the performance and the influence of automated testing compared with manual testing. We evaluate how team size and software layers can influence automated testing. We also evaluate soft features from performance side which includes cost, code coverage and driven-design. The details of each hypothesis are explained below.

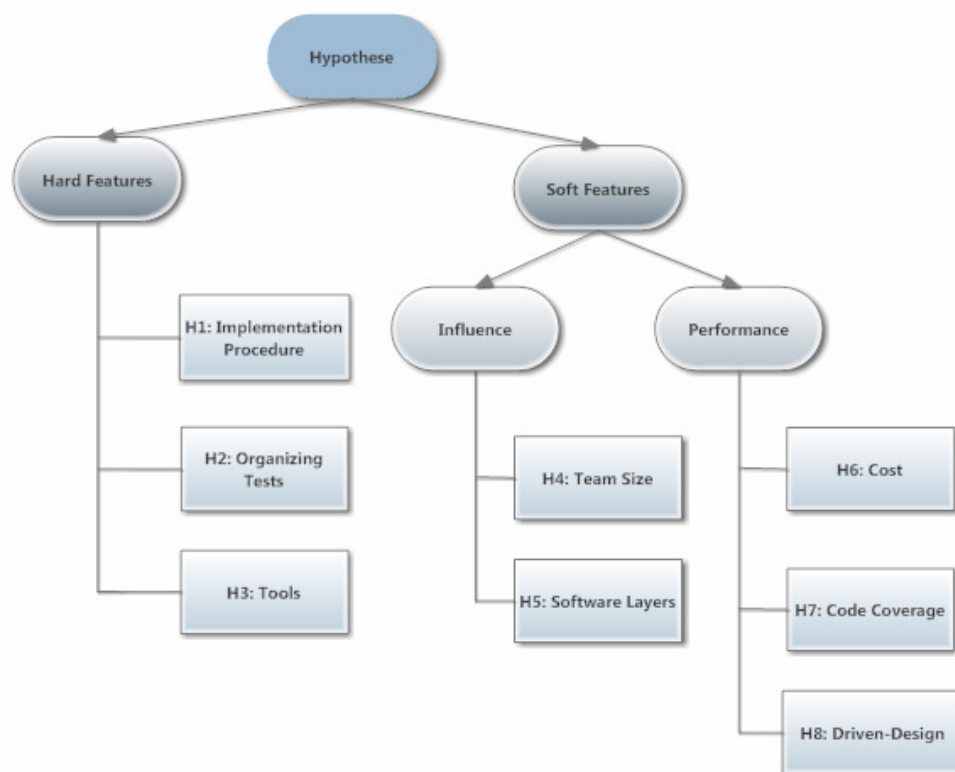


Figure 4.1: Structure of hypotheses

H1: The automated testing in Bikube 2011 can be implemented with the procedure of training, implementing and maintenance.

It has been suggested in chapter 2.2.4 that the cost of automated testing is consisted of training, implementing and maintenance. So we assume training, implementing and maintenance are the essential steps when a manual testing team wants to implement automated testing .

H2: The way of organizing manual tests and automated tests is different.

From chapter 3, we noticed the problem of organizing the current manual tests, so we assume that the way of organizing the automated tests should be different from the way of organizing the manual tests in order to avoid the problems.

H3: The combination of different automation tools can result better automated testing.

From the pilot study, we noticed that one capture automation tool was used for capture/replay function. We assume that more than one automation tool will be used to fulfil different functions during the implementation so that the automated testing can be well performed.

H4: The cooperation in the small developing team is more efficient than that in the big team.

From the case studies [16], we can see that both small develop team (2-5 persons) and large develop team can successfully transfer from manual testing into automated testing. So we assume that the team size will not influence the success of automated testing. Furthermore, the pilot study in chapter 2.2.4 suggested that working in a small developing team made people cooperate closer.

H5: Automated testing performs better than manual testing in all the layers.

From the analysing of current testing in Bikube, we noticed that manual testing is only used at the user interface layer. And in the pilot study, automated testing is also implemented at the user interface layer. This indicated that both manual testing and automated testing can be implemented at the user interface layer.

Since manual testing requires human execution (automated testing requires machine execution), an accessible interface is needed for manual testing. However, in other layers, there is no directly interface to be used. So we assume that automated testing performs better than manual testing in all the layers.

H6: Automated testing costs more than manual testing.

The unit of cost is time, for example, day, hour, minute. It has been generally regarded that using automated testing is more expensive than using manual testing even automated testing can run 24 hours a day without any human supervision. Thus, hypothesis H6 suggests a comparison between automated testing and manual testing in terms of cost.

H7: It is easier to increase the code coverage of automated testing than that of manual testing.

The experiments in [17] and [18] indicate that automated testing can offer higher code coverage comparing with manual testing. So we assume that the code coverage of automated testing can be easily increased than that of manual testing.

H8: Automated testing can drive the design of new functionality in Bikube 2011 while manual testing cannot give any benefit.

According to the usage of automated testing in test driven development, the develop team will implement the tests instead of requirement specification to design different functions. So we assume that automated testing can be beneficial in driving design.

4.2 Procedure Design

When a develop team wants to transfer from manual testing into automated testing, there are several steps needed to be done. We need to firstly get the copy of Bikube 2011 and secondly to design the implementation procedure with the experience from other studies. Basically, we set up the environment and configure the software for the case study. After the preparation work, we design the procedure of implementing automated testing in Bikube 2011.

4.2.1 Preparation

Bikube is the basic platform in our case study; all the experiments will be conducted on it. However it is not wise to take the real system which will be deployed later, as our case study platform. The reason is obvious. We might write wrong code, change things that we are not supposed to change or even delete some essential parts in the system. So we need to configure a replication of Bikube 2011 in local machine for the case study instead of accessing to the real database server and web server.

The following screenshot is the system information in the local machine. The system version is Windows 7 Enterprise with Intel Core i7 processor and 4GB RAM. With our experience, this system is good enough to conduct the case study, but when there is lots of information needed to be processed, the response time might be slow down.

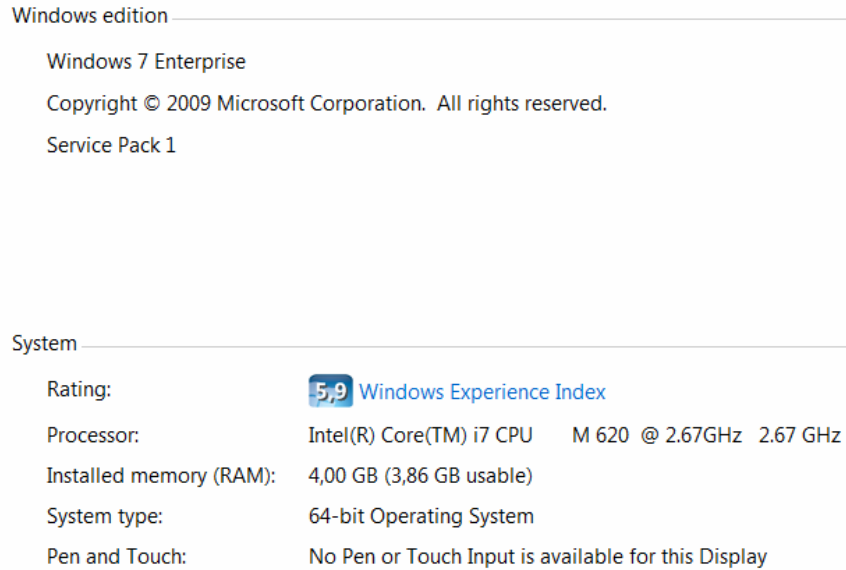


Figure 4.2: System environment

Then we need to install a local database server in our machine to store all the project information due to the absence of a remote database server. We choose SQL Server Developer Edition 2008 R2 as our database server since this is the version that has been used in the real server. Then we restore the database locally from the Bikube solution. We encountered some problems during installation and solved them by changing the owner and some settings. We ignore the details of how to fix the problem because the focus of this thesis is software testing.

Afterwards, since the solution uses ASP.NET MVC 3 framework to build scalable, standards-based web applications, we also need to install MVC 3 to enable related function.

After we set up the basic environment, we need to publish Bikube so that people can access to it. At beginning, we planned to publish the Bikube portal by IIS as how real Bikube is been published. But then we found the web config file is too complicated to be set ready. So we choose a simple way to publish the Bikube portal by debugging the whole solution in Visual Studio. And we get the website address <http://localhost:54321/>.

4.2.2 Procedure

The procedure of implementing automated testing will be defined in this chapter. When traditional manual testing teams want to implement automated testing, there are three steps that we can follow: training, implementation and maintenance. In the training part, we learn all the necessary elements for performing the implementation and the maintenance. Under the implementation part, we firstly find out what we should automate, following by how to automate and which tools to use. In the last step, we figure out what we should maintain and how to maintain.

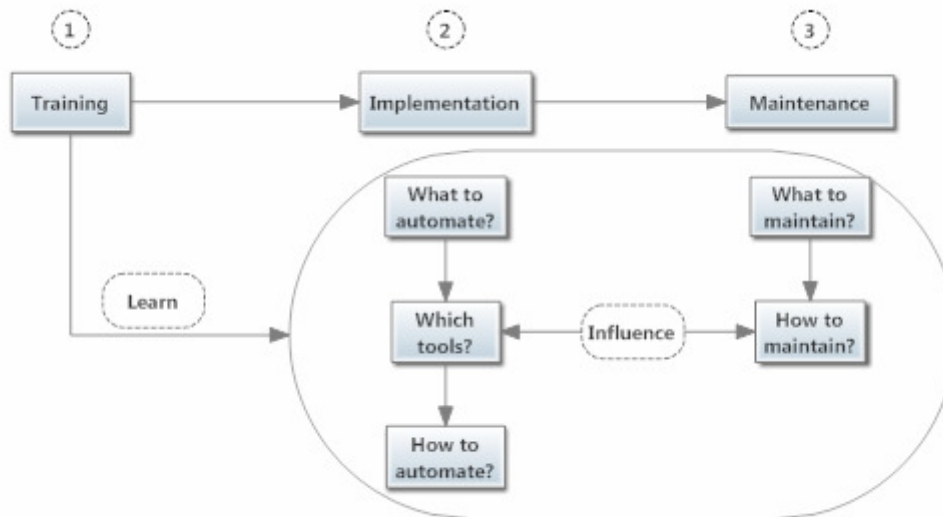


Figure 4.3: The procedure of implementing automated testing

4.2.2.1 Training

Training can be time consuming and high cost because team members might need to learn how to program or get used to the automation tools as well as the target software. So we start from reading the basic software testing books and videos several months before the master project started in order to gain necessary information. Everything we learn is the basic of performing the later steps. So we need to focus on the related knowledge of performing implementation and maintenance which is illustrated in figure 4.3.

4.2.2.2 Implementation

After self-training is finished, we move towards to the implementation. There are three aspects which are related to the implementation. We need to figure out what we need to do in these aspects.

1) What to automate?

Almost everything in software can be tested. However, we cannot automate everything because we cannot afford it. Tests are only good if they are likely to find bugs. We'd like to ask ourselves several questions to choose the right tests.

- Do the automated tests provide benefits in the future?
- Is the tested feature used frequently?
- How often is the feature likely to be changed in the future?
- Is the test likely to find regression bugs?

From the testers' view, we start by automating regression tests. These tests are run frequently and rerun in order to make sure that things which used to work are still working after new changes have been injected. Regression tests are tedious and being highly repeated in the same scenario test.

From the developers' view, the areas of the software that they are most worried about (core or easy to be broken parts) are also wise to be tested automatically. But for the parts that have a high degree of likelihood to change, they are not good candidates because too much maintenance work would need to be done later.

From the stockholders' view, the decision of whether or not to automate the test process should be driven by Return on Investment (ROI) considerations.

The architecture of Bikube 2011 is divided into 3 layers (Figure 4.4): Client/Front End, Service Layer and Data Access Layer. Client/Front End is also known as user interface. So we can do the main implementation parts based on software structure layers and select specific tests from each layer.

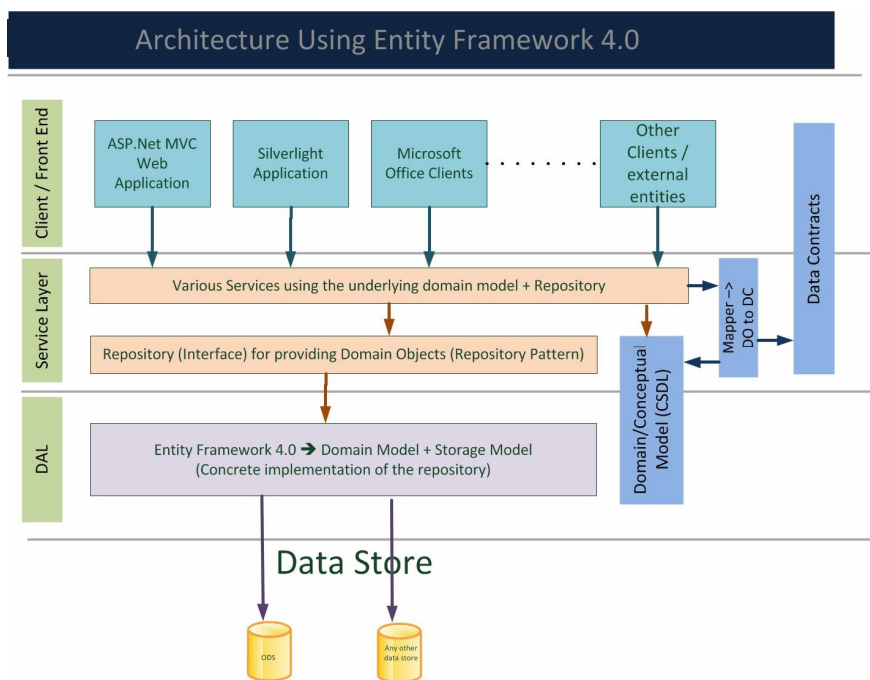


Figure 4.4: Bikube 2011 architecture using Entity Framework 4.0

2) Which tools to select?

We select the tools based on how we want to conduct the automated testing. The use of right testing tools can make testing easier, more productive and more effective. We should look from two sides:

- Activities or tasks (like test design, test execution and test evaluation) in the software testing fulfilled by tools.
- Specific functions performed by tools like capture, playback and code coverage.

The above two sides are required by almost all the software when automated testing needs to be implemented. There are of course some other aspects that we might look into like generating test data automatically.

There exist huge amounts of tools in the market; it is important and complicated to select the tool. However, selecting tools elaborately is essential.

The idea of software test automation is to let the computer simulate what a tester does when he manually runs a test on the target application. In manual testing, user interface testing is straightforward and relatively easy since all the actions are visible for the testers. Sometimes large amounts of test data need to be added to the same test scenario by hand. Too many repeatable tests are tiresome for human but not for machine. We need to have a tool to recognize the UI objects and click actions to perform the automated testing. In the other layers, we need to use automated frameworks to simplify the code writing. The selection of frameworks and tools will be discussed in the chapter 4.3.

3) How to automate?

We need to well document the tests. For doing this, we will design the tests based on the requirement so that we can make sure that the tests truly test the important parts against the software. So it is necessary to get developers, testers, preferable customers together to specify requirements specification clearly.

Test Suite: is a collection of test cases that are intended to be used to test a software program. A test suite connects with one requirement.

Test Case: includes a sequence of test steps.

Test Steps: provides the detailed semantics on how each test case is to be validated and ultimately completed

Based on requirements specification of Bikube 2011, we organize tests by creating test suits and test cases to reduce the size of tests. Meanwhile we define all the tests step by step in order to make them repeatable. Thus we are able to divide each test as small and detail as possible. The way to organize the test suits and test cases are tree-model which is illustrated below.

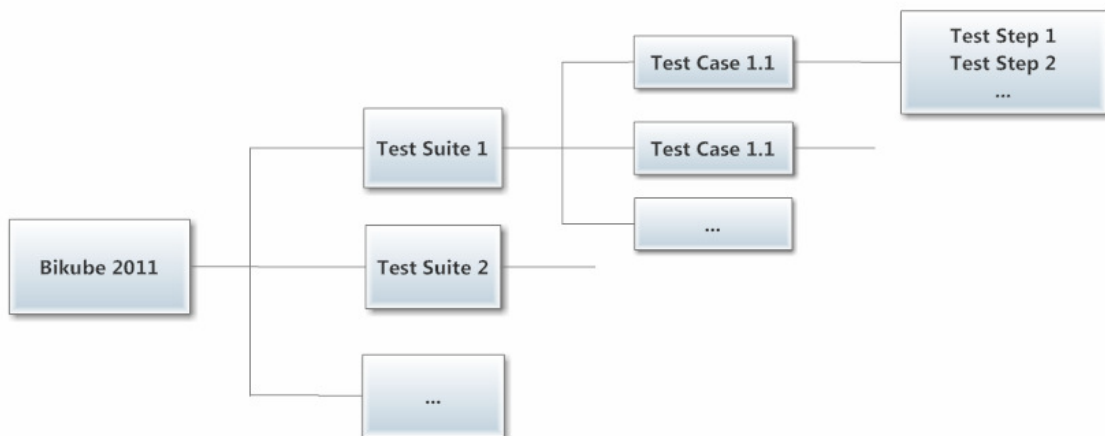


Figure 4.5: The way of organizing the tests

4.2.2.3 Maintenance

Maintenance is the last step in the procedure. We shall ask ourselves two questions about maintenance.

1) What to maintain?

Automated tests can be used again and again only in the condition that no changes has been made in the software. Once some changes are made in the software, the maintenance is required and meanwhile it can be highly time consuming. We should focus on the software parts that have been modified and try to update the tests as soon as possible.

2) How to maintain?

As mentioned in chapter 2.2.3, there exists a close relationship between maintaining and implementing. It has been illustrated in Figure 4.3. So when we design the tests, we try to make them require the minimum efforts of the maintenance.

The maintenance of automated tests can be really easy if the tests are well designed and organized during the implementation. We need to create the tests from the smallest level of business value and use them to build more complex concepts. The tests need to be as small as possible but with relevant requirement. When the target tests are small, we can easily locate the specific function/part that we need to fix.

When the system's user interface changes, it requires a lot of efforts to update the automated test scripts so that they can still test the program accurately. When the user interface language changes (such as from Norwegian to English), it is hard to revise the scripts so that they can still test the program accurately. Just using capture/playback may be effective, but it is not wise to use capture/playback to create all the test suite. This will make the scripts extremely hard to maintain when application modifications are made.

Input data should not be directly coded into the script of the automated tests. Because sometimes we need to modify the input data, if the data is directly coded into the script, we have to look through all the places where the data is used and make the changes. It requires lots of maintenance job. So it is better to store the data in a file or excel and retrieve them as many times as we wish. When some changes are made, we only need to maintain one file instead of the different locations in the code.

In this thesis, we require a smart design of automated tests in order to reduce the maintenance efforts when some changes are made. The tests need to be maintained over the product life just like any program/system would require maintenance.

4.3 Implementing Automated Testing

After having the knowledge of the case study procedure about how to conduct automated testing in Bikube 2011, we are able to implement it in the real software. This part is the second step in the case study procedure: implementation. In this step, firstly, we select the target parts in Bikube 2011. These parts will be tested automatically. Secondly, we choose the tools which can fulfil the automation tasks. Thirdly, we write automated tests in the target parts with selected tools.

4.3.1 Selection of Testing Objects

In this case study, the automated testing will not cover 100% of the components or the methods in each layer. Instead, we will test some of the most typical parts in each layer and we believe it is enough to give the idea about the process of implementing the automated testing in Bikube 2011.

For the higher level, the testing approach is mainly based on the Telerik Test plug-in in Visual Studio combined with code modification. For the lower layers, like the service layer and the data access layer, they require coding efforts and usage of frameworks.

In the user interface layer, we are only going to do automated testing in Filter part. The main reason for choosing Filter part to test is that, testing the filters involves the majority of the technologies which have been used in the Bikube portal, for instance, typing text, selecting dropdown list, selecting pane&icon, clicking button, sending information to the background.

Furthermore, Filter is one of the most important and most frequently used functions in the portal. In a project management system, how well the filters are can directly influence the overall system performance due to the necessity of sorting huge amount of projects. Quickly locating the right project must rely on the filters.

In the service layer, we need to test the methods. We take one of the method 'retrieving geography information' to test because this method shares some common features with other methods when we want to create a project, for instance, most of the methods cannot be used before a project is created. And in a document management system, creating projects is one of the most important activities.

Furthermore, Asplan Viak As has lots of customers who spread around the whole of Norway. So the correct location information of different customers is important and it can also tell which office is in charge of a specific project.

In the data access layer, since this layer is directly connect to the database, so how the objects reflect the database is significant. The data is required to be retrieved correctly. We need to test the mappings of the domain objects and the underlying database 'Project' table. Since projects are the main focus in Bikube, we would test how the elements (name, leader, status, etc.) in projects can be mapped.

4.3.2 Selection of Testing Tools and Frameworks

Firstly we select the tools and the frameworks which will be used for implementing the automated tests. Secondly, we select a code coverage tool. One of the most frequently used features in test quality evaluation is code coverage which offers a quantitative measure for the quality of tests. Or we can say it helps us to find out whether or not the software is being thoroughly tested. Automated code coverage tool is essential to the software testing because with it we can know what is covered and what needs to be covered.

4.3.2.1 Visual Studio 2010

Visual Studio 2010 is the development tool for Bikube 2011 and C# is the development language; hence our default choice of testing environment is Visual Studio. There exist three editions of Visual Studio: Professional, Premium and Ultimate.

The developers are currently using Visual Studio Professional. However our choice is the Ultimate edition due to its outstanding features in testing. Ultimate has advanced built-in testing tools which offer comprehensive features not found in other Visual Studio editions. One of the most special features is coded UI tests, which automate the testing of user interfaces in web and Windows-based applications.

One of the challenges in UI test automation is how to recognize the UI objects as well as human actions by the machine without any errors. Not every tool can be used in all kinds of scenarios. So, after the basic testing tool for user interface has been chosen, it is necessary to do some sample tests to check how the tool works and what the result is.

We pick up a user story which is bounded with the requirement: any login user can create a 'Forespørsel'. Then we use coded UI test to fulfil this user story. However, some problems occur: 1) Any drop down pane is impossible to be selected for future actions. 2) It is impossible to verify some components in user interface. Obviously, coded UI test is not suitable for our requirement of conducting tests on user interface. We can hardly use the tool to recognize the UI objects. There is no doubt that we need to find a new tool for automated testing which can be built inside Visual Studio to test the user interface.

4.3.2.2 Telerik Test Studio

There exist over 20 different automation tools [19], it is not so easy to find a proper tool but it is vital to have one. After a deep discussion with the developer, we find out that Telerik ASP.NET MVC has been used to develop Bikube user interface (we have already installed MVC during the preparation). We find a testing tool Telerik Test Studio which is developed by the same company. Therefore we try to implement the user story in Telerik Test Studio (See Appendix B). The whole user story has been transferred into 31 steps which include actions and verifications. We can modify the code behind the steps to fulfil specific requirements (See Appendix B). The sample test totally reaches our expectation of the user interface test both in capture action and verification parts. Hence, we decided to use Telerik Test Studio as a plugin in Visual Studio to fulfil the user interface tests. Telerik test studio plugin also supports C#.

For the rest of the automatic testing in other layers, we need to find a proper framework.

4.3.2.3 Mspec

Mspec (short for Machine Specification) is a Behavior-Driven Design (BDD) framework for Microsoft.NET. BDD is an evolution in the thinking behind Test Driven Development [20]. Mspec is one of the most widely used frameworks in BDD for designing automated tests. BDD uses a very specific and small vocabulary to minimise the miscommunication among developers, testers, analysts and managers. With this framework, all involved members can not only understand the meaning of tests, but also they will be on the same page of development.

A test with Mspec needs four pieces of information to be established.

1. Defining variables and test name
2. Setup the context
3. Check functionality (which will be specified in class name)
4. Check result

An example of Mspec main code is written below.

[Description]

```
Establish context = () => {};
```

```
Because of = () =>{};
```

```
It should_return_oppdrag_containing_name_specified = () =>{};
```

The way of using Mspec can be compared with normal testing framework to have a better understanding [21]:

[Description] = TestContext

Establish context = SetUp

Because of = Also SetUp, but happens after Establish context

It = Test

MSpec uses named delegates and anonymous functions instead of methods and attributes in order to make the tests easy to be read.

All the variables are defined at the beginning of the test body, right after [Description]. **Establish** and **Because** methods are executed before every test. With MSpec, the **Establish** and **Because** anonymous methods are executed only once for every context [22]. Also the fields in the contexts should be set to static so that the anonymous methods can access them. The result check starts with the word "should" after **It**.

We run these Mspec tests with the help of TestDriven.NET plugin to yield the output in the Visual Studio window.

4.3.2.4 NDBUnit

'Mocking' the data layer of Bikube 2011 is not very easy and straightforward since the layer connects to the Microsoft SQL Server database. Dealing with the data properly is the key to make the data access layer tests work well. There are two strategies that are best suited: Typemock Isolator tool (commercial) or the NDbUnit framework (free open source). In our thesis, we would like to use NDbUnit.

NDbUnit is an Apache 2.0-licensed open-source project which is written in C# and supports Microsoft SQL Server 2008. We can also say that NDbUnit is a .NET library for managing database state and is a critical tool when we test the data access layer. It can be used to the tests with low complexity.

NDbUnit uses .NET dataset *xsd* file and *xml* file to load the test data in the database. For the relevant tests, a dataset needs to be created, along with the required test data in *xml* form. The dataset from the required *xsd* file must fully match the database schema. Dataset must be equivalent to the Table in the database [23].

4.3.2.5 Ncover

Ncover is a convenient tool of checking code coverage inside the software. There are two kinds of code coverage which are generated by NCover: symbol coverage and function coverage.

Symbol coverage checks the code coverage by sequence points. The code coverage here measures how many sequence points have been executed. Sequence points are very small code parts which can be either completely executed or not completely executed.

Symbol coverage tells which code has been executed and which code has not in a very detailed way. One of the disadvantages in symbol coverage is that the code must be totally executed to tell whether conditional statements (such as 'if' statements) have executed both their 'true' and their 'false' blocks. [24]

Function code coverage checks the code coverage by using function/method. Method or function coverage measures how many methods have been executed. The method could exit partway through. For example, a method will still be regarded executed even there is only one 'if' statement has been executed. A function coverage of 50% means that half of the methods under the module, namespace or class have been called, and half have not been called. Method coverage gives a very high level overview of how well a project is covered. [24]

In order to run Ncover directly from Visual Studio to test service layer and data access layer, we set Ncover as an external tool inside Visual Studio. Alternatively, we can use TestDriven.NET to run Ncover in Visual Studio as well.

For checking code coverage in user interface layer either for automated testing or manual testing, we need to set up the “Path to application to profile” and “Arguments for the application to profile” like the following graph to enable Ncover detect Bikube application.

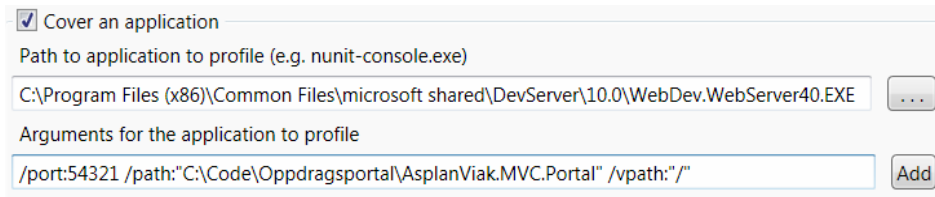


Figure 4.6: Setting up Ncover for the user interface

4.3.3 Experiment of Automated Testing

We implement the automated testing in all the layers in Bikube 2011 architecture.

4.3.3.1 Automating User Interface Testing

Client/Front End is also known as user interface. The main purposes of UI test is validating UI objects, checking functional flows and verifying displayed output data which are generated from lower layers.

This layer is directly exposed to the end users. The testing conducted here is usually system testing and being done manually due to the complexity of the interfaces. There are thousands of projects stored in Bikube 2011. In order to quickly locate the right project, filtering and sorting are important to support this function. According to the requirement specifications, we pick up several requirements is Filter part, and instead of testing all the features manually; we write the test suites against the requirements and then automate them.

When we test the user interface manually, we can do any actions without limitation. However, it is difficult for the machine to perform the same since complicated codes will be required. In automated testing, we divide each test as small and simple as possible to avoid the complexity of test. The following screenshot shows how the filters look like in the web portal.

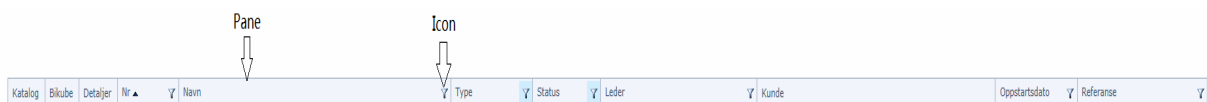


Figure 4.7: Outlook of the filters

Test Suites

Each test suite is related to one requirement. Based on the test object and the requirement specifications of the Bikube 2011 in the user interface, we list three requirements which can be used to connect to the test suites.

- **Requirement 1:** Clicking the filter panes allows the information in the corresponding row being sorted by specific orders.
- **Requirement 2:** Clicking the filter icons shows the sub-filter panes up.
- **Requirement 3:** Typing specific information in sub-filter can result the corresponding row to show specific information.

Test cases

The test case should be written based on the test suite and consists of test steps in order to easily show how the test works and what the test case aims to do. For example, the test case **Click filter pane ‘Nr’** can be fulfilled in three steps. This test case connects to the requirement 1.

1. Navigate to <http://localhost:54321/Home/All>
2. Click Nr Filter
3. Expectation: The numbers in the row sorted from small to large.

We summarized all the test cases which connect to the requirement 1 in the following table.

Test Suite: Requirement 1			
Test Case	<i>Step 1</i>	<i>Step 2</i>	<i>Expectation</i>
1.1 Click filter pane ‘Nr’	Navigate to http://localhost:54321/Home/All	Click Nr Filter	The project numbers in the row sorted from small to large.
1.2 Click filter pane ‘Navn’		Click Navn Filter	Projects are sorted from A to Z by name.
1.3 Click filter pane ‘Type’		Click Type Filter	The projects with property ‘tilbud’ show first and sorted by number.
1.4 Click filter pane ‘Status’		Click Status Filter	The projects with property ‘Activ’ show first and sorted by number.
1.5 Click filter pane ‘Leder’		Click Leder Filter	Projects are sorted from A to Z by Leder.
1.6 Click filter pane ‘Kunde’		Click Kunde Filter	Projects are sorted from A to Z by Kunde.
1.7 Click filter pane ‘Oppstartsdato’		Click Oppstartsdato Filter	Projects are sorted by date.
1.8 Click filter pane ‘Referanse’		Click Referanse Filter	Default sort by database.

Table 4.1: Test cases with requirement 1

The test cases are written based on the steps and organized in Telerik Test Studio. We can see how the tests look like in Telerik Test Studio from the following screenshot.

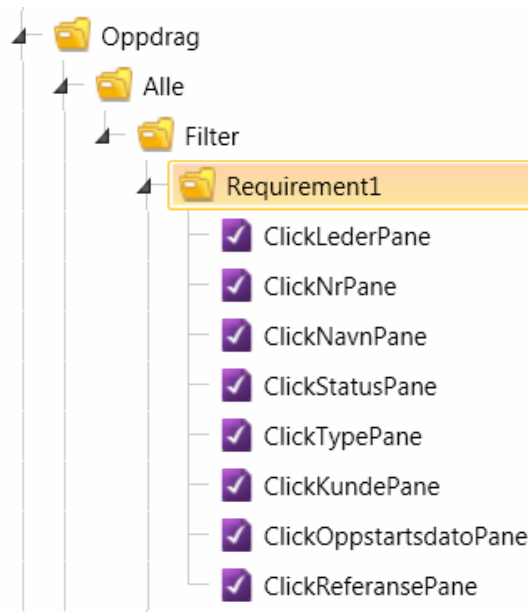


Figure 4.8: Test cases in Telerik with requirement 1

We summarized all the test cases which connect to the requirement 2 in the following table.

Test Suite: Requirement 2			
Test Case	Step 1	Step 2	Expectation
2.1 Click filter icon 'Nr'	Navigate to http://localhost:54321/Home/All	Click Nr icon	Sub-filter panes show up
2.2 Click filter icon 'Navn'		Click Navn icon	
2.3 Click filter icon 'Type'		Click Type icon	
2.4 Click filter icon 'Status'		Click Status icon	
2.5 Click filter icon 'Leder'		Click Leder icon	
2.6 Click filter icon 'Kunde'		Click Kunde icon	
2.6 Click filter icon 'Oppstartsdato'		Click Oppstartsdato icon	
2.7 Click filter icon 'Referanse'		Click Referanse icon	

Table 4.2: Test cases with requirement 2

The test cases are organized in Telerik Test Studio and shown in the following screenshot.

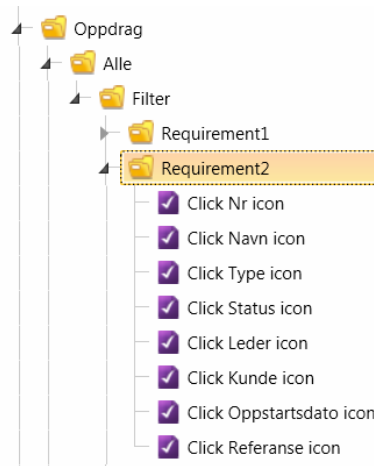


Figure 4.9: Test cases in Telerik with requirement 2

We summarized all the test cases which connect to the requirement 3 in the following table.

Test Suite: Requirement 3				
Test Case	Step 1	Step 2	Step 3	Expectation
3.1 Search project by specific Nr	Test Case 2.1	Select 'Is Equal To' and Type '529109' in Sub-filter panes	Click 'Filter'	Only one project shows up with Nr 529109
3.2 Search project by specific Navn	Test Case 2.2	Select 'Contains' and Type 'bistand barentswatch' in Sub-filter panes	Click 'Filter'	All the projects which include 'bistand barentswatch' show up
3.3 Search project by specific Type	Test Case 2.3	Select 'Oppdrag' in Sub-filter panes	Click 'Filter'	All oppdrag type show up
3.4 Search project by specific Status	Test Case 2.4	Select 'Avsluttet' in Sub-filter panes	Click 'Filter'	All Avsluttet status show up
3.5 Search project by specific Leder	Test Case 2.5	Select 'Is Equal To' and Type 'Lin Egholm' in Sub-filter panes	Click 'Filter'	All projects which belong to Lin Egholm show up
3.6 Search project by specific Kunde	Test Case 2.6	Select 'Is Equal To' and Type 'Kristiansand Kommune' in Sub-filter panes	Click 'Filter'	Projects with Customer 'Kristiansand Kommune' show up
3.7 Search project by specific Oppstartsdato	Test Case 2.7	Select 'Is Equal To' and Type '08.02.2012' in Sub-filter panes	Click 'Filter'	Projects with Start time '08.02.2012' show up
3.8 Search project by specific Referanse	Test Case 2.8	Select 'Is Equal To' and Type 'Ikke angitt' in Sub-filter panes	Click 'Filter'	Projects with Referanse 'Ikke angitt' show up

Table 4.3: Test cases with requirement 3

The test cases are organized in Telerik Test Studio and shown in the following screenshot.

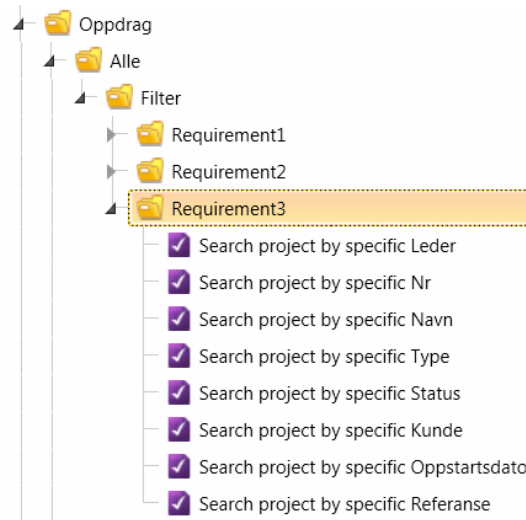


Figure 4.10: Test cases in Telerik with requirement 3

Test Design

The tests are exported from Telerik Test studio into Visual Studio and then the codes behind the tests are modified in order to fulfil specific tasks, for example, verification.

For the above test cases which connect to requirement 1 and requirement 3, code modification is required. The first two steps in requirement 1 ‘Navigate’ and ‘Click’ can be easily recoded by Telerik Test Studio capture action tool, but we need to add code behind the verification part. The basic idea is to find the project table from the webpage and select the corresponding column. Then we retrieve the information from each row and store them into a list. Once all the data is stored in the list, we can easily verify whether the data fulfil our expectation or not.

The most important and difficult step here is to retrieve the table from the webpage because the table can be only targeted down by its number (This is how the software being designed). Hence, we have to look through the original html code behind the webpage to locate the target table and then create a ‘oppdragTable’ to store it. The code is illustrated below.

```
ArtOfTest.WebAii.Controls.HtmlControls.HtmlTable oppdragTable =
Pages.AlleOppdrag.Get<ArtOfTest.WebAii.Controls.HtmlControls.HtmlTable>(
"cellspacing=0", "tagIndex=20", "tagname=table");
```

Once we get the table, we make a loop to verify the data in the list. The detailed code for test case 1.1 **Click filter pane ‘Nr’** can be found in appendix C as an example.

However, some tests are quite simple which do not require any code modification. For example, when we design the test cases for requirement 2, we simply record the action in the user interface and verify the sub-filter is visible by the default tool in Telerik Test Studio.

Test Execution

After finishing the tests design, we can simply run them in Visual Studio Test List without any human supervision. During the execution, all run-time information will be recorded. And after execution, logs files will be created to identify whether a test is Pass or Fail and notify what the exceptions are. The test result will be returned and displayed automatically with **Result** (Pass/Fail status), **Test name**, **Project** (where the tests are located) and **Error Message**. In the test result, the number of the tests which are executed for each test suite is also displayed.

The test results of the above three test suites are listed below.

Result	Test Name	Project	Error Message
Failed	ClickKundePane	BikubeTestProject	Overall Result: Fail...
Passed	ClickTypePane	BikubeTestProject	Overall Result: Pass...
Passed	ClickNrPane	BikubeTestProject	Overall Result: Pass...
Passed	ClickLederPane	BikubeTestProject	Overall Result: Pass...
Passed	ClickNavnPane	BikubeTestProject	Overall Result: Pass...
Passed	ClickStatusPane	BikubeTestProject	Overall Result: Pass...
Passed	ClickOppstartsdatoPane	BikubeTestProject	Overall Result: Pass...
Passed	ClickReferansePane	BikubeTestProject	Overall Result: Pass...

Figure 4.11: Test Result of Requirement 1

Result	Test Name	Project	Error Message
Passed	Click Oppstartsdato icon	BikubeTestProject	Overall Result: Pass...
Passed	Click Nr icon	BikubeTestProject	Overall Result: Pass...
Passed	Click Type icon	BikubeTestProject	Overall Result: Pass...
Passed	Click Referanse icon	BikubeTestProject	Overall Result: Pass...
Passed	Click Status icon	BikubeTestProject	Overall Result: Pass...
Failed	Click Kunde icon	BikubeTestProject	Overall Result: Fail...
Passed	Click Navn icon	BikubeTestProject	Overall Result: Pass...
Passed	Click Leder icon	BikubeTestProject	Overall Result: Pass...

Figure 4.12: Test Result of Requirement 2

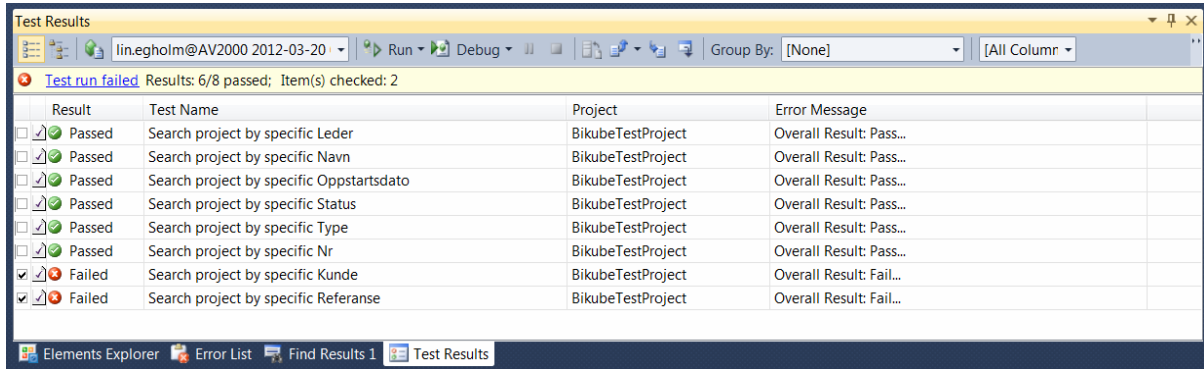


Figure 4.13: Test Result of Requirement 3

Problems Identifying

From the above results, we find out that 4 tests have failed. We identify the error by checking the test log information and summarize them in the following table.

<i>Test</i>	<i>Bug/Problem</i>
1.6 Click filter pane ‘Kunde’	Pane ‘Kunde’ is NOT enabled
2.6 Click filter icon ‘Kunde’	Icon ‘Kunde’ is NOT available
3.6 Search project by specific Kunde	Pane ‘Kunde’ is NOT enabled
3.8 Search project by specific Referanse	An alert window pops up

Table 4.4: Bug/Problem in test cases

Clearly identifying the problems is essential in the process of automated testing. The executed test cases together with the test results will be automatically exported after each run and stored in a repository.

Problem Fixing

These failures will be reported to the developers and modified by them. With automated testing, the developers can replay the tests whenever they want to re-create the bug/ problem or check the log file to identify the problem. However, problem fixing will not be the focus here since developers will do it and it is not included in software testing.

A possible way to solve the error in test case **2.6 Click filter icon ‘Kunde’** is to add a new filter icon; the main code is explained below.

```

public ArtOfTest.WebAii.Controls.HtmlControls.HtmlSpan FilterSpan6
{
    get
    {
        return
        Get<ArtOfTest.WebAii.Controls.HtmlControls.HtmlSpan>("id=Oppdrag",
        "|", "tagIndex=span:6");
    }
}
    
```

After the problem has been fixed, we can replay the test automatically and obtain the results immediately to see whether the problem has been fixed or not.

4.3.3.2 Automating Service Layer Testing

The service layer is a central layer in the system architecture. When performing actions in the user interface, it calls the methods or functions from the service layer. When methods or functions require data access, it will call the data access layer. So we can say in this layer, our aim is testing the methods or functions which are used to fulfil different services.

Services in Bikube 2011 can be generally divided into four parts: project (including tilbud, forespørsel, oppdrag and intern aktivitet) service, person service, customer service and lookup service.

Actually, the above automated user interface testing in filter is a part of project services which calls the filter function in the service layer. In the user interface layer, a whole project table is required before we can conduct testing. However, in the service layer, we do not want to wait until the project table is finished before we can test the functions/methods. The main idea here is using the mock object framework, Rhino Mocks [25] to do the tests in this layer to reach our expectation.

However, the friendly user interface control tool is replaced by pure code design when we conduct automated testing here. This will be difficult for the person who has not developing experience and not have any knowledge of the programming language or tools. Being well trained is extremely important when comes to this layer.

All the tests should be based on one requirement. So we take one of the requirements which is related to the lookup service. Under this requirement (also regarded as test suite), we establish three test cases. Each test case corresponds to one method, like ‘getting fylke list’ method.

- **Requirement 4:** When creating projects, geography information can be retrieved correctly.

Test Suite: Requirement 4			
Test cases	<i>Step 1</i>	<i>Step 2</i>	<i>Expectation</i>
4.1 Getting fylke list	Creating a project	Checking the content of lists	All the fylke show up
4.2 Getting region list	Creating a project	Checking the content of lists	All the region show up
4.3 Getting kommune list	Creating a project	Checking the content of lists	All the kommune show up

Table 4.5: Test cases with requirement 4

The related ‘retrieving geography information’ method at user interface layer is shown in Figure 4.14. If we want to test the method manually, we can create a project first and then test the function by counting the amount of items in the list. In the automated testing, no real project is required to create. So in the service layer, we can say automated function tests are most suitable and effective for testing method.

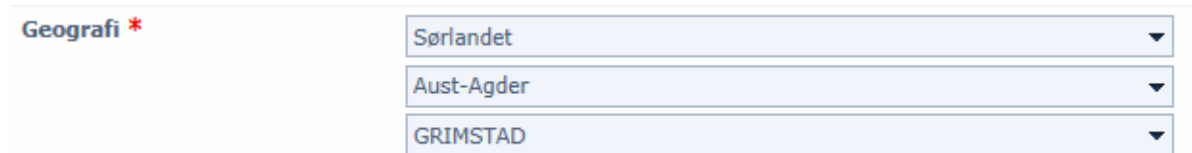


Figure 4.14: Selecting geography in user interface

The reason of using mock is that we haven’t actually implemented a particular object which is required for the tests. We want to unit test the method that relies on a **Project** object, but during the implementation of requirement 4, we have not had a chance to implement the **Project** object yet. Another reason is that sometimes the actual object might access to an external resource such as database which makes the actual object be too slow to use and drag down the speed of running the tests. If we want to run the tests, the geography records must be retrieved from a database table. Retrieving data from a database is slow even though the amount of data might be small. So the mock object is acting as a replacement for the real object so that we can build our tests that depend on that object. In other words, when testing the service layer, we mock the behaviour of the data access layer and really just test the service methods or functions.

For using mock, we just need to simply add a reference to the Rhino.Mocks.dll assembly. We aim to test three methods (three test cases) under `LookupService`. They are `GetFylkeList()`, `GetRegionList()` and `GetKommuneList()`.

Here, we use test case 4.1 as an example to test method `GetFylkeList()`. Firstly, we call the static method `MockRepository.GenerateMock<T>()` (no return value) to generate a stub for abstract class. After we generate the stub, we can treat the stub like a normal class. The following code mocks an object to store the fylke list.

```
_readRepositoryMock = MockRepository.GenerateMock<IReadRepository>();
_readRepositoryMock.Stub(x => x.FindAll<GEO_FYLKE>()).Return(_fylkeList);
```

The fylke list is created by calling a `FakeObjectBuilder` class.

```
_fylkeList = FakeObjectBuilder.CreateFakeList<GEO_REGION>(_fylkeCount);
```

We use this list as input to set objects in `LookupService` and then we call the method `GetFylkeList()` in `LookupService` to verify the result of this method. The returned list can be compared with the one we created in order to verify the method.

The result of the three test cases is illustrated in the following graph. We can see all the tests passed and it only took around 1.5 seconds to execute each test. The result also shows the advantage of Mspec which gives added structuring results which are easier to be understood. From the result, we can say that requirement 4 is realized by the geography look up service.

```

----- Test started: Assembly: AsplanViak.ODS.Services.Test.dll -----

Getting region list, when getting region list
» should return the region list

1 passed, 0 failed, 0 skipped, took 1,50 seconds (MSpec).

----- Test started: Assembly: AsplanViak.ODS.Services.Test.dll -----

Getting kommune list, when getting kommune list
» should return the kommune list

1 passed, 0 failed, 0 skipped, took 1,51 seconds (MSpec).

----- Test started: Assembly: AsplanViak.ODS.Services.Test.dll -----

Getting fylke list, when getting fylke list
» should return the fylke list

1 passed, 0 failed, 0 skipped, took 1,57 seconds (MSpec).

===== Total Tests: 3 passed, 0 failed, 0 skipped, took 6,25 seconds =====

```

Figure 4.15: Test result of requirement 4

4.3.3.3 Automating Data Access Layer Testing

The data access layer (DAL) provides simplified access to data and directly connects to the database on the server. So in this layer, not only Mspec is needed, but also the help with NDBUnit is essential to deal with data.

Bikube 2011 interacts with the database almost at every stage which makes the testing on this level essential. We discovered that unit tests are not suitable here since they only test the units of the application individually. Our main purpose is to test how different units interact with the database. Therefore, integration tests are more suitable under this circumstance to test the data access layer.

- **Requirement 5:** The mappings of the domain objects and the underlying database *Project* table are correct (Impedance mismatch verification) when searching project by status and name.

Each test is separately organized in a C# file. There are 7 test cases and 3 of them are negative test which means we search wrong/invalid data in the tests and an error message should be returned to make the tests pass.

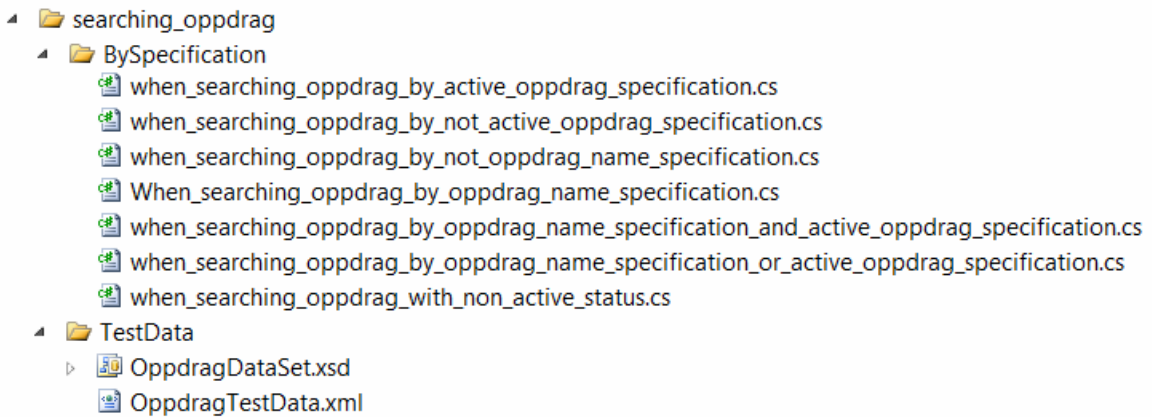


Figure 4.16: Tests with requirement 5

When we conduct the tests on the database, we must make sure that one test will not modify the values in the database that can fail another test. But we still need to test the data access components on the actual data. So running tests on the real database is not a good idea, we might ruin important data. We use NDBUnit to create just enough data from the local database for running the tests on in order to have a controlled number of the data. This can simplify the verification of the results.

This mocked-out data access layer can reduce run time to almost zero. In order to tell the test how to connect to the test database, a special Helper.cs file with a ConnectionString is used to call the test database.

```

public static string
GetDatabaseConnectionStringWhichContainsTestData()
{
    return
    ConfigurationManager.ConnectionStrings["AsplanViak.Repository.En
tityFramework.IntegrationTest.Properties.Settings.ODSConnectionS
tringWhichContainsTestData"].ConnectionString;
}

```

When the integration test starts, as necessary, data is inserted into the database and then the tests run. During the run, tests may insert or remove any data. Once the test is complete, the database needs to be returned to its original state before the tests. Even if the test fails, the database should be returned to its original state. We illustrate the detailed steps below.

➤ Generating Test Data

First of all, for the tests which cover the query part of the data access layer, we need to have the test data that the test will work upon. Since we do not want the data of each test overlap, all the tests should generate their own test data and clean it after running, so the data does not interfere with the results of the other tests.

We use NDBUnit to load the test data from the database using dataset and *xml* file. The dataset should be created directly from the server explorer in Visual Studio. The server explorer displays the local database. We can find the corresponding table which holds required data for the test in question and then drag it into the dataset. This is the easiest way to create the dataset.

Then we need to create an *xml* file in Visual Studio and add the dataset schema in the *xml* file property so that Visual Studio can provide intelligence to the *xml* file, which offers us the default setting of the controlled data. Afterwards we can start adding data in the *xml* file with the help of dataset schema.

➤ Loading and Cleaning Test Data

All the tests in data access layer should subclass from `auto_cleaning_db_context`. The base class indicates the details about the actual data loader and ensures to keep the database clean after every test. In this class, it establishes the `ConnectionString` to the test database and creates a method `LoadData` for the tests to pass in the dataset schema and *xml* files, it then loads the data from the *xml* file. After the test data has been loaded, the method is also responsible for cleaning the insert data. So in the test we only need to simply load and clean the data by calling the method `LoadData`.

The following screenshot is the running result of the above test cases and all of them have passed.

```

Search Oppdrag, when searching oppdrag by active oppdrag specification
» should return oppdrag list satisfying active oppdrag specification
1 passed, 0 failed, 0 skipped, took 7,61 seconds (MSpec).
Search Oppdrag, when searching oppdrag by not active oppdrag specification
» should return oppdrag list satisfying specification
1 passed, 0 failed, 0 skipped, took 20,99 seconds (MSpec).
Search Oppdrag, when searching oppdrag by not oppdrag name specification
» should return oppdrag list satisfying specification
1 passed, 0 failed, 0 skipped, took 17,94 seconds (MSpec).
Search Oppdrag, when searching oppdrag by oppdrag name specification
» should return oppdrag list satisfying oppdrag name specification
1 passed, 0 failed, 0 skipped, took 5,20 seconds (MSpec).
Search Oppdrag, when searching oppdrag by oppdrag name specification and active oppdrag specification
» should return oppdrag list satisfying specification
1 passed, 0 failed, 0 skipped, took 5,08 seconds (MSpec).
Search Oppdrag, when searching oppdrag by oppdrag name specification or active oppdrag specification
» should return oppdrag list satisfying specification
1 passed, 0 failed, 0 skipped, took 8,20 seconds (MSpec).
Search Oppdrag, when searching oppdrag with non active status
» should return oppdrag which is not active
1 passed, 0 failed, 0 skipped, took 16,00 seconds (MSpec).

===== Total Tests: 8 passed, 0 failed, 0 skipped, took 88,35 seconds =====

```

Figure 4.17: Test result of data access layer

4.4 Maintenance

This part is the last step in the case study procedure: maintenance. In this case study, we did not experience the changes in the software which may require the maintenance in the test scripts. We maintain the tests that the related requirements have been changed.

The maintenance what we focus here is mainly on the methods which were used to minimize the efforts of the maintenance work in the case study. The way of modifying the test scripts can be very different which depends on the test cases. But all the tests can share some common features to let the maintenance work as easy as possible. The methods which we have used in the case study for minimizing the efforts of the maintenance work are described below.

- **Organizing the test suites well**

We have a detailed plan about how to automate the tests which makes the maintenance effective and efficient. An essential part of the test plan is the definition of the test suites, the test cases and the test steps. Each test suite connects to a single requirement. So when requirement changes, we can locate the corresponding test suite easily and simply modify the test cases and test steps.

- **Designing and maintaining the tests by the same person**

In the case study, designing and maintaining the tests by the same person saved us lots of time. There is no extra effort needed to understand the tests code. It is easy and efficient to maintain the tests since the tester has already known the details of each test and the system code. However, when the tester tries to maintain/fix other people's tests, he has to spend time on understanding the tests first.

- **Treating tests like any other software code**

Test cases and their associated test scripts, whether recorded or programmed, should be treated like the real software code. We paid lots of attention on how to design better test scripts by selecting suitable test tools and frameworks. We also regard our test suites as assets that belong to the whole development team rather than just the testers.

- **Close cooperation between testers and developers**

We improve the testability of the test cases through the close collaboration between the testers and the developers. This makes the maintenance work much easier. We used the test cases as a way to communicate with the developers about the software quality. We also located the tests in the software solution repository which can be accessed by the developers. So the developer can run the tests themselves when they try to fix the bugs. The developers will notify the testers about the changes in the software once they make, so that the testers know what to maintain immediately.

The above methods are easy to be adopted. Well designing and planning the tests minimize the efforts which are spent on locating the tests. Designing and maintaining the tests by the same person reduce the difficulty of maintaining the scripts. Close cooperation between the tester and the developers speed up the maintenance procedure. However, when some

changes (such as machine change, operation environment change or the whole requirement specification change) occur in the development of Bikube 2011, it still requires lots of efforts to correct the test scripts or introduce new tests. So trying to avoid unnecessary changes from day one is significant.

4.5 Results Evaluation

After the implementation has been done, we would like to evaluate our process based on the hypotheses. These hypotheses are related to the specific research questions in the chapter 1.2 in order to solve the main problem. The results of hypotheses are presented in the following table.

Hypotheses	Related research questions	Result
H1: The automated testing in Bikube 2011 can be implemented with the procedure of training, implementing and maintenance.	1	T
H2: The way of organizing the manual tests in current Bikube 2011 and the automated tests in case study is different.	1	T
H3: The combination of different automation tools can result better automated testing.	1	T
H4: The cooperation in the small developing team is more efficient than that in the big team.	2	T
H5: Automated testing preforms better than manual testing in all the layers.	2	F
H6: Automated testing costs more than manual testing.	2	F
H7: It is easier to increase the code coverage of automated testing than that of manual testing.	2	T
H8: Automated testing can drive the design of new functionality in Bikube 2011 while manual testing cannot give any benefit.	3	T

Table 4.6: Hypotheses and results

4.5.1 Hard Features

H1 predicts the implementation procedure of the automated testing. We follow the procedure by starting with training ourselves in the area of automated testing. The knowledge that we have learned is carried out in the chapter 4.2.2. The knowledge helped us to be aware of what and how we need to do when we implement and maintain the automated testing in Bikube 2011. The procedure is successfully adopted in this thesis (in chapter 4.3 and chapter 4.4).

H2 predicts how the automated test cases should be organized. We noticed that the way of organizing is different between the current manual tests in Bikube 2011 (chapter 3) and the automated tests in the case study at user interface layer (chapter 4.3.1.1). We found out that current manual test cases are not ‘small’ enough to be totally used in automated testing. More specifically, we can say that the current manual tests are actually described at a requirement level, too general. So in the case study, we have to cut down the size of manual tests and make them as detailed as possible but cover with one specific requirement as well to be used in automated tests. As a result, we can easily follow the steps which are defined in the test cases to write automated tests scripts.

The process of defining test cases before starting automation is an excellent way to ensure that there is a shared understanding between all involved members and the actual requirements being developed are tested. And in fact, the manual tests can be organized in the same way as the automated tests to obtain the advantages.

H3 predicts the number of tools which are used for automated testing, is not only one. And the proper combination of different tools can result better automated testing. Obviously, in chapter 4.3.2, we introduced two automation tools, two automated testing frameworks and one code coverage tool. Also, we use Test.Net to run the automated tests. The combination of different tools allows us to test the different software layers easily and efficiently. Using proper tools is one of the keys to set up automated testing successfully.

4.5.2 Soft Features

H4 predicts the cooperation in the small developing team is more efficient than the big team. Our team consists of two developers who work in Asplan Viak and one tester who works with automated testing, the number of three is defined as a small develop team.

In our case, the benefit of such a small team is that we can work closer; all the defects can be reported immediately to the team. And once developers change some parts in the software, there is a comprehensive transfer of knowledge to the tester who creates and maintains the scripts. There is no communication gap in the small team.

H5 predicts whether or not the automated testing performs better than the manual testing in all the software layers. Tests at user interface layer provide a high level of confidence both for automated testing and manual because you can see exactly what is going on when the tests are executing. But they are expensive to build and fragile to maintain. Manual testing is

useful for the exploring of bugs at the user interface layer while automated testing is more suitable to perform highly repeated tasks.

Only testing one layer in an entire project is not a good choice. In the lower layers, we can check the integration of internal methods, the functions and so on. UI based tests should only be used when the UI is actually being tested or there is no practical alternative.

In the case study, we used the automated testing to test the service layer and the data access layer successfully. However, the manual testing cannot perform well in these layers because 1) In the server layer, we have to create a real project in manual testing before the function can be tested which add extra workload for the tester. 2) In the data access layer, we have to compare the target results with the data in the database manually, which is extremely easy to introduce errors and make the test result incorrect.

Hence, we can say that the automated testing and the manual testing have their own advantages when being implemented in the user interface layer. But in the service layer and data access layer, automated testing is much more effective to be used than the manual testing.

H6 predicts that the cost of automated testing is higher than that of manual testing under the same requirement. We use the test suites from chapter 4.3.3.1 and write the manual tests based on these test suites since it has been concluded in **H2** that the test suits in automated testing and manual testing can be the same. The reason for doing so is that the automated tests and the manual tests are created based on the same requirement which can make the comparison feasible.

Automated testing has a high start cost when a team intends to transfer from manual testing. In our experiment, the training period is 2 months before we can start up automated testing.

The following graph illustrates the implementation and maintenance cost for both manual testing and automated testing when we try to build a specific test suite at the first time. Implementation cost of automated testing is much higher than that of manual testing. For conducting the first test suite in this case study at user interface layer, it requires 290 minutes to implement automated testing while 20 minutes for manual testing. When any changes are introduced in the test suite, it requires around 60 minutes to modify the scripts in automated testing for a signal test suit. But when user interface is totally changed or most of parts are changed, the cost of maintenance in automated testing will equal to the cost of implementation because all the design has to be done from the start. However, it only costs 20 minutes for manual testing no matter what is changed in the user interface.

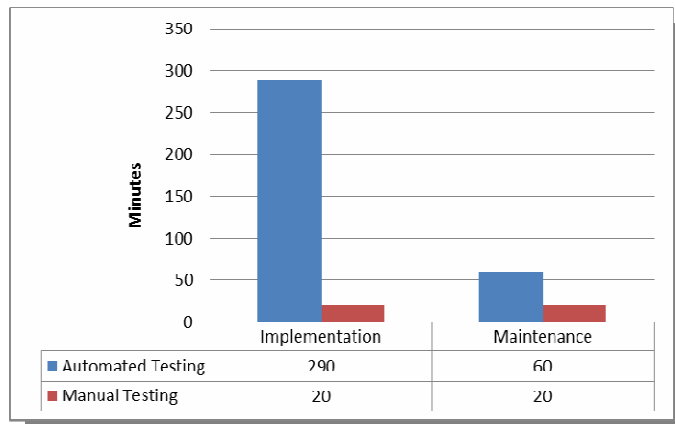


Figure 4.18: Comparison of implementation and maintenance cost

Now we look into the cost inside implementation (Figure 4.19). It is divided into three parts: designing test cases, designing scripts and execute tests. From the following figure, we can see that it takes the same time (around 10 minutes) to design the test cases for both automated testing and manual testing against the same test suite.

Designing the scripts occupies approximately 95% cost in the whole implementation of automated testing while it doesn't require any script design when we implement the manual testing.

It takes less than 1 minute to execute the automated testing while we have to spend 10 minutes to execute the manual testing. In automated testing, we successfully increased the execution speed for tests which can be executed very quickly and do not require any application configuration.

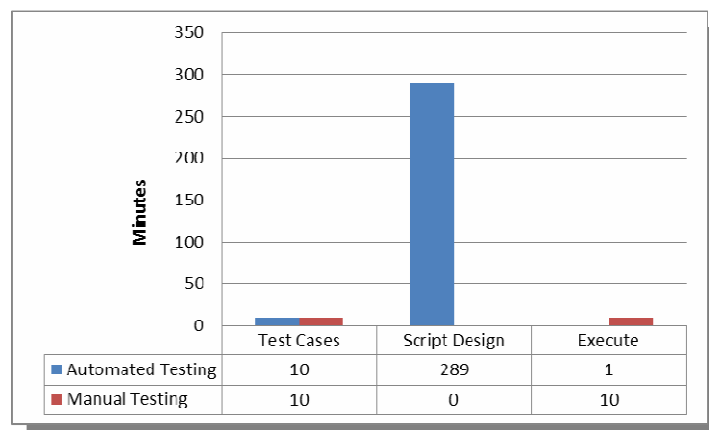


Figure 4.19: Implementation of first test suite

During the experiment of automated testing in chapter 4.3.3.1, the first test suite requires 290 minutes to be implemented, the second test suite only takes us around 100 minutes to implement and we just spend 50 minutes on the implementation of the third test suite. These three test suites aim to test the same function, so after the scripts of the first test suite has been designed, we use less and less time to design the rest of the test suites since we get more and more experience. The costs spend on implementing these three test suites for filter function (chapter 4.3.3.1) are illustrated below.

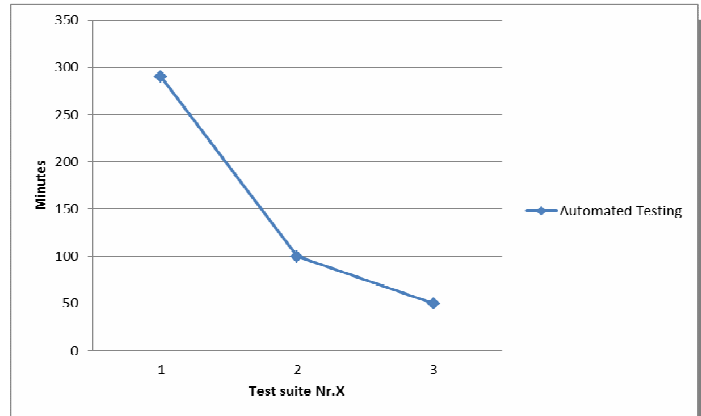


Figure 4.20: The cost of implementing different test suites

From the above data, we would like to find the breakeven point where the automated testing and the manual testing cost the same during the implementation of each test suite. We suppose no changes are made in the user interface which means maintenance cost is not counted here. The accumulating cost of automated testing almost doesn't grow while the manual testing goes up following linear function with slope 1. The following three graphs show the accumulating cost of automated testing and manual testing for all three test suites. The breakeven points of first test suite is 15th run, which means the automated testing cost more than manual testing before the 15th run, same in the 15th run and less after the 15th run. The breakeven points of second and third test suites are 5th run and 3rd run respectively.

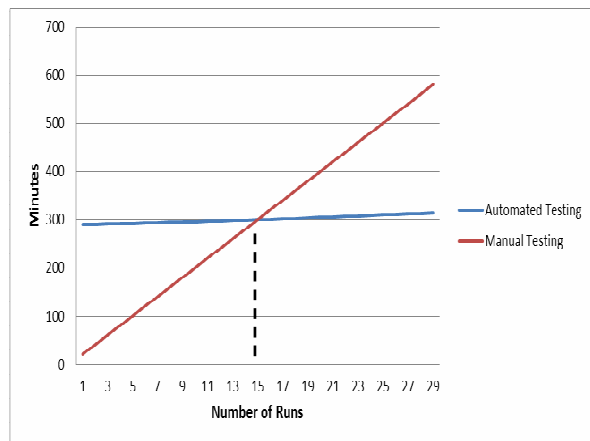


Figure 4.21: Cost of first test suite with requirement 1

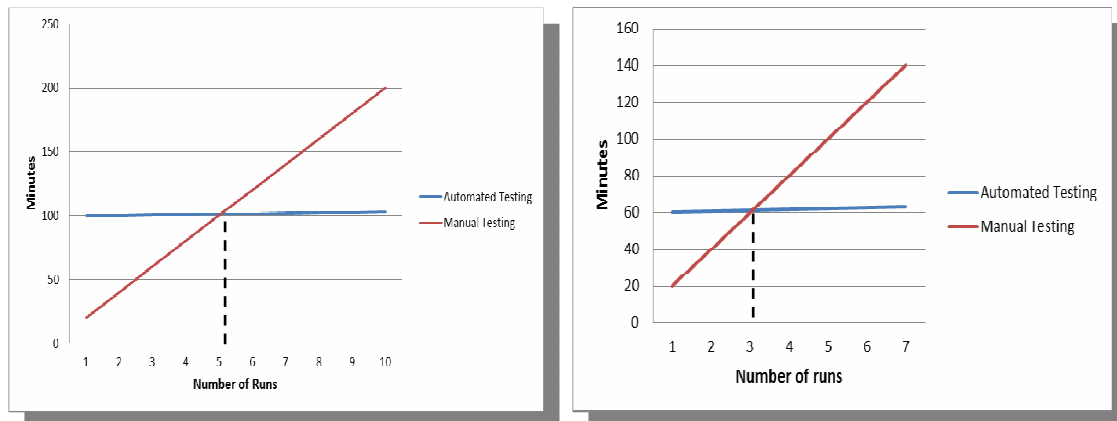


Figure 4.22: Cost of first test suite with requirement 2 (left) and 3 (right)

H7 predicts the code coverage of the automated testing is easier to increase than the code coverage of the manual testing. Code coverage can be checked with the help of Ncover. The process of setting up Ncover is explained in chapter 4.3.2.5.

For the user interface layer, we compare the code coverage of automated testing and manual testing against the same test suites (test suites 1, 2 and 3 in chapter 4.3.3.1). We use the same test cases and steps; the only difference is that the automated testing is run by tool while the manual testing is run by hand. The code coverage results are that the symbol coverage is 6.48% and the function coverage is 11.73% both in the automated testing and the manual testing. It is not surprising that the code coverage of automated testing and manual testing are the same since both of them test against the same suites.

The function coverage is higher than symbol coverage indicates that when the function/method is executed, not all the paths in that function/method have been used for testing. For example, the function will still count as executed in the function coverage even it only throws an Exception. The detailed information of symbol coverage and function coverage are shown in the following picture. From the code coverage, we can find out which parts of the code are not tested in Bikube 2011. However, in the user interface layer, it is difficult for us to know which specific components in UI are related to which specific codes since the tests in this layer is acceptance testing. If we want to improve the code coverage in the user interface, we have no choice but randomly click around to see if we can find the untested code.

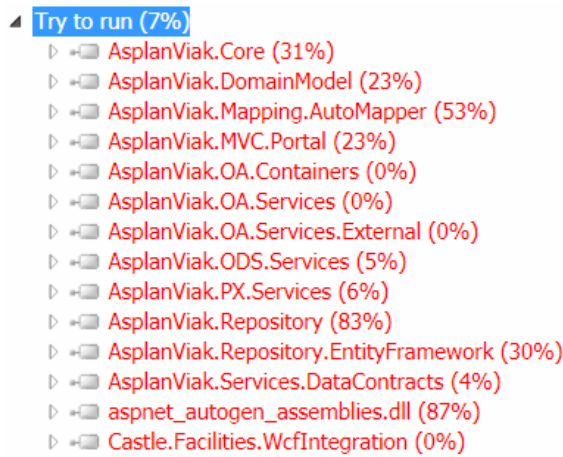


Figure 4.23: Symbol coverage

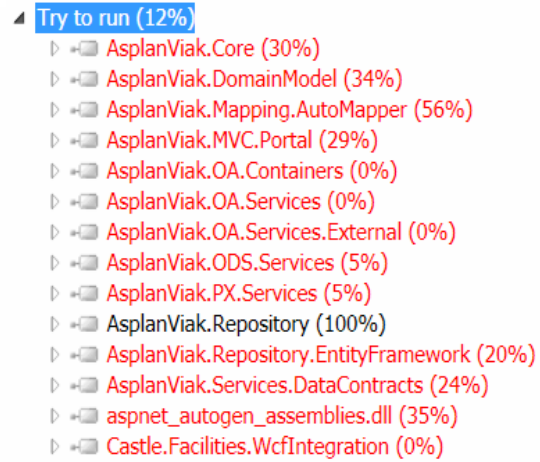


Figure 4.24: Function coverage

In the service layer, we tested “retrieving geography information” in the lookup service. We can see (Figure 4.25) that this specific function has 100% code coverage and cover 42% of the whole lookup service. In this layer, the tests are designed directly against the functions, so it is easy to increase the code coverage for the specific areas.

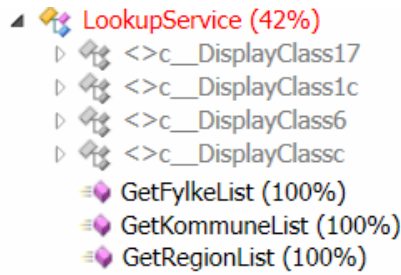


Figure 4.25: Code coverage of test suite 5 “retrieving geography information”

In the data access layer, we use the integration testing to check the mappings of the domain objects and the underlying database ‘Project’ table are correct when searching the projects by status (Active or not) and name. Each test covers several functions which are related to the same requirement. The result is shown in Figure 4.26. The symbol code coverage is 2% of the whole Bikube since we only did automated testing against one test suite. The related functions in the domain models and the repository have been fully tested with 100 % code coverage.

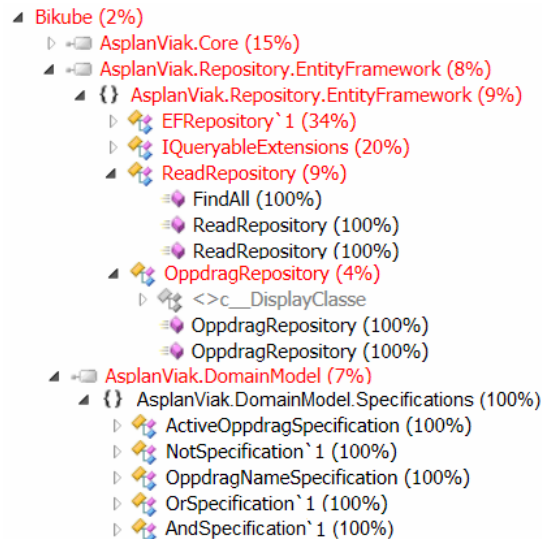


Figure 4.26: Code coverage of test suite 6

With the above results, we cannot simply think that automated testing offers better code coverage than manual testing. In the user interface layer, as long as using the same test suites, automated testing and manual testing can have the same code coverage. But in other layers, the code coverage of automated testing can be quite high since they are tested against the specific functions or modules. And we can easily control (increase) the code coverage to reach an acceptable percentage.

H8 predicts the driven-design possibility for automated testing and manual testing. Driven-design means that the tests are written to define the functionality before the software code is written. The developers design the software code based on the tests. Only when all the tests are pass, the code is considered complete.

Many studies [4][6][26] have indicated that automated testing can be used to drive design in all the different kinds of software. We can design a specific function to pass the tests and modify it over and over against the tests. With automated testing, we can quickly execute the tests whenever a code change is made. Most likely, we can execute tests hundreds of times while building the application. The tests run very fast, do not take work to setup and no need to access to a database.

So based on the above knowledge, automated testing should also be able to drive the design of new functionality in Bikube 2011. However, manual testing cannot drive the design of new functionality in Bikube since the manual tests cannot be designed before the application is finished.

5 Recommendations

We have noticed that automated testing has many advantages from the literature reviews. So is it really worthwhile to implement automated testing during the development of Bikube? Should Asplan Viak use automated testing when they design or upgrade a new version of Bikube?

The results evaluation suggested that it is worthwhile to implement automated testing due to the high execution speed, easily-raised code coverage, possible low cost and driven-design. Automated testing can also access to all the layers to make the testing more thorough. Furthermore, our team size can efficiently increase the cooperation between the developers and the testers.

In order to even better implement the automated testing, we are going to give some recommendations based on the case study.

- **Organizing the tests**

The test cases should be always designed first based on the requirements and then coming along with the scripts. How good the tests are organized can directly influence the implementation and maintenance. Studies indicate that about half of the testers' task during software maintenance is understanding the code if they do not design the tests by themselves [8]. Good test cases can explain the code rather than uses descriptive words.

- **Combining automated testing with manual testing**

Automated tests are not a replacement for the manual testing. Although sometimes it does not cost much just to finish one or two automation tasks, not every test is suitable to be automated. Writing automated tests scripts is very time consuming and requires maintenance. Automating everything is not a good idea.

A proper mixture of both manual testing and automated testing can help us achieve detecting defects more efficiently and cost less. In practice, the detection of functional defects at the system level is still largely dependent on the contribution of human testers doing manual testing. Human is definitely more creative than machine, so most of the new defects are found by manual testing, and automated testing is suitable for taking over simple and repetitive tasks from human testers in order to free up time for creative manual testing, not to replace it [13,18].

In Bikube 2011, we suggest that testing the important functions in the service layer using automated testing in order to reach a high code coverage of functions/methods, which can give us high confidence on how well the functions/methods can be performed. In the data access layer, the automated testing can well access to the database and verify the connection. In the user interface layer, we use automated testing in the most often used /important

features. And we use the manual testing to explore testing since human is much more creative than the machine.

The team should strive for a high degree of automation and use manual testing where human judgment and interpretation is most beneficial.

- **Adequate automated tests**

The testing effort should match the desired system quality. In order to achieve adequate number of automated tests, it is important to well define the requirement-bounded test suites and then decide whether or not we need to automate them. We need to have collaboration with business and technical stakeholders to make sure there is a shared understanding of requirements.

Not only can the requirement influence the number of automated tests, but also the execution frequency. We should automate the parts which require frequent execution. From the above results evaluation, a test which will be run over 15 times is a good candidate to be automated. Automating everything is nice but not necessary; there is no point to automate a test if it will only be used once.

- **Start testing in the early stage**

We test Bikube at a late stage which means most of the code has been already designed. Automated testing will even be better if it penetrates through each stage of the development from the beginning. The code coverage can be higher when the tests are designed with corresponding functions which makes Bikube more reliable. Starting testing in the early stage allows us run automated testing constantly from the very beginning rather than once at the end of the development. We discover defects immediately when it is least expensive to fix. Thus, we always have control of the quality of the product during the whole development life cycle.

- **Choosing suitable tools and frameworks**

Automated testing is good only with proper tools and frameworks. Choosing right tools and frameworks at the beginning of the project is essential for us to work more efficient during the implementation and improve the performance of testing. Especially, we do not want to change to a new tool in the middle of development just because we suddenly find the old one is not suitable. Changing to a new automation tools will cost company tremendous money.

In the case study, we have spent much time on selection tools which is one of the keys lead us to perform good automated testing. The basic testing platform is Visual Studio. And we select Telerik test plug-in from dozens of automated testing tools. In the user interface layer, Telerik test plug-in can perfectly catch the mouse actions and verify the components, which saved us lots of time for not coding. We also introduced two automated testing frameworks in order to better design the tests and make the tests easy to be maintained.

Some additional tools are used for the purpose of making tests easy to handle. For example, TestDriven.NET makes it easy to run automated tests and code coverage with a single click.

6 Discussion

The case study enables us to verify the hypotheses and leads us to the recommendations. Some of the hypotheses are false; we would like to find out why and how it can influence the recommendations. In addition, there are other aspects which we would like to discuss based on the methodology that we have chosen in this thesis. We would like to find out whether or not each part in the methodology is good and enough for leading us to the recommendations.

6.1 Verification of Hypotheses Results

From the evaluation, we noticed that not all the hypotheses are true. Some results from the case study support an alternative theory. However, the alternative theory is more like a modification of the hypotheses than a opposite theory. So, our recommendation is still supporting the implementation of automated testing in Bikube 2011 as well as using the automated testing in other similar projects under some certain conditions. We would like to discuss the results evaluation in terms of organizing tests, team size and cost in order to better explain why automated testing is worthwhile to use.

➤ Organizing tests

The test cases of automated testing in the case study are different from the ones of manual testing in Bikube 2011. So all the test cases have to be redesigned when we implement the automated testing. Discarding the old manual test cases may worry the company since they have to redesign the test cases from the beginning, which takes time. However, from the knowledge in chapter 3, the manual test cases have to be redesigned anyway since they are too general and difficult to be executed the same way every time.

➤ Team size

During the case study, we find a positive effect of being a small team. A small team size not only reduces the communication difficulty between developers and testers but also improve the work efficiency. The automated testing in our team is highly feasible.

➤ Cost

Our hypothesis suggests that automated testing is more expensive to use than manual testing. During the case study, we found out that the cost of automated testing was very high at the beginning because extra training and scripts coding are required which consume lots of time. However, training is a one time thing for the same type of software when we implement the automated testing. The scripts coding skills will also improve when we wrote more tests. The cost of automated has a high start but tends to be lower and lower. Although automated testing is more expensive than manual testing at the beginning, after the breakeven point, the cost of automated testing keeps in a stable level while manual testing grows linearly.

6.2 Performance of Methodology

We start with literature review. There are so many related papers which are presented in the research work, the specific papers that we chose for this thesis are based on our problem. Our problem is in software testing domain, more specifically, automated testing. The opposite notion to automated testing is manual testing. So we emphasised the relationship and difference between manual testing and automated testing in the literature review. Furthermore, we learned the lessons and tips of implementing automated testing from other people's work.

In the current state of Bikube 2011, we analysed the manual testing which has already been done in Bikube. We focused on the weak points of current manual testing in order to see if automated testing can make the weak points stronger in the case study.

How we come up with the hypotheses? The hypotheses are highly based on the literature review and the current state of Bikube 2011. We made our assumptions by following the logic of testing process. We started with the elements which can possibly influence the implementation procedure suggested by literature review.

There are many ways to give the hypotheses, we focus on

Usually testing software starts with unit testing, following by function testing and ending with acceptance testing, a procedure of testing from the small parts to the big parts. However, in the case study, we conduct the testing based on different layers because we focused on the feasibility of automated testing instead of specific order. The testing which has been conducted in the service layer and the data access layers is function testing. And the acceptance testing is more likely in the user interface layer.

From the case study, we can see that each layer was tested separately and the integration between each layer is not specified. Actually the integration has already been injected into the tests. When we tested the methods in the service layer, some of them will call the data access layer, instead of using mocking, we can retrieve the data from the database, then this tests can be regarded as integration testing between service layer and data access layer but done in the service layer. So the integration is been covered inside some of the tests and we just need to changing a way of doing it.

When we execute tests, they are run in a specific order instead of random. If tests are dependent on one another, the order is typically fixed. For example, the 'changing user profile' test must follow after the 'log in' test. If tests are independent, then their order is not important since they are self-contained and they do not impact other tests. There is no obvious advantage gained by randomizing the order of tests.

7 Conclusion and Future Work

In this thesis, we try to figure out whether or not it is worthwhile to implement automated testing in the document management system Bikube 2011. The reason for doing this is that we hope automated testing would offer some good features like fast execution and better code coverage which we cannot gain from the current software testing in Bikube 2011.

We gave some hypotheses against the problem from different aspects. The hypotheses were based on literature review and current testing state in Bikube 2011. In order to solve the problem, we conducted a case study. In the case study, we offered a reasonable procedure to let our team implement automated testing. By following the procedure, we successfully implemented automated testing in all the layers in the Bikube architecture. Then we evaluated the results of the case study.

The results show that automated testing may not offer very high code coverage at the beginning but it is easy to increase it since the tests are organized in a manner that we can easily identify the relationship between codes and tests. We can run automated tests in less than one minute without supervision which reduces the development iterations. Possible high code coverage and fast execution speed give us more confidence on each stage of the product development life cycle. The results also shows that automated testing somehow can drive the design of the software. As long as we well control the cost of automated testing by only automating the suitable tests and identify the breakeven point, automated testing can cost less than manual testing.

In summary, the major perceived benefits of automated testing include quality improvement through a better test coverage, and that more testing can be done in less time. However, the case study indicated that to achieve a better test coverage and less cost, automation alone is not enough, but human involvement is needed in the selection of test cases. Based on the case study, we suggested that automated testing together with our recommendations can be implemented in the development of Bikube 2011.

Our automated tests are initiated manually, there exists some risk that they are not being run regularly and therefore may in fact be failing at some point. Therefore, our future focus will be continuous integration (CI) to ensure that all the automated tests can run regularly in order to realize even more cost and rapid feedback benefits.

It is difficult to write error-free code. What we can do is trying to identify these errors as early as possible and reducing the amounts of errors as much as possible. Finding errors early is a lot more cost effective that finding them later. We can use automated testing to achieve these requirements, thus we can save a lot of time and money. With a nearly error-free software, the company can maximize the value of the software.

Reference

- [1] Jeff Levinson, “Software Testing with Visual Studio 2010”, March, 2011
- [2] <http://sharepoint.microsoft.com>, February. 13, 2012
- [3] Charette, R.N., “Why software fails”, Spectrum, IEEE, 42(9), page: 42 - 49 ,2005
- [4] Goutam Kumar Saha. “Understanding software testing concepts” February 2008
- [5] Edward Kit, “Software testing in the real world”,1995
- [6] Kolawa, Adam; Huizinga, Dorota (2007). “Automated Defect Prevention: Best Practices in Software Management”. Wiley-IEEE Computer Society Press. p. 74. ISBN 0470042125
- [7] Woi Hin, Kee, “Future Implementation and Integration of Agile Methods in Software Development and Testing”. Innovations in Information Technology, 2006
- [8] J. Bach, "Test Automation Snake Oil," 1999
- [9] A. Bertolino, "Software Testing Research: Achievements, Challenges, Dreams," in Future of Software Engineering: IEEE Computer Society, 2007, pp. 85-103.
- [10] ZHU Xiaochun, ZHOU Bo, LI Juefeng, GAO Qiu. “A Test Automation Solution on GUI Functional Test”
- [11] C. Persson and N. Yilmaztürk, "Establishment of Automated Regression Testing at ABB: Industrial Experience Report on ‘Avoiding the Pitfalls’," in 19th International Conference on Automated Software Engineering (ASE’04): IEEE Computer Society, 2004.
- [12] Katja Karhu, Tiina Repo, Ossi Taipale, Kari Smolander. ”Empirical Observations on Software Testing Automation”
- [13] S.P. Ng, T. Murnane, K. Reed,D. Grant, and T.Y. Chen, “A preliminary survey on software testing practices in Australia”, Software Engineering Conference, 2004. Proceedings. 2004. Australian, 2004 Page(s):116 – 125
- [14]A Test Automation Solution on GUI Functional Test
- [15] M. Fewster, "Common Mistakes in Test Automation," Grove Consultants 2001.
- [16] Dorothy Graham; Mark Fewster. “Experiences of Test Automation: Case Studies of Software Test Automation”. Addison-Wesley Professional. January 09, 2012
- [17] Janzen, D, Saiedian, H, “Test-Driven Development: Concepts, Taxonomy, and Future Direction”, Computer, Volume: 38, Issue:9 ,page: 43 – 50, 2005
- [18] Andersson, C. and P. Runeson, “Verification and validation in industry - a qualitative survey on the state of practice,” Proceedings of International Symposium on Empirical Software Engineering, 2002, pp. 37-47.
- [19] http://en.wikipedia.org/wiki/Test_automation.
- [20] <http://behaviour-driven.org/> 20.April.2012
- [21]Aaron Jensen,<http://codebetter.com/aaronjensen/2008/05/08/introducing-machine-specifications-or-mspec-for-short/> 20.April.2012
- [22] <http://elegantcode.com/2010/02/19/getting-started-with-machine-specifications-mspec/>
- [23] <http://code.google.com/p/ndbunit/>
- [24] <http://docs.ncover.com/best-practices/code-quality-metrics/>,August.1.2011
- [25] <http://hibernatingrhinos.com/open-source/rhino-mocks>

- [26] David S. Janzen, Hossein Saiedian, “Does Test-Driven Development Really Improve Software Design Quality?”, *Software, IEEE*, Volume: 25, Issue:2, page: 77 – 84,2008
- [27] <http://www.microsoft.com/visualstudio/en-us/products/2010-editions/ultimate/overview>
- [28] http://msdn.microsoft.com/en-us/library/ff649520.aspx#mtf_ch02_softwaredevelopment

Appendix A - Glossary

Exploratory Testing : can be regarded as a blind test. The tester does not have specifically planned Test Cases, but one goal of exploring the software features and discovering unknown bugs. Testers may use their imagination to find all the possible scenarios. Exploratory testing is only test type that can help in discovering hidden bugs.

Regression Testing : is the process to test changes to software programs to make sure that the older code still works with the new changes (for example, bug fixes or new functionality) that have been made. Regression testing is a normal part of the application development process. These regression tests ensure that subsequent changes to the code do not break sections that already work [27].

Requirement specification: is a complete description of the behavior of a system to be developed.

Software development life cycle: is a structure imposed on the development of a software product.

Sharepoint: designed to cater to web requirements common for most organizations.

































Waterfall Model: is a 'top down' approach through the phases of requirement analysis, design, implementation, testing, integration and maintenance without revisiting the steps. The defects cannot be detected until and unless the testing stage is reached.

Extreme Programming : follows neither a purely sequential approach nor a purely iterative approach, the preliminary design work is reduced into small and simple identified tasks and feedback of the software is returned as soon as it is developed [28].

Return on Investment (ROI): the benefit (return) of an investment is divided by the cost of the investment; the result is expressed as a percentage or a ratio.

Appendix B - User Story

User story in Telerik Test Studio: any login user can create a 'Forespørsel'

1	<input checked="" type="checkbox"/>	 Navigate to : 'http://localhost:54321/'	▼
2	<input checked="" type="checkbox"/>	 Click 'NyTableCell'	▼
3	<input checked="" type="checkbox"/>	 Click 'ForespørselSpan'	▼
4	<input checked="" type="checkbox"/>	 Enter text 'test' in 'GenereltOppdragNavnText'	▼
5	<input checked="" type="checkbox"/>	 Enter text 'gri' in 'SearchKundeComboboxInputText'	▼
6	<input checked="" type="checkbox"/>	 Wait for element 'Span' 'is' visible.	▼
7	<input checked="" type="checkbox"/>	 Click 'MEiendomASListItem'	▼
8	<input checked="" type="checkbox"/>	 Enter text 'This is a test' in 'GenereltBeskrivelseTextArea'	▼
9	<input checked="" type="checkbox"/>	 Click 'OpenTheSpan'	▼
10	<input checked="" type="checkbox"/>	 Verify element 'Div' 'is' visible.	▼
11	<input checked="" type="checkbox"/>	 Click 'WednesdayLink'	▼
12	<input checked="" type="checkbox"/>	 Click 'SelectSpan'	▼
13	<input checked="" type="checkbox"/>	 Wait for element 'Div0' 'is' visible.	▼
14	<input checked="" type="checkbox"/>	 Click 'SørlandetListItem'	▼
15	<input checked="" type="checkbox"/>	 Click 'SelectSpan0'	▼
16	<input checked="" type="checkbox"/>	 Wait for element 'VelgListItem' 'is' visible.	▼
17	<input checked="" type="checkbox"/>	 Click 'AustAgderListItem'	▼
18	<input checked="" type="checkbox"/>	 Click 'SelectSpan1'	▼
19	<input checked="" type="checkbox"/>	 Wait for element 'Div1' 'is' visible.	▼
20	<input checked="" type="checkbox"/>	 Click 'GRIMSTADListItem'	▼
21	<input checked="" type="checkbox"/>	 Click 'SelectSpan2'	▼
22	<input checked="" type="checkbox"/>	 Wait for element 'Div1' 'is' visible.	▼
23	<input checked="" type="checkbox"/>	 Click 'BoligerListItem'	▼
24	<input checked="" type="checkbox"/>	 Click 'FullførLink'	▼
25	<input checked="" type="checkbox"/>	 Click 'OppretteNyOppdragSubmitSubmit'	▼
26	<input checked="" type="checkbox"/>	 Wait for element 'NyOppdragCreateStatusMessageDiv' 'is' visible.	▼
27	<input checked="" type="checkbox"/>	 Verify 'TextContent' 'Contains' 'Melding sendt.' on 'MeldingSendtDiv'	▼
28	<input checked="" type="checkbox"/>	 Click 'OppdragImage'	▼
29	<input checked="" type="checkbox"/>	 Click 'MineSpan'	▼
30	<input checked="" type="checkbox"/>	 Verify 'TextContent' 'Contains' 'test' on 'TestTableCell'	▼
31	<input checked="" type="checkbox"/>	 Verify 'TextContent' 'Contains' 'Lin Egholm' on 'LinEgholmTableCell' 	▼

The code behind the above user story.

```
namespace BikubeTestProject
{
    [TestMethod()]
    public void CreateForespørsel()
    {
        // Launch an instance of the browser
        Manager.LaunchNewBrowser();

        // Navigate to : 'http://localhost:54321/'
        ActiveBrowser.NavigateTo("http://localhost:54321/");
    }
}
```

```

// Click 'NyTableCell'
Pages.SisteOppdrag.NyTableCell.Click(false);

// Click 'ForespørselSpan'
Pages.Ny.ForespørselSpan.Click(false);

// Enter text 'test' in 'GenereltOppdragNavnText'
Pages.NyForespørsel.GenereltOppdragNavnText.Wait.ForExists(10000);
Pages.NyForespørsel.GenereltOppdragNavnText.Text = "test";

// Enter text 'gri' in 'SearchKundeComboboxInputText'

Pages.NyForespørsel.SearchKundeComboboxInputText.ScrollToVisible(ArtOfTest.WebAii.Core.ScrollToVisibleType.ElementTopAtWindowTop);
ActiveBrowser.Window.SetFocus();
Pages.NyForespørsel.SearchKundeComboboxInputText.MouseClick();
Manager.Desktop.Keyboard.TypeText("gri", 50, 100);

// Wait for element 'Span' 'is' visible.
Pages.NyForespørsel.Span.Wait.ForVisible();

// Click 'MEiendomASListItem'
Pages.NyForespørsel.MEiendomASListItem.Click(false);

// Enter text 'This is a test' in 'GenereltBeskrivelseTextArea'
Pages.NyForespørsel.GenereltBeskrivelseTextArea.Wait.ForExists(10000);
Pages.NyForespørsel.GenereltBeskrivelseTextArea.Text = "This is a test";

// Click 'OpenTheSpan'
Pages.NyForespørsel.OpenTheSpan.Click(false);

// Verify element 'Div' 'is' visible.
HtmlDiv Div = Pages.NyForespørsel.Div;
Assert.IsTrue(Div.IsVisible());

// Click 'WednesdayLink'
Pages.NyForespørsel.WednesdayLink.Click(false);

// Click 'SelectSpan'
Pages.NyForespørsel.SelectSpan.Click(false);

// Wait for element 'Div0' 'is' visible.
Pages.NyForespørsel.Div0.Wait.ForVisible();

```

```
// Click 'SørlandetListItem'
Pages.NyForespørsel.SørlandetListItem.Click(false);

// Click 'SelectSpan0'
Pages.NyForespørsel.SelectSpan0.Click(false);

// Wait for element 'VelgListItem' 'is' visible.
Pages.NyForespørsel.VelgListItem.Wait.ForVisible();

// Click 'AustAgderListItem'
Pages.NyForespørsel.AustAgderListItem.Click(false);

// Click 'SelectSpan1'
Pages.NyForespørsel.SelectSpan1.Click(false);

// Wait for element 'Div1' 'is' visible.
Pages.NyForespørsel.Div1.Wait.ForVisible();

// Click 'GRIMSTADListItem'
Pages.NyForespørsel.GRIMSTADListItem.Click(false);

// Click 'SelectSpan2'
Pages.NyForespørsel.SelectSpan2.Click(false);

// Wait for element 'Div1' 'is' visible.
Pages.NyForespørsel.Div1.Wait.ForVisible();

// Click 'BoligerListItem'
Pages.NyForespørsel.BoligerListItem.Click(false);

// Click 'FullførLink'
Pages.NyForespørsel.FullførLink.Click(false);

// Click 'OppretteNyOppdragSubmitSubmit'
ActiveBrowser.Window.SetFocus();

Pages.NyForespørsel.OppretteNyOppdragSubmitSubmit.ScrollToVisible(ArtOfTest.WebAii.Core.ScrollToVisibleType.ElementTopAtWindowTop);
Pages.NyForespørsel.OppretteNyOppdragSubmitSubmit.MouseClick();
```

```
// Wait for element 'NyOppdragCreateStatusMessageDiv' 'is' visible.
Pages.NyForespørsel.NyOppdragCreateStatusMessageDiv.Wait.ForVisible();

// Verify 'TextContent' 'Contains' 'Melding sendt.' on 'MeldingSendtDiv'
Pages.NyForespørsel.MeldingSendtDiv.AssertContent().TextContent(ArtOfTest.Common.StringCompareType.Contains, "Melding sendt.");

// Click 'OppdragImage'
Pages.Ny.OppdragImage.Click(false);

// Click 'MineSpan'
Pages.SisteOppdrag.MineSpan.Click(false);

// Verify 'TextContent' 'Contains' 'test' on 'TestTableCell'
Pages.MineOppdrag.TestTableCell.AssertContent().TextContent(ArtOfTest.Common.StringCompareType.Contains, "test");

// Verify 'TextContent' 'Contains' 'Lin Egholm' on 'LinEgholmTableCell'
Pages.MineOppdrag.LinEgholmTableCell.AssertContent().TextContent(ArtOfTest.Common.StringCompareType.Contains, "Lin Egholm");

    }
}
}
```

Appendix C - Sample Code

The sample code of the test case **Click filter pane ‘Nr’** is presented below. The verification of this test is generated by writing code instead of action recording.

```
using System.Collections.Generic;
using ArtOfTest.WebAii.Controls.HtmlControls;
using ArtOfTest.WebAii.Core;
using ArtOfTest.WebAii.TestTemplates;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace BikubeTestProject
{
    /// <summary>
    /// Summary description for ClickNrPaneUnitTest
    /// </summary>
    [TestClass]
    public class ClickNrPane : BaseTest
    {
        #region [Setup / TearDown]
        private TestContext testContextInstance = null;
        /// <summary>
        /// Gets or sets the VS test context which provides
        /// information about and functionality for the
        /// current test run.
        /// </summary>
        public TestContext TestContext
        {
            get
            {
                return testContextInstance;
            }
            set
            {
                testContextInstance = value;
            }
        }

        //Use ClassInitialize to run code before running the first test in the class
        [ClassInitialize()]
        public static void MyClassInitialize(TestContext testContext)
        {
        }

        private Pages _pages;

        public bool verifyOrder(Pages p)
        {
            Pages.AlleOppdrag.NrLink.Click();

            ArtOfTest.WebAii.Controls.HtmlControls.HtmlTable oppdragTable =
            p.AlleOppdrag.Get<ArtOfTest.WebAii.Controls.HtmlControls.HtmlTable>("cellspacing=0", "tagIndex=table:2
            0", "tagname=table");

            Log.WriteLine(oppdragTable.InnerText);

            int r = oppdragTable.Rows.Count;
        }
    }
}
```

```

List<string> list = new List<string>();

bool result = true;

for (int i = 1; i < r; i++)
{
    HtmlTableRow row = oppdragTable.Rows[i];

    HtmlTableCell cell = row.Cells[4];

    list.Add(cell.TextContent);

    Log.WriteLine(cell.TextContent);
}

for (int j = 1; j < list.Count; j++)
{
    if (j + 1 == list.Count)
    {
        break;
    }

    else
    {
        if (list[j + 1].CompareTo(list[j]) < 0) result = false;
    }
}
return result;
}
public Pages Pages
{
    get
    {
        if (_pages == null)
        {
            _pages = new Pages(Manager.Current);
        }
        return _pages;
    }
}
// Use TestInitialize to run code before running each test
[TestInitialize()]
public void MyTestInitialize()
{
    #region WebAii Initialization

    // Initializes WebAii manager to be used by the test case.

    Initialize(false, this.TestContext.TestLogsDir, new
TestContextWriteLine(this.TestContext.WriteLine));

    SetTestMethod(this, (string)TestContext.Properties["TestName"]);

    #endregion
}

// Use TestCleanup to run code after each test has run
[TestCleanup()]
public void MyTestCleanup()

```

```

{
    #region WebAii Cleanup

    // Shuts down WebAii manager and closes all browsers currently running
    // after each test. This call is ignored if recycleBrowser is set
    this.Cleanup();

    #endregion
}

//Use ClassCleanup to run code after all tests in a class have run
[ClassCleanup()]
public static void MyClassCleanup()
{
    // This will shut down all browsers if recycle Browser is turned on. Else
    // will do nothing.
    ShutDown();
}

#endregion

[TestMethod()]
public void ClickNrPane()
{
    // Launch an instance of the browser
    Manager.LaunchNewBrowser(BrowserType.InternetExplorer);

    // Navigate to : 'http://localhost:54321/Home/All'
    ActiveBrowser.NavigateTo("http://localhost:54321/Home/All");

    // Click 'NrLink'
    Pages.AlleOppdrag.NrLink.Click();

    // Verify the column is started ascending
    Assert.AreEqual(true, verifyOrder(Pages) );
}
}
}

```


Appendix D - Log Trace

Overall Result: Fail

```
-----
'20.03.2012 09:04:38' - Using .Net Runtime version: '4.0.30319.261' for tests execution.
'20.03.2012 09:04:38' - Starting execution...
'20.03.2012 09:04:42' - Detected custom code in test. Locating test assembly: BikubeTestProject.dll.
'20.03.2012 09:04:42' - Assembly Found: C:\Users\lin.egholm\Documents\Test Studio
Projects\BikubeTestProject\TestResults\lin.egholm_AV2000 2012-03-20 09_04_38\Out\BikubeTestProject.dll
'20.03.2012 09:04:42' - Loading code class: 'BikubeTestProject.Click_Kunde_icon'.
-----
'20.03.2012 09:04:42' - Using 'InternetExplorer' version '9.0' as default browser.
'20.03.2012 09:05:00' - 'Pass' : 1. Navigate to : 'http://localhost:54321/Home/All'
'20.03.2012 09:05:00' - 'Pass' : 2. Click 'KundeTableHeader'
'20.03.2012 09:05:06' - 'Fail' : 3. Verify 'InnerText' 'StartsWith' 'A' on 'SpareBank1TableCell'
-----
Failure Information:
~~~~~
Content.InnerText of 'SpareBank1TableCell' does not match!

Match Type: 'StartsWith'
Expected Result: 'A'
Value at time of failure: 'SpareBank 1 SMN Kvartalet AS'
-----
'20.03.2012 09:05:06' - Detected a failure. Step is marked 'ContinueOnFailure=False' aborting test execution.
-----
'20.03.2012 09:05:06' - Overall Result: Fail
'20.03.2012 09:05:06' - Duration: [0 min: 23 sec: 676 msec]
-----
'20.03.2012 09:05:07' - Test completed!
```

Appendix E - Gantt Chart

During the project, the following Gantt chart has been used to illustrate the project schedule. The project started in January, and finished by the end of May.

#	Task	Description	Dur	2012				
				Jan	Feb	Mar	Apr	May
	Master Thesis		142	[Gantt bar spanning Jan to May]				
1	Introduction		4	[Gantt bar in Jan]				
1.1	Background of the thesis	Describe why and how Asplan Viak is interested in this topic.	3	[Gantt bar in Jan]				
1.2	Importance of the topic	Describe how this topic can benefit academic research and real industry.	2	[Gantt bar in Jan]				
1.3	Problem Definition	What should be solved in the thesis? And what is our goal?	1	[Gantt bar in Jan]				
2	Related Work		32	[Gantt bar spanning Jan to Feb]				
2.1	Automatic and Manual	Giving the definition of these two test types as well as the advantage and disadvantage. Explaining why automatic testing draws much attention.	13	[Gantt bar in Jan]				
2.2	Test-first and Test-last	Giving the definition of these two testing methodologies and how they are related with automatic testing. Looking through previous similar studies to help us to determine which one is better for our application in terms of team scale, application levels and software requirements specification.	20	[Gantt bar spanning Jan to Feb]				
3	Case Study		80	[Gantt bar spanning Feb to Apr]				
3.1	Define Hypotheses	What are our possible hypotheses against the problem?	2	[Gantt bar in Feb]				
3.2	Case Study Design	Defining the procedure. How should the case study be conducted? How many subjects are involved? What are possible case study variables that we cannot control?	4	[Gantt bar in Feb]				
3.3	Analyzing Bikube 2011	Finding out what has been already done and how is the result.	6	[Gantt bar in Feb]				
3.4	Understanding the Testing Tool	Studying how to use the testing tool to perform and code automatic testing.	9	[Gantt bar in Feb]				
3.5	Implementation	Using automatic testing to test some modules in software and get feedback from developers.	60	[Gantt bar spanning Feb to Apr]				
3.6	Validation	What is the result?	20	[Gantt bar spanning Apr to May]				
4	Discussion and Conclusion		46	[Gantt bar spanning Apr to May]				
4.1	Results analysis	Verifying the hypotheses are true or false and explaining why. How well does our solution solve the problem or satisfy the requirements?	26	[Gantt bar spanning Apr to May]				
4.2	Conclusion	Briefly outline what we have done, and finally make a statement that we have actually solved the problem or proved or disproved the hypothesis.	6	[Gantt bar in May]				
4.3	Revise	Reading through and fixing the report.	14	[Gantt bar in May]				
5	Report	Writing report	142	[Gantt bar spanning Jan to May]				