Master Thesis
Software Engineering
Thesis no: MSE-2007:18
June 2007

An automated testing strategy targeted for efficient use in the consulting domain

Teddie Stenvi

School of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

This thesis is submitted to the School of Engineering at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:
Author(s):
Teddie Stenvi
Address: Jaktstigen 18
22652 Lund
E-mail: teddie@stenvi.se

External advisor(s):
Per Sigurdson
Testway AB
Address:
Hans Michelsengatan 9
211 20 Malmö

University advisor(s):
Dr. Robert Feldt
Department of Systems and Software Engineering

School of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Internet : www.bth.se/tek
Phone : +46 457 38 50 00
Fax : + 46 457 271 25

ABSTRACT

Test automation can decrease release cycle time for software systems compared to manual test execution. Manual test execution is also considered inefficient and error-prone. However, few companies have gotten far within the field of test automation. This thesis investigates how testing and test automation is conducted in a test consulting setting. It has been recognized that low test process maturity is common in customer projects and this has led to equally low system testability and stability. The study started with a literature survey which summarized the current state within the field of automated testing. This was followed by a consulting case study. In the case study it was investigated how the identified test process maturity problems affect the test consulting services. The consulting automated testing strategy (CATS) been developed to meet the current identified challenges in the domain. Customer guidelines which aim to increase the test process maturity in the customer organization have also been developed as a support to the strategy. Furthermore, the study has included both industrial and academic validation which has been conducted through interviews with consultant practitioners and researchers.

Keywords: Consulting, Testing, Requirements, Process Improvement.

TABLE OF CONTENTS

1 INTRODUCTION

Software testing is a practice that is neglected in many development projects due to budget and time constrains. In the test consulting domain, the testers and test managers change domains frequently due to large sets of customers involved. This chapter will present the motivation for this thesis project followed by the aims and objectives and research questions. The research methodology will be briefly introduced followed by an outline for the rest of the report.

1.1 Background

Executing manual test cases several times is inefficient and error-prone and by automating these, the tests can be improved in later development phases, resources may be freed and the release cycle time may be decreased [Keller05]. Acting as a consultant in the test consulting domain infers some special issues that need to be handled in regards to the automation of the manual test cases in the customer development projects. The development process maturity often differ between the customers and with this in mind, the automated test procedures, methods and approaches used by the consulting firms must be adapted to suit the different customer domains and the distinct projects within these domains.

If automated testing is not considered in the architecture and design, it will be decrease the possibilities of automating the test cases in the later phases [Keller05]. This can pose problems for a test consultant that arrives in late phases of development where these items are hard to change for the sake of automating the test cases. As mentioned by Keller et al. [Keller05], the success of the automated tests are dependent on the test automation strategy that describes which test types that are to be performed, such as for example, integration tests, reliability tests and functional tests.

There are development methodologies that support automated testing, such as test driven development. Such practices can in fact reduce the defects in the software products and this is partly because it enables automated test cases to be written before the actual problem solution implementation [Williams03]. However, the consulting domain differs from traditional software development in the sense the consultants arrive in various phases of development depending on the contract with the given customer. It would hence be an advantage if the consultant could guide the early development phases in a direction which would facilitate automated testing in the later phases when the consultant arrives.

With such guidance, executable test frameworks, such as the unit testing framework JUnit [Noonan02], could be introduced in the early stages of development which could help in the early detection of defects. This would also facilitate the regression testing that is needed after a change has been made in the software artefacts which in turn save the effort and cost of manual re-testing. In many software disciplines, the possibility of artefact reuse is discussed as a means of decreasing the development costs with the advantage of increased quality in regards to the iterated improvements made to the reused artefact. Such reuse could be enabled with the introduction of automated test cases which could be beneficial in the sense that the consultant could gather a test case collection and thereby bring the test cases from one customer to another.

Automated testing is not the best verification technique for every single scenario, many other factors needs to be considered before making the decision to automate the test case such as what artefact that are to be tested, how many times the test are to be run and how long time it will take to implement the test suite [Keller05]. However, having them gives the advantage of being able to run them more frequent and improves the quality of the test cases.

As mentioned, it is very difficult to add automated test cases in late development phases in projects which have not taken automation into account in the architecture and design. In traditional software development organisations it would be possible to change the development method to for example, test-driven development in order to prepare automated test cases in the early phases. Such change would open up the possibility of introducing executable test frameworks which in turn could help to find errors in the early stages of development. As the hired test consultant, this is not possible to the same extent whereas the consultant often arrives in a phase where the development artefacts have already been produced which makes it feasible to adapt the traditional automated testing practices to cope with this situation.

Few of the customers of these consulting firms have gotten far in the field of test automation which introduces a gap between the state-of-the-art research of test automation and the industrial implementation of such. This thesis investigates how the traditional automated testing practices can be adapted in these kinds of situations and also examines if it is possible to guide the customers, which have not gotten very far in the field of automation, in their early phases of development in a direction to facilitate automated testing in the phase where the consultant arrives.

1.2 Aims and objectives

This aim of this thesis project was to report on the difficulties within the test consulting domain in regards to the automated test methods and processes used. With this information in mind, an automated testing strategy and customer guidelines has been constructed with the aim of making these methods and processes more adaptable between different customer domains. The objectives which were formed prior to the study are primarily described in the list below:

- Identify which automated testing methods, approaches and strategies that are used in the consulting domain.
- Identify how these automated testing methods, approaches and strategies differ from the corresponding ones used by standard development companies and the ones considered state-of-the-art.
- Construct a theoretical hybrid strategy for automated testing, targeted for efficient adaptation in the consulting domain, with guidelines for easier adoption.
- Validate the adaptation efficiency of the strategy in the consulting domain.
- Validate the feasibility and cost effectiveness of the proposed strategy in the consulting domain.

1.3 Research questions

With the aims and objectives in mind, the following set of research questions was constructed:

RQ1: Which testing methods, approaches and strategies for automated testing are considered state-of-the-art?

RQ2: What automated testing methods, approaches and strategies are currently used by testing consulting firms?

RQ3: How do the testing and test processes for consulting firms differ from the corresponding ones used by traditional software development organisations?

RQ4: What common factors of these can be identified for effective use across different customer domains?

RQ5: Are there potential for reuse of automated test cases between different testing consulting clients and domains?

RQ6: What problems exists in regards to testability in customer projects?

RQ7: How can the automated testing methods, approaches and strategies be transformed and combined in order to be more flexible in the dynamic environments of consulting firms?

1.4 Research methodology

In order to get a sufficient amount of information, the study has been divided into three main parts where each will form a part of the report;

- Literature survey.
- Case study.
- Validation.

An extensive literature study has been conducted which was indented for the identification of which automated testing methods, approaches and practices are considered state-of-the-art. This study was indented to answer some of the research questions which were directed at the comparison to the results spawned by the case study.

The industrial case study included interviews, surveys and questionnaires. The interviews of this case study were performed with company personnel at different levels in the test consulting organisation. This was done in order to get the views from a tester in a specific project as well as a test manager which act over several projects. With the combined results from these activities, sufficient information was acquired for the construction of the strategy and guidelines.

The last phase of the study was the validation of the strategy and guidelines in the consulting domain. This validation was performed through interviews with a consultant testers and test managers of the consulting firm where the industrial case study was performed. Furthermore, a validation interview was performed with a customer of the consulting firm. These interviews were conducted in order to assess the estimated efficiency and feasibility of the strategy in a live consulting setting. Furthermore, an interview with a researcher within academia was performed in order to assess the academic value of the study.

1.5 Thesis outline

This section provides the chapter outline of the thesis.

Chapter 2 (Automated Software Testing) begins with an introduction to software testing and basic concepts in Section 2.1 and 2.2. Section 2.3 provides a discussion of verification-oriented development methodologies. The following section (Section 2.4) discusses automated testing opportunities in more depth. Section 2.5 concludes the chapter with a summary and discussion of methods, approaches and strategies that are deemed relevant for the consulting domain.

Chapter 3 (Methodology) contains a discussion about the study design. The sections in this chapter contain flowcharts with attached discussions of each activity conducted throughout the study.

Chapter 4 (Test consulting) introduces the consulting domain in Section 4.1. This is followed by a discussion of the software development and testing differences between consulting

firms and standard development companies in Section 4.2. A case study has been conducted at Testway which is a consulting firm in a southern part of Sweden and Section 4.3 describes the consulting view and services provided by this organization.

Chapter 5 (Consulting Automated Testing Strategy (CATS)) propose an automated testing strategy which has been developed for efficient use in the consulting domain. An overview of the strategy is provided in the Section 5.1. This is followed by sections which describe the core phases of the strategy; Section 5.2 (Preparation phase), Section 5.3 (Execution phase) and Section 5.4 (Post execution phase). As a concluding part of the chapter (Section 5.5), a couple of pitfalls which could be avoided when applying the strategy is introduced and discussed.

Chapter 6 (Customer Guidelines) propose customer guidelines which are developed as a complement to the automated testing strategy mentioned above. The aim of these is to facilitate system and acceptance testing in the customer development projects. The chapter starts with an introduction to the guidelines in Section 6.1. Since the current main challenges are related to requirements and lack of early verification activities in the customer projects, the following sections (Section 6.2 and 6.3) give pointers on what should be considered in these two areas in order to increase the system testability and stability.

Chapter 7 (Discussion) starts with an discussion of the lessons learned in Section 7.1 and continue with an validity discussion in Section 7.2 where validity strengths and threats are introduced. This chapter is concluded with a discussion based on the original research questions.

Chapter 8 (Conclusions) draws conclusions based on the thesis results.

Chapter 9 (Future work) gives directions for future work that the author considers relevant based on the current state of the automated testing strategy and customer guidelines.

2 AUTOMATED SOFTWARE TESTING

This chapter introduces some key elements in the field of software testing and provides a summary of what is considered state-of-the-art. An introduction to software testing is given in Section 2.1. There are several development methods that focus on the testing aspects of development; they are covered in Section 2.2. In section 2.3, different levels of testing are discussed which could be used depending on the development status. Of course, there are several advantages of automated testing but also many challenges and these issues will be discussed in Section 2.4. To conclude the chapter, the last section covers state-of-the-art techniques, methods and approaches to testing and particularly automated testing that aim to solve these challenges.

2.1 Software testing in general

In every large software development project, there exist several defects in artefacts such as requirements, architecture, design in addition to the source code, each of which decrease the quality of the product. Software testing practises are used to ensure quality of software items by finding these defects. The overall development cost can be decreased by finding these defects early in the development process rather than later [Lloyd02][Juristo04]. For example, consider performing a bug fix to a set of requirements after the implementation has been completed. When performing such change, the already implemented source code may now be based on an incorrect set of requirements. This means that the existing functionality may not be needed after all, rendering the development effort useless. The longer a defect goes unnoticed, the more software artefacts are being developed in parallel. When the defect finally is discovered, these developed artefacts may need changes as a result which in turn increase the time required for bug fixing. This makes is beneficial to conduct the testing practices continuously throughout all development phases. By finding the defects continuous, this feedback can be delivered to the developer responsible for bug-fix immediately thus limiting the affected artefacts that need to be changed [Saff04a].

Agile development methodologies have evolved which accommodate the need for continuous testing. Traditionally, every development phase produces the complete set of artefacts before proceeding to the next phase. The main distinction between the agile approaches and traditional ones is that the agile projects are broken up into several releases which are given to the customer throughout the project. In agile methodologies, large sets of documentation are also avoided in favour of strong communication within the development team. Since it is hard to maintain such close communication in large teams, these approaches are considered to be better suited for smaller project teams [Merisalo-Rantanen05]. Extreme programming (XP) [Beck99] is an agile methodology which emphasises test-driven development. This simply means that the tests shall drive the development forward and in the case of XP, the testing practices stress the implementation of executable unit test cases.

In many organisations there is a reluctance to adapt testing practices due to a misconception that these practices would increase the cost of development. This is not the case in reality since the maintenance and bug fixing required often produce larger total costs without these practices. The lack of enthusiasm for software testing can decrease when the quality benefits are made more visible to the organisations [Bach01]. Also, in my experience, software developers do not consider writing test cases as productive. This is also a misconception since these tests contribute to the increase in quality while decreasing the total development effort at the same time.

Software testing can roughly be divided into several methods and levels each of which has distinct responsibility of testing [Rakitin01]. The methods include black-box and white-box

testing which is discussed below. Software levels include unit, integration, system and validation testing each of which will be introduced in Section 2.2.

The most commonly cited statement in software testing is probably the one published by Dijkstra in 1972, and will also be cited below because it proves a good point which applies to both the black-box and white-box approach.

"Program testing can be used to show the presence of bugs, but never to show their absence!" [Dijkstra72]

2.1.1 Black-box testing

Often, it can be useful or even necessary to test software without any knowledge about the internal structures of system; this is called doing a black-box testing [Rakitin01]. This type of testing aims to view the system as a black-box where the testers finds defects by trying to make the system behave in a way that does not corresponds to the system specifications [Myers04]. Black-box testing is about achieving a high coverage of the functional requirements which in turn needs to be gathered in one way or another. These requirements could be formalized in system requirements specifications or in the case of more agile approaches the tests could be based on the user stories provided by an on-site customer. In development methodologies that are supposed to base the design on the requirements specifications, such as the waterfall model, poorly written or low amounts of documentation can pose problems. In these cases it is difficult to generate the expected output for the test cases which in turn leads to problems when the results of the tests are to be inspected [Xie06].

Statement coverage is a measure of how many of the code statements that is executed by the executed test cases. Because black-box testing is only concerned with the behavioural issues, the structural concern is neglected which means that statement coverage is not considered at all. In order to achieve this type of coverage, the grey-box and more especially the white-box approach should be used.

2.1.2 White-box testing

Contrary to the black-box method that tests the system without knowledge of the internal structure, this information is known when using the white-box approach [Myers04]. Contrary to the black-box method, the test cases are designed based on the internal statements, branches and paths in the white-box approach [Rakitin01]. With this in mind; a good knowledge of the system design can be beneficial in the construction of test cases.

```
log_file *example(int a, int b) {
        if ( a == 7 && b > 623512 )
                return NULL; /* Should never happen */
        return logger;
}
```

Example 1 – White-box testing example

In Example 1, a function which returns a log structure with the exception when a == 7 and b > 623512 is illustrated. In complex systems, there are many such possible branches which make it difficult to ensure full code coverage since test inputs need to be generated for each and every one of these branches. In fact, in many large scale applications it is simply too time-consuming to run all possible combinations [Myers04]. White-box testing is useful in order to achieve high statement coverage because with this method the code structures are visible, information that can be used when constructing the test inputs. This means that the amount of test vectors can be limited which is a necessary means to decrease the execution time for running an extensive test suite.

The main limitation of the white-box approach is that it only focuses on the implemented structures of the system. To ensure that the requirements are satisfied, the black-box approach should be used. However, the white-box approach is indeed necessary due to the ability of achieving high coverage and combining this method with the grey-box and the black-box approach would be appropriate to get the most complete testing [Cole00].

2.1.3 Grey-box testing

The grey-box method is an uncommonly used concept which is a combination of the black-box and the white-box approach, and the mixture of these colors is also why it is called grey-box [Büchi99]. It has the visibility of the module interfaces which the black-box does not while it do not contain the information about their internal structures which the white-box approach do. With the data structure information, the grey-box testing type is used by methods that act in the integration test level and use the structure design specification to get the acceptable input and output for the interfaces [Sneed04]. The main purpose with the method is to see if the interactions to and from the component interfaces corresponds to the behavior described by their corresponding documentation. This is also a difficulty one face when using the approach since many applications lacks the formal descriptions of what input and outputs are valid for these interfaces and in which cases exceptions are thrown.

2.2 Test levels

Software testing can be divided into several so called test levels which basically describe where to focus the testing [Rakitin01]. This means that each level has a distinct testing responsibility such as individual module testing at one level and the module integration at another. These levels are introduced through the V-Model which describes four separate levels namely; Unit, integration, system and the validation testing level. This model is derived from the classic Waterfall development model [Sommerville04][Pyhajarvi04]. The V-Model with its subsequent levels is illustrated by Figure 1.

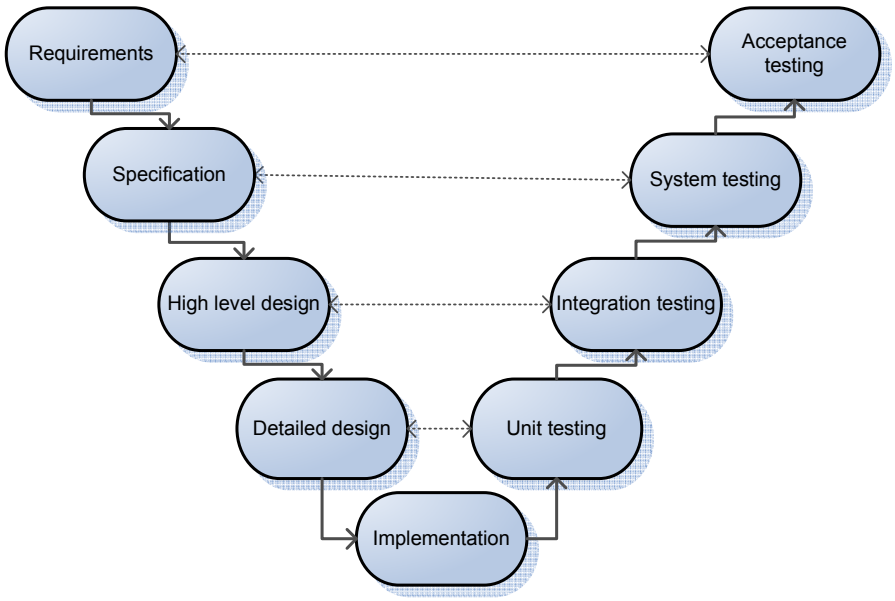


Figure 1 –V-Model of testing

Each of these levels has a distinct testing responsibility which is described below.

- Unit testing. This level verifies if the implementation of the individual modules described by the detailed design behaves in an acceptable manner. However, it could also be used to ensure the correct behavior of the units by using a black-box approach.

- Integration testing. The integration testing level focus on the high level design which usually contains cooperating architectural artifacts. This means that this level verifies if the implemented interactions between modules are correct.
- System testing. The system testing level ensures that the complete system is behaving in acceptable manner. It acts with the system specification as the basis and the input source to this test level comes from the developers.
- Acceptance testing. This testing is usually done by the end-user or customer and verifies if the requirements are fulfilled by the implementation with the requirements specification as a basis. The main difference between this level and the system testing level is that the source of input comes from the customer instead of the developers.

This particular model has several disadvantages, one of them being the fact that it is based on the Waterfall model [Pyhajarvi04]. The V-model assumes that the development phases are completed in the order described by Figure 1. In the agile development environment, this model needs to be modified so that the unit test cases may be written for a small set of requirements instead of testing the complete implementation of the requirements specification. The model may however be appropriate in several cases where the clear distinction between the development phases needs to be known. For example, a consulting firm that needs to sign-off a particular deliverable to the customer may prefer this model over the agile approach where the boundaries are fussy. More information about these particular testing levels is found below with a discussion of the automation possibilities of each level.

2.2.1 Unit testing

Unit testing is meant as a means of testing software components in isolation with disregard to the rest of the system thus to verify that the single units of software meets the requirements or its design intentions, depending on the development method [Runeson06]. This type of testing can be done manually but is often automated in order to increase the efficiency since such tests usually require minimal human attention which in turn decreases the execution time.

An executable test is a test case that can be executed by a computer system. The automation is usually done by implementing executable test code with the responsibility of executing procedures and functions with a specified range of test vectors. As mentioned, it is often hard to test all of the source code statements due to the large amount of possible branches. Procedures exist such as the use of randomised unit testing which is an approach that has been proven successful in regards to unit testing [Yong05]. This technique aims to automatically generate unit test cases and thereby decrease the manual effort that is usually needed to construct these.

In regards to automated testing, the implementation of unit tests in form of source code can have several benefits. First of all, this enables the possibility of repeating the same test over and over again without the need for large amount of tedious manual labour [Runeson06]. This is obviously an advantage when building up a regression test suite in the sense that the decreased manual efforts will lead to decreased costs which can be used for an eventual expansion or improvement of the test suite. Another benefit that may not be as apparent is the reuse possibilities of unit test cases among several projects which can be very useful in the consulting domain (which is the focus of this thesis).

There are several frameworks available for executable unit testing, the most known being JUnit [ObjectMentor01] that is used for unit testing of Java based classes and methods. Since the introduction of this framework, the benefits have been recognised and frameworks for other languages have been developed with similar features. As an example, there is an executable unit test framework called the TSQLUnit framework [Ekelund02] which is based

on the xUnit framework and targets the T-SQL database language developed by Microsoft. With this extensive support, the unit test cases may be automated without large restrictions in the various programming languages. This is of course a major advantage in the sense of reuse because the test suites may now be classified for different types of domains where some languages are particular useful.

As mentioned, unit testing aims to test software components in isolation but it can be hard to separate one unit from another due to large dependencies among them [Tillmann06]. By using so called mock objects, the surrounding environment for the object under test are simulated. Consider a class that needs to be tested, class C. This class is in turn dependent on some methods in class C'. A mock object is used to simulate objects such as C' in order to ensure that the input and output between C and C' is correct. The main purpose with this is to make sure that an eventual found defect is caused by the unit under test and not some other object in its environment. By simulating the environment in this way, the execution time can be reduced since the operations done by C' is kept to a bare minimum [Saff04b].

2.2.2 Integration testing

It is common practice to initiate the integration testing phase when the individual units have passed through unit testing with sufficient quality. This is where these individual components are grouped and tested together through the corresponding interfaces of the units [Leung97]. According the Keller et al. [Keller05], this is a part of testing that is often neglected in favour of other testing measures such as unit testing. However, integration testing is very important because many defects are discovered when the units need to cooperate. Individual unit may work fine alone but most often; defects are revealed when other units try to use their interface. This can derive from, for example, misinterpretations made by separate developers of the unit responsibility which can lead to failures in the interaction between them.

A common mistake that can be made when doing integration testing is to test the component interactions through the user interface alone which is more like the system test approach [Leung97]. Such approach to integration testing can have some disadvantages because it is not guaranteed that the user interface provides entry points for all underlying functionality delivered by the components external interface. This means that some application logic will be untested and such problems can be avoided through bypassing the user interface when performing the integration test [Keller05]. This way all the functionality provided by the external interfaces may be exposed to the test cases. It has been mentioned by Keller et al. that test cases for GUI components are hard to automate which makes it feasible to disregard the user interface in this level of testing [Keller05].

2.2.3 System testing

After the integration testing phase has been completed, the system testing is initiated which targets the system functionality [Leung97]. This phase is a black box approach which should be performed without the knowledge of the systems internal structures. In order to generate good test cases that accurately tests the functionality, the requirements need to be well defined and unambiguous.

Because this level of testing only focuses on the behavioural aspects of the system it can be hard to automate in regards to the structure of the requirements specification. In many cases, the specification documents are written in natural language which implies that some requirements may be ambiguous and unclear which in turn affects the testability. Manual testing in this case may be more appropriate since it can be hard to properly construct an application which successfully can derive the correct behaviour from these documents.

Nebut et al. attempts to combat the problem with deriving behaviour out of specifications by introducing a contract language which can be used to formulate the requirements in such a

way that test cases can be derived through documents written using the language [Nebut03]. This approach attempts to formulate use cases and scenarios, specify all acceptable test inputs and outputs in these and then generate test cases with these artefacts as input [Nebut03]. Such approach may seem feasible in theory but system requirements document written in formal languages tend to be hard to understand and thereby be less useful in other development practices such as in software design. In fact, many companies today prefer the use of informal notation because of the increased understanding of these compared to use cases, scenarios and formally written requirements.

2.2.4 Acceptance testing

This process usually involves the customer to a great extent. Its focus is to ensure that the system fulfils the agreed upon requirements i.e. the acceptable behaviour and this is done by letting the customer or end-user be involved. As can be seen in Figure 1, this is the last level in the V-Model which implies that defects found here can be costly. Therefore, it would be appropriate to develop the test cases for this level early on, based on the requirements together with the customer. By involving the customer in this manner, the requirement defects could be found early instead of in the actual test case execution later on. It is also worth to mention that test-driven methodologies goes one step further and lets the customer take full responsibility for the acceptance tests which force this person to be involved in the process.

Miller and Collins states that the customers should not start writing these acceptance test cases too early in the development due to the lack of system understanding at this point in time [Miller01]. In my opinion, it could however be useful to do this early on in the sense that changes to the test cases throughout the project will increase the system understanding. This could increase the probability of achieving correct and complete test cases in time for the final execution when the system is completed.

It is a misconception that acceptance testing cannot be automated and in fact, some agile methodologies require it. Several frameworks have been proposed. For example, the JAccept suite by Miller and Collins [Miller01] which targets user scenarios in Java applications by letting the customer in an agile setting write these test cases in a tool. Another framework is the one proposed by Talby et al. in [Talby05]. It has been identified by Talby et al. that some formalism is required in the system test specifications if these behaviors are to be automated [Talby05]. Their framework formalizes the specifications to the extent that they can be used for automation as well as be read by non-technical stakeholders. This is a large benefit in the sense that training stakeholder in formal languages is often not feasible or desired. However, because acceptance testing most often targets the graphical user interface and involves the customer it can be still be hard to automate. First of all, the frameworks should not be technically challenging for the novice customer, otherwise the customer will not be able to form complete tests. Because there are many graphical components involved, it can take significant time to keep the frameworks up-to-date which is due to the large changes that often occur in for example the Java SDK. With this in mind, this level can be automated as discussed but it is often not economically viable to do so.

2.3 Verification-oriented development methods

Traditional development models such as the widely known waterfall model divide the development into distinct phases with strict separators [Sommerville04]. This poses several problems in regards to the testing phase which is initiated after the implementation has been concluded. If a strict waterfall approach is used, most of the defects will be discovered in late phases of development which has proven to be very costly [Graham93][Boehm01] [Juristo04]. As opposed to iterative development, the test-oriented development methods integrate the quality aspects into the process itself by performing the testing activities continuous rather than sequential. It is said that the test cases drives the development forward since that the implementation is designed to ensure that the test cases pass [Williams03].

This section presents two of these methodologies and gives a brief discussion of the feasibility of these.

2.3.1 Test-driven development

In test-driven development (TDD), unit test cases are designed based on the requirements rather than the implementation. The production code is designed to pass the unit tests which in turn are designed to fulfil the requirements [Williams03]. A small set of unit tests are written prior to the production code which is then implemented directly after in an iterative manner throughout the development process. There are several advantages that make this practice attractive which are also discussed in [Williams03];

- Early defect detection. Because the automated test cases are available before the source code unit is developed, the implemented code can be tested as soon as it has been developed. This means that possible defects may be corrected early which decreases the costs in the sense that it avoids the discovery of these defects at later stages in development where they are more costly to fix.
- Regression testing. If the practices are followed to the letter, there should be automated unit test cases for every production unit. This makes this approach very attractive in situation where regression testing is essential because every source code unit may be re-tested through their corresponding unit test case.

Due to the fact that the test cases are written prior to the implementation, the testability will increase in the sense that non-testable code will not be implemented at all. However, this approach may also decrease the design documentation that is usually produces with more traditional development methods [George04]. Without this documentation, the implemented design may be hard to understand for new developers. As mentioned by George et al., the rational regarding the structure of the system may not be documented either which can lead to even larger misunderstandings [George04]. However, these are issues that can be dealt with during the development process and thereby be avoided.

George et al. conducted an experiment described in [George04] where TDD was compared to the traditional waterfall model. It was determined by George et al. that the code quality is increased with the TDD approach but that it was more time consuming than the traditional approach [George04]. However, this experiment did not consider maintenance time after release. As TDD aims to provide larger quality than products developed by the waterfall model the total development time of the waterfall approach may be increased if the maintenance time after release is considered. Another interesting observation made by George et al. was that some developers did not produce the necessary unit tests in the traditional approach after the production code had been implemented [George04]. This makes TDD even more appropriate for organisations where quality assurance are of the essence in the sense that developers are more or less forced to make unit test cases which in turn increases the testability of the source code.

In development projects where the production code comes prior to the test cases, it is common that functionality is developed which will be discarded at later phases. Agile methodologies define this as the You Ain't Gonna Need It (YAGNI) phenomenon. By using the test-driven state-of-mind, the test cases are meant to discover unnecessary functionality before it is implemented in the application. In other words, if the functionality may be needed later on, develop it when this time comes instead of when it is estimated that the functionality may become necessary [Jeffries07]. It also relates to testability since the developers will avoid the complexity of implementing functionality that might be removed when it is discovered that the functionality is incorrect. Pancur et al. has done an empirical study where they compared TDD with, what they call and iterative test-last (ITL) approach by using university students in their senior year [Pancur03]. The result from this experiment show that the students think of TDD as ineffective and that the two development approaches did not differ that much. In my opinion, this result is tainted because of the use of students

instead of practitioners in industry. Students will only deliver the product or laboratory assignment and then move on to the next course which means that they will not experience the low maintenance benefits gained by using TDD. With this in mind, the only visible aspects to these students is the initial overhead in regards to test case development time using TDD. However, this time would be decreased if the bug-fixing time would be included. There has been empirical studies such as the one conducted by Bhat and Nagappan where they empirically evaluated TDD against a non-TDD approach in two case studies [Bhat06]. These results, which were conducted with professional developers, showed that it took longer time to develop software with TDD but it increased the code quality significantly when compared to the non-TDD approach. However, it did not described if the overall development time included eventual maintenance time needed for bug-fixing after release which could have altered the results in favor of test-driven development.

2.3.1.1 Extreme programming

One of the most famous agile development methods that advocate test-driven approach is Extreme programming [Abrahamsson03]. Extreme programming introduces twelve core practices namely; Planning game, Small releases, Metaphor, Simple design, Tests, Refactoring, Pair programming, Continuous integration, Collective ownership, On-site customer, 40-hour weeks, Open workspace and Just rules as first introduced by Kent Beck in [Beck99]. The on-site customer practice of XP is particularly interesting to testing and it states that a customer representative should be on-site 100% of the development time. This customer delivers short user stories of some wanted functionality and these can be considered the equivalent to the requirements specifications used in other development methodologies. The development is then conducted in small iterations where the design and user acceptance tests are based on these stories. It is important to have a single customer that can correctly represent the end-users of the system and who has sufficient time for the project. Johansen et al. describes the need for a customer that can explain the requirements to the developers [Johansen01]. This type of clarification is particularly important in extreme programming since there is limited documentation of the requirements and because the primary testing focus is put on unit and acceptance testing both of which are based on the requirements. The XP paradigm advocates that the initial user stories should be kept short until the time of implementation where the on-site customer is asked for further details [Wells99] which go hand in hand with the YAGNI concept described in Section 2.3.1. As a consequence of this concept, the design should be simple which in turn increases the testability needed for the unit and acceptance test.

The extreme programming description found in [Wells99] states that there should be unit tests for every production code unit which facilitates the regression testing needed between releases. Another interesting issue in regards to acceptance test is that it is the responsibility of the customers to form these tests so that they can be automated by the testers later on. This is an excellent way to get a fair amount of customer involvement since it ties the customer to the project which can be utilized for increased developer understanding of the customer need. It is also worth to mention that the acceptance tests are constructed for one iteration at a time. This has the benefit that it minimizes the risk of getting to far away from the customer which could become a problem if acceptance tests for all iterations were to be developed all at once. The traditional V-Model described in Section 2.2.1 places the test levels, including unit and acceptance level, in a sequential order which do not work in the XP methodology. However, the levels still apply with the distinction that they are used continuously throughout the development instead of sequential with the aim to begin the levels prior to the implementation. It is most common to implement executable test cases for the production units and the primary used unit test frameworks today inherit from the xUnit framework, which also includes the JUnit framework that is further described in Section 2.5.7 where a code example can be found as well.

A difficulty with test-driven methodologies such as extreme programming is that they are relatively new in comparison to other models such as the waterfall model which means that their worth has not yet been definitely determined. However, there are some papers which evaluate the XP paradigm empirically. Abrahansson gives some empirical data in [Abrahamsson03] where a XP project is conducted in two releases. The results from this study showed that learning experiences of the methodology practices was conducted in the first release which affected the second release positively in terms of estimation accuracy and developer productivity. Koskela and Abrahamsson has also published a later paper which targets the customer-on-site practise in XP and they claim that even though the customer was 100% available, the actual work done in development was more close to 21% of the total time [Koskela04]. These studies do however have some drawbacks since they use students as their subjects and use a fellow researcher as the on-site customer, a bias also recognised by the authors in [Koskela04]. As mentioned by Abrahamsson, it can be difficult to compare empirical data collected from different organisations since each organisation adopts different practices and conducts them in dissimilar ways [Abrahamsson03]. This is partly due to the fact that the extreme programming methodology only provides guidelines in regards to which practices that may be adopted and does not dictate that every single practice should be used. Merisalo-Rantanen et al. made an empirical study where a critical evaluation of the extreme programming methodology was conducted [Merisalo-Rantanen05]. They argue that the methodology is too dependent on skilled individuals and that the methodology itself is mostly derived out of other development paradigms. It is also recognized by Merisalo-Rantanen et al. that extreme programming needs further study in order to validate how it applies to large scale project since the practices are more focused on small teams that have good communication skills [Merisalo-Rantanen05]. Another challenge relates to how the management and developers are to be convinced of the benefits gained by adopting the development methodology. This is described as how to sell the practices by Johansen et al. in [Johansen01]. Because it has not yet been empirically proven that the adoption of these practices actually provides added value in form of productivity and product quality it can be hard to convince these people to move from a well established set of development practices to this new one. It can be concluded that this methodology needs further focus in terms of empirical studies to determine its worth.

2.3.2 Behaviour driven development

A recent effort has been made to combine the test-driven development methodology with domain driven design in an attempt to get the benefits from both into a unified development method called behavior-driven development (BDD) [BDD07]. To my knowledge, this approach has not yet been evaluated empirically so the method will be discussed here out of a speculative perspective based on the information found in [BDD07].

As the name implies, this development method focuses on the behavior of the system, which is usually described by the system requirements specification in non-agile methodologies such as the waterfall model. One of the aims with agile and the test-driven part of BDD is to minimize such documentation and instead have a customer on site which mediates the requirements through brief user stories and more detailed ones when the functionality is actually needed [Jeffries07]. The test-driven part also aims to increase the shared requirement understanding between customer and developer. Test cases are designed with the purpose to test that the system fulfils the acceptable behavior [BDD07]. In other word, if the output from the test cases corresponds to an acceptable behavior, the test has passed. With the behavioral focus, strong cooperation among the various stakeholders is needed which is the reason behind the customer-on-site practice. If understanding is not mutual, proper test cases would not be possible because the correct output would not be known. In organizations where the requirements tend to be ambiguous it could be a risk of adopting this approach without proper education in the field of requirements engineering. A similar need in regards to requirements elicitation is also recognized by Murnane et al. in [Murnane06]. If the correct behavior cannot be properly elicited through the various stakeholders, the test

cases would probably be incorrect which would affect the final implementation. Murnane et al. discusses in [Murnane06] that proper input/output elicitation is needed to ensure the effectiveness of black-box testing approaches which is usually the case when testing behavioral artifacts.

Similar to test-driven development, the test cases are written prior to the implementation of the production code which means that defects in the requirements may be detected at the early stages [BDD07]. As mentioned, finding defects early is very cost effective and this certainly applies to requirement faults which can be time consuming and hard to correct after implementation. In regards to automated testing, this development method seem as friendly to executable test frameworks as the test-driven approach which can reduce costs in favor of early defect detection.

Even though this methodology is new, there has been an attempt to support it through frameworks such as JBehave [JBehave07] that targets the Java programming language and RSpec [Hellesøy05] for Ruby. The JBehave framework is similar to the JUnit framework in the regards to the structure and is described further in Section 2.5.8.

2.4 Automated testing opportunities

Manual execution of test cases is considered inefficient and error-prone and it is often possible to increase the efficiency by automating these which also relives the workload of the testers [Keller05]. By introducing automated test cases to the development process, the testing cost also decrease and some of the tedious manual labour is avoided. However, in addition to the opportunities it provides, there are several challenges as well. It does take some time to develop these automated test cases and several considerations should be taken before their implementation. If test cases are to run several times which is the case in for example regression testing, it may prove beneficial to automate them so that the resources needed for the re-run can be put to better use [Keller05].

Even with the introduction of automation it is most often impossible to achieve full test coverage due to the large amount of different states and branches that a software product may enter [Whittaker00]. This introduces the issue that handles which artefacts that are important enough to be considered for coverage of the automated test cases. However, it should be noted that striving for full coverage is not always the most appropriate measure for fault detection. This is due to the fact that the defects often have different severity while the test cases differ in terms of cost [Elbaum01].

A test strategy of an organisation describes which types of tests that is to be conducted and how they should be used within the development projects [Keller05]. When forming this strategy it is important to consider which tests that is to be executed and when they are to be executed and as Keller et al. states, it can be hard to run certain tests at the incorrect test level. For example, an integration test would not be the most feasible approach to use when trying to find defects in the internal structures of a particular module. Instead, perhaps a unit testing approach should be used in that state of development.

A large amount of software development companies today are far behind in this field of automation and sometimes, the testing resources are allocated after the product has been developed. Such behaviour can inflict serious problems to the product quality. It is hard to develop automated test cases in late development phases when automation issues have not been considered in the architecture and design. In this section, several challenges as well as possible benefits imposed by automated testing will be discussed, issues that should be taken into consideration when forming the automated test strategy for the different projects in software development organisations.

2.4.1 Reuse

In most development stages, there has been a focus of component reuse which has several advantages. First of all, the component can be written once and used many times which saves development effort. It also has quality benefits because the component may be refined and improved over time. This practice can be used for requirements, design artifacts and source code components and it can also be applied to the automated test cases. With this kind of reuse, the benefits discussed such as quality refinement is transferred to the test cases as well and first-class test cases is very important in testing. For example, with poor quality, false positives may be found instead of real defects which can lead to unnecessary manual labor. This is an issue that can be remedied with sound reuse.

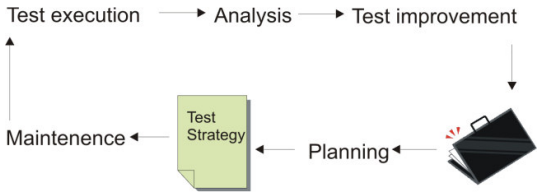


Figure 2 – Reuse strategy example

To get a reusable quality test suite it could be appropriate to extend the normal test case development process briefly described by Keller et al. in [Keller05]. Figure 2 gives an example of how the test suite can be improved along sides the ordinary development. It contains the following stages;

- Planning. This phase includes consulting the test strategy to see if the test case chosen from the test suite corresponds to the current testing goals.
- Maintenance. Often, when test cases are brought from the test suite, they need some maintenance so that it can be adapted to the current setting. This state takes care of the possible modifications needed.
- Test execution. In this stage, the test is executed in order to find eventual defects and more importantly for the reuse issue, return test data to the next stage.
- Analysis. Analysis in regards to test reuse is concerned with how the test case performed, if it fulfilled its purpose. Some measurements may be needed, depending on the current goals of the test strategy.
- Test improvement. With the results provided by the analysis part, the test case may now be improved before it is returned into the test suite that is illustrated as a black portfolio in Figure 2

Notice should however be taken to the fact that the aim of the test improvement stage is to improve the test suite in favor of the production software quality and not only the test cases themselves. In other words, have the software quality aspects in mind when modifying and improving the test cases so that the goals provided by the test strategy are not neglected.

2.4.2 Regression testing

After a change has been made in a software artefact it is usually a good idea to re-run previous test cases to ensure that the change did not affect other system components which have previously passed tests. This is called regression testing. It is a common belief that automated test cases will find many new defects continuously throughout the development process and according to Kaner this is not the case [Kaner97]. Kaner states that most defects that the automated test cases find are at the first execution right after the test case design [Kaner97]. Still, these test cases are most useful. Consider the fact that re-iteration of old test cases are needed in order to guarantee that changes in the software have not introduced faults into the already tested components. Without these automated test cases this has to be done manually and the testing cost increases for every manual test case execution. Now, because of the automation, this tedious work and large costs can be avoided simply by the re-

execution of the test cases automatically, a large benefit also acknowledged by Keller et al. [Keller05].

In large software systems where there are copious amounts of test cases, there can be some problems in regards to the time and resources needed for the execution of all tests. Granted, automated test cases embedded together with the production source code may decrease the execution time compared to manual testing but it may still take a very long time do a full automated regression test. By using prioritisation of the source code units, the regression test case suite may be constrained which could save time when doing the regression testing. There are several test case prioritization techniques that can be used for the selection of test cases and it depends on the testing goals which one that should be chosen [Elbaum01].

Another more immediate challenge is the maintenance of large suites of test cases. Consider that these test cases assume that the methods and constructors have a particular defined header that expects a particular set of input. Such a simple question can pose serious problems in regards to the cost of maintaining large scale automated test suites due to the continuous code changes. Meszaros et al. has proposed solutions to these problems into a suite they call the Test Automation Manifesto which is shown below and first introduced in [Meszaros03].

Principle	Rationale
Concise	As simple as possible and no simpler.
Self checking	Test reports its own results; needs no human interpretation.
Repeatable	Tests can be run many times in a row without human intervention.
Robust	Test produces same result now and forever. Tests are not affected by changes in the external environment.
Sufficient	Tests verify all the requirements of the software being tested.
Necessary	Everything in each test contributes to the specification of desired behaviour.
Clear	Every statement is easy to understand
Efficient	Tests run in a reasonable amount of time.
Specific	Each test failure points to a specific piece of broken functionality; unit test failures provide “defect triangulation”.
Independent	Each test can be run by itself or in a suite with an arbitrary set of other tests in any order.
Maintainable	Tests should be easy to understand and modify and extend.
Traceable	To and from the code it tests and to and from the requirements.

Table 1 – The Test Automation Manifesto. (From [Meszaros03])

The twelve principles seen in Table 1 from [Meszaros03] would be appropriate to consider when designing the test cases. An interesting issue also covered by these principles is the aim of providing easy-to-read test cases. This is especially important to the test maintenance in the sense that it is hard for developers to keep every test case in memory. Complex tests give longer maintenance time which led to larger overall testing costs. These principles of course consider more issues than maintainability, such as traceability which is a considerable asset to have. Without such traceability it would be hard to see which particular behavior that has passed or been failed by the test case.

2.4.3 Coverage issues

With large software systems, it is almost impossible to achieve full test coverage because of for example the large amount of different branches and states that can occur in the program execution [Whittaker00]. Coverage is a general concept which can be divided into criterion such as statement, branch and path coverage criterion [Zhu97]. The coverage criterion efficiency in regards to found defects largely depends on the application type and

complexity. A brief introduction to the above mentioned criterion is described below and alternate descriptions of these can also be found in [Zhu97].

- Statement coverage. It can be hard to reach all statements in the sense that some statements are rarely executed throughout the program and this criterion focuses on executing each source code statement.
- Branch coverage. This criterion targets the branches that an application may enter depending on some condition. (e.g. if and switch case statements)
- Path coverage. This criterion focuses on the different possible paths that are executed throughout the functions. It checks if each possible path in a function has been followed.

In the cases where full coverage is actually possible some other challenges are raised instead. Full coverage does not necessarily mean that all defects are discovered [Juristo04]. This is because different types of tests are necessary for the coverage of the large amount of possible data structures that may be executed throughout the source code [Juristo04]. Example 2 is a real simple illustration of a C function which suffers from a possible boundary violation.

```
char example(int n) {
        char buf[] = {'a','b','c'};
        return buf[n];
}
```

Example 2 – Buffer overflow example

As can be seen in Example 2, a buffer overflow may occur if n > 2 thus it would be appropriate to include test cases with boundary checks to find this defect. This example is of course a real simplification but consider the lines of code in large industrial applications and the complexity of these. If full coverage is achieved it may be possible to cover each and every of the boundary violations, but there are many other types of defects that need to be considered as well, for example control flow defects. With this in mind, it would probably not be possible to have full coverage in regards to all types of defects. This introduces test selection where one issue is to weight coverage extent against the defect types that needs to be considered.

2.4.4 Test selection

When full coverage of all source code and all types of defects are not feasible it is important to make serious considerations about which artefacts that should be considered for testing. In regards to the automated test selection it is discussed by Keller et al. [Keller05] that test that lie at an inappropriate level is hard to execute no matter if they are automated or done manually. This should be taken into account when doing the test selection because as Keller et al. describes, tests that are hard to do manually is often equally or even harder to automate [Keller05]. With this in mind, organization should be aware of that automation may not solve issues related to such manual execution difficulties of test cases. Instead, it should be considered which manual labor that is most economical viable to automate.

Before the actual testing is initiated, it is also important to determine what is to be achieved with the test cases and particular how many times they are expected to be executed whereas tests that is to be run once hardly deserves to be automated [Zhu97]. Zhu et al. discusses the importance of knowing the testing objectives through the use of a test adequacy criterion [Zhu97]. This test adequacy criterion is a measure of the feasibility of a given test and a various amount of these has been introduced in to the field of software testing according to Zhu et al. [Zhu97]. This measure is connected to the coverage in the sense that the selected criterion steers which type of coverage is to be achieved. Keller et al. describes that organisations tend to over automate in the sense that they try to automate all manual test cases [Keller05]. This could pose problems because some items are in fact less time consuming to do manually such as, according to own experience, GUI testing. Large

amounts of change may increase the total cost of the automated test cases and this is because these test cases need to be maintained when the change occur [Kaner97]. GUI components fall into the category of components that is exposed to frequent change, an issue also described in [Kaner97]. It could thus be a good idea to complement automated test cases with manual ones in an attempt to get as high return of investment as possible.

As mentioned by Juristo et al., a common problem is that software testers often rely on their competence and experience when a choice is to be made among the various existing testing techniques and methods [Juristo04]. Without actual proof of the feasibility of the testing methods, the choice may suffer from inefficiencies and low coverage of important software artefacts [Juristo04]. Juristo et al. proposes in [Juristo04] that the knowledge about test technique selection should spawn from empirical studies that prove their benefits. This would impose a more engineering like approach to the software testing process which in turn would increase the maturity of the process, according to Juristo et al. It is of course an advantage if the benefits are proven and an engineering approach is used but in my opinion, the competence, experiences and intuition of the developers should not be neglected when doing testing.

2.4.5 Test data generation

As mentioned above, full coverage is nearly impossible to achieve in most cases and this brings forward the issue of which test data that is appropriate to generate in order to maximise the coverage of the given criteria. Each set of input to a function is called a test vector and in most cases, several of these vectors need to be generated in order for the test case to be somewhat efficient. It has been determined by Xie that commercial tools often generate redundant test cases [Xie06], an issue that is dealt with in their approach to automated testing. This is an important challenge to deal with in test data generation because of the increased execution time that comes with large amounts of test vectors.

There are several approaches to test data generation and the three of these are described briefly below which is also described by Pargas et al. in [Pargas99].

- Random generation. In this approach, the data is randomised into the test vectors, often iteratively in attempt to execute a chosen statement.
- Path-oriented generation. This approach uses the various paths visible in the source code to generate test data which triggers the execution of selected paths in the application.
- Goal-oriented generation. In this approach, a statement is selected for execution and no matter which path or branch that needs to be entered; the test data is generated in an attempt to execute the particular statement.

When the number of statements, paths and branches increase it also enhance the difficulty of data generation.

```
int example(char a, int b, int c, float d) {
        switch (a) {
                case 'b':
                        if (b == c)
                                return (d == 0.1 ? 1 : 0); /* Defect */
                        else {
                                if (b < 0)
                                        return ( b == c ? 1 : 0);
                        }
                break;
        }
}
```

Example 3 – Path-orientation example

Consider Example 3 that is related to the path-oriented approach. To reach the code which is marked as a defect (0.1 is a double which will result in a mismatch), the test data generated

need to be a == ‘b’, b == c. Otherwise, the path is unchecked and the defect remains unnoticed. Full path coverage is most often impossible in large scale applications due to the vast amount of possible paths. Two approaches which target these issues through automation will be discussed in a later section.

2.4.6 Test analysis

When the test cases are executed with the test vector as input, the intent is to provide output to some entity that has the responsibility of verifying that the behaviour of the procedures and functions are correct. The documentation produced that describes the behaviour, i.e. the requirements specification, the architectural and the design documents is appropriate to use in test data generation as well as test analysis [Xie06]. In case of more agile approaches, this behaviour could be derived from the customer and user stories instead. Because the output should reflect the behaviour of the system it would be appropriate to generate the output vectors based on these artefacts [Xie06]. However, as discussed by Xie [Xie06], there is often insufficient documentation in software development projects. Furthermore, Xie propose a framework in [Xie06] with the aim of increasing the effectiveness of automated testing when such artefacts are missing.

As mentioned by Yong and Andrews [Yong05], manually checking output values can be an exhaustive task which can be relieved by introducing automated test oracles which basically is a program that checks the output given by the application under test. One of the most concerning problems with these oracles is how to replace the instincts of a human controller with automated software which is an area suitable for further research.

2.4.7 Testability

Testability is basically a measure of to which extent a software product can be tested, however several definitions exists as described in [Mouchawrab05]. With poor testability, fewer defects will be discovered and the quality of the product will be lower than it could have been with more testable structures.

In regards to test consultants which can arrive to the project in late phases it may not be possible to change the design of the already implemented system so that the testability can be increased. If the situation occurs where the developers deliver code with low testability to the testers it can lead to inefficient software testing. Some examples of problems that can contribute to low testability are presented below:

- Ambiguous requirements. If the developers produce ambiguous requirements, it will be hard to write sufficient test cases for the system testing.
- Complex design. If the design is to complex, it will be hard to automate the traceability back to these entities from the source code. In fact, bad design may disable the possibility of automation to a great extent [Keller05].
- Complex source code. If the source code is to complex, it will impose a long learning time for the testers which may lead to inefficient test cases.
- Maintainability. It is not enough to make the architecture testable, the test cases developed along sides the other software artifacts has to be maintained as well [Kaner97].

Design pattern testability is an interesting notion which is strongly related to the complex design problem. As a brief introduction, design patterns are recurring design decisions taken during development and these are further described in [Larman05]. Design pattern testability is used to control design patterns to avoid a decrease in system testability. Baudry et al. introduces the concept of testability anti-patterns which represents bad design decisions which increases the testing effort needed to ensure that the component has been properly tested [Baudry03]. It would be appropriate to use these patterns to ensure that the design patterns used do not resemble the ones which provide low testability.

2.4.8 Test strategy

To achieve success when adopting testing practices in the organization, a strategy is needed which contains the testing objectives i.e. the goals that are to be reached [Keller05].

As mentioned by Keller et al., an automated test strategy describes what types of tests that is to be conducted in the development projects and at which test level they belong [Keller05]. It is important that the test cases are located at the correct level because usually these different levels have a distinct set of goals and objectives which may not be appropriate for the given test case [Keller05]. For example, a boundary check at the integration test level will probably not discover errors in single branches of code that are not accessed through the component interface. Such test would be more appropriate to have at the unit test level where the probability of defect discovery is higher. Such issues need to be dealt with because inefficient test strategy may result in lower software quality in the end.

It could also be a good idea to consider development policies in the strategy in regards to the test collection that is implemented throughout the projects. If the test suites are not developed using the same engineering policies as the other software artifacts, the test cases would probably become inefficient. Using an ad-hoc approach could be hazardous to the quality of the test suite which in turn could propagates to the quality of the actual production source code in the sense that bad tests may fail in achieving the overall goals describes in the strategy.

Testability is discussed above as an important factor in regards to automation. This is an issue that would be appropriate to cover in the test strategy as a policy. The strategy may state that the architecture should be designed with testability in mind and also contain a description of what testable architecture means in the particular organization. It would be beneficial to involve the developers, testers and managers when taking these decisions so that they do not feel uncomfortable with these definitions. If this is not done, the strategy will probably be ignored and the effort wasted.

2.5 Relevant methods, approaches and strategies

There are numerous frameworks available that covers different criterions. Several testing frameworks are available which supports the automation of test cases, not only to automate the test cases themselves but also to adapt other frameworks to fit several application domains. This section will introduce frameworks, methods and strategies that are considered to be useful primarily in the test consulting domain where the testing criteria often change but also as possible solutions to the challenges discussed in section 2.4. In table 2, a brief overview can be found for each method, approach and strategy that will be further described in this section.

Title or Author	Overview
Directed Automated Random Testing [Godefroid05]	An approach which automatically generates test drivers and automatically parses component interfaces to also generate test cases. Uses the structural visibility to direct the execution to particular branches.
Structurally guided black-box testing [Kantamneni98]	Combined a black-box with a white-box approach to guide the automated testing. Targets nested branches which are considered hard-to-reach.
A framework for practical, automated black-box testing of component-based software [Edwards01]	An approach which tests individual components by providing test case wrappers with an entry point to the component under test. Automatically generates both test drivers and test cases.

Korat: Automated Testing Based on Java Predicates [Boyapati02]	Uses formal JML specifications of the system to derive the acceptable behaviour and automatically generates test cases with java predicate methods. Also generates test oracles which check the results from the predicate methods.
Feedback-directed Random Test Generation [Pacheco07]	A recent approach which start by automatically generating test sequences by using a random testing approach. Then it continues by using the results from the previously executes test method sequences in order to guide the testing.
Systematic Method Tailoring [Murnane06]	This method enables current black-box techniques to be broken down into atomic rules [Murnane05] which later can be used to tailor black-box methods to suit specific software domains.
JUnit [Beck98]	This is a unit testing framework based on the xUnit family. It enables developers to write executable test cases for their Java based source code in a relatively easy way. The foremost used framework in the test-driven development paradigm.
JBehave [JBehave07]	A unit testing framework similar to JUnit which focus on the validation of behaviour instead of the unit design. Can be used for the unit testing process in behaviour-driven development.

Table 2 – Section overview

2.5.1 Directed Automated Random Testing

Godefroid et al. has proposed the DART approach (Directed Automated Random Testing) which aims to provide complete automation of the testing procedure thus removes the need of manually writing test drivers [Godefroid05]. The approach has been divided into three distinct techniques by Godefroid et al. and these are described below.

- Automated extraction. Extracts the interfaces provided by the application that is to be tested. An internal stack that corresponds to these interfaces is then built into the memory structure of the DART application. The purpose of this stack, besides knowing the function inputs, is to keep track of the current branches that have been tested.
- Automated generation. By interface examination, test cases that aim to provide random testing towards the interfaces are generated.
- Dynamic analysis. When the initial vector has passed through the application during execution, the results are checked. If a defect has been detected, this is reported.

In article [Godefroid05] the technique is described for C code, but it could probably be applied to any language if these syntaxes are considered in the implementation of the automated extraction module. The most interesting issue, besides the fact that the approach is completely automated, is that the branch that has been covered is marked as done in the stack and a dynamic calculation is done. In this phase, the program tries to generate test vectors that will reach certain branches that not yet have been covered. This is done dynamically through analyzing the results from the previous execution and thereby generating a test vector that will reach the next branch through execution.

This approach has several benefits opposed to pure random testing. First of all, it can be determined if a branch can be covered at all thus conclude if it is reachable which is also mentioned in [Godefroid05]. Secondly, because the test vectors are calculated and not randomized after the initial attempt the testing time can be decreased. This is due to the unnecessary exhaustive testing that is avoided in this approach which is necessary for pure random testing to be efficient. The approach could also be beneficial if the testing strives for high coverage. Because the test vectors are issued through calculations with the branches as a basis, no unnecessary execution is needed. An additional advantage identified by the authors of [Godefroid05] is that every defect which is discovered is guaranteed to be correct thus no false positives will be issued. However, if the internal structures are not known, this approach would not be appropriate whereas it can be considered a white-box approach to testing.

2.5.2 Structurally guided black box testing

This framework which was introduced by Kantamneni et al. [Kantamneni98] combines black-box testing with white-box testing due to the difficulties of getting high branch coverage through using a strict black-box approach.

It has been identified that these nested control statements are particularly hard to cover and in regards to this assumption, a new term called potential of a branch has been introduced by Kantamneni et al. [Kantamneni98]. Kantamneni et al. describes a potential as being basically a count of the nested branches in the code that not yet has been covered by the test cases. The main focus of this approach is to cover these hard to reach, nested control statements. In order to do this, a so called guiding mechanism has been introduced by Kantamneni et al. which are used to steer the test cases towards these hard to test branches. This mechanism is used after the easier branches have been covered, which is done initially. A more detailed description of this approach can be found in [Kantamneni98].

The founders of this approach have put the approach to the test in an experiment described in [Kantamneni98]. However, the applications used in the experiment had a low code size which could affect the results in the sense that the industrial applications where the approach would be appropriate often contain much more lines of code [Kantamneni98]. In any case, the experiment showed that the application interfaces had to be adapted for the sake of interoperability which may not be feasible in industrial applications. However, for these particular applications it was concluded by Kantamneni et al. that overall, the approach gave larger coverage and less needed test vectors than a standard random testing approach. An interesting observation made by Kantamneni et al. is that low testability affected the result in one of the applications where the approach gave the same result as the random testing approach. It can be argued that if the tested applications have been implemented with testability in mind would favor this approach in regards to the number of test vectors needed.

Granted, by using black-box testing it may be hard to cover certain branches and this approach may be appropriate to increase the coverage of hard to test branches which indeed exist in many applications. However, black-box techniques are mainly considered when the internal structures are not known and if this would be the case, this approach could not be used due to the involvement of the white-box specific techniques.

2.5.3 A framework for practical, automated black-box testing of component-based software

Software components today are often built with reuse in mind due to the cost benefits that is gained through the build once and reuse approach. Edwards has recognised this and developed an automated test framework for reusable software components which is described in [Edwards01]. This approach has three main parts as also described by Edwards;

- Automatic generation of built-in test (BIT) wrappers. A BIT wrapper surrounds the component under test. It contains two layers with the inner layer connected to the

actual component. It has been concluded by Edwards that these wrappers should not interfere with the normal behaviour of the tested component and that the component should not be altered for the sake of the wrapper.

- Automatic generation of test drivers. By parsing the component interface, test drivers can be automatically be generated.
- Automatic generation of test cases. This step includes the generation of component test cases as well as the generation of test oracles which are to check that the input and output corresponds to a correct behaviour.

As described by Edwards, the purpose of the two layer approach is to have the inner layer responsible for checking the internal component state while the outer layer handles input and output checking in regards to the client code operations. Careful consideration has been taken to not affect the client code which uses the component in the production code [Edwards01]. To note here is that the oracles also should check that the production code does not try to use the component in an incorrect way [Edwards01]. This means that for both incorrect inputs and outputs to and from the component, notifications are to be made from the oracle [Edwards01].

For the automation to be efficient when using the approach, the components should be described in a formal behavioural language and in the trial the components are describes using the RESOLVE language [Edwards01]. If this is not the case, it requires human interaction for the creation of the test cases and wrappers [Edwards01]. This is also the main difficulty in using this approach in the sense that formal languages are rarely used in industry. If the behavioural description is not present, the correct behaviour must be established through the stakeholders or else the correct input and output of the built-in tests cannot be verified by the test oracles.

As described by Edwards, the approach has been evaluated for simple component and the approach needs to be evaluated for more realistic industrial components to better ensure the validity [Edwards01]. This approach could be beneficial to use due to its attractive automation focus for black-box components but in organizations which have informal behavioral descriptions of the components; much manual labor is still needed with the approach.

2.5.4 Korat: Automated Testing Based on Java Predicates

The object-oriented programming language Java has a large set of various data structures that can be used for different purposes. Java predicates are simply methods which return Boolean values depending on the outcome of the method call. Boyapati et al. has introduces the Korat framework which tests Java structures with the use of these predicates [Boyapati02]. This includes a complete automated test suite where test cases, test oracles are generated based on a formal class description based on the Java Modeling Language (JML). As mentioned earlier, it may be hard to convince developers to adapt formal modeling languages and this is partly because the transition to formal modeling requires that persons change their way of thinking. However, as also mentioned in [Boyapati02], by using modeling languages with likeness to the programming language itself, the transition these programmers face are now limited.

To briefly introduce this framework, the following discussion is largely derived from [Boyapati02] where a complete description can be found. By automatically deriving class information from the formal JML specification, a skeleton of a Java predicate is automatically generated. This is done by checking the acceptable input and outputs described in the language as well as what constitutes an exception. These predicates can be considered as the automated test cases provided by the framework. The purpose of them is to return either true or false, depending on if their internal structures find defects in the tested Java structures. When the test cases are executed, it depends on the type of data that is entered, if

it is valid or exceptional data sent to the structures. A valid data set should trigger the acceptable output and the invalid should result in the exceptional behavior described by the formal model. Test oracles are also generated which executes these test cases and interpret the results thus increasing the automation. There are two primary strengths of this framework, one being the partitioning of the search space with is done by pruning away unnecessary test cases. The other is the division of candidate objects into separate domains which results in that only one candidate from each of these domains needs to be executed for sufficient test results. Boyapati et al. considers the framework to be effective which can be traced to the search pruning technique and search space partitioning used by the approach. However, due to the fact that programmers need to modify the test cases manually from time to time, this framework cannot be considered completely automated but it does provide large automation benefits.

2.5.5 Feedback-directed Random Test Generation

Random test generation is a commonly used approach that basically produces tests randomly for a given set of methods. The random approach has been shown to be effective because it gives high code coverage [Yong05]. However, as mentioned by Pacheco et al., other approaches such as chaining may give larger coverage than the random approach [Pacheco07]. This may be because the technique does not reason about particular branches, paths or statements which could be vulnerable. Instead it generates and hopes to get as high coverage as possible through the execution. Also, if the test results are not analysed before running additional tests, redundant and unnecessary tests may be produced and executed in the consecutive testing cycle [Pacheco07].

In a recent paper, Pacheco et al. introduces a technique based on random test generation with the distinction of using feedback from previously executed test cases. A detailed description of this approach can be found in [Pacheco07] which is the basis for this section. One of the main features of the approach is the use of sequences which basically is a sequence of method calls which are going to be executed in the test case. Pacheco et al. introduces an interesting notion of reducing the needed test cases. For each new sequence that are generated, the old ones are examined to make sure that no test redundancy is issued to the collection, the purpose being to maximize the unique number of states that a particular object can enter. The creation of new sequences is referred to as extending the suite of sequences by Pacheco et al. As for the algorithm, there are four primary attributes that needs to be considered;

- Classes. This is collection of classes that are going to be tested by the sequences.
- Contracts. One of the most interesting attributes is the contracts collection. This specifies what to consider when executing the sequences. In the default setting as mentioned by Pacheco et al., the API description for the classes is used to determine if the behaviour is accurate. An advantage with the approach is that the number of contracts can be increased by the user thus it enables the testers to derive the contracts based on behavioural descriptions. These can, for example, be derived from user stories which are common in the agile environment. The results of testing against these contracts are also evaluated by the automated test oracles contained in the approach.
- Filters. To restrict the extension of the sequences and thereby the search space, filters are used which could be feasible in several cases such as when a sequence are known to produce a specific behaviour at some point in the method. Pacheco et al. gives an example in [Pacheco07] where a run-time exception is known to happen at a specific point in the method. Thereby it would be unfeasible to use this sequence for further creation of new ones in the sense that the method will stop at that known point either way.
- Time limit. Every testing technique contains some way of knowing when to stop testing and this is called a stopping criterion. This particular approach used a time limit to restrict the testing which is sent initially to the algorithm.

The evaluation described in [Pacheco07] concludes that Feedback-directed Random test generation can give high coverage but more importantly, high defect discovery. This approach seems attractive because of the expected high coverage and ability to tailor contracts. It could be used for behavioral testing as well in regards to requirements because of the ability to create custom-made contracts. However, this approach needs further evaluation so that the industrial value can be established because the current evaluation only covers framework classes provided by the Java and .NET libraries.

2.5.6 Systematic Method Tailoring

It has been recognised by Murnane et al. that there can be difficult to adapt current black-box testing techniques to fit different application domains and they propose an approach for dividing current techniques into atomic rules [Murnane05]. Different black-box techniques target specific types of defects and some of these techniques may target some of the same types. The approach divides these techniques into distinct rules, the number depending on how many different scenarios that the technique cover. A rule in this sense can be, for example, a test of a specific item such as a lower boundary check connected with either invalid or valid input depending on the purpose with the test. As can be realised, the most commonly used techniques have large sets of these atomic rules which implies that not all can be used in every single application domain. By the technique breakdown to atomic rules, each rule can be executed alone to test if the output given by the specifications is achieved. Because every single rule may not apply in all domains this also gives an advantage in the sense that distinct rules may now be selected to match the current domain and project which means that the redundancy of using several complete techniques are avoided.

Murnane et al. has also proposed the Systematic Method Tailoring approach [Murnane06] which is based on the Atomic rule approach. This approach is quite interesting because it is used to tailor black-box approaches to be efficient in distinct project domains. With this approach the atomic rules can be collected into the rule set by using three separate procedures as also described in [Murnane06];

- Selection-Based tailoring. By applying this procedure, the rules are taken from current black-box techniques and put into the rule-set.
- Creation-Based tailoring. Often, the tester experiences are used to test software artefact rather ad-hoc and it has been recognised by Murnane et al. that it is beneficial to support this [Murnane06]. In this procedure, the rules by the testers themselves based on their previous experiences in the particular domain.
- Creation-Based tailoring via Selection. In this procedure, existing rules are the basis for creating new rules i.e. they are combined in order to create a new rule for the rule-set. As mentioned in [Murnane06], the instincts of the testers are often used to combine rules in black-box testing techniques which make this support attractive.

These approaches combined could be very useful in the consulting domain both because of the frequent domain change that consultants experience and the possibility of reuse. Test cases can be written for a particular rule and when this rule is selected for use in another domain, this test case can be modified or directly used in that domain as well.

2.5.7 JUnit

JUnit is an executable testing framework that enables developers to write automated test cases for classes, methods and packages they have written using the Java programming language [Beck98]. In organizations where Unit testing is adapted, the JUnit and other xUnit frameworks are the ones that are primarily used. This can be traced to their early arrival to the development community but also to the simple structure of the frameworks. The benefits imposed by unit testing frameworks have been recognized by several IDE vendors. Netbeans and Eclipse for example, has built-in support for JUnit which makes test case creation for particular classes, packages and individual methods a couple of clicks away. It also

facilitates test-driven development since the test cases can be developed prior to the implementation of the production code. Of course, it will not compile until the production code is implemented which is good since it ensures that the source code will be designed based on the test case in order to get a compilation.

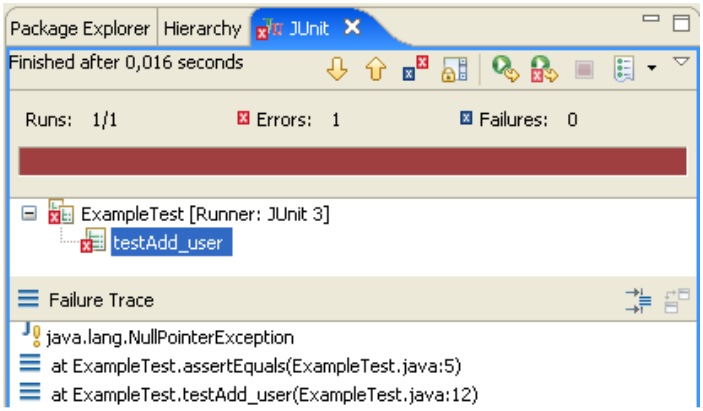
```
import junit.framework.TestCase;
public class ExampleTest extends TestCase {

    protected void assertEquals(User expected_user,User user) {
        assertEquals(expected_user.GetName(), user.GetName());
        assertEquals(expected_user.GetUser_id(),user.GetUser_id());
    }
    public void testAdd_user() {
        Example ex = new Example();
        User user = ex.add_user("John Doe");
        User expected_user = new User("John Doe",1);
        assertEquals(expected_user,user);
    }
}
```

Example 4 – JUnit code example

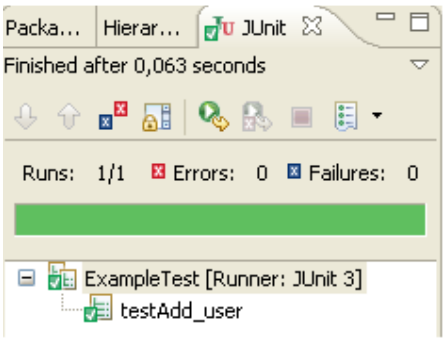
A typical JUnit test case is shown in Example 4 where an instance method is tested for a class and the result is validated for accuracy. An interesting feature of the framework is that the assert methods can be tailored which has been done for the assertEquals method in the example. Instead of using the standard implementation, the method is overridden so that the data contained in two User objects is compared to see if the data is equal instead of their pointer address which is checked through the framework method. Note that the testAdd_user method only tests that the Example class corresponds to its design intentions and that the test does not related to any sort of functional requirement. In the next section, another framework will be discussed which can be used to test for accurate behavior of units.

If other test cases are to use objects similar to the Example ex variable, these can be put as instance variables of the class that extends TestCase in the JUnit package to save resources. Such instance variables are called a Test Fixure which is further described by Beck and Gamma in [Beck98].



Example 5 – JUnit fail example

As mentioned, the test aims for validation of the expected output for a unit and a failing test case is showed in Example 5 where the test described in Example 4 has failed. As can be seen in the JUnit GUI illustration in the example, a failure trace is given so that the test case can be examined and thereby the failing source code unit. In this case, the code returned null instead of the expected User instance, shown in Example 4.



Example 6 – JUnit success example

Now, because of the early feedback in terms of a failing test case, the developer can alter the production code under test directly and rerun the test case(s) that failed. In example 6, it is shown how the JUnit GUI looks after a successful test run as illustrated by the green color and a zero count of both errors and failures.

Some of the most interesting concepts in JUnit will be described briefly below as an overview of the framework structure.

- TestCase. Classes that contain JUnit test case methods inherit from this class. This is also where a possible test fixture can be placed to share resources between test cases. A test case typically contains several test case methods, each of which tests some aspect of the same unit of code which means that the unit test responsibility is placed on the whole test case class.
- TestSuite. This class is used to group several test cases together. A benefit of this described by Beck and Gamma is that several individual developers can write their own test cases and later group them together to one test suite which can be executed as a whole [Beck98].
- Assertions. Every test case method can have several assertions such as the one shown in Example 5. There exist a large variety of different assertion classes, for example assertNotSame, assertNull and many others. The first parameter to most of these is the expected value for the second parameter which can be an instance of a class or some other return value from a method in the unit. When the assert method sense that this value or object does not correspond to the expected one, the test case has failed and a notification is given to the tester.

In addition to these classes, there needs to be some way of running the test cases and test suites and this is done by a so called TestRunner. The framework delivers both a console-based and a GUI based test runner. These runners visualize which tests that have passed or failed, the GUI-based output from Eclipse is shown in Example 5.

2.5.8 JBehave

As opposed to JUnit, the JBehave framework [JBehave07] focuses on validating the expected behavior of a particular unit which makes it suitable for behavior-driven development methodologies. The framework is similar to the JUnit framework in regards to its structure, classes and methods.

```
import org.jbehave.core.minimock.UsingMiniMock;

public class ExampleBehaviour extends UsingMiniMock {
    public void shouldAddUser() {
        Example ex = new Example();
        User u = ex.add_user("John Doe");
        User added_user = ex.get_user(u.GetUser_id());
        ensureThat("John Doe", is(added_user.GetName()));
    }
}
```

Example 7 – JBehave code example

The framework can test for several issues related to the functional requirements such as if one method is run after another. In Example 7, a behavioral version of the JUnit illustration shown in Example 4 is given. Instead of ensuring that the underlying structure of the unit is operational, it tests a user story which states that it shall be possible to add a user to the system. This responsibility is put on the Example object according to the systems specification which makes it natural to use this unit for testing. A User object is returned from the add_user method and to actually make sure that the User is added, the user_id of this instance is again used through another method to ensure that the correct user is acquire d from the system.

```
<terminated> ExampleBehaviour (9) [JBehave] C:\Program Files\Java\jre1.6.0\bin\javaw.exe (2007 mar 11 12:05:48)
F
Time: 0.797s

Failures: 1.

1) Example should add user:
java.lang.NullPointerException
        at ExampleBehaviour.shouldAddUser(ExampleBehaviour.java:7)
```

Example 8 – JBehave fail example

The ensureThat method is quite similar to the assert methods provided in by the JUnit framework and as shown in Example 8, this behavioral test failed and a failure trace is given so that the test case can be found and the production code corrected. In this case, it was a null pointer exception which was traced to an error of adding the user_id to the added user.

```
<terminated> ExampleBehaviour (8) [JBehave] C:\Program Files\Java\jre1.6.0\bin\javaw.exe (2007 mar 11 11:44:22)
.
Time: 0.062s

Total: 1. Success!
```

Example 9 – JBehave success example

And once again, the test case(s) can simply be re-run to ensure that the failing requirements have been fixed which the case as shown in Example 9 was. It can be concluded that both the JUnit and JBehave frameworks can be beneficial to have in any development setting due to its simplicity and its advantage in regards to regression testing.

3 METHODOLOGY

This chapter will provide the design of the phases in the thesis project, starting with an overview which describes the complete chain of activities. After this, each distinct phase and its purpose will be presented by itself.

3.1 Overview

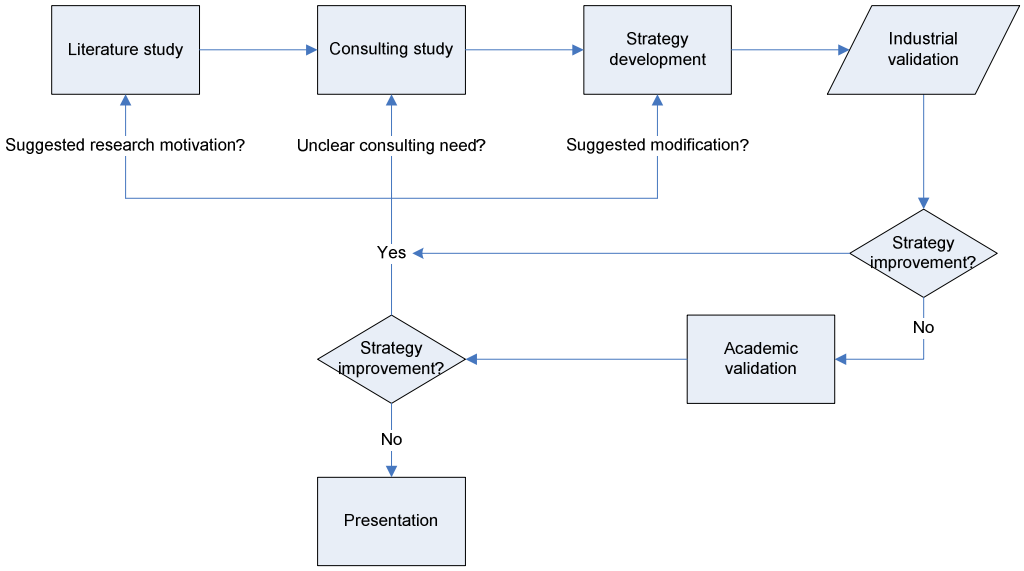


Figure 3 – Study design overview

As can be seen in Figure 3, the study was divided into five different phases which was iterated throughout the thesis project. An extensive literature survey was conducted in the former part of the study which resulted in relevant testing background for use in the consulting study. In the consulting phase, the consulting domain and related challenges was studied which resulted in useful information for the strategy development phase. This part included development of the actual automated strategy as well as the customer guidelines which was the primary focus of both the industrial and academic validation. To strengthen the validity further, the report was iterated several times with the supervisor in order to get valuable feedback which could be used to improve the thesis.

3.2 Literature study

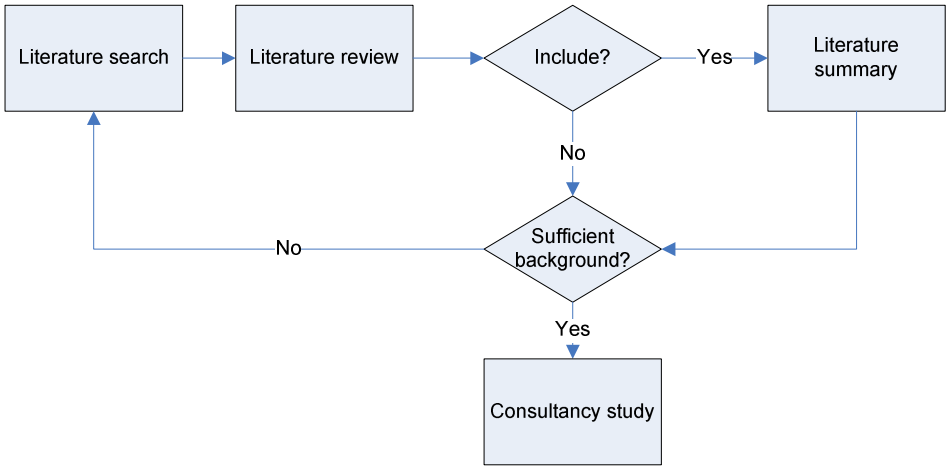


Figure 4 – Literature study design

In order to get the current state of research within the field of automated software testing, the literature study was initiated as the first phase of the thesis project. Figure 4 shows that this study was iterative, where careful consideration of each research paper was evaluated for quality aspects and relevance. To increase the chance of sufficient research quality, acknowledged literature databases was used and only peer reviewed material was issued for review. The primary intention was to find research that was empirically evaluated since this was considered important for the thesis. This importance was derived from the fact that the proposed strategy in the thesis is indented for use in a live industrial setting and not foremost an academic one. As can be seen in Figure 4, it was decided based on these premises if the study should be included or discarded from the thesis. For each included research paper, a summary of the most relevant parts for the study was chosen and discussed in the thesis. This work was iterated to the point where the acquired background was considered strong enough for the consulting study. However, at some points during the rest of the thesis project, this phase was iterated once more in order to find support for issues related to the customer guidelines and automated testing strategy.

3.3 Consulting study

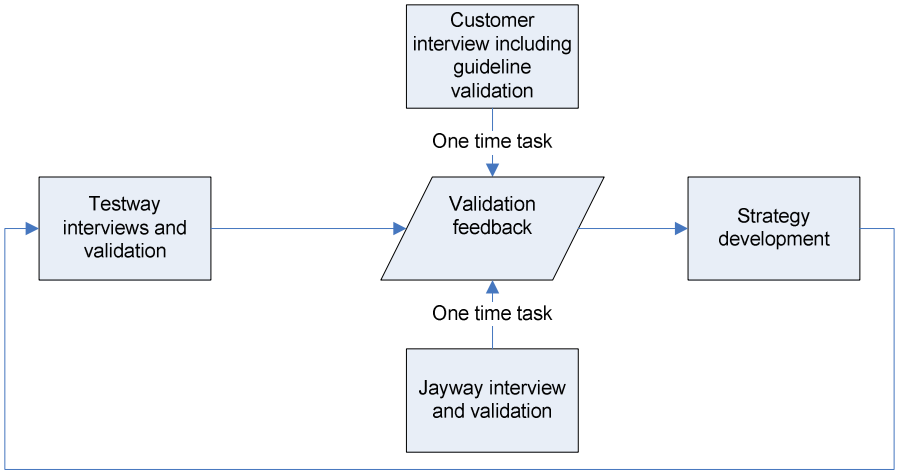


Figure 5 – Consulting study design

Fortunately, a test consulting firm offered the needed resources for the thesis. First of all, interviews were scheduled with an automated test consultant at the firm so that an overview of the automated testing practices could be acquired. Since the thesis author got the possibility to be on-site at the offices of the test consulting firm, interviews could be scheduled to coincide with the consultant pit stops to the offices. The interviewees were selected in order to acquire a complete picture of the test consulting domain, from technical specifics to test management and consulting management issues. Interview questions were designed based on the specialization of the particular interviewee. For example, when an interview was to be scheduled with a test manager, the questions were designed to elicit information about test management issues in the customer projects. Also, more technical aspects were covered through the interviews with the test automation consultant. The most relevant information gathered from these interviews about the consulting domain, the strategy and the customer guidelines was summarized into the thesis. Note from Figure 5 that additional one-time interviews were scheduled with a customer of the test consulting firm and a consulting development company (Jayway). The customer interview served as a validation point which helped to improve the strategy and the guidelines based on customer feedback of its perceived worth. A further discussion of the results from this validation point will be provided in Section 7.

3.4 Strategy development

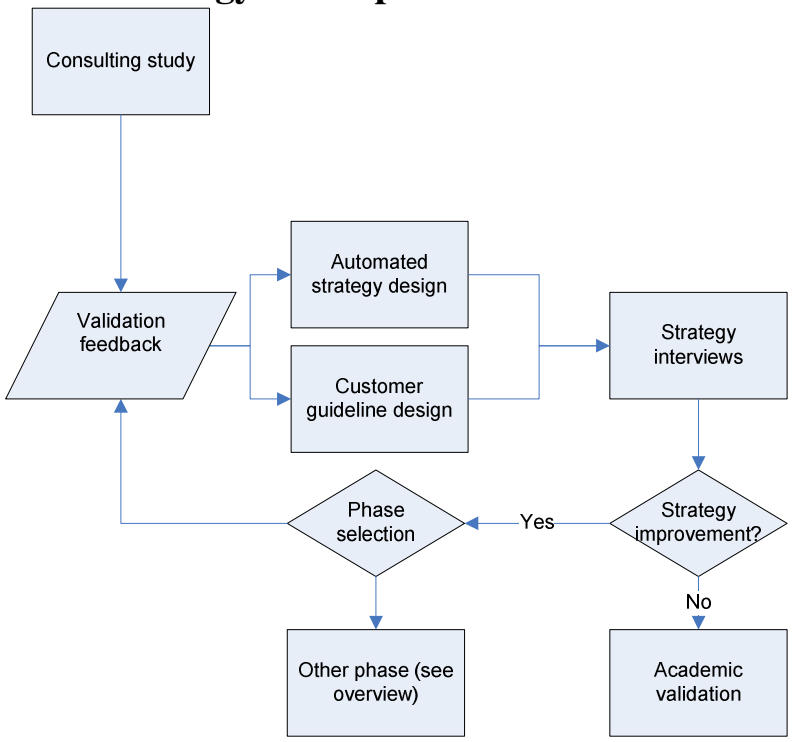


Figure 6 – Strategy development design

By evaluation of the feedback provided by the consulting study described in Section 3.3, this phase targeted the actual creation of the automated strategy and the customer guidelines. It started out with initial feedback from the automated test consultant at Testway where it was identified that the most common problem is related to the system requirements. This fact along with the fact that the consultants use system and acceptance testing most frequently formed the structure of the customer guidelines which serves as a complement to the automated testing strategy. As these are closely related, it was appropriate to design them in parallel as shown in Figure 6. The results were then validated through structured interviews at Testway which either led to another iteration of the design or a satisfactory results which initiated the academic validation. This extra validation process was needed in order to ensure the validity of the study in academia. Note that the phase selection choice in Figure 6 refers to the iteration of the literature study, the consulting study or the strategy development as shown in the overview in Section 3.1.

3.5 Academic validation

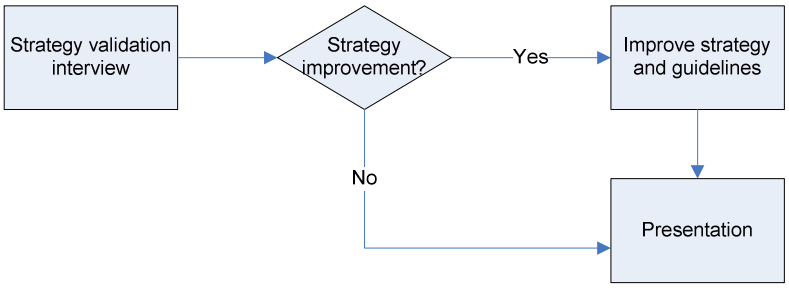


Figure 7 – Academic validation design

In order to strengthen the validity of the study further, one academic researcher within the field of validation and verification was contacted so that his view of the academic worth of the strategy and guideline could be assessed. As can be seen in Figure 7, an interview was conducted once due to the given time constraints of the thesis project. The acquired data from the interview was analyzed and a decision was made whether to improve the strategy and guidelines or not. As it turned out, some modification was needed. Further academic support was needed for the motivation sections in the strategy and the guidelines. Also, the automated tool selection section was considered weak. These sections were expanded and pointer support in terms of relevant research references was added. The overall academic validation showed that the approach can be generalized to traditional development as well since the pointers are not specific for consulting projects. Furthermore it was concluded that the pointer structure is relevant and the purpose of the strategy is visible. However, this researcher suggested additional academic validation by involving other researchers for the requirements and metrics pointers since these was outside his area of expertise. Unfortunately, this validation is left as future work since there was not enough time for these additional validation meetings.

4 TEST CONSULTING

This chapter starts with an introduction to firms in the consulting domain. Section 4.1 gives a consulting overview and discusses the role of the consultant. The following section (Section 4.2) is focused on development and testing differences between consulting firms and their customers. Next, a deep going discussion of the applied test consulting practices at Testway is given. Common for all sections in the chapter is that the information is derived from discussions with the development and test consulting firms in the case study which was previously described in Section 3.3.

4.1 Introduction

4.1.1 Overview

In general, consulting firms employ persons which have strong competence in a specialized knowledge domain and provide services to customers within this domain. The main characteristic of a consultant is that the domain knowledge is generally greater for the consultant than for a regular employee in the customer organization. Contracts are signed by the consulting firm and the customer that describe the services which shall be provided by the consultants. A general fact is that such contract is signed for a particular time period. If there is time left after the task has been completed, it is usual for the consultant to suggest additional tasks to be performed thus filling the gap in the contract. The opposite, however, can be difficult since it can be hard to persuade the customer to accept additional task that would affect the original contract or project deadline when it is discovered that additional services is needed. Three different types of consulting contracts have been identified through interviews with Testway and these are described in table 3.

Contract type	Description	Discussion
Project commitment.	The consulting firm takes full responsibility for a development project. The development can be done in-house at the consulting firm or at the customer organization depending on the customer arrangement.	The consultants cooperate and provide specialized competence for each development activity which reflects on the quality of the software since the process maturity can be expected to be greater for each development area.
Component commitment.	The consulting firm takes full responsibility for a particular project component or development phase. The particular project determines if it is possible to do this in-house at the consulting firm or if it needs to be performed in the customer organization.	In case a development phase or project component is considered important, it is beneficial to involve consultants that can provide strong knowledge and process maturity within that particular field to guarantee added quality.
Specialist consulting.	The customer hires a consultant for a specific task where the domain knowledge of the	The customer need not employ a full-time person for the specific task. Furthermore, since the consultant is focused on the particular problem area

	consultant is related to the specific task. When the task has been completed, the consultant leaves the customer organization.	while the employee often has broader knowledge, the competence within the task domain is generally greater for the consultant. However, it is more expensive to hire a consultant than to use a regular employee.

Table 3 – Consulting contracts

The following chapter defines consulting firms as companies that employ several persons and offer the services of these persons to their clients as consultants. It has been recognized that some consulting companies try to make their consultants key persons in the customer organizations. This way, the contracts can be prolonged and the number of leased personnel increased. Others, including Testway, tend to avoid this with the motivation of keeping their consultants flexible and their domain knowledge up-to-date. Since consulting firms differ a lot on this point, the question whether or not to make the consultants key players in the customer organization is still open for debate.

4.1.2 Role of the consultant

The primary responsibility of the consultant is to deliver the services described by the contract within the given time frames. Before an assignment starts, the consultants often make a plan based on the expected parameters of the particular project. Glass recognizes this as a problem since it is hard to know the exact valid conditions for the particular customer domain [Glass98]. It is important to realize that the plan may need alterations later on due to unexpected issues that can come up at the customer site.

In theory it is sometimes appropriate to guide the customer in directions that differ from the original problem statement. In other words, there is sometimes a need to convince the customer that the actual problem is something else than was originally contracted. In practice this can be a sensitive and hard issue to approach as a consultant. Some clients may feel uncomfortable for an external person to tell them what to do. However, these persons are also aware of the fact that problems do exist since the consultant contract has been initiated in the first place.

4.2 Differences between consulting and standard development

4.2.1 Development differences between consulting firms and their customers

As previously described, several types of consulting contracts exist. For specialist consulting, it is harder for the consultant to affect the development methodology since it depends on the process maturity at the customer site. Since the process maturity varies among the different customers, the difference between the used development methodologies differs as well. However, for project and component commitment contracts, the development methodology reflects the competence and process maturity in the consulting firm which is generally high within their areas of expertise.

Through discussion with Jayway, which is a consulting firm that specializes in Java development, differences between consulting and standard development been discovered. The development methodology used in the in-house project commitment projects at Jayway is very focused on testing aspects through the use of the test-driven extreme programming paradigm combined with the scrum methodology [Schwaber01] for project management. The consultants have determined that the quality awareness and test process maturity has been high in these in-house projects. This has had a positive effect on the software products. A comparison has been made to a particular customer project where the automated unit and

integration testing practices were neglected since the project management had not scheduled creation of executable tests. The test focus in this case was on the system test level which resulted in a system testing which found relatively simple defects which should have been found through unit testing. As previously described, system test shall focus on behavioral aspects. When defects slip through the unit and integration test level to the system test, the behavioral aspects of the system gets unreachable. The cost for system test increases because it has to be iterated several times due to the fact that the unit and integration defects have to be fixed before the next system test iteration. Another factor which leads to this increase in cost is the added lead time for the testers which had to wait for the individual developers that was responsible for the bug fixing.

4.2.2 Testing differences between consulting firms and their customers

Test consulting implies a focus on the validation and verification parts of the software process. However, for specialist consulting assignments, it is common that the test consultants only performs the system and acceptance testing at the customer site while the customer system developers are responsible for the unit and integration testing activities. In the current situation, such developers often have low test process maturity which comes from lack of experience, education and proper attitude towards software testing. This also leads to low amount of testing in the projects. Since the consulting firm may offer specialist consulting assignments such as unit and integration testing education for developers the test consultants can get involved in these levels as well but from a slightly different perspective.

When the consulting firm has been contracted for a project or component commitment, the testing influence is generally greater since they are controlled by the boundaries of the contract and not the higher management at the customer. In this situation, the development and testing processes can be adopted according to the existing special qualifications residing in the consulting firm. This may have positive effects on the software quality since the development methodologies can be used where test educated developers write the unit tests and where the system testers may influence the testability through the requirements and design.

4.2.3 Gap between consulting and reviewed research

There are no studies to the knowledge of the author that describes how current research within the field of software testing can be applied to a consulting setting in an efficient manner. Much research assumes standard development in the project where items such as the development methodology may differ but not the organization that surrounds it. This means that the worth of this research within the consulting settings is unclear. For specialist consulting assignments, the challenge revolves around the dynamic adaption of the research approaches to different development states.

4.3 Consulting at Testway

As previously described, Testway tries to avoid making their consultants key players in the customer organization. The precise time-span for a consulting assignment differs depending on the wishes of the individual consultant. Three to six months is the preferred contract span for one consultant that was interviewed during the case study while others prefer more long-term contracts. There are currently eight consultants in the firm and one manager which imply that the firm can be classified as a small consulting firm. Testway is considered to be a test specialist firm where the consulting services constitute one part of the operation together with other services such as education within their field of expertise. By providing education through specialist consulting services, the knowledge of how to solve problems within the testing area can be transferred to the customers directly and thereby avoid making the consultants key players in the customer organization. This means that it will be possible for the ordinary personnel to perform the testing practices themselves when the consultant leaves the organization. However, many customers do not strive for education within the field. Rather they need the consultant to perform some testing activities for them which is the

most common scenario. It is also worth to mention that Testway has customers from several applications domains from industrial automation companies to pure software development firms. The size of the customer organizations also vary from small up to medium and large scale companies.

As described earlier, three main types of consulting contracts has been identified and Testway provides services according to all three of these. Table 4 gives a brief summary of the consulting services provided within the three contract types at Testway. These are also described at [Testway06].

Service	Description	Contract type(s)
Test management	Consultant test managers plan, manage and follows the test practices in the customer projects. These projects may involve testers in the consultant organization as well as other consultant testers.	Project commitment. Component commitment. Specialist consulting.
Test process improvement	The current test process in the customer organization is assessed and evaluated for areas which could be improved. This is done by combining the Test Process Improvement (TPI) model [Koonen99] with the TPI Automotive model [Sogeti04].	Specialist consulting.
Test strategy	This service helps customer organizations arrange and plan testing practices for organization wide or project specific testing.	Specialist consulting.
Test automation	Test consultants can implement test automation, identify automated testing opportunities and suggest automated testing tools for organization wide or project specific test automation.	Project commitment. Component commitment. Specialist consulting.
Load and performance testing	Test consultants perform load and performance testing on selected software components using automated testing tools.	Project commitment. Component commitment. Specialist consulting.
Training	Testway offers seminars within the field of software testing as well as courses which lead to test certifications. Furthermore, training on-site in the customer organization is also performed on demand.	Specialist consulting.

Table 4 – Testway consulting services

4.3.1 Current state

Currently, there is no official automated testing strategy at Testway since it is hard to develop a strategy which covers all customer assignments. Instead, the strategy is built ad-hoc at the customer site, depending on the different variables in the particular project. The consultants mostly act at the system and acceptance test level which implies that sufficient requirements understanding are needed for the assignments. Unfortunately, it has been recognized that the quality of the requirements in the customer projects vary depending on the process maturity of the particular organization which becomes a testing problem when the consulting assignment begins. This is also the reason why the company advocates the use of development methodologies where testing initiated done early such as iterative and test-driven development.

Currently, the requirements problems are solved on-site by speaking with the involved stakeholders in attempts to elicit the requirements after development. In agile development methodologies this may be only a minor problem since agility often has the benefit of close contact with on-site stakeholders. Most customers intend to use some sort of iterative process but unfortunately it is common that this end up with a more traditional approach where the software product is delivered to the tester after implementation. It can be difficult to perform requirements elicitation in the testing phase of such methodologies since the development methodology does not require the presence of the original stakeholders in that phase of development.

4.3.2 Test levels

The attitude towards testing differs a lot depending on test process maturity in the customer organization. The customer decides which test levels are to be used and the test consultant has little say in the matter. However, when the test management services have been contracted, the influence possibilities are greater in comparison to the other services.

As mentioned above, the most common test levels used are system and acceptance testing. The test consultants rarely act at the unit test levels since these tests are expected to be performed by the developers themselves. Pyhajarvi and Rautiainen have recognized that this is a common way of looking at the test levels in organizations which uses the traditional V-Model [Pyhajarvi04]. However, if the test process maturity is low in the customer organization, this practice is often done in an unguided manner or not at all. Because the customer decide which consulting service to contract, it is hard to get them to realize that proper unit testing can increase the quality of the software product in terms of added stability and system level testability. In rare cases, customer organizations contract consulting services which involve developer unit testing training at the customer site and this point to high test awareness in the organizations. In the cases where the test maturity level has been low it has been hard to convince the developers to adapt unit testing since they have problems visualizing the expected benefits.

Table 5 gives two examples of previous unit testing consulting assignments conducted by Testway. As mentioned above, the system test level is the most common level and two typical examples of automated system testing assignment is given in the table as well.

Service	Description	Test level	Contract type
Training	The customer had started adopting agile development in the organization and set the branch coverage goal to 95% for their unit test cases. They initiated a consulting contract for unit testing training of their developers in the project.	Unit test level.	Specialist consulting.
Test automation	Unit tests were required by the	Unit test	Specialist

	customer development policy which is unusual in this domain and posed the same requirement on the consulting contract. This assignment involved creation of automated test cases in the customer project.	level.	consulting.
Test automation and training.	A regression test suite was required by the customer and thus created by the test automation consultant. Furthermore, a test automation framework was created to support this regression suite which was to be used by the developers themselves. The assignment was concluded by developer training of this framework.	System test level.	Specialist consulting.
Test automation and test strategy	There was much legacy code in the customer organization that did not have automated tests. The automated strategy that was compiled for this organization determined that every new feature and required bug fix were to be automated but not the existing code. This strategy proved successful and resulted in a large regression test suite which was created by the test consultant.	System test level.	Specialist consulting.

Table 5 – Testway consulting assignment examples

4.3.3 Reuse challenges

There is no test case reuse strategy at the company but there is a sound reason for this. Since the customers pay the consulting firm for the development and tailoring of automated test cases, these test cases cannot leave the customer organization. However, there are possibilities for reuse since each individual consultant gather their own set of domain knowledge which can be shared with the other consultants that act in other organizations. In other words, it may be possible to reuse the knowledge of the various consultants in order to increase the total knowledge in the consulting firm. This is done to some extent already in form of ongoing seminars where the different consultants share their experiences through lecturing. As mentioned by one of the consultants at Testway, the individual knowledge of test methodologies are constantly reused and improved in the sense that these are tailored to fit into each new customer site. The notion of knowledge reuse has been taken into consideration and is introduced as a step in the automated testing strategy which is further described in chapter 5.

4.3.4 Customer development issues

Consultants can arrive in several phases of development and the most usual scenario is that the consultant arrives in the testing phase in a waterfall-like development model. This can be particularly hard for a consultant due to the learning curve often needed for sufficient testing. It has been recognized by the consultants that the time schedule for the testing phase in sequential development methodologies are often decreased in favor of the other development phases. Most customers that use a traditional development methodology deliver an implementation to the testers after it is completed. This means that the time spent on testing is not actually the time that often is required but the time that is left after the other phases has received their fare share of time. There have also been attempts to inform the management at the customer site of the product quality drawbacks that this imposes. However, there has been a trend that the management is more interested in meeting the deadlines than the

delivery of a quality product. Note that managers may be willing to add more resources in form of people to the testing project but not more development time. This is challenging since the extra added persons needs to be brought up to speed and trained which take further time from the actual testing. The project managers closest to the project commonly understand the need for further testing but are often constrained by time schedules imposed by higher management. On the upside, these facts are about to change since more and more customers starts to realize the benefits gained by thorough software testing.

4.3.5 Automated testing

Not many of the customer organizations have gotten very far in the field of automated software testing. It has been identified that this does not come from lack of developer knowledge of automation but from higher level management that expect shorter development time for each new project. Test automation is not scheduled since this is expected to increase the development time. Creating automated test cases will cause initial development overhead but compared to manual execution, overall execution time will be decreased for each test case regression. This fact is often not taken into consideration when estimating the test execution.

One interesting issue related to the automated testing practices at a customer is that the developers had recognized a need for a commercial automated testing tool in several projects but that the project manager did not want to spend project resources on a tool which were to be used over the entire organization. This way, the purchase of the tool was postponed with the intent that it could be bought in the next project where there were more resources. However, the same problem of course occurred here as well. What can be learned from this is that such purchases should be brought up to the organizational level so that the tools are indeed brought into the organization. On the other side, it should be noted that careful tool selection is needed to avoid the bias imposed by commercial tool vendors which only displays how easy it can be to test certain items. Most often, the tools needs to be complemented with manual testing due to missing features such as the inability to test several applications sequentially. For example, a test application may be able to do a system test on an application that adds data to a database while it lacks the possibility to launch a test script that checks the actual database contents in sequence which could be appropriate for the test case to be complete. In such case, a manual effort is needed for the test script. Such chains of interactions are needed in system and acceptance testing since the levels test implemented behavior which can be spread over several applications. This does not mean that the test application should not be purchased but it does mean that it shall be noted that such features is missing in the tool.

A typical scenario for automation is that the customer has some existing sets of manual test cases that they want to automate in order to save resources in form of manual testers. Usually, this type of assignment starts with a workshop where the customer and consultant sit down and discuss which of these test cases would be appropriate to automate. Test selection is used and the test cases up for automation are eventually prioritized. Factors such as how tiresome the manual test cases are to do manually, how prone they are to change and if they even can be automated is considered when doing this prioritization.

In most cases, it is not possible to automate every of the manual tests and the strategy instead is to write executable test cases to get large system coverage. A technique that is used by the automation consultants is partition testing where similar tests are gathered into collections that corresponds to different parts of the system under test. This way, each part of the system gets some sort of testing which is considered better than to focus the testing efforts to some single component. However, for some customers there are critical components that needs to receive higher priorities and in these cases, partition testing is not the most appropriate way to go. In these test cases, it is usual to include techniques such as boundary checking. Furthermore, a data-driven approach is often attractive for the test cases.

The programming language used when doing automated testing depends on the language used in development. Many projects are web based which leads to languages such as C# and Visual Basic. Ruby is another language that is common in web based testing due to its possibilities of testing code written using other languages.

It has been recognized by the consultants that it is important for the test automation that testability is designed into the software. These assignments sometimes require the software to provide so called *software hooks* so that the test cases can interact with through these hooks to verify that a given input gives the correct output. If such issues are not taken into consideration in the design it is hard to automate tests for certain components and application types.

It has been recognized that the management at the customer site often require statistics about the progress of the automated testing which is a sign that they are involved in the process to some extent. However, the management tends to view the number of test cases as a good measurement of this progress and not the quality of them which would be a better measurement.

5 CONSULTING AUTOMATED TESTING STRATEGY (CATS)

This chapter starts with an introduction to the automated testing strategy where the scope and motivation is described. Thereafter, an overview can be found which illustrates the strategy as a flowchart. The core of the strategy is then discussed in the following three sections, starting with the preparation phase which is then followed by the execution phase and finally the post execution phase. There are some pitfalls that could be avoided when following the strategy and some of these are discussed in Section 5.5 which is also the final section.

5.1 Overview

The automated testing strategy is developed for use by consultants that primarily deal with test automation in software development projects but some parts of the strategy may be useful for manual testing as well.

5.1.1 Strategy concepts

Strategy pointers in the following sections are distinct tips that can be applied in different phases of the testing project with the intent to increase the efficiency of the testing practices. The pointers can be applied independently of each other, depending on the parameters of the current development project and organization. As for the different phases of the strategy, these are not to be confused with the phases of the used development methodology since the strategy phases are independent of the development methodology. The main concepts of these phases are to increase software testability and stability, increase the effectiveness of the test execution and to improve the strategy and customer guidelines with the execution results as the input source. The pointers are structured so that the test consultant can assess the pointers independently and choose which pointers that applies in the current development phase.

5.1.2 Strategy scope

In section 4.1.1, it was stated that there are several forms of consulting and the primary scope for this strategy is to be efficient in specialist consulting projects where the test input comes from development projects where other teams has done the actual development. It may be possible to adapt the strategy to project and component commitment projects as well but this is out of scope for this thesis.

5.1.3 Severity scale

The severity scale in table 6 is used by the “Prioritize defects” pointer which can be found in Section 5.3.2. As mentioned for the particular pointer, the found defects need to be prioritized so that the most critical defects can be found in the defect report. Of course, if the organization has a defect reporting system which has another priority scale, this could be used instead since the main point is that the defects should be prioritized in one way or another.

Severity	Description
5	Critical defect
4	Serious defect
3	Defect
2	Minor defect
1	Insignificant defect

Table 6 – Severity scale

5.1.4 Automation prioritization scheme

The prioritization scheme in table 7 is used by the "Prioritize the tests selected for automation." pointer in Section 5.3.1. The pointer describes that the test selected for automation should be based on the corresponding requirement prioritization and this scheme is meant to describe the importance of automating the test cases.

Priority	Description
5	Critical
4	High
3	Normal
2	Low
1	Minor
0	Should not be automated

Table 7 – Automation prioritization scheme

5.1.5 Motivation statement

As mentioned, testability and stability is important for the quality of the software release. Because it is not always the case that the customer has the correct understanding of their current problems in terms of automated software testing, it is necessary to guide this understanding in some situations. Furthermore, if the consulting firm can motivate the use of advanced testing methodologies in favor of increased software quality, it also increases the value of the service set provided by the firm.

If the customer contacts the consulting firm in the start-up phase of their development project it can be appropriate to take some measures to ensure that testability and stability is reached. Otherwise, when the customer organization has low testing maturity, the requirements for example may not be testable due to ambiguities. Furthermore, insufficient use of unit and integration testing also introduces low stability which affects the system testing. The system test may find defects that should have been discovered by previous test levels, defects that differ from the goal of finding behavioral defects.

To alleviate these problems, the preparation phase is introduced where such issues are handled. Unfortunately, it is not always possible to have such a large impact in the customer organization so the strategy must cater both for situations where we have and situations where we do not have a high level of testability and stability in the target system. This is why the execution phase provides pointers that can be applied even when the testability and stability has not been affected in the preparation phase. This is needed since the test execution will differ due to the issues that the preparation phase is expected to handle. The strategy needs constant improvements in order to stay efficient and effective. This is the reason for the post execution phase which is the last phase of the strategy. Here, the test execution is analyzed to find areas in which the strategy is weak or not up-to-date with the current state-of-the-art in automated testing and software development. This can then lead to strategy improvement proposals and discussions.

5.1.6 Structure of strategy

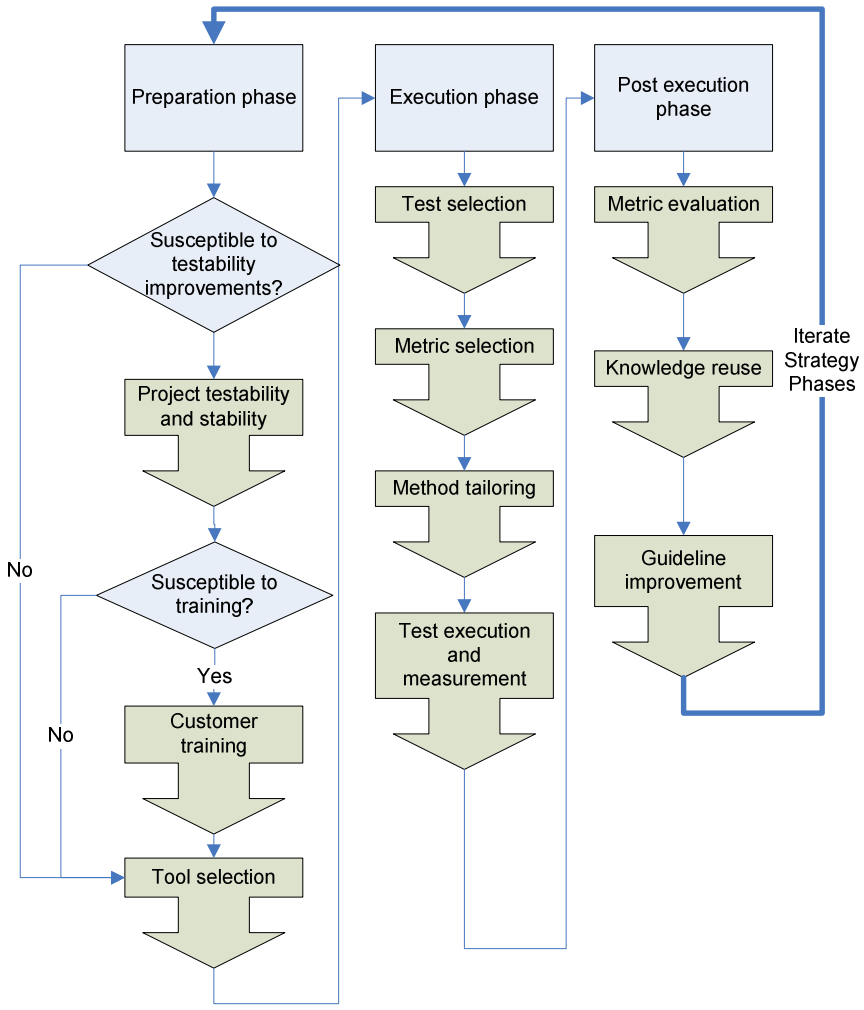


Figure 8 – Automated strategy overview

As shown in Figure 8, the strategy consists of three main phases; Preparation, Execution and Post execution phase. Note that the strategy phases are independent of the development methodology and should not be confused with the development phases of the current methodology. Each phase of the strategy has a set of tasks, each of which is responsible for some aspect of the total quality assurance process. The purpose of the strategy is to increase the efficiency of the testing, both within the current project as well as other projects in the consulting organization. The testability and stability are the first targets of the strategy as can be seen in the preparation phase in the figure which is necessary in order to facilitate the test automation which is done in the execution phase. The execution phase is where the actual testing is performed and where the test methodologies are adapted to the current situation. Metrics should be collected during execution so that the test results can be documented and reported to the management. The metrics also serve as a means of strategy and guideline improvement which is done in the post execution phase. Note that this phase also includes knowledge reuse which aims to improve the total knowledge within the consulting firm so that the experiences collected by the individual consultants are shared. The figure illustrates

that the strategy phases are iterative which means that the actual strategy is iterative even if the current development methodology is sequential.

For example, if the project uses the traditional waterfall model where the software is delivered to the consultants in the test phase, the testability and stability focus in the preparation phase should target future releases from the organization since it is too late to affect this for the software under test. When the tool selection step is started, the current testability and stability should be assessed so that it can be determined what related problems the tool needs to circumvent in the execution phase.

5.2 Preparation phase

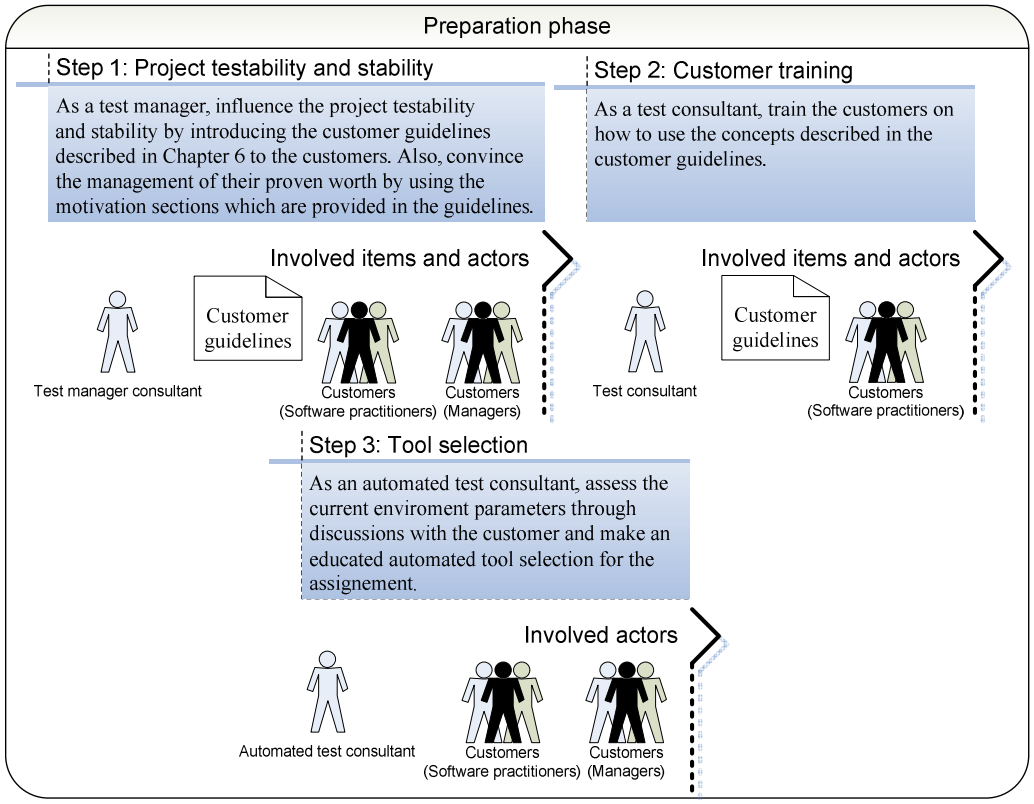


Figure 9 – Preparation phase

An overview of the steps in the preparation phase is given in Figure 9 which also illustrates the involved items and actors for each step in the phase. As can be seen in the figure, the phase consists of three steps; Project testability and stability, Customer training and Tool selection each of which will be introduced in this section. Many factors can impact the testability and stability in late phases of development such as complex design and ambiguous requirements. In order for the customer to get full value of the consultant services they need to facilitate testing by considering such factors in early development. The preparation phase is the first step of the strategy which begins with customer preparation since it aims to affect the testability and stability of the customer project in a positive direction. This can be considered the hardest step in most situations since customers seldom realize the importance of early testing activities, testable requirements and testable design which makes it hard but necessary to influence this. In the traditional waterfall model for example, it is hard to affect the testability and stability of the current release when it has been delivered to the testing phase. However, the preparation phase of the strategy may be used to affect the testability and stability of upcoming releases or projects since it is designed to increase the test process

maturity of the entire customer organization. The first subsection focuses on the testability and stability of the software development projects and gives advice on how this could be increased. This is followed by the customer training step which may be required if the customer organization has agreed to adopt certain guideline pointers. As can be seen in Figure 8, these two steps are optional since it may not be possible to influence the customer organization for one reason or another. The final subsection deals with issues that should be taken into consideration when choosing among different automated tools for the test execution.

5.2.1 Project testability and stability

As mentioned, system and acceptance testing targets the requirements with the distinction that the input to the acceptance testing comes from the customer while the inputs to system testing come from the testers or developers. The main difference between the strategy and the guidelines which are provided in Section 6 is that the guidelines are intended for customer use and the strategy is indented for consultant use. The guidelines in Section 6 are indented to influence the testability and stability of the software development projects at the customer site. Table 8 contains a pointer which aims for increased testability and stability of the software development projects at the customer site.

Pointer	Description	Motivation
Customer guidelines.	Try to convince the customer of the testability and stability benefits gained by adopting the guidelines in Section 6 in regards to the consulting services which they need.	As a test consultant it is important to realize that it is your responsibility to make sure that sufficient testing is performed. When there are problems with the software testability and stability this issue should be dealt with in order to give the customer full value of the consultant services. The importance of testability for test automation is also recognized by Pettichord in [Pettichord02] where three main issues are identified as important in this regard; Cooperation between testers and developers, team commitment and early involvement of the testers.

Table 8 – Project testability and stability pointer

5.2.2 Customer training

Training on how to use the guidelines can be an effective way to help the customer increase the testability and stability in the project. Introduce this as an optional step to the customer and if the customer is willing to accept such training, let consultant train the developers in how to use the guidelines. Table 9 contains pointers that focus on customer training issues.

Pointer	Description	Motivation
Developer and manager persuasion.	Persuade the developers and managers of the benefits that are gained with the adoption of the guideline pointers. Bring forth previous quality results that can be traced to the guideline adoption at the particular organization where this was visible. Since it is likely to have several customers which act in the same application domain, the primary focus should be to show	It is important to convince the developers and managers of the benefits that come out of each relevant guideline pointer since the adoption level will decrease if these are not visible. As mentioned by Pettichord, it is important to have a full team commitment if high testability is to be achieved in the software projects [Pettichord02] which can be achieved by convincing them about the benefits

	previous successful results from an application domain similar to the current one.	that comes with this type of testability.
Developer training.	If the concepts of the pointers require education, provide proper training so that the pointers can be successfully implemented by the developers themselves. Gable addresses the importance for the consultant to have a superior knowledge set compared to the ones held by customers in the domain [Gable03]. With this in mind, make sure that the concepts are properly understood prior to the developer training.	Without proper understanding of the pointers, these may not be implemented to deal with the goal for which they were intended. In this case, the testability and stability will not be increased to the extent that was intended.

Table 9 – Customer training pointers

5.2.3 Automated tool selection

It is most common to use some sort of automated testing tool, framework or script language for the creation of the automated test cases. There are several tools available for automated system testing such as Watir [Rogers07], SilkTest [Borland07] and many more, each of which has its own advantages and disadvantages. However, organizational needs should be considered as well as the customer setting before acquiring a tool. In case a commercial tool is considered, its applicability in several application domains should be considered as well, otherwise the tool may end up on the company shelf and never be used again which probably means that the return of investment for the tool will be low. In addition to this, many tools lacks the ability to perform specific subtasks which are usually performed manually after the test case has been executed. Make careful assessments and involve consultants from other project in order to get a united view of the tool under observation so that such missing features are brought forward and discussed prior to the assignment.

Since automated system testing tools such as SilkTest uses a record and playback approach, it assumes that the functionality is in place prior to the test case creation which implies that it cannot be used in a test-driven development setting where the test cases should be produced prior to the production code. In such cases, consider script based languages that can be used to communicate with applications through some communication protocol such as the Component Object Model for a windows setting [Microsoft07].

Note that for the system testing phase, it is important to choose a tool that can target the current system requirements in an efficient manner, an issue that should be considered in the selection as well. Table 10 describes key factors that should be considered when doing the tool selection. Note that in order to stay efficient, this list is supposed to be extended as new factors are discovered at customer sites.

Pointer	Description	Motivation
One-time projects.	If it is unlikely that a similar project is to be conducted in the future, it is appropriate to select a tool where the learning cost combined with the purchase cost does not outweigh the current	Be careful to purchase tools to the organization that is unlikely to be used in the future since one project will probably not produce the return of investment needed for the purchase and learning time of a particular tool. Poston and Sexton mentions that testers needs training in tool operation, tool input preparation and tool output use and mentions

	expected return of investment.	that these three activities should be included in the cost estimations when considering the tool [Poston92].
Consider execution analysis.	Many tools lack sufficient execution results analysis. This may require additional analysis to be done manually after the tool has been executed. The expected costs for this should be considered when selecting the tool.	As previously described, an oracle is a program that automatically checks the results from a test execution. It has been recognized by Yang et al. that most tools still require some human interactions for creating the test oracles [Yang06] which should be based on the behavioral specification. The results analysis is a large and important part of the total test case execution and if the tool has little oracle support, it may be appropriate to look for another tool. This importance has also been recognized by the development consulting company where the case study was conducted.
Test case design.	Design the test case structure prior to the tool selection. Then select a tool which has the proper support for the implementing the test suite. In other words, do not let a particular tool guide the test case design.	The stability of the test suite decreases if it has to be designed to cope with the limitations imposed by an already selected tool. It has been recognized by test managers at Testway that unstable test case design increases the maintenance time of the automated test case suites.
Integration support.	Ensure that the tool has integration support for the development environment used in the project.	The importance of this tool feature has been elicited through an interview with a development consulting firm. This coincides with their continuous integration practice since the build-in support for unit testing speeds up the test execution.
Tool evaluation.	Conduct a tool evaluation of current available tools where the tools are compared against each other with the current project parameters as the relevant support criterion. Poston and Sexton have proposed a structured method for conducting tool selection which could be used for guidance [Poston92].	A complete review may be needed for the tool selection to be efficient. As mentioned by Poston and Sexton, if the results from the tool evaluation are not quantifiable, the managers may not be convinced that the tool is worth purchasing [Poston92]. With this in mind, a structured evaluation would be appropriate so that the most appropriate and efficient tool is chosen.

Table 10 – Automated tool selection pointers

5.3 Execution phase

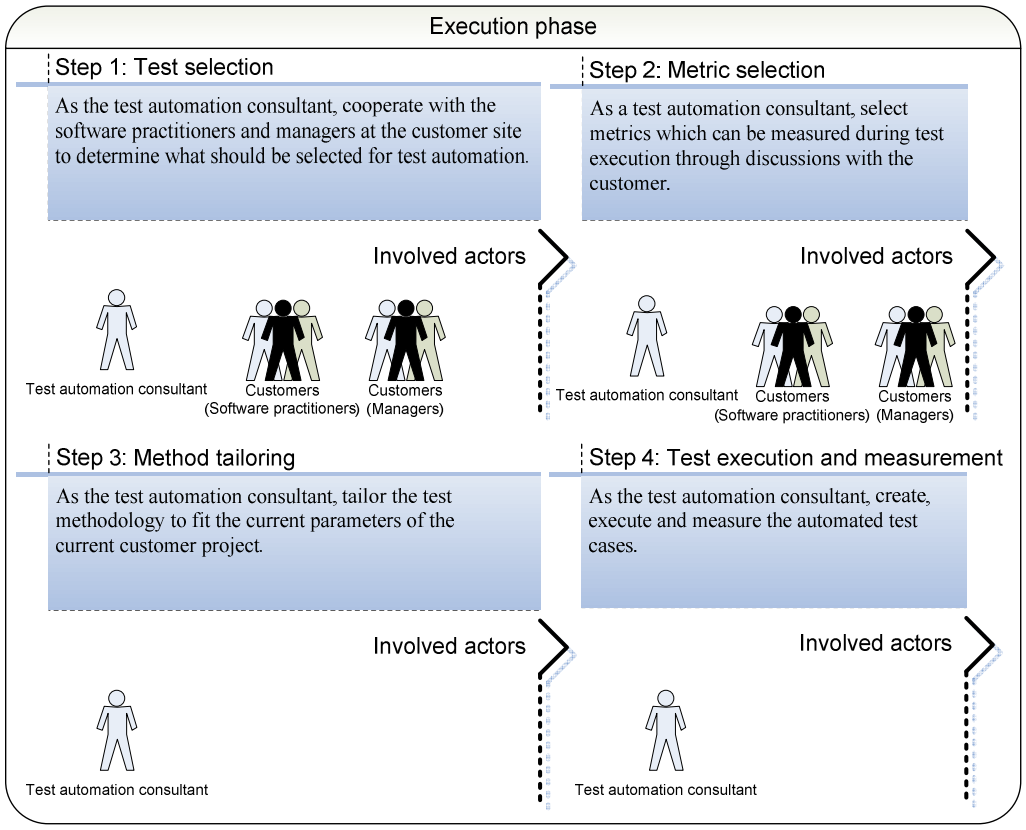


Figure 10 – Execution phase

An overview of the steps in the execution phase is given in Figure 10 which also illustrates the involved actors for each step in the phase. As can be seen in the figure, the execution phase contain four steps; Test selection, Metric selection, Method tailoring and Test execution and measurement. The execution phase contains pointers which should be considered when starting the actual test execution. This section starts with a subsection that focuses on issues that should be taken into consideration when doing a test selection for a particular project. Section 5.3.2 considers metrics that should be collected during test execution. Section 5.3.3 describes how to tailor the testing methodologies towards the different customer projects and domains. The last section describes how to perform the actual test execution and how to measure the execution based on the metrics chosen in the metrics selection step.

5.3.1 Test selection

The test selection is important for quality assurance purposes since it affects the quality and test execution performance if unimportant items are selected for testing. Table 11 contains pointers which should be taken into consideration when performing this test selection.

Pointer	Description	Motivation
Design walkthrough.	Try to get the developers to provide a design walkthrough so that this information could be taken into consideration in the test selection. Document these meetings in some manner, for example, audio recordings or written summary notes.	Since discussions with Testway has shown that many projects lack the needed design documentation, it would be appropriate to collect this information through other means such as a design walkthrough. Furthermore, Aurum et al. has done a research review in the field of software inspections which is similar

		to walkthroughs [Aurum02]. In this review they conclude that such inspections are considered cost-effective in regards to defect removal and increased software quality.
Prioritize the tests that are selected for regression test.	Prioritize the automated tests according to the relevant criterion. There are techniques available such as the Echelon approach by Srivastava and Thiagarajan which uses binary versions and the coverage information about the old version to determine the most effective prioritization of the regression test suite [Srivastava02].	Since time and resource constraints may affect the amount of tests that can be executed, it is important to have priorities for the test cases so that this can be taken into account prior to the execution. Furthermore, as recognized by Srivastava and Thiagarajan, effective test prioritization can save time and resources in early development [Srivastava02]. Furthermore, is has been empirically proven by Elbaum et al. that the fault detection rate differs a lot depending on the used prioritization technique [Elbaum00] which makes technique selection an important issue.
Prioritize components.	If automated tests are to be developed for developed system components that do not have previous manual tests, special attention should be put on the component criticality. Furthermore, have regression testing in mind and estimate how many times the test needs to be executed. If changes to the component may affect other components, the test would probably need to be re-executed several times.	As described by Kaner in [Kaner97b], it is not economically defendable to automate tests that only need to be executed a few times due to the large initial overhead in creating, verifying and documenting automated test cases.
Prioritize the tests selected for automation.	Prioritize the tests that are selected for test automation. Since tests that are considered hard to execute manually are hard and time consuming to automate [Keller05], these should be given lower priority. The prioritization should be based on the requirement prioritization and the scheme in 4.1.4 would be appropriate to use for setting this priority. Also provide a motivation to why the priority has been set.	As mentioned by Keller et al. tests that are hard to do manually are even harder to automate [Keller05]. This implies that such manual tests should be given lower priority since these will probably not give a sufficient return-on-investment.
System partitioning.	Investigate the system structure and define partitions so that tests can be categorized. In this way, system coverage can be	This way, each system partition gets some amount of testing which is beneficial in case full coverage is not possible. This has been proven successful in some of the automated

	measured. After this has been done, select tests with a fairly even spread over the different partitions.	testing assignments done at Testway.
Follow the test automation manifesto.	Create test cases that are easy to read and maintain. The test automation manifesto described in [Meszaros03] provides guidelines towards a more maintainable test suite.	Meszaros et al. recognized the importance of maintainable test suites and developed the test automation manifesto which serves as guidelines towards this type of maintainability [Meszaros03]. It is also mentioned in [Meszaros03] that the authors have created automated test cases that require less refactoring when following the manifesto. However, since this study has yet to be empirically evaluated, the impact on testability has not been proven.

Table 11 – Test selection pointers

5.3.2 Metric selection

In order to judge the effectiveness and efficiency of the current testing practices, they need to be quantified and measured. To do this, a relevant set of metrics should be collected during test execution. These metrics should later be used as an indication of the current quality of the software product. Furthermore, they also serve as a means of improvement for the customer guidelines. Table 12 contains relevant metrics that can be used during the test execution.

Metric	Description	Motivation
Requirements coverage.	Measure the requirement coverage of the system. As mentioned by Lipaev, the requirements coverage analysis should determine two things; How complete testing that has been achieved in regards to the requirements and what additional test cases are needed in order to achieve full coverage [Lipaev03]. Lormans et al. describes requirements views as a means of keeping track of the requirement test coverage and propose a method for this type of requirements traceability in [Lormans06].	In case a system test is performed, the requirements coverage is a good quality measure since it can be determined how much of the functionality has been tested.
Number of defects.	Note the number of defects.	According to a consultant at Testway, the project management in customer organizations is often interested in the number of found defects which they use as a quality indication. Collect this number to satisfy the demand.

Prioritize defects.	Prioritize each defect according to a classification scheme. If there are no support for this in the defect reporting tool used, use the severity scale found in Section 5.1.2. If no defect reporting tool is currently used, Bugzilla [Bugzilla07] is recommended since it has advanced reporting features.	Since it is possible that every defect cannot be scheduled for immediate attention, it is important for the tester to provide an indication of the severity of the found defects.
Log execution time.	Log the execution time for the first execution of a test case. If the system partitioning pointer described above is adopted it could be beneficial to measure the execution time for an entire partition instead since a single test case has low execution time.	Since large test suites can take significant time to execute, each test case should be measured in terms of time so that the time for a full regression test can be estimated prior to the execution. As mentioned by Hayes this can also be useful in order to find performance problems [Hayes95]. With this measure in hand, the most critical test cases could be scheduled depending on the available time left for test execution.
Test execution progress.	Keep a progress report of how many tests that have been executed and how many of them has failed. This pointer applies if the test execution spans over several days. The progress report should be updated on a daily basis.	First of all, if the testing time gets cut, it serves as a quality measure since it can be determined in percentage how much of the system that has been tested. Secondly, it can serve as a stability measure. For example, in the case where 40 out of 50 test cases failed in execution, the software release was probably not ready for system testing in the first place. It is also recognized by Galli et al. that a single defect often results in several failing test cases [Galli03] which can be a result from this type of low system stability.
Note defect cause.	Note the cause of a found defect or failure. Note that a system test may not reveal this information but it can help the debugging process if this can be provided.	The noted defect causes are important for future projects since these could be used to convince the management to allocate resources for further guideline pointers. They could also be useful for the enhancement of the guidelines as indicators of how well an adapted pointer works in practice. If the defect causes is not related to the intent of the guideline pointers that are adopted it can be an indication that the guideline pointer is not implemented correctly.

Resources.	Note how many people that are involved in the testing along with the time spent for each of these.	This information is needed for the validation of the roles section in the guidelines. The need for resource allocation information was elicited by an interview with a consultant test manager at Testway which considered this to be needed information for convincing upper management to adopt the guideline pointers.

Table 12 – Metric selection pointers

5.3.3 Method tailoring

Since the starting time for each testing project differs, the methodologies used needs to be adapted to fit in to the particular state of each given project. In table 13, a tailoring pointer is presented which can be useful. Keep in mind that the table is not final and should be extended or modified when the pointer efficiency is measured.

Pointer	Description	Motivation
Adapt the test methodology.	Adapt the test methodology to fit the current customer project parameters. Consider the systematic method tailoring approach introduced by Murnane et al. in [Murnane05]. This approach breaks down black-box methods to individual rules, each of which can be combined to form a hybrid method which could be applied to the current situation.	As mentioned above, Murnane et al. recognized difficulties in the using black-box testing techniques in different application domains [Murnane05]. As their approach deals with changing application domains which includes the consulting domain, this approach could be appropriate to use for the method tailoring process done by the consultants.

Table 13 – Method tailoring pointer

5.3.4 Test execution and measurement

In this phase, execute the tests and measure the results in regards to the selected metrics. It is important to ensure that sufficient data is collected for each relevant metric so that is actually can be used in the metric evaluation step in the post execution phase. Table 14 contains pointers which should be considered in this step which related to both the actual execution and the test measurement activity. As for the other tables, these are not final so these are expected to be adapted and improved when their worth is proven in industrial projects.

Pointer	Description	Motivation
Requirements traceability.	Log each found defect to some tracking system where the corresponding requirement is stored as well.	This is important for the traceability issue since the requirement version may be changed prior to the bug fix which can be confusing when looking at the bug report. The importance of being able to trace a defect to its corresponding software artifacts has also been recognized by Yadla et al. which also propose an Information Retrieval technique that focus on defect to

		requirements traceability [Yadla05].
Analyze results.	Even though the selected tool or test script includes a test oracle, the correctness of these should be verified prior to reporting the issue as a bug.	Issuing false positives as defects to the reporting tool may take focus of the more serious defects. This is why careful consideration should be taken in order to verify that a defect actually has been found. The importance of limiting the number of false positives was elicited through discussions at Testway.
Use parameterized tests.	Develop test cases that take input through method parameters.	By using parameterized, also called data-driven test cases, the actual input data may be developed in the consulting organization and thereby reused among several customers. As mentioned by Tillmann, the tests can be instantiated by other test cases with a range of input parameters which differs from traditional test cases which are restricted to a particular set [Tillmann05]. This typically led to fewer needed test cases since the created ones can be reused in the same manner as ordinary methods which also decrease the maintenance time of the test suite.

Table 14 – Test execution and measurement pointers

5.4 Post execution phase

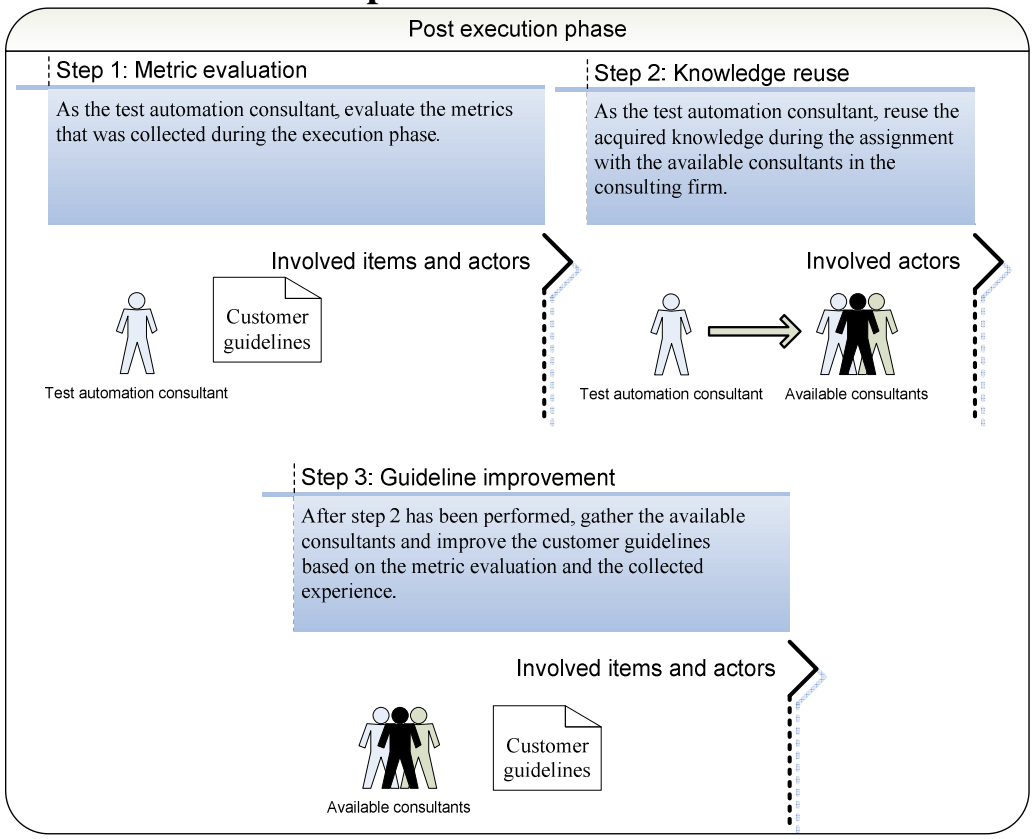


Figure 11 – Post execution phase

An overview of the steps in the post execution phase is given in Figure 11 which also illustrates the involved items and actors for each step in the phase. As can be seen in Figure 11, the phase involves three steps; Metric evaluation, Knowledge reuse and Guideline improvement which are introduces below starting with the metric evaluation step.

5.4.1 Metric evaluation

This step involves the evaluation of the metrics that was collected during the execution. Keep in mind that the evaluation should focus on the current customer project but also as a means of improvement for this strategy and the customer guidelines. Table 15 contains pointers which should be taken into consideration when evaluating these metrics.

Pointer	Description	Motivation
Combined focus.	Focus on the strategy and guidelines improvement as well as the current customer product.	Since the strategy and guidelines are intended to increase the quality of the customer products, it is important to keep these up-to-date with the current state of the art with in the field of software testing.
Evaluate tool	Use the metrics to evaluate the efficiency of the automated testing tool. Appropriate tool evaluation checklists can be found in [Poston92].	Poston and Sexton recognize the importance of measuring the company work products quantitatively [Poston92]. These results serve as a means of estimating the return on investment.

Table 15 – Metric evaluation pointers

5.4.2 Knowledge reuse

Since the consultants are spread over several customers and application domains, each individual faces specific challenges related to these domains. If this knowledge can be reused, the total knowledge in the consulting firm can be increased which benefits customers in projects other than the one where the experience was collected. Gable mentions that the knowledge held by each individual can be divided into codified and un-codified knowledge where the former can be transferred fast while the latter is related to individual actions and experiences and thereby more difficult to reuse [Gable03]. However, Gable has recognized that both types can be transferred which could be useful within the consulting firm in order to stay competitive and up-to-date. Table 16 contains pointers related to the knowledge reuse issue.

Reuse step	Description	Motivation
Seminars.	Let the consultants hold seminars from time to time where the interesting experiences, challenges and solutions are shared in the organization.	This practice was already initiated at Testway as a means of in-service training for the consultants, a practice called Test forum at the company. Gable mentions guided learning, formal training and knowledge creation activities as factors for successfully sharing individual knowledge with other individuals in a group [Gable03].
Experience reports.	Collect interesting challenges and solutions from the individual assignments to an organization wide online forum.	Since each individual consultant gather personal experiences through their assignments, it could be beneficial for knowledge collection in the entire organization if these are shared. However, Gable mentions that individual experiences and action patterns are better shared through

		direct communication [Gable03] so this pointer should applied for codified knowledge which Gable considers to be easier to transfer.

Table 16 – Knowledge reuse pointers

5.4.3 Guideline improvement

The customer guidelines are designed with extensibility in mind since it has been recognized that flexible adaption to customer domains are important for the life span of these guidelines. After the metric evaluation step, there is sufficient information available that can be applied when making this improvement and the pointers in table 17 are recommended.

Step	Description	Motivation
Adopted pointers.	Assess the metrics collected about the adopted pointers and check to see how the pointers can be modified for increased success in regards to project testability and stability.	It is important that the guideline pointers are kept up-to-date with the customer project state since the test process maturity are expected to increase for each project that adopts a certain set of pointers.
State-of-the-art research.	Search for recently made, empirically evaluated case studies that have been done in the field of software testing in order to find new inputs for the guidelines and the strategy.	Since the organizations needs to be informed of the benefits gained by adopting the guideline pointers, it is appropriate to find new pointers and support old ones through studies which has proven their worth.

Table 17 – Guideline improvement pointers

5.5 Strategy pitfalls

The following section describes some possible pitfalls of the strategy which should be considered and thereby avoided throughout its use in the organization.

5.5.1 To ambiguous automation

Often, organizations tend to set up high automation coverage goals which may not be economical viable. It should be noted that it may not be suitable to automate every test case in a software project. Economical aspects should be considered prior to the automated test case creation. For example, if a test case is expected to be executed once, it is not economically viable to automate. Furthermore, every pointer in the strategy may not be applicable for every organization so the most appropriate ones for the particular assignment should be chosen.

5.5.2 Low testability

Since automation is dependent on high testability for the test case design, the test automation may become limited if the customer optional steps are neglected. To limit this effect, the low testability issues should be taken into consideration in the tool selection step prior to the execution phase.

5.5.3 Selling the guidelines to practitioners

It can be hard to sell the guideline pointers to software practitioners even with sound motivations if the test process maturity is low at the organization. This issue could be dealt

with by expanding the motivation sections so that the worth is proven from angles which are more desired by the given customer.

6 CUSTOMER GUIDELINES

This chapter describes the guidelines that are meant to increase the test process maturity and thereby the testability and stability in software development projects at the customer site where consultants perform the system and acceptance testing. It starts with an introduction section where the needs are motivated along with an overview to the guideline design. The identified current challenges in the consulting domain are related to requirements and neglected verification practices in the early phases in the development projects where the consultants act. Section 6.2 provides requirements pointers which targets the requirements engineering practices in the software development organizations. Section 6.3 deals with the second set of challenges which relates to neglected verification practices and gives pointers to general practices which can be used to test and design the system in the early phases of development in favor of the system and acceptance test.

6.1 Introduction

As a complement to the automated testing strategy developed for use by test consultants, the following guidelines is intended for the customers and provide directions towards more testable and stable software applications in the customer projects. The guidelines are divided into so called pointers and each of these gives a specific tip of what should be done to facilitate the system and acceptance test. These pointers are intended to be implemented by the developers in the customer projects and be motivated by the consultant test manager by using the motivation sections for each pointer. As previously described, without system testability and stability in the release, the lead time for the consultant testers will increase which in turn decrease the efficiency of the system and acceptance test. The guidelines has been designed based on consultant experience and empirically evaluated studies which proves their usefulness in development projects. Adoption of the pointers will increase the system testability and stability in the development projects and this will maximize the return of investment of the consulting services when the contract has been signed for the system and acceptance test.

6.1.1 Motivation statement

Historically, consultants have struggled with challenges related to the requirements and lack of early verification practices in the development projects when the system and acceptance testing have been initiated. For example, when a requirement is too complex, a test case cannot be properly traced from the failing source code entity to a single corresponding requirement. Of course, it is possible to fix such defects in most situations as well but it can be very time consuming which in turn leads to unnecessary cost. For example, in some consulting projects, it has been necessary to rewrite the requirements at the customer after implementation so that the system and acceptance testing could be performed. In another project, the release was so instable when delivered to the system test due to lack of early verification activities that the release needed to be sent back to the developers for bug fixing after minimal system testing. This in turn increased the lead time for the consultant testers since the system test could not continue before the defects had been fixed. If such effort could be avoided, it would save resources that could be spent elsewhere. As a first initiative towards solving these issues, guidelines that target such problems have been developed which is intended to help the customer to facilitate system and acceptance testing. These, together with the automated testing strategy developed for the consultants will hopefully bring the quality of the software development projects forward.

6.1.2 Guideline concepts

As a reference, this section will start by comparing these customer guidelines with the Capability Maturity Model Integration (CMMI) [CMMI02]. CMMI contains two different kinds of representation which are introduced in [CMMI02] as continuous and staged representation. The continuous representation in CMMI uses capability levels and is

organized so that the order of processes to improvement can be selected by the organizations without the restrictions that the staged levels impose [CMMI02]. The staged representation on the other hand contains maturity levels each of which has predefined sets of process areas and the process improvement shall be done in a predetermined order [CMMI02]. Capability levels focus on a specific process area contrary to the maturity levels which span several process areas. The common idea is to start the process improvement at the first level and work towards the higher levels. The customer guidelines differs but can still be compared to the continuous representation in CMMI since they are designed for organizations which need to focus on specific process problems instead of improving the complete process chain. This led to the concept of guideline pointers which is the basis of the customer guidelines.

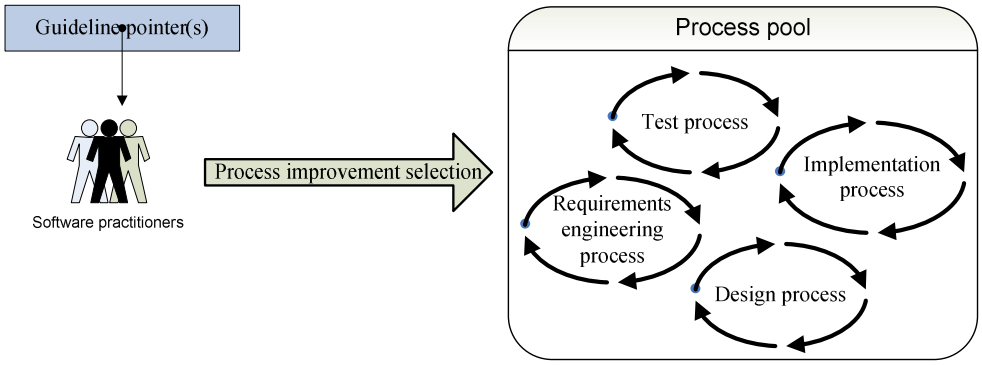


Figure 12 – Guideline pointer concept

Guideline pointers in the following sections are distinct tips for process improvements that can increase system testability and stability in order to facilitate the system and acceptance test. As illustrated in Figure 12, the pointers informs and motivates software practitioners of appropriate process improvement activities and these practitioners then choose which process to improve based on the current process state in the organization. The guidelines pointers have been designed to be independent of each other so that separate sets of pointers can be compiled and thereby customized for specific customer settings. Two main challenges have been identified as the basis for these pointers; requirements engineering practices and lack of early verification activities in the development projects. Since it has been recognized that design for testability is important for system and acceptance testing, such pointer has also been included in the general verification pointer section. A guideline checklist which summarizes the upcoming pointers can be found in Appendix A which can be used by developers at the customer site to verify that the planned pointers have been implemented.

6.1.3 Prioritization legend

Table 18 describes the prioritization levels which can be found in the guideline tables. Each pointer below is prioritized according to this table and the priority has initially been subjectively assigned according to the experience of the author. The pointer prioritization is expected to be modified in the post execution phase of the automated testing strategy after the guidelines has been evaluated based on the collected metrics.

Priority	Description
5	Critical
4	High
3	Normal
2	Low
1	Minor

Table 18 – Prioritization levels

6.1.4 Pointer table legend

For each pointer in the guideline tables below, there are five attributes attached. These are described in table 19.

Headline	Description
Pointer	A descriptive name of the pointer
Description	The actual pointer which describes what should be influenced in the customer project.
Motivation	A motivation to why the pointer should be adopted.
Priority	The priority refers to the criticality of the pointer. The priority will dynamically change based on the metric evaluation in the automated testing strategy. (Prioritization legend can be found in Section 6.1.3)
Roles	The roles section contains the team roles that are affected by the pointer adoption.

Table 19 – Pointer table legend

6.1.5 Structure of guideline pointers

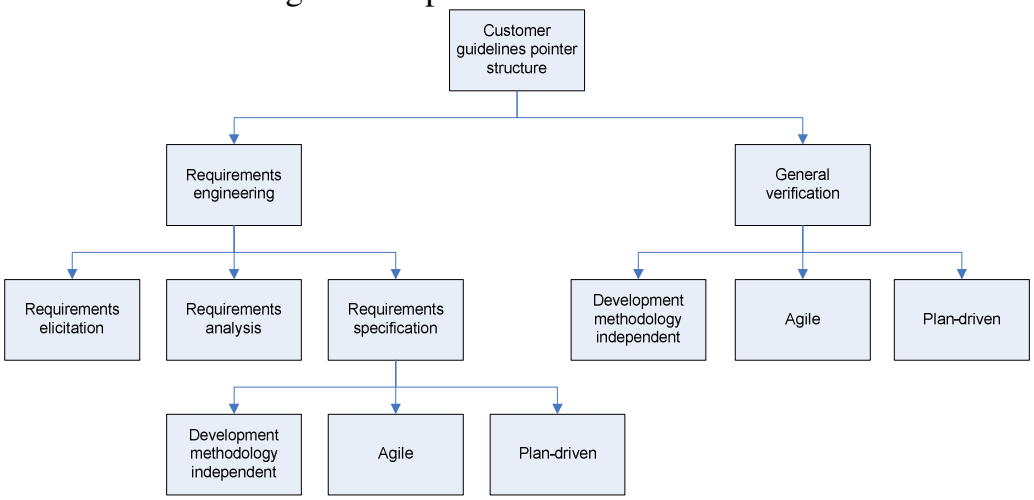


Figure 13 – Guideline pointer structure

As illustrated in figure 13, the customer guideline pointers are structured into two main categories; Requirements engineering and general verification pointers. As for the requirements engineering category, three requirements engineering phases has been considered especially important for system level testability and these phases have been divided into subcategories as shown in the figure. It has also been identified that requirements are specified in different manner depending on if the development methodology is agile or plan-driven. This is why the requirements specification subcategory is further divided into development methodology independent, agile and plan-driven pointers. The other main category, General verification, has been introduced in order to strengthen the system stability by early verification activities in the projects. Since verification differs to a great extent between agile and plan-driven methodologies, this section has been subcategorized into such sections.

6.2 Requirements engineering pointers

Requirements engineering contains several distinct phases and three of these have been identified as important for the testing practices, namely the elicitation, analysis and specification phases. Each of these contains challenges in regards to system and acceptance testing. The following section provides a set of pointers which relates to each of these phases. In many cases in industry there are low requirements engineering maturity and these guidelines provide pointers which can be adopted at several maturity levels so that

organizations can choose which pointers that is appropriate, if not all. The testability is expected to increase for each pointer that is adopted.

6.2.1 Requirements elicitation pointers

In some organizations, a completed requirements documents are handed over to the development organizations by their customer with the purpose of acquiring some sort of software system. However, the most common initiation of a requirements phase is to elicit requirements from a relevant set of stakeholders. Since the software is based on the requirements, it is important for both the development and the software testing to have solid requirements elicitation. Table 20 gives pointers on what needs to be considered in the elicitation phase in order for the system and acceptance testing to have a good basis later on.

Pointer	Description	Motivation	Priority	Roles
Ask reasoning questions.	Ask questions that forces the stakeholder to reason and motivate the requirements. In the market-driven context, this pointer could be applied in workshops with market sample representatives.	According to Pitts and Browne, this type of reasoning increases the stakeholder comprehension [Pitts07] which in turn leads to increased accuracy of the requirements. Test consultants at Testway have recognized the importance of accurate requirements for system and acceptance testing. This is because a found defect may be accurate behavior according to the requirements understanding of the developers which often differ from the consultant understanding of the same requirement.	3	Requirements engineers Customers
Prioritize requirements.	Prioritize each requirement using an appropriate prioritization method such as AHP or CV which are described in [Berander05].	In some cases, full system testing is not possible. If the requirements are not prioritized in these situations, it is difficult to make an efficient selection of requirements to put under test. This in turn can lead to non-optimal system testing since the selection may be at random. Turk has also recognized the importance of requirements prioritization and considers implementation costs as a primary factor to consider in this phase [Turk06].	4	Requirements engineers Testers Customers
Avoid asynchronous questioners.	Avoid using asynchronous questioners (where the answers are expected at later dates).	Lloyd et al. conducted an empirical study where it was concluded that asynchronous questioners lowers the quality of the requirements specification [Lloyd02].	3	Requirements engineers Customers

Table 20 – Requirements elicitation pointers

6.2.2 Requirements Analysis pointers

The analysis and the elicitation phase are often done in parallel. When a requirement has been elicited through for example a stakeholder workshop it should be analyzed to ensure that it fulfills certain parameters such as measurability and more importantly for testing, testability. Table 21 contains pointers that should be considered when doing this analysis.

Pointer	Description	Motivation	Priority	Roles
Avoid dividable requirements.	Try to avoid requirements that can be broken down into several requirements.	Dividable requirements are possible to test but it is hard to trace a failing test case to a particular requirement if it is dividable. A consultant at Testway mentioned that this can be a problem when the consultant arrives in late development phases.	4	Requirements engineers
Ensure measurability.	Ensure that it is possible to measure the fulfillment of non-functional requirements.	A general fact about requirements is that they should be measurable and verifiable. Turk gives a couple of example of terms that should be avoided such as “Easy” and “Fast” in [Turk06] with the motivation that such terms means different things depending on the reader.	3	Requirements engineers Testers
Ensure requirements testability.	Base test-related questions on the requirements in order to ensure that they are possible to test. If the question cannot be answered, the requirement probably needs modification.	According to Gelperin, asking these types of questions early is more important for software quality than the actual test execution [Gelperin88].	4	Testers
Avoid contradictions.	Make sure that the different requirements do not contradict each other.	If a set of requirements contradict each other, it is possible that a test case is passed based on the wrong conditions.	4	Requirements engineers
Analyze assumptions.	Verify that the elicited requirements are based on correct assumptions.	If the requirement differs from the one wanted by the customer, the testing will be inaccurate in any case. Pitts and Browne addresses the	5	Requirements engineers Customers

		importance of challenging the made assumptions continuously since requirements tend to start out with some level of ambiguity [Pitts07]. As mentioned above, test consultants address the importance of requirement understanding since it affects the system testability due to the possibility of false positives in the defect report.		

Table 21 – Requirements analysis pointers

6.2.3 Requirements specification pointers

It is important to document in a way so that a common understanding between developers and stakeholders is achieved. However, there is also a need to increase the tester understanding, especially when the tester has not taken part in the actual development. Plan-driven and agile methodologies differ a lot when it comes to requirements representation since the latter aim to minimize the overall development documentation in favor of flexibility. The pointers have therefore been divided into three categories; Development methodology independent, agile methodology and plan-driven pointers. These pointers are presented in table 22, 23 and 24 starting with the ones which are development methodology independent in table 22.

6.2.3.1 Development methodology independent pointers

Pointer	Description	Motivation	Priority	Roles
Consider requirements traceability.	Ensure that defects can be traced back to their corresponding requirements when the system and acceptance testing begins. For example, Yadla et al. reports that Information Retrieval techniques have been proven successful in tracing requirements to their corresponding defects [Yadla05].	If the failing component cannot be traced to its corresponding requirement(s) it is hard to determine which part of the total requirements has failed. This importance has also been recognized by a test consultant at Testway which considers this to be a common issue.	4	Requirements engineers Developers Testers
Ensure understandability.	Ensure that the specified requirements are possible to understand by all stakeholders, developers and testers in late development phases.	Damian et al. conducted an empirical study where the perceived need for requirements understanding in late development was evaluated [Damian03]. About 85% of the	5	Requirements engineers Developers Customers Testers

		asked engineers perceived that such understanding had a large impact in the test phase.		

Table 22 – Development methodology independent requirements specification pointers

6.2.3.2 Agile methodology pointers

Pointer	Description	Motivation	Priority	Roles
Store rationales.	Store the rationales behind why a requirement exists. Sauer has introduced the Event-Based Design Rationale Model for storing such rationales in an agile setting and describes this further in [Sauer03].	Documenting rationales can avoid confusion for both developers and testers in later phases. As previously mentioned, Sauer mentions that such rationales are beneficial for an individual which has not been involved when the decision was made [Sauer03].	3	Requirements engineer
Allocate time for workshops.	Allocate time for future workshops with the consultant tester where the requirements are discussed in the case when the consultant has not been involved during early development.	Such workshops are useful for agile testing since the documentation can be expected to be incomplete.	3	Managers Requirements engineer Developers Customers Testers
Complement user stories or backlogs with test stories.	Make simple test stories attached to the user stories or backlog (depending on the requirements representation) with brief testing suggestions for the feature. Also, let another developer read the test story to ensure that the meaning is clear.	User stories or backlogs are often the only documentation of requirements in agile settings. This is why it would be appropriate to at least make sure that these are understandable and testable.	2	Developers Testers

Table 23 – Agile methodology requirements specification pointers

6.2.3.3 Plan-driven methodology pointers

Pointer	Description	Motivation	Priority	Roles
Avoid ambiguity.	Make sure that there is no room for several interpretations of the requirements.	If the requirements can be interpreted in several ways, the tester may pass a test that	4	Requirements engineers Testers

	Rosenberg et al. describes further relevant tips and metrics that are important for achieving high quality requirements in [Rosenberg98].	actually should fail due to a misinterpretation. The ambiguity issue is also recognized by Rosenberg et al. as a requirement quality factor which can affect the system and acceptance test [Rosenberg98].		
Store rationales.	Store the rationales behind why a requirement exists. Detailed descriptions are appropriate for this development setting since time with the customer often cannot be guaranteed in late development phases.	Documenting rationales can avoid confusion for both developers and consultant testers in late phases. Sauer recognizes that such rationales are useful for individual which has not been involved when the decision was made [Sauer03].	4	Requirements engineer
Ensure requirements comparability.	Ensure that the requirements are comparable to each other. The Requirements Abstraction Model (RAM) has been developed by Gorschek and Wohlin for the market-driven domain. By using this model, the requirements can be abstracted or broken down into four levels of abstraction which serves to make the requirements more comparable to each other [Gorschek06].	Gorschek and Wohlin have identified that comparable requirements are necessary for effective prioritization and release planning [Gorschek06].	3	Requirements engineer
Structure requirements in logical order.	Structure the requirements specification in a logical order.	Damian et al. has identified good requirements specification structure as important for propagating the system features among stakeholders [Damian03].	3	Requirements engineer

Table 24 – Plan-driven methodology requirements specification pointers

6.3 General verification pointers

Depending on the development methodology, it may not be possible to affect the requirements when these guidelines are introduced since the consultant may arrive at different stages of development. Tables 25, 26 and 27 provide general pointers that can be applied to both design and implementation-specific items in order to increase the testability and stability in favor of the system and acceptance testing. Since agile methodologies usually differ a lot from plan-driven ones in regards to the testing practices used, the pointers for these has been divided into two separate sets.

6.3.1 Development methodology independent pointers

Pointer	Description	Motivation	Priority	Roles
Maintain the requirements.	Update the requirements when system changes occur.	Even if small changes are issued, the represented requirements should be updated to reflect the change. As mentioned by Graham, even the smallest changes can have large impacts on the testing [Graham02].	4	Requirements engineer Developers
Use change management routines.	Store the rational for accepted changes.	If the rational is neglected, it may be difficult for a tester to understand the correct system behavior which is crucial for the test case design. Furthermore, as described by Sauer, these rationales can also illustrate dependencies between different decisions taken during development [Sauer03].	3	Change control board Developers
Involve testers early and continuously.	Involve the testers early and continuous throughout the development phases.	Graham has identified that such early involvement of testers can save time and resources since this leads to early defect detection and thereby early feedback to developers [Graham02].	3	Managers Testers
Design for testability.	Consider what needs to be tested by automation early on and provide software hooks for the input and output to these components. Pettichord gives further practical advice on how to	Testway has recognized that if the applications do not provide access points to the critical components, they are hard or impossible to test through automation. Pettichord has also recognized the importance of testability	5	Designers Developers Testers

	make the software testable for test automation in [Pettichord02]. Gao et al. introduces testable beans which can be used to facilitate testing in component based software [Gao02]. Also consider using testability anti-patterns [Baudry03] to ensure that the design patters used gives sufficient testability.	for test automation. Gao et al. mentions that high component testability may decrease the overall testing cost [Gao02] which is another important factor in software development. Furthermore, as described by Baudry et al. bad design decisions can increase the testing time needed to test the system [Baudry03].		

Table 25 – Development methodology independent general verification pointers

6.3.2 Agile methodology pointers

Pointer	Description	Motivation	Priority	Roles
Enforce test-first practice.	Ensure that the test cases, especially the unit tests, are implemented prior to the relevant production code.	It can be tempting to abandon the practice where the test cases are written prior to the production code due to expected increase in development time. However, this often led to larger overall development time since the maintenance time may be greater without the test-first practice according to an interviewed development consulting firm. Furthermore, Erdogmus et al. conducted an empirical study where the results showed that the involved test-first practitioners were more productive than the ones that tested after implementation [Erdogmus05].	5	Developers (Possibly testers if customer training is needed)
Create simple unit tests and follow the automation test manifesto.	Keep the implemented unit tests simple; set an appropriate maximum limit of the line of code for	If a unit test case contains a large amount of code it can be a sign that the unit under test are to complex and needs to	4	Developers (Possibly testers if customer training is needed)

	these that will be upheld in your particular project. A similar tip along with 11 more can be found in the Test Automation Manifesto introduced by Meszaros et al. in [Meszaros03]. It is recommended that this manifesto is considered before implementing the unit test cases. Also, consider the Feedback-directed Random test generation approach introduced by Pacheco et al in [Pacheco07] when creating the unit tests.	be broken down. Brown and Tapolcsanyi discuss the importance of keeping the unit tests easy to write and easy to use with the maintenance needed for large unit test suites in mind [Brown03]. This importance has also been recognized by Meszaros et al. and is discussed in [Meszaros03]. However, as previously mentioned, the test automation manifesto has yet to be empirically evaluated which means that the impact on testability has not been proven. Furthermore, as previously described, previous research concludes [Pacheco07] that Feedback-directed Random test generation can give high coverage but more importantly, high defect discovery.		
Use code coverage through personal code ownership.	Let each developer have personal ownership of their developed code. Then use a tool which tracks the unit test coverage of the checked-in code.	The applicability of this pointer has been successfully proven in a consulting development firm where interviews have been conducted. By the introduction of a companywide code coverage goal, the personal sense of responsibility has increased in the organization since each downfall in coverage can be directly traced to an individual developer. Note that if the extreme programming methodology is used,	4	Managers Developers

		this pointer could collide with the collective ownership practice proposed by Beck in [Beck99].		

Table 26 – Agile methodology general verification pointers

6.3.3 Plan-driven methodology pointers

Pointer	Description	Motivation	Priority	Roles
Allocate sufficient time for testing.	Allocate testing time along with the time allocation for other development phases. Do this early to ensure that the testing phase gets sufficient time for quality assurance.	It is a common mistake that testing time is allocated too late in the project according to a consultant test manager at Testway. This means that the amount of testing is determined by how much time the other development phases needs. This can be a major contributor to poor software quality since the testing practices needs their fare share of time.	3	Managers
Use continuous integration practice.	Issue small releases during the development cycle and focus on continuous integration with unit test and integration test coverage.	This is actually an extreme programming practice [Beck99] that could be applied to the plan-driven approach as well. It has been determined through interviews with a consulting development firm that large releases may decrease the total automated testing coverage. If the release is to large and complex when delivered to the test phase, insufficient code coverage is common. This usually turns out in a system testing that only detects small errors that cannot be traced to requirements, such as null-pointer exceptions that should be caught through unit testing.	4	Managers Developers Testers

Table 27 – Plan-driven methodology general verification pointers

7 DISCUSSION

This chapter contains discussions regarding the proposed automated testing strategy and the customer guidelines. The first section (Section 7.1) provides a discussion of lessons learned during the case study while Section 7.2 continues with a validity discussion based on four types of validity. The last section (Section 7.3) contains describes how the research questions has been answered during the thesis projects and give elaborated answers to each question.

7.1 Lessons learned

This section will provide a discussion of the perceived applicability of the proposed strategy and guidelines in a live consulting setting. Since only static validation has been collected, the discussion will be based on the opinions of the consultants at Testway, the opinions of one of their customers and the opinions of the thesis author.

7.1.1 Strategy applicability

The applicability of an automated testing strategy within the consulting domain was considered to be low by an automated test consultant at Testway due to the changing parameters in the different customer environments. This opinion was taken into consideration during the strategy creation which is the reason for the general nature of the strategy and guideline pointers. Due to this generalization, the thesis author expects that the flexibility of the strategy between customer domains has been increased compared to the original design which had more detailed pointers.

Many of the pointers in both the strategy and customers guidelines refer to relevant studies that further describe how to conduct certain practices. It is the opinion of the author that this implies initial overhead in regards to learning time if the referred concepts are unknown at the time of strategy adoption. However, the author also expects that the strategy efficiency will be greater when these concepts have been accepted by the strategy practitioners. Furthermore, it is the belief of the author that the guidelines have larger possibilities of being adopted by the customers if the person who has taken the role as consultant test manager in the strategy (Figure 9, Step 1) has complete understanding of the pointer benefits described by the motivation sections.

7.1.2 Customer guideline applicability

In the initial stage of the strategy and guideline development, the guidelines were meant to be delivered as a guide to the customers. The customers were then to follow these guidelines themselves in their early development phases in order to improve the testability and stability of the software project. However, the industrial validation showed that it would be difficult to use the guidelines in this manner so these were modified in order to be used through a test manager from the consulting firm. This way, the consultant test manager can convince the developers to adopt the pointers and train them accordingly. This is also the motivation for the empirically evaluated studies which supports the pointers in the guidelines. The motivations are supposed to be used in the persuasion of the developers and management at the customer site.

Due to this modification, a validity discussion with a consultant test manager was conducted. Through this discussion, it was concluded that there indeed are problems in regards to the test process maturity in many organizations which often infer low system testability and stability. With this in mind it was also concluded that the guidelines is needed in order to increase the testability and stability of these projects since the guidelines also motivate why changes needs to be made. The test manager mentioned that it would be appropriate to include information of which roles in the organization that needed to be allocated for adopting each pointer. This was considered especially important since it was perceived that managers would require this information to allocate resources for the pointer

implementation. As can be seen in the guideline pointers in Chapter 6, a roles section exist which has been included due to this perceived importance.

Furthermore, if the project managers are serious about the quality assurance process the test manager did not see further difficulties in applying the guideline pointers in order to facilitate the system and acceptance test.

7.2 Validity assessment

Since the thesis project has been a qualitative one, a search was made for validity criterion suitable for these kinds of studies. As described by Trochim in [Trochim06], some researcher's claim that validity issues in qualitative studies differ from the ones discussed for quantitative research. Trochim further describes that quantitative studies contain methods and result data which cannot be found in qualitative studies [Trochim06]. Based on this assumption, this section provides a validity discussion based on the four criterion introduced by Lincoln and Guba in [Lincoln85] for qualitative research.

7.2.1 Credibility

This section will discuss how the participants in the case study experience the credibility of the automated testing strategy and customer guidelines within their environment. This kind of validity is important since the strategy and guidelines has only gone through a static validation in terms of interviews and general discussions. The validation has been conducted through discussions with a consultant test managers and an automated testing consultant at Testway. In addition to this, an interview with one of their customers was conducted.

While the consultant test manager was interviewed for the validation of the part of the strategy which included the customer guidelines, the automated test consultant was interviewed to validate the credibility of the automation specific part of the strategy. The purpose of the customer interview was to get their point of view of the customer guidelines.

Since system testability and stability was considered to be the main challenges in most situations, the consultant test manager perceived the focus in the preparation phase as credible. Furthermore, it was also mentioned that strong motivations of the guidelines used in this phase was needed in order to convince the developers and managers in the customer organization. As described, a customer interview was also conducted in order to get customer validation of the guidelines. They consider testability and stability to be of great importance for the system testing, acceptance testing and overall product quality. Furthermore, they recognized the importance of having people in the organization which are test-oriented since the testing practices may not be adopted otherwise. They also consider the guideline approach to be appropriate for facilitating system testing in customer project but also that it can be hard to convince low test maturity organizations to adopt the pointers. The customer mentioned that there is a threshold that needs to be crossed before the practitioners perceive the benefits. However, after this threshold has been crossed, they mentioned that an organization rarely switches back to their old routines.

Unfortunately, the use of an automated testing strategy in the consulting domain was perceived as difficult by the automated test consultant due to the changing parameters at different customer sites. However, this has been taken this into consideration in the design of the automation specific parts of the strategy and thereby they have been designed for practitioners who move between different customer domains and development phases. Therefore, it is suggested as future work to include a dynamic validation of the strategy so that the feasibility of the strategy can be addressed from a live industrial perspective.

7.2.2 Transferability

This section will provide a discussion about how well the approach is transferable to other settings than the one for which it was originally intended. As described throughout the thesis,

the primary focus has been to produce an approach suitable for a consulting setting. In this setting, the application domain is expected to change due to the consultant movement between customers. To achieve this goal, the strategy and guideline pointers have been generalized to the extent that they can be adopted independent of the current parameters in the customer domain. Due to this design, the approach may be transferred to ordinary development settings where the application domain is static. However, in this case it could be appropriate to extend the guidelines with more specific pointers for the particular domain since the strategy no longer needs the flexibility that the generalization provides. This kind of extension is possible due to the dynamic structural design of the guidelines. In fact, since the strategy and guideline pointers are expected to be modified, they have been designed to support this which makes a setting transfer possible.

As mentioned in Section 7.2.1, the customer validation showed the importance of having at least one person who is aware of the quality benefits that testing provides. Since the preparation phase in the strategy involves a consultant test manager which motivates the guideline pointers to the software practitioners and managers, a similar person is needed in the traditional organization as well. Without such person, it would be hard to cross the threshold described in Section 7.2.1 and this could infer a problem if the strategy and guidelines is needed in an organization which currently have low test process maturity. Furthermore, since the proposed strategy and guidelines refer to academic studies in the pointer motivation sections it can be hard to transfer the approach to organizations which are not susceptible to the results made by such studies.

7.2.3 Dependability

Trochim describes that quantitative studies use replicability in their validation process to ensure that the results can be replicated by other researchers [Trochim06]. Trochim also mentions dependability as an alternate way for qualitative researchers to describe how the changing environment where the study was conducted has affected the research [Trochim06]. Since replicability has been found to be inappropriate for this thesis project due to the qualitative nature of the study, this section will instead describe how the consulting setting where the study was conducted has influenced the research.

The main impact that the environment has made on the research is the abstraction levels of the pointers in the strategy and customer guidelines. As described above, the pointers have been designed to be general with the intent to be adoptable for several application domains. If the approach would have been tailored for a particular environment, the pointers would have included more domain specific details. However, since the pointer structure allows dynamic modifications, these can be extended by organizations to include such details when the need occur.

Also, as previously described, the preparation phase is dependent on a consultant test manager who is responsible for motivating the guideline pointers. If a similar person who can take this responsibility is not available in the organization that is about to adopt the strategy, it could result in a low adoption level of the pointers.

7.2.4 Confirmability

Trochim mentions confirmability as a validity type which relates to how the study results can be confirmed by others [Trochim06]. Since researchers often introduce validity threats in form of personal bias, this section will describe how the view of the thesis author differs from the point of view of other researchers.

As illustrated in Figure 3 (Section 3.1), the study started with a large literature survey which formed the initial point of view of the author regarding the current state-of-the-art within the field of testing and more specifically automated testing. Since the author has limited experience in research evaluation, this can pose a threat to validity because it is possible that

the evaluated studies have limited relevance for this particular thesis project. This threat was handled through the academic validation described in section 3.5. As described, some modifications were needed and have been implemented since the discussion with the academic researcher. Furthermore, the concept of the strategy and guidelines was considered feasible. However, further research validation was suggested for the metrics and requirements pointers and these pointers need further confimability to ensure academic relevance.

7.3 Answering research questions

This section will revisit the research questions that were initially formed in the early phases of the thesis project. Section 7.3.1 provides a flowchart of how the questions were answered and more elaborated answers is provided in Section 7.3.2.

7.3.1 Overview

Some of the research questions have been dependent on the results from the previous ones and the workflow of these and their answers are illustrated in Figure 14.

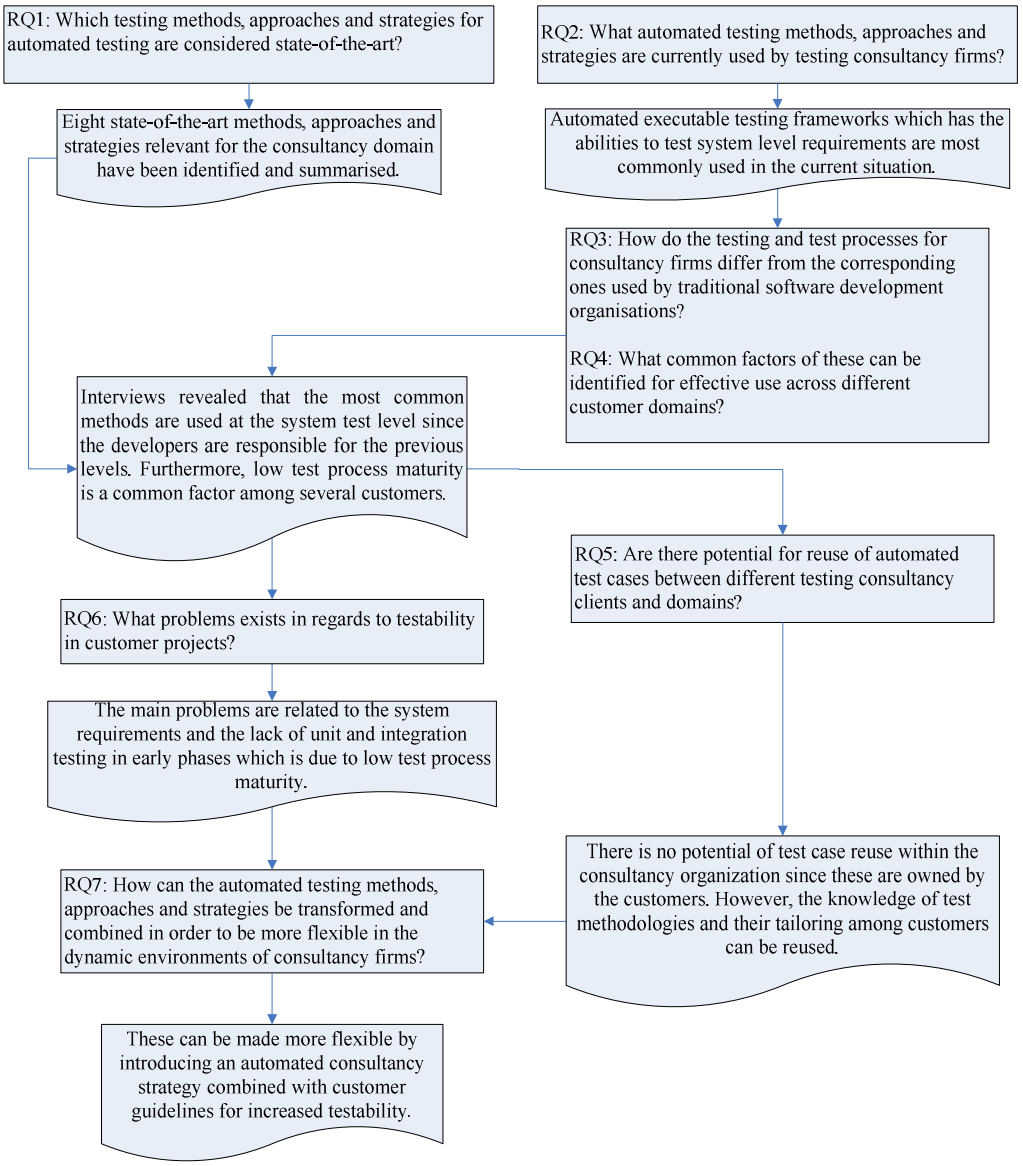


Figure 14 – Research questions workflow

7.3.2 Elaborated answers to research questions

RQ1) *Which testing methods, approaches and strategies for automated testing are considered state-of-the-art?*

As described in Section 3.2, a literature survey was done in order to find research relevant for the consulting domain. Through this survey, eight relevant state-of-the-art approaches were found which was deemed relevant for use in this domain. The primary focus was to find automated testing methods suitable for use in the system and acceptance testing levels. However, since the consultants act in unit and integration testing levels in some situations it was necessary to introduce methods which can be relevant for these situations as well. This led to the summary of three black-box techniques, three black-box/white-box hybrid techniques and two frameworks for unit testing suitable for both agile and plan-driven development settings. More deep going summaries of these can be found in Section 2.5.

RQ2: *What automated testing methods, approaches and strategies are currently used by testing consulting firms?*

Since the most common test levels are system and acceptance test, it is important that the automated tools have sufficient support for testing system level requirements. For example, tools such as Watir [Rogers07] are common for system testing of web applications. In general, it is important that the tools are flexible enough to support custom modifications to the test cases. Script languages such as Ruby have also been recognized as valuable due to its capabilities to test modules written using other programming languages. Furthermore, it has been recognized that partition testing is suitable in the cases when full test case automation is not possible since this can be used to measure how much testing each system partition has received.

RQ3: *How do the testing and test processes for consulting firms differ from the corresponding ones used by traditional software development organisations?*

It has been identified that the main difference is that the test level responsibility differs between test consultants and the developers in the traditional software development organizations. It is generally considered that unit and integration testing shall be handled by the developers doing the implementation while the test consultants are most often responsible for the system and acceptance testing. Furthermore, it has been recognized that the test process maturity is generally higher in the test consulting firm in comparison to the customer organizations.

RQ4: *What common factors of these can be identified for effective use across different customer domains?*

Low test process maturity in the customer organizations has caused low system stability due to low usage of unit and integration testing. Furthermore, since the system and acceptance tests use the system level requirements these are especially important. Unfortunately, it has been recognised that the requirements process maturity is low as well which has lead to low testability of the requirements. These problems combined have negatively affected the system and acceptance testing done by the test consultants. In summary, the two identified factors that most commonly needs improvement is the requirements engineering process and general verification practices where the responsibility lies on the developers.

RQ5: *Are there potential for reuse of automated test cases between different testing consulting clients and domains?*

Since the customers pay for the automated test case creation, these test cases are not allowed to leave their organisation. This means that there are no possibilities for this kind of reuse. However, it has been recognized that the test methodology is constantly reused by each individual consultant among different customers. The proposed strategy has taken this a step further in order to allow knowledge reuse from an individual consultant to the rest of the available consultants in the consulting firm. Such reuse benefits the consulting firm since the knowledge set and experiences of each individual can be transferred which in turn increases the total knowledge in the organization.

RQ6: *What problems exists in regards to testability in customer projects?*

Low testability and stability is a common problem which can be traced to low test process maturity in some organizations. The requirements are often poorly formulated in the situations where they exist and in many cases, there are no documented requirements at all. Furthermore, it has been recognized that the low process maturity has inferred low adoption of testing practices in the early phases of development. Also, low test process maturity has affected the software design in the sense that testability has not been build into the system which restricts automated tools that rely on software hooks which was earlier described. These factors combined have impacted the efficiency of the system and acceptance testing in a negative way.

RQ7: *How can the automated testing methods, approaches and strategies be transformed and combined in order to be more flexible in the dynamic environments of consulting firms?*

These can be made more flexible through the use of an automated testing strategy which can be applied in several application domains. Since there are often problems with low testability and stability due to low test process maturity in the customer projects, these problems must be handled prior to the consultant assignments. This in turn can be solved by introducing customer guidelines which targets the most common test process problems in the organizations. A dynamic structure is needed for the automated testing strategy and customer guidelines for them to stay efficient. The suggested pointers in both the strategy and the guidelines need to evolve when the test process maturity increases in the customer organizations. This can be handled by modifying, removing or adding pointers to the current set due to the dynamic structure. Furthermore, it is important that the pointers of the strategy and customer guidelines are supported by either previous consultant experiences or empirically evaluated studies which prove their worth. Otherwise it can be hard to convince practitioners of their value in their domain.

8 CONCLUSIONS

The consulting automated testing strategy (CATS) along with its supporting customer guidelines was developed for consulting domains where the practitioners act in changing application environments. CATS is divided into three steps where the first step targets the system testability and stability, a step which should be done prior to the actual test automation. The second step handles issues that should be taken care of in the test execution phase. As for the final step of CATS, it is focused on strategy and customer guideline improvements. Both CATS and the guidelines was developed in cooperation with a test consulting firm where it was recognized that the most common challenges is related to requirements engineering practices and early verification activities in the customer projects. These problems have caused low testability and stability in the customer projects which has inferred problems in the system testing level where the consultants mostly act. Furthermore, low testability and stability often increase the lead time for the testers since the system test often finds defects that should have been found by proper unit and integration testing which is the responsibility of the developers in the projects.

CATS use the customer guidelines in order to increase the test process maturity in the customer organizations which can solve the current lack of high testability and stability in the development projects. Both the strategy and the customer guidelines have been validated through industrial discussions in a consulting firm and by discussions with researchers in academia. An automated test consultant perceived that an automated testing strategy would to be difficult to apply in the consulting domain due to the changing parameters at different customer sites. To handle this issue, CATS has been generally designed to be flexible with the intent to be useful in different customer settings. The customer guidelines on the other hand were perceived by a consultant test manager as useful in the customer organizations due to the increase in software quality that they are expected to bring. However, it is also believed that there may be some problems to convince the management and developers of the benefits gained by adopting the guideline pointers. Relevant academic references and previous consultant experiences was provided in the motivation section in order to solve this issue and it was concluded by both the case study and the academic validation that this kind of motivations is feasible.

Since CATS is developed for consulting practitioners which act in several development domains, this strategy can be generalized to more static development settings as well. The strategy and guidelines can be dynamically modified to suit specific organizations. This way, they can be used at different test process maturity levels. With these strengths, the approach can evolve alongside the increasing test process maturity in the organizations which adopts it.

9 FUTURE WORK

Only static validation has been performed through interviews within the consulting firm, a relevant customer of this firm and researchers in academia. It would be appropriate for future researchers to assess the strategy and guidelines through dynamic validation by letting consulting customers use the guidelines before the consultant starts the assignment using the automated strategy. This way, an eventual increase in quality could be monitored and documented which would prove the worth of this study as well.

In the middle of the thesis project, it was discussed whether or not to build a tool for the customer guidelines which could generate specific sets of guidelines dynamically by assessing certain customer parameters. This idea was formed through discussions with the thesis supervisor. Such tool is suggested as future work since it could be beneficial for consulting firms that have adopted the automated testing strategy and the customer guidelines. Furthermore, since knowledge reuse is a part of the automated testing strategy, it could also be worth to include further research on knowledge management issues as a complement to that strategy step.

In regards to the guidelines, these could be extended with additional pointers in the requirements engineering field. As previously described, the requirements and metric pointers in the strategy and guidelines would benefit from further academic validation for their motivation sections. Furthermore, to enable the creation of the tool mentioned above, it would be appropriate to modify the guidelines to that customer parameters can be mapped against certain pointers.

There was a suggestion from one of the consultant test managers that it could help the customer guideline adoption if the pointers could be mapped to specific development phases in the methodologies used in the customer organizations. A study where different methodologies are organized and put in relation to the customer guidelines could increase the efficiency of the strategy and customer guidelines.

It may be possible to adapt current strategy to in-house project commitments as well but such case study was out of scope of this thesis. It would increase the validity of the current study if the strategy along with the guidelines would be empirically evaluated in a consulting in-house commitment project.

10 REFERENCES

[Abrahamsson03] Abrahamsson, P. (1-6 Sept. 2003). Extreme programming: first results from a controlled case study. *Proceedings on 29th Euromicro Conference, 259- 266*.

[Aurum02] Aurum, A., Petersson, H., & Wohlin, C. (September 2002). State-of-the-Art: Software Inspections after 25 Years. *Software Testing Verification and Reliability, 3(12), 133-154*.

[BDD07] BehaviourDrivenDevelopment. (n.d.). *Retrieved May 22, 2007, from http://behaviour-driven.org/*.

[Bach01] Bach, J. (2001). James Bach on Explaining Testing to Them. *Software Testing & QUality Engineering, 6(3)*.

[Baudry03] Baudry, B., Traon, Y. L., Sunyé G., & Jézéquel, J.-M. (2003). Measuring and improving design patterns testability. *9th IEEE International Software Metrics Symposium (METRICS'03), 50–59*.

[Beck98] Beck, K., & Gamma, E. (1998). Test infected: programmers love writing tests. *Java Report, 3(7), 37–50*.

[Beck99] Beck, K. (1999). Embracing change with extreme programming, *Computer, 10(32), 70-77*.

[Berander05] Berander, P., & Andrews, A. (2005). Requirements Prioritization. *In Engineering and Managing Software Requirements, eds. A. Aurum, C. Wohlin, 69-94*.

[Bhat06] Bhat, T., & Nagappan, N. (2006). Evaluating the efficacy of test-driven development: industrial case studies. *Proceedings of the 2006 ACM/IEEE international symposium on International symposium on empirical software engineering, 356 – 363*.

[Boehm01] Boehm, B., & Basili, V. (January 2001). Software Defect Reduction Top 10 List, *IEEE Computer, 1(34), 2-6*.

[Borland07] Borland. (2007). Automated Software Regression Testing & Functional Software Testing - from Borland. *Retrieved May 22, 2007, from http://www.borland.com/us/products/silk/silktest/index.html*.

[Boyapati02] Boyapati, R., Khurshid, S., & Marinov, D. (July 2002). Korat: Automated Testing Based on Java Predicates. *ACM International Symposium on Software Testing and Analysis (ISSTA), 123-133*.

[Brown03] Brown, M. A., & Tapolcsanyi, E. (2003). Mock Object Patterns. *10th Conference on Pattern Languages of Programs*.

[Bugzilla07] The Mozilla Organization. (1998-2007). Bugzilla. *Retrieved May 22, 2007, from http://www.bugzilla.org/*.

[Büchi99] Büchi, M., & Weck, W. (1999). The greybox approach: when blackbox specifications hide too much (Technical Report 297). *Turku Centre for Computer Science*.

[CMMI02] CMMI Product Team. (2006). CMMI for Development (version 1.2) (Technical Report CMU/SEI-2006-TR-008). *USA, Pittsburgh: Carnegie Mellon University, Software Engineering Institute*.

[Cole00] Cole, O. (2000). White-Box Testing. *Dr. Dobb's Journal: Software Tools for the Professional Programmer, 3(25), 23-28*.

[Damian03] Damian, D., Chisan, J., Vaidyanathsamy, L., & Pal, Y. (2003). An industrial case study of the impact of requirements engineering on downstream development. *In Proceedings of the International Symposium on Empirical Software Engineering (ISESE), 40-49*.

[Dijkstra72] Dijkstra, E. W. (1972). Notes on structured programming. In Structured Programming, *eds. Dahl, O.J., Dijkstra, E.W., & Hoare, C.A.R. London: Academic Press*.

[Edwards01] Edwards, S. H. (2001). A Framework for Practical, Automated Black-Box Testing of Component-Based Software. *Software Testing, Verification and Reliability, 11(2)*.

[Ekelund02] Ekelund, H. (December, 2002). TSQLUnit unit testing framework. *Retrieved May 22, 2007, from http://sourceforge.net/projects/tsqlunit*.

[Elbaum00] Elbaum, S., Malishevsky, A. G., & Rothermel, G. (August 21-24, 2000). Prioritizing test cases for regression testing. *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis, 102-112*.

[Elbaum01] Elbaum, S., Malishevsky, A., & Rothermel, G. (May 12-19, 2001). Incorporating Varying Test Costs and Fault Severities into Test Case Prioritization. *Proceedings of the 23rd International Conference on Software Engineering, 329-338*.

[Erdogmus05] Erdogmus, H., Morisio, M., & Torchiano, M. (2005). On the Effectiveness of the Test-First Approach to Programming. *IEEE Transactions on Software Engineering, 3(31), 226-237*.

[Gable03] Gable, G. (2003). Consultants and Knowledge Management. *JOURNAL OF GLOBAL INFORMATION MANAGEMENT, 3(11)*.

[Galli03] Galli, M., Nierstrasz, O., & Wuyts, R. (2003). Partial ordering tests by coverage sets (Technical Report IAM-03-013). *Switzerland, Bern: Universitat Bern, Institut für Informatik*.

[Gao02] Gao, J. Z., Gupta, K. K., Gupta, S., & Shim, S. S. Y. (February 04-06, 2002). On Building Testable Software Components. *Proceedings of the First International Conference on COTS-Based Software Systems, 108-121*.

[Gelperin88] Gelperin, D., & Hetzel, B. (June 1988). The growth of software testing. *Communications of the ACM, 6(31), 687 – 695*.

[George04] George, B., & Williams, L. (2004). A structured experiment of test-driven development. *Information and Software Technology 46, 337-342*.

[Glass98] Glass, R. L. (December, 1998). How Not to Prepare for A Consulting Assignment and Other Ugly Consultancy Truths. *Communications of the ACM, 12(41), 11-13*.

[Godefroid05] Godefroid, P., Klarlund, N., & Sen, K. (2005). DART: Directed Automated Random Testing. *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, 213-223*.

[Gorschek06] Gorschek, T., & Wohlin, C. (2006). Requirements Abstraction Model. *Requirements Engineering, 1(11), 79-101*.

[Graham02] Graham, D. (2002). Requirements and Testing: Seven Missing-Link Myths. *IEEE Software, 5(19), 15-18*.

[Graham93] Graham, D.R. (8 Dec 1993). Testing, verification and validation. *Layman's Guide to Software Quality, IEE Colloquium on*.

[Hayes95] Hayes, L. (1995). The Automated Testing Handbook. *The Software Testing Institute, Texas*.

[Hellesøy05] Hellesøy, A., Baker, S., Chelimsky, D., Takita, B., Astels, D., & Redpath, L. (2005). *Retrieved May 22, 2007, from http://rubyforge.org/projects/rspec*.

[JBehave07] JBehave. (2007). *Retrieved May 22, 2007, from http://jbehave.org/*.

[Jeffries07] Jeffries, R. E. (1999-2007). *Retrieved May 22, 2007, from http://www.xprogramming.com/*.

[Johansen01] Johansen, K., Stauffer, R., & Turner, D. (2001). Learning by Doing: Why XP Doesn't Sell. *Proceedings of the XP/Agile Universe*.

[Juristo04] Juristo, N., Moreno, A. M., & Vegas, S. (March, 2004). Reviewing 25 Years of Testing Technique Experiments. *Empirical Software Engineering, 1-2(9), 7-44*.

[Kaner97] Kaner, C. (1997). Improving the Maintainability of Automated Test Suites. *Proceedings of the 10th International Software/Internet Quality Week, 4(4)*.

[Kaner97b] Kaner, C. (April 1997). Pitfalls and strategies in automated testing. *Computer, 4(30), 114-116*.

[Kantamneni98] Kantamneni, H. V., Pillai, S.R., & Malaiya, Y.K. (1998). Structurally Guided Black Box Testing (Technical report). *USA, Ft. Collings: Colorado State University, Dept. of Computer science*.

[Keller05] Keller, R.K., Weber, R., & Berner, S. (May, 2005). Observations and lessons learned from automated testing. *On Proceedings of the 27th International Conference on Software Engineering, 571-579*.

[Koonen99] Koonen, T., & Pol, M. (1999). Test Process Improvement: A Practical Step-by-Step Guide to Structured Testing. *Boston: Addison-Wesley Longman Publishing Co*.

[Koskela04] Koskela, J., & Abrahamsson, P. (2004). On-Site Customer in an XP Project: Empirical Results from a Case Study. *In Proceedings 11th European Conference on Software Process Improvements (EuroSPI 2004), 1-11*.

[Larman05] Larman, C. (2005). Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design (3rd edition). *USA, New Jersey: Prentice Hall*.

[Leung97] Leung, H. K. N., & WONG, P. W. L. (1997). A study of user acceptance tests. *Software Quality Journal, 2(6), 137-149*.

[Lincoln85] Lincoln, Y. S., & Guba, E. A. (1985). Naturalistic inquiry. *USA, Beverly Hills: Sage*.

[Lipaev03] Lipaev, V. V. (November, 2003). A Methodology of Verification and Testing of Large Software Systems. *Journal Programming and Computer Software, 6(29), 298-309*.

[Lloyd02] Lloyd, W.J., Rosson, M.B., & Arthur, J.D. (2002). Effectiveness of elicitation techniques in distributed requirements engineering. *In 10th Anniversary IEEE Joint International Conference on Requirements Engineering, 311-318*.

[Lormans06] Lormans, M., Gross, H.-G., Deursen, A. van., Solingen, R. van., & Stehouwer, A. (October 2006). Monitoring Requirements Coverage using Reconstructed Views: An Industrial Case Study. *In Proceedings of the 13th Working Conference on Reverse Engineering (WCRE'2006), 275–284*.

[Merisalo-Rantanen05] Merisalo-Rantanen, H., Tuunanen, T., & Rossi, M. (2005). Is Extreme Programming Just Old Wine in New Bottles: A Comparison of Two Cases. *Journal of Database Management, 5(16), 41-61*.

[Meszaros03] Meszaros, G., Smith, S. M., & Andrea, J. (2003). The Test Automation Manifesto. *XP Agile Universe Conference, 73-81*.

[Microsoft07] Microsoft Corporation. (2007). Microsoft Windows 2000 Scripting Guide - COM Objects. *Retrieved May 22, 2007, from http://www.microsoft.com/technet/scriptcenter/guide/sas_vbs_wcmr.mspx?mfr=true*.

[Miller01] Miller, R. W., & Collins, C. T. (July, 2001). Acceptance Testing, *Procs. XPUniverse*.

[Mouchawrab05] Mouchawrab, S., Briand, L. C., & Labiche, Y. (2005). A measurement framework for object-oriented software testability. *Information and Software Technology, 15(47), 979-997*.

[Murnane05] Murnane, T., Hall, R., & Reed, K. (2005). Towards Describing Black-Box Testing Methods as Atomic Rules. *29th Annual International Computer Software and Applications Conference (COMPSAC '05), 1, 437-442*.

[Murnane06] Murnane, T., Reed, K., & Hall, R. (2006). Tailoring of Black-Box Testing Methods. *Australian Software Engineering Conference, 292-299*.

[Myers04] Myers, G. J., Badgett, T., & Sandler, C. (2004). The ART of SOFTWARE TESTING (2nd ed.). *USA, New Jersey: John Wiley & Sons, Inc*.

[Nebut03] Nebut, C., Fleurey, F., Le Traon, Y., & Jézéquel, J.-M. (November 17-21, 2003). Requirements by Contracts Allow Automated System Testing. *Proceedings of the 14th International Symposium on Software Reliability Engineering, 85-96*.

[Noonan02] Noonan, R. E., & Prosl, R. H. (February 27 - March 03, 2002). Unit testing frameworks. *Proceedings of the 33rd SIGCSE technical symposium on Computer science education, 232 - 236*.

[ObjectMentor01] ObjectMentor, Incorperated. (2001-04). JUnit, Testing Resources for Extreme Programming. *Retrieved May 22, 2007, from http://www.junit.org/.*

[Pacheco07] Pacheco, C., Lahiri, S. K., Ernst, M. D., & Ball, T. (May 23-25, 2007). Feedback-directed random test generation. *In Proceedings of the 29th International Conference on Software Engineering (ICSE'07).*

[Pancur03] Pancur, M., Ciglaric, M., Trampus, M., & Vidmar, T. (2003). Towards empirical evaluation of test-driven development in a university environment. *EUROCON 2003, 2, 83-86*.

[Pargas99] Pargas, R.P., Harrold, M. J., & Peck, R. R. (1999). Test-Data Generation Using Genetic Algorithms. *The J. Software Testing, Verification and Reliability, 9, 263-282*.

[Pettichord02] Pettichord, B. (October 2002). Design for Testability. *In Pacific Northwest Software Quality Conference*.

[Pitts07] Pitts, M. G., & Browne, G. J. (2007). Improving requirements elicitation: an empirical investigation of procedural prompts. *Information Systems Journal, 1(17), 89-110.*

[Poston92] Poston, R.M., & Sexton, M.P. (1992). Evaluating and Selecting Testing Tools. *IEEE Software, 3(9), 33-42*.

[Pyhajarvi04] Pyhajarvi, M., & Rautiainen, K. (2004). Integrating Testing and Implementation into Development. *Engineering Management Journal, 1(16), 33-39*.

[Rakitin01] Rakitin, S. R. (2001). Software Verification and Validation for Practitioners and Managers. *Artech house Inc.*

[Rogers07] Rogers, P., & Pettichord, B. (n.d.). Watir: Web Application Testing In Ruby. *Retrieved May 22, 2007, from http://wtr.rubyforge.org/.*

[Rosenberg98] Rosenberg, L., Hammer, T., & Huffman, L. (1998). Requirements, testing and metrics. *In Proceedings of the 15th Annual Pacific Northwest Software Quality Conference*.

[Runeson06] Runeson, P. (2006). A Survey of Unit Testing Practices. *IEEE Software, 4(23), 22-30*.

[Saff04a] Saff, D., & Ernst, M. D. (July 12-14, 2004). An experimental evaluation of continuous testing during development. *Proceedings of the 2004 International Symposium on Software Testing and Analysis, 76-85*.

[Saff04b] Saff, D., & Ernst, M. D. (June 7-8, 2004). Mock object creation for test factoring. *In ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'04), 49-51*.

[Sauer03] Sauer, T. (2003). Using design rationales as agile documentation. *In Proceedings of the 18th International Workshops on Enabling Technologies: Infrastructures for collaborative enterprises, 326–331*.

[Schwaber01] Schwaber, K., & Beedle, M. (2001). Agile Software Development with Scrum. *USA, New Jersey: Prentice Hall*.

[Sneed04] Sneed, H.M. (24-26 June 2004). Program comprehension for the purpose of testing. *In Proceedings of the 12th IEEE International Workshop on Program Comprehension, 162-171*.

[Sogeti04] Sogetti. (2004). TPI Automotive (version 1.01). *Retrieved May 22, 2007, from http://www.flow.nl/images/TPI_automotive_version_1%5B1%5D.01_tcm6-30253.pdf*.

[Sommerville04] Sommerville, I. (2004). Software Engineering (7th ed.). *Boston: Addison-Wesley*.

[Srivastava02] Srivastava, A., & Thiagarajan, J. (2002). Effectively prioritizing tests in development environment. *ACM SIGSOFT Software Engineering, 4(27), 97 – 106*.

[Talby05] Talby, D., Nakar, O., Shmueli, N., Margolin, E., & Keren, A. (2005). A process-complete automatic acceptance testing framework. *Proceedings. IEEE International Conference on, 129-138*.

[Testway06] Testway. (2006). Testway - Certified Test Specialists. *Retrieved May 22, 2007, from http://www.testway.se/services_e.html*.

[Tillmann05] Tillmann, N., & Schulte, W. (2005). Parameterized unit tests. *ACM SIGSOFT Software Engineering, 5(30), 253 – 262*.

[Tillmann06] Tillmann, N., & Schulte, W. (2006). Mock-object generation with behavior. *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06), 365-368*.

[Trochim06] Trochim, W. M. (October 20, 2006). The Research Methods Knowledge Base (2nd Edition). *Retrieved May 22, 2007, from http://www.socialresearchmethods.net/kb/*.

[Turk06] Turk, W. (2006). Writing requirements for engineers [good requirement writing]. *Engineering Management Journal, 3(16), 20-23*.

[Wells99] Wells, D. (1999, 2000, 2001). *Retrieved May 22, 2007, from http://www.extremeprogramming.org/*.

[Whittaker00] Whittaker, J.A. (2000). What is software testing? And why is it so hard?. *IEEE Software, 1(17), 70-79*.

[Williams03] Williams, L., Maximilien, E. M., & Vouk, M. (2003). Test-driven development as a defect-reduction practice. *IEEE International Symposium on Software Reliability Engineering, 34-45*.

[Xie06] Xie, T. (2006). Improving Effectiveness of Automated Software Testing in the Absence of Specifications. *22nd IEEE International Conference on Software Maintenance, 355-359*.

[Yadla05] Yadla, S., Hayes, J. H., & Dekhtyar, A. (September 2005). Tracing Requirements to Defect Reports: An Application of Information Retrieval Techniques. *Innovations in Systems and Software Engineering: A NASA Journal, 2(1), 116-124*.

[Yang06] Yang, Q., Li, J. J., & Weiss, D. (2006). A survey of coverage based testing tools. *International Conference on Software Engineering, Proceedings of the 2006 international workshop on Automation of software test, 99 – 103*.

[Yong05] Yong, L., & Andrews, J.H. (November 08-11, 2005). Minimization of Randomized Unit Test Cases. *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering, 267-276*.

[Zhu97] Zhu, H., Hall, P. A. V., & May, J. H. R. (December, 1997). Software unit test coverage and adequacy. *ACM Computing Surveys, 29(4), 365 – 427*.

11 APPENDIX A – CUSTOMER GUIDELINE CHECKLIST

Requirements engineering pointers – Elicitation	Methodology	Check
Ask reasoning questions.	Independent	
Prioritize requirements.	Independent	
Avoid asynchronous questioners.	Independent	
Requirements engineering pointers – Analyses	Methodologies	Check
Avoid dividable requirements.	Independent	
Ensure measurability.	Independent	
Ensure requirements testability.	Independent	
Avoid contradictions.	Independent	
Analyze assumptions.	Independent	
Requirements engineering pointers – Specification	Methodology	Check
Consider requirements traceability.	Independent	
Ensure understandability.	Independent	
Store rationales. (Event-Based Design Rationale Model)	Agile	
Allocate time for workshops.	Agile	
Complement user stories or backlogs with test stories.	Agile	
Avoid ambiguity.	Plan-driven	
Store rationales. (Detailed descriptions)	Plan-driven	
Ensure requirements comparability.	Plan-driven	
Structure requirements in logical order.	Plan-driven	
General verification pointers	Methodology	Check
Maintain the requirements.	Independent	
Use change management routines. (Store the rational for accepted changes)	Independent	
Involve testers early and continuously.	Independent	
Design for testability.	Independent	
Enforce test-first practice.	Agile	
Create simple unit tests and follow the automation test manifesto.	Agile	
Use code coverage through personal code ownership.	Agile	
Allocate sufficient time for testing.	Plan-driven	
Use continuous integration practice.	Plan-driven	