Pimp My Test Process – Introducing Test Automation and Process Maturity in an IT Consulting Context

by

Eric Lindahl

LIU-IDA/LITH-EX-A—12/011—SE

2012-03-20
Final Thesis

Pimp My Test Process – Introducing Test Automation and Process Maturity in an IT Consulting Context

by

Eric Lindahl

LIU-IDA/LITH-EX-A—12/011—SE

2012-03-20

Supervisor: Thomas Lööf
Examiner: Kristian Sandahl
Abstract

Ipendo Systems in Linköping, Sweden, is a small IT consulting firm developing among other things custom portal applications based on Microsoft Sharepoint 2010. The purpose of this thesis is to investigate whether the test tools TestComplete and LoadComplete provide sufficient compatibility for functional and non-functional testing of Ipendo Systems’ Sharepoint based applications, as well as design testing related activities to be incorporated into their existing software development framework. A test process maturity framework was chosen and applied for the design of the testing activities, while the test tool investigation resulted in guidelines on how to apply the tools effectively and circumvent any issues discovered.
# Contents

1 Introduction ........................................... 1
1.1 Background ........................................... 1
1.2 Purpose ................................................ 2
1.3 Specific topics ....................................... 2
1.4 Method ............................................... 2
1.5 Demarcation .......................................... 3
1.6 Acknowledgements .................................... 3

2 Theoretical framework ................................. 4
2.1 Software testing basics ............................... 4
2.1.1 Introduction ........................................ 4
2.1.2 Basic terminology ................................ 4
2.1.3 Why testing is important ......................... 5
2.1.4 Characteristics of a good test case ............... 6
2.1.5 Types of testing and their metrics ............... 7
2.1.6 Limitations of testing ............................ 8
2.2 Testing techniques ................................. 8
2.2.1 Testing levels ..................................... 8
2.2.2 Test case design techniques ..................... 9
2.3 Automating the testing process .................... 14
2.3.1 Basic concepts — why automate? ............... 14
2.3.2 Limitations and risks of automation .......... 15
2.3.3 Best practices - what to automate, when and how .. 18
2.4 Testing process maturity framework ............... 19
2.4.1 Introduction ....................................... 19
2.4.2 Overview of the TMMi model .................... 20
2.4.3 Process areas of level 2 ........................ 24

3 Methodologies, products, and tools ................. 28
3.1 Solutions Experts' development methodology .......... 28
3.2 Overview of Microsoft Sharepoint 2010 .............. 30
3.3 System under test .................................... 33
3.3.1 Overview ......................................... 33
3.3.2 Uploading and assigning to a case .............. 34
## CONTENTS

3.3.3 Handling case assigned documents .......................... 35
3.3.4 Searching .................................................. 36

3.4 Test tools ....................................................... 37
3.4.1 TestComplete ............................................... 37
3.4.2 LoadComplete ............................................. 40

3.5 Applying the test tools ....................................... 41
3.5.1 General thoughts .......................................... 41
3.5.2 TestComplete and Outlook ............................... 42
3.5.3 TestComplete and Word .................................. 44
3.5.4 TestComplete and the DMS ............................... 45
3.5.5 LoadComplete and the DMS .............................. 50

4 Solution .......................................................... 53
4.1 Best practice for the tools .................................. 53
4.1.1 Controlling the test environment ....................... 53
4.1.2 How to apply TestComplete ............................. 53
4.1.3 How to apply LoadComplete ............................ 55

4.2 Propagating changes through traceability ................... 57
4.3 Designing a test process ..................................... 57
4.3.1 Introduction ............................................... 57
4.3.2 Top-level views, goals, and schemes .................... 58
4.3.3 Inception phase activities ................................. 59
4.3.4 Elaboration phase activities ............................. 60
4.3.5 Construction phase activities ............................ 62
4.3.6 Other RUP phases ........................................ 64

5 Discussion ....................................................... 65
5.1 Evaluation ...................................................... 65
5.2 Raising the TMMi level ...................................... 66
5.3 Dimensions of cost effectiveness ............................ 66

6 Conclusions ....................................................... 68
6.1 Future implications .......................................... 68
6.2 Results and generalizability ................................. 68
Chapter 1

Introduction

1.1 Background

Ipendo Systems AB in Linköping, Sweden, is a small IT consulting firm wishing to expand upon their testing activities. The company has two main business areas - Ipendo Solutions and Solutions Experts - where the former develops a system for handling and maintaining intellectual properties called Ipendo Platform, and the latter offers strategic IT consulting services and develops customized solutions for facilitating their customers’ operations. Among the specialities of Solutions Experts are integration expertise, business intelligence, and portal based solutions.

While Ipendo Solutions develops its system from the ground up using ASP.NET, the Solutions Experts business area employs different technologies based on the project at hand. For their portal based solutions they mainly implement Microsoft Sharepoint 2010 as a base platform, adding custom functionality according to the customers’ individual needs. This creates a peculiar issue while testing, as testing related activities must be focused on verifying the custom functionalities and not the already existing functionalities of the underlying platform.

In the spring of 2011 Ipendo Solutions incorporated a tool for automated testing after an extensive evaluation of the various tools available on the market. The chosen tool was TestComplete, which since then has been employed to record and execute automated test scripts for Ipendo Platform. Management at Ipendo Systems now wishes to implement usage of TestComplete together with the recently released sister tool LoadComplete into the development process of Solutions Experts as well, launching a pilot project to investigate applicability for the various projects at that business area.
1.2 Purpose

The purpose of this thesis is to analyse how automated testing using TestComplete and LoadComplete should be applied to the Sharepoint based portal applications developed at Ipendo Systems. The results from the analysis are to be used as a basis for activities in a testing process to be integrated into the development process of the Solutions Experts business area.

1.3 Specific topics

- How should TestComplete and LoadComplete ideally be used for testing Sharepoint based applications?
- How should the traceability between the software requirement specification, test cases, and the results of automated test runs be realized so that changes propagate correctly?
- How should activities concerned with (automated) testing be designed in a project framework such that cost effectiveness is achieved?

If TestComplete proves adequate compatibility, automated test scripts are to be implemented for simple build verification testing. As for LoadComplete, a proof-of-concept for compatibility will suffice.

1.4 Method

A literature study on the topic of automated GUI testing will be conducted for analysing the results of the trial of TestComplete and LoadComplete. For the process part of this thesis, a process maturity model will be chosen and used as a framework for designing testing related activities.

Empirical data will be gathered during the course of using TestComplete and LoadComplete for recording and executing test scripts based on user tasks commonly performed within the system under test. Any issues arisen during this phase will be noted and analysed further using the above described literature.

Feedback from developers currently working with the system under test will also be gathered using unstructured open-end interviews. Input and wishes concerning their future testing activities will be considered during design of the future testing process.
1.5 Demarcation

The possibility of incorporating other computer aided software engineering tools than TestComplete or LoadComplete has been defined to be outside the scope of this thesis by management at Ipendo Systems AB. Therefore, tools which could aid in further automating parts of the testing process - such as model-based automatic test case generation tools - will not be considered for Ipendo Systems’ testing activities. [10]

1.6 Acknowledgements

Thanks to Micaela Kvist, manager at Solutions Experts as well as the Ipendo Systems supervisor for this thesis, for giving me the opportunity to tackle this project. My hat’s off as well to the developers at Solutions Experts who contributed to this thesis with valuable input and break time antics — Gustaf Armgarth, Niclas Gustafsson, Erik Strid, and Robin Wallin.

Ole Vestergaard, Nordic sales representative for TestComplete and LoadComplete, also deserves a nod for his interest and support for the tool investigation part of this thesis.

Last but not least, gratitude of gargantuan proportions go to Kristian Sandahl at Linköping University for providing continuous and invaluable feedback and support.
Chapter 2

Theoretical framework

2.1 Software testing basics

2.1.1 Introduction

There are many definitions for the concept of software testing. Broadly, testing can be described as an activity whose purpose is to evaluate a product’s quality and conformance with its requirements, and improving upon these factors by identifying and isolating defects and problems. A different and stricter definition states that software testing deals with the dynamic verification of the behaviour of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behaviour. [5]

Dynamic in this case means executing the code on the pre-defined inputs. Finite refers to the possibility of even simple systems having so many theoretically possible test cases that the complete test set can be considered infinite, requiring a trade-off due to limited resources. That is why appropriate test techniques have to be selected and applied in order to maximize the effectiveness of the process and the resulting test set. It must also be possible, although not always easy, to discern whether the observed behaviour and output of the tested system is acceptable or not, forcing the tester to define the expected outcome. Observed behaviour can be checked against user expectations (testing for validation), some sort of specification (testing for verification), or anticipated behaviour based on implicit requirements or reasonable expectations. [5]

2.1.2 Basic terminology

Software engineers, like humans in general, tend to sometimes make mistakes in their work, which may manifest as faults or defects in the resulting code. Faults can take a number of forms, such as incorrect steps, processes, data definitions, or something as simple as using the wrong logical operator (e.g. typing a "<" character instead of ">"). This is what commonly is referred to as a bug. [6]
As the code is executed, the software will not be able to perform as specified due to the containing fault, meaning the fault results in a failure. A particularly severe system failure may also be referred to as a crash. It is important to note that testing only reveals failures, but it is the underlying faults that can and must be identified and removed. [5, 6]

Generally, testing is done with a specific objective in mind, for example executing an isolated code snippet or verifying system conformance to a specific requirement. These activities benefit greatly from designing coherent sets of system inputs, execution conditions, and expected results based on the objective at hand. Such a set is called a test case and they are to be properly documented with the above properties and provided with a unique ID for reference purposes — as is generally done with most artefacts in the software development cycle. A battery of test cases belonging together in some way — as in sharing objectives or executing code within the same module — is called a test suite. [6, 7]

The agent — human or machine — who decides whether a test case execution is successful (the test passed) or not (the test failed) is called an oracle. Failed tests are to be documented in a so called incident report. [7]

2.1.3 Why testing is important

Generally, the longer faults linger without being discovered, the more costly it will be to isolate and correct them. Therefore, it is preferable to verify the software’s conformance to its specification and locate deviations as early as possible. Please refer to figure 2.1 for an illustration of the relative costs involved in correcting faults at different stages in the software development process.

<table>
<thead>
<tr>
<th>Stage</th>
<th>Relative Repair Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>1-2</td>
</tr>
<tr>
<td>Design</td>
<td>5</td>
</tr>
<tr>
<td>Coding</td>
<td>10</td>
</tr>
<tr>
<td>Unit test</td>
<td>20</td>
</tr>
<tr>
<td>System test</td>
<td>50</td>
</tr>
<tr>
<td>Maintenance</td>
<td>200</td>
</tr>
</tbody>
</table>

Figure 2.1: Relative cost to repair a fault in different stages [11]

The information gained when running tests is of great importance to multiple stakeholders to the software development process. The data on the state of the product can help managers make ship or no-ship decisions so as to avoid premature releases, thereby reducing the need for maintenance and/or technical support. [15]

For safety critical systems, correcting defects that could cause corrupt data or even accidents such as personal injuries is imperative. Depending on the judicial
climate, the following lawsuit after such an incident could be devastating for the company. Therefore, testing could be seen as a type of risk management — avoiding or mitigating the likelihood of unspecified behaviour in the system under test. [15, 17]

All in all, investing in testing early will increase the project costs on a short term basis, giving the pay-off of reducing the total cost of system ownership through improved quality. [2]

2.1.4 Characteristics of a good test case

A well designed test case generally conforms to the following guidelines [17]:

**It has a reasonable probability of finding a fault**
When developing test cases and in need of new ideas, work backwards from the image of how the system could fail. If the system *could* fail this way, how would the fault be caught and identified?

**It is not redundant**
If two tests look for the same fault using the same method, why run both?

**It is the best of its breed**
In a group of similar tests, some tests can be more effective than others. Since resources generally are scarce, the tester wants to single out the test cases most likely to find and isolate the fault.

**It is neither too simple nor too complex**
The tester could save some time by combining several test cases into one test case. Caution needs to be taken though, as test cases become harder to understand and execute as they grow in size. Test cases involving too many steps and interactions with a system also might yield inconclusive results if they fail.

**It makes system failures obvious**
Extending the last thought somewhat, testers need to take extra care that they do not miss failures because the system output is hard to interpret. Test cases should be designed in such a way that as little time as possible is wasted trying to understand if the test case passed or failed — and in the case of system failure, what was the cause of the failure.
2.1.5 Types of testing and their metrics

There are two different basic approaches to testing. **Black box** testing is defined as testing using only requirements and specifications as a basis. Black box testing requires no knowledge or familiarity with the underlying code or the implementation of the system. The sole focus is analysing behaviour and outputs generated in response to pre-selected inputs, system states, and execution conditions. Any kind of black box testing is by definition **dynamic**, meaning code is executed during the test. [6, 7]

The effectiveness of a black box test suite can be tricky to measure. Since the system is represented by a black box, the tester cannot see what parts of the system are exercised during the tests. Instead, black box test suite effectiveness can be given as the percentage of use cases or requirements covered. However, this says nothing of how thoroughly each use case or requirement is verified. [7]

**White box** testing, on the other hand, takes the internal mechanisms of the system into account. Different types include manipulating the system into executing specific paths through the code or verifying correctly defined branching in logical clauses. Therefore, a certain level of programming skills is required. [6, 7]

While a majority of these techniques are dynamic, white box testing also encompasses all types of **static** testing, meaning no code is executed. Instead, the code and algorithms are checked for general sanity, correct syntax, spelling errors, or general room for improvement. [17]

One might argue that according to the stricter definition of software testing given in section 2.1.1, the concept of static testing is to be regarded as an oxymoron, as software testing by default involves executing code (“**dynamic** verification of the behaviour of a program”). Although some static testing techniques employ tools to simulate code execution, static testing activities as a whole do differ significantly from dynamic techniques in that they lack pre-defined test cases with specified inputs and expected outputs. These activities still generate the same type of results though — defects are corrected, code quality is improved upon, and overall system quality is increased. Therefore, static testing activities are quite viable additions to any tester’s tool set, not be overlooked in his or her software testing endeavours.

As the tester has full insight into the system’s internals, measuring a test suite’s effectiveness is greatly facilitated compared to white box testing’s darker sibling. Data regarding the percentage of lines of code executed and paths taken during the test is readily available, meaning metrics such as line or path coverage — the percentage of lines of code or possible paths executed through the test suite — can be calculated with ease. [7, 17]
2.1.6 Limitations of testing

Unfortunately, it is easy to ascribe too high confidence in a system due to a set of passed tests. The tester and the company’s management have to keep in mind that any passed tests do not guarantee 100% stability in that particular module, or infer conformance to requirements in other parts of the system. As exhaustive testing is not practically feasible in real world software, testing can only be used to prove the existence of faults, but not the absence of them. The system could still fail due to any of the millions of untested possible combinations of conditions and states. [5, 16, 17]

Also, testing activities can neither create quality, nor take it away. If a series of test cases results in a complete system crash, the testing team did not “break the product” — it came to them already broken. Testers may work under quality assurance flag, but testing activities’ resulting artefacts (i.e. test results, bug reports) are only a part of the entire company’s collective quality assurance effort. Quality comes from the people who build the product, not from the testing department. [16]

2.2 Testing techniques

2.2.1 Testing levels

Traditionally, testing and by extension test case design, is performed at three different levels. [5, 7]

- **Unit testing** means testing the smallest pieces of software created by a developer. A unit may constitute an entire class or an individual method within a class, and these units need to be tested in isolation to ensure correct behaviour in their own.

- **Integration testing** is the next step up from testing units. It is possible for isolated units to function properly in a testing environment but cause failures once they are integrated into the rest of the system. This creates the need to integrate in steps, emulating simplified versions of the surrounding units and executing them as their own system to see how the unit in question behaves in a larger context. More of the simulated surrounding units are replaced with their real counterparts as the unit under test proves to harmonize with the rest of the system.

- **System testing** focuses on faults on the top level of integration testing, where all parts of the system are fully integrated. Various approaches are available at this level of testing, such as testing functionality, regression, security, installation, load handling, or localization. More on the various techniques available in system testing in section 2.2.2.
• **Acceptance testing** involves the last stage of testing in the development life cycle. When completed successfully, the customer will accept the software in the state it is in at that time, after which ownership of the software is transferred to the customer. Ideally, acceptance testing is be performed on-site together with the intended end-users of the system.

This classical view of testing levels is not applicable to all systems, as the model ignores transparency issues when constructing software on top of pre-existing platforms for which one does not have the source code. It is also assumed that a significant amount of time passes between each level, which is far from the case today where development of dedicated web applications can go into production within a matter of hours. Therefore, an alternative set of levels is suggested for these types of situations. [7]

- **Code quality** encompasses various static testing of the code.
- **Functionality** verifies system conformance to functional requirements.
- **Usability** is concerned with the system’s ease of use as specified in the non-functional requirements.
- **Performance** verifies system conformance to non-functional requirements bearing on variables such as response and processing times, energy consumption, and reliability.
- **Security** is tested to verify protection against potentially harmful system intrusion and theft of sensitive data. Handling of specific threats may be specified in the non-functional requirements.

As the system under test is developed on top of a pre-existing platform, the built in security measures in Microsoft Sharepoint are assumed to be "good enough" by management at Ipendo Systems and will not merit further testing at this time.

Usability testing is a completely different area of software engineering altogether. With one foot in cognitive psychology and the other in software development, this aspect of testing is outside the scope of this thesis.

The remaining levels — code quality, functionality, and performance — will be further discussed in the process of developing testing activities for Ipendo Systems.

### 2.2.2 Test case design techniques

As one of the goals of testing is to expose defects as far as possible, many techniques have been developed in order to use the testing department’s resources more effectively and efficiently for fulfilling this goal. These techniques may differ a lot in their approach but what they all have in common is being systematic and methodical in breaking down the system to a simplified version of itself before basing test case design on the remaining, albeit dimensionally reduced part. [5]
CHAPTER 2. THEORETICAL FRAMEWORK

Exploratory testing
Using this technique, the tester dynamically designs and executes test cases on-the-fly. This skill improves as the tester learns more about the software, its market, its risks, and the ways in which it has previously failed. These tests are more powerful than older tests, as they are based on the tester’s continuously increasing knowledge and skill. [5, 16]

Equivalence class partitioning
If a tester judges a group of tests to involve the same input variables, result in similar operations in the system, and affect the same output variables, the tests are considered to be equivalent. Therefore, one or a few test cases from each equivalence class is chosen to represent that class, and the rest is discarded as they do not add anything to the test suite. If it were not for this technique the only viable methodical approach would be to employ exhaustive testing, meaning every possible input is tested until resources run out. [16, 17]

Equivalence classes for invalid input is often one of the best sources for bugs. Few programmers take the time to thoroughly account for a module’s invalid or unexpected inputs. Therefore, the more types of invalid inputs are tested for, the more faults will be discovered. [17]

Boundary value testing
While equivalence class partitioning is generally seen as the most basic test design technique, it lacks more specific targeting into areas traditionally more fault-ridden than other areas. Analysing and testing around the boundaries between equivalence classes — where applicable — very often exposes more defects per input tested than inputs from within the classes in question. [7]

Most mistakes made by programmers are so simple they should not have been made in the first place. Despite this, it is very common for incorrectly defined boundaries to sneak through while handling input, such as the example with incorrect logical operators mentioned at the beginning of this chapter. A programmer accidentally defining a boundary as ‘>’ instead of ‘>=’ may look innocent enough of a mistake, but it can be very difficult to pinpoint if it goes unnoticed and causes some sort of failure. Therefore, boundary value testing is practically a necessity wherever user input data is handled. [7, 16, 17]

Regression testing
Regression testing involves re-running the same tests after a change in the software. A change might be anything from a bug fix to re-factored code to added functionality in a module. Performing regression testing has the goal of verifying that the system has not regressed between versions, meaning the change has caused something that used to work to now be broken. [5, 16]
Typically, over time a regression test library will be developed. The idea would be that whenever a new build of the system would be submitted for testing, every test in the library should be run. Therefore regression tests are among the types of tests that benefit the most from being automated. [17]

The drawback to regression tests is that they are among the types of tests least likely to expose new bugs in the system. The tests might have exposed faults at some point in time — some companies’ regression test libraries consist solely of tests that have found bugs — but those bugs were found and fixed a long time ago. The tests will probably not find any faults again in their lifetime, old or new, although it is good to make sure. [17]

Smoke testing
Smoke tests are a type of side-effect regression tests with the goal of proving that a new build or version is broken to the extent that it does not merit further testing. The expression smoke testing comes from the world of hardware development where if for example a new circuit board was to be tested, if you get smoke upon connecting it to electricity, you are basically done with testing already. [16]

Smoke tests are often automated and standardized from one build to the next. They test functionality expected to work, and if they fail, it is suspected that something basic is broken. Therefore smoke testing is also referred to as build verification testing. [5, 16]

If project resources for testing are scarce, it is preferable to prioritize smoke testing as that particular approach exercises relatively large parts of the system compared to the time required to design the test cases. Smoke testing, if applied correctly, is therefore generally seen as giving a lot of information per invested resource unit. [24]

Scenario testing
Scenario tests are tests derived from use cases. Using this technique facilitates using use case coverage as a metric. [16]

The process starts by analysing the use case’s pre- and post-conditions and its typical flow of events. The analysis itself also acts as a form of inspection as it will help point out inconsistencies and mistakes in the design of the use case.

The output of the analysis is a path flow diagram, where any unique path may constitute its own test case. Because the theoretical number of possible test cases may be rather overwhelming, some prioritization is needed. Primarily test cases are to be based on the main success scenarios, after which the alternative scenarios ending in unwanted system responses should be handled.
After weighing the cost of building and exercising a certain test case against the probability of it finding defects — with every defect’s inherent impact potential accounted for — the tester can narrow the number of test cases down to an efficient method of testing the use case. [4, 7]

**Performance testing**

These types of tests are usually run as benchmarks, measuring system response times and throughput. The data is compared to non-functional requirements to see if and/or where the system needs optimization.

Performance tests can also expose faults by comparing performance data between different builds of the system — a significant change in the response time of a certain function might indicate defects in the code. A particular request being completed three times faster or slower than normal is suspicious either way as it is evident that something fundamental has changed. [16, 17]

It is important to note that it is not necessarily changes in your own software that cause performance changes. Updates, patches or service packs issued to third-party components or underlying frameworks may have a noticeable impact on system performance as well. [16]

In any case, a best practice is to have the test environment mimic the production environment as closely as possible. Thanks to virtual machines, most hardware combinations can be emulated and tweaked at the press of a button, which greatly simplifies a performance tester’s work. Any anomalies increase the degree of uncertainty in the results. [20]

**Stress/load testing**

Stress or load testing involves exposing the system to many simultaneous demands for resources. With increasing load, interesting data concerning for example hardware adequacy and concurrency issues can be collected concerning the pattern of events leading up to a possible system failure. This data will point to potential vulnerabilities that might be exploited during less extreme circumstances. [5, 15, 16, 20]

Stress tests sometimes receive critique for lacking credibility due to not being representative of real-world usage of the system under test. As these types of tests are unrealistic in their design, some stakeholders may dismiss results due to lacking understanding of test applicability. [20]

Also, data from a stress test induced failure may be hard to collect unless an appropriate tool is used, or the tester is extremely familiar with the system and can interpret all of its peculiarities. [15]
CHAPTER 2. THEORETICAL FRAMEWORK

Tuning activities
Performance and stress/load testing techniques can be used as parts of a process known as tuning. It is an iterative process that usually is separate from a project’s general performance testing approach. [20]

The tests are conducted as a joint effort between a testing and tuning team in a fully controlled and dedicated test environment. As tests reveal unfavourable results in system performance, the combined team enters a diagnosis and reparation stage — tuning — where the output is proposed changes to the system and/or the test environment. These changes may not be suggested to immediately repair faults, but rather make temporary adjustments to deliberately magnify performance issues for diagnostic purposes, or to alter the test environment to see if response times decrease. [20]

After the tuning phase is completed, the test environment is generally reset to its initial state. Any proposed changes to the test environment and system under test deemed to be worthy of implementation are kept, while the rest is discarded. Finally, the performance tests are run again for regression purposes so as to prove that the correct changes have been integrated. [20]

As heavy focus is put on quick iterations, the cooperative testing and tuning team is to execute and re-execute their tests and implement the proposed changes in rapid succession. Such a process requires a high degree of commitment, meaning it can take exponentially more time if dedication is lacking. [20]

Static review techniques
The basic idea is that a group of people get together and review system code. Depending on the specific review technique used, the group may be as small as two people or as large as the entire development team, and the developer who wrote the code snippet under review may or may not be a part of the group. As with testing in general, the main objective is to identify issues, as opposed to solving them.

The designer can take a leading role in the process and show the rest of the group, step by step, how the system behaves in response to test data supplied beforehand. This simulation, also called a walkthrough, can expose awkwardness, redundancy, and other missed details. [17]

An other alternative is bringing a group of about seven people together where the developer of the code snippet in question is not present. The reviewers should have read the code in advance and are to question and discuss the design choices in the meeting. This is called a formal review meeting. [17]
Recent research has suggested that calling meetings for code review purposes is a rather costly activity for a development team. In an extensive case study done by Cisco Systems including 50 developers performing 2500 code reviews over 10 months' time, it was concluded that employing more lightweight code review activities — essentially removing the large in-person meetings — saved on average 6.5 man hours per review. Suggested alternatives to meetings were sitting down with the developer in question in front of the code and discussing face-to-face, passing the code around attached to emails, or employing a dedicated tool. Among the best practices gathered from the case study were going slowly through the code — no more than 200-400 lines of code during a 60-90 minute sitting, keeping check lists short, and annotating the code before starting the review using an automatic code metric gathering functionality or tool. [30, 31]

Common for all of the techniques are — since they generally involve a lot of people — that they require a considerable deal of management commitment to be successfully implemented. [17]

2.3 Automating the testing process

2.3.1 Basic concepts — why automate?

To employ automated tests is to execute a sequence of interactions with a system without human intervention. In theory, automating test execution should eliminate the human error factor and provide a significantly shorter feedback loop as to the state of the system under development — possibly leading to labour cost savings down the line thanks to reduced overall testing effort. [3]

In the case of web applications, automating tests is generally done using a capture/playback based GUI testing tool. As the tester moves through the steps specified in the test case, the testing tool records each interaction and the system's responses to these interactions. The resulting output is test scripts which can be played back considerably quicker compared to performing the same test manually. The tester can then use his or her time elsewhere during the duration of the test, and simply read the test execution output summary when the test is finished. Also, the risk of the tester becoming tired, inattentive, or habituated with performing the same test case repeatedly by hand — thereby missing defects — is mitigated by automation. [3, 8]

In essence, it all boils down to this: computers are faster, cheaper, and more reliable than humans — therefore automate. Unfortunately, the naivitée in this approach can lead to rather reckless automation efforts as there are quite a few more issues to clear before getting on the automation train. [3]
2.3.2 Limitations and risks of automation

Test tool issues

Ironically, test tools have a bad reputation for themselves being buggy. Examples include tools that are resource demanding to the point that they incapacitate the tester’s equipment, rendering the application under test untestable using the tool.

The most common type of problem found in test tools is compatibility issues. The tool may be incapable of recording or playing back GUI tests in web pages built on certain platforms or using particular third-party plug-ins, or the tool may behave in unexpected ways due to defects in hardware drivers installed in the operating system. [16]

As automated test scripts have to either be recorded, programmed, or a combination thereof, the skill of the automation engineer has a clear impact on the quality of the resulting script. Scripts should have an appropriate amount of lee-way in them such that when they fail, they do so because of a proper defect being discovered — not because of something trivial and irrelevant for the test. Some tools can be overly sensitive and fail test cases because of simple things like varying sizes of graphical controls, even by a couple of pixels in size. It is the automation engineer’s task to make sure the scripts fail for the right reasons and return conclusive results displayed in a clear way. [3, 16, 19]

Changes in the GUI are a natural occurrence during the software development cycle — especially if an iterative development process is employed. Testers and automation engineers need to solve the problem of test script maintenance somehow — the last thing they need is an entire test suite broken after a minor user interface adjustment. This risk can be mitigated by selecting the right test tool (some tools handle these occurrences better than others), and designing test cases and their resulting test scripts with maintainability in mind. [3, 14, 35]

Another aspect to keep in mind is a tool’s specific error recovery process when running entire suites of test scripts. Some tools stop the entire execution of the suite if one script fails, which is impractical if the suite is designed to run without human testers overseeing the test run. The tool needs to be able to recover and fulfill the test run. [35]

Given these issues, developers often require of testers that they replicate every defect found with automation outside of the test tool before taking the bug report seriously. This adds yet another complication to analysing automated test failures. [16]

Also, a test tool will not teach a company’s testers how to test, although some tool vendors may try to propagate the idea. Their test tools come pre-packaged
CHAPTER 2. THEORETICAL FRAMEWORK

with rudimentary advice on testing, but the suggestions are neither well-informed nor tailored to that company’s needs. If the state of a company’s testing is already confused, introducing a test tool will not improve the situation.

What is needed is for there to be a proper process in the first place, and this process needs to be in somewhat working order before introducing rather complex activities like automation. [1, 16]

Automated tests do not replace manual testing

The value of a test come from the information it provides. Estimating this value is a complex issue, where the question of automating said test will provide a return of investment widens the problem all the more. [9]

The most common approach this issue is to compare the costs of automated tests to the costs of running the same tests manually. Please refer to equations 2.1 and 2.2 below. [9]

\[
\begin{align*}
\text{Manual testing cost} &= \text{Manual preparation cost} + \left( N \times \text{Manual execution cost} \right) \\
\text{Automated testing cost} &= \text{Automated preparation cost} + \left( N \times \text{Automated execution cost} \right)
\end{align*}
\]

(2.1) \hspace{2cm} (2.2)

\( N \) represents the number of times the test is run during its life cycle.

These formulas capture the common observation that automated testing generally has higher upfront costs while providing reduced execution costs. One can then compare the lower total costs for manual testing for low values of \( N \) with the higher returns of investment in the automation alternative the more the test is executed.

However, these equations fail to account for the fact that the values are fundamentally incomparable. Performing a test by hand brings an entire spectrum of human capabilities to fruition, such as the ability to improvise new tests or notice things that were not anticipated beforehand. Automated tests could not even dream of replicating this rich intellectual process — which is why it is nonsensical to try to brand automated tests as automated human testing. [16]

Even if the tests were comparable in information gained from the results, consider the situation where for example \( N = 50 \). Since it is extremely unlikely that any stakeholder within the software development project will try to justify the value of the information gained from running the same manual test 50 times — the information is not worth the cost — the point is moot. It is not a question of choosing to run a test manually or to automatize its execution — for higher values of \( N \) manual execution is not even a viable option. [16]
CHAPTER 2. THEORETICAL FRAMEWORK

Manual testing and automated testing are really two different processes, as opposed to two different approaches of executing the same process. Their intricacies are different, and they supply different kinds of information. Therefore, trying to compare them in terms of project resource consumption or typical defect finding rates is meaningless.

Test automation should generally be treated as one part of an excellent multi-pronged test strategy and as a complimentary activity to the bread and butter that manual testing poses, rather than letting automated testing dominate the process and thinking that the cumulative testing effort can be reduced thanks to cost savings. [1, 3, 35]

Automation is a significant investment

It is a common misconception that employing automated testing results in a higher number of faults found in a shorter amount of time, yielding reduced project costs. What exposes and isolates defects are test cases — converting test cases into automated test scripts and executing them multiple times with the same inputs to the system will rarely find other defects than when the scripts are executed the first time. Actually, informal surveys reveal that even projects with significant, well-designed automation efforts report that their automated regression tests suites find about 15 % of total defects reported. [1, 16]

Actually, a prerequisite for there to be any chance of reaching profitability in introducing automation to a project is that the product is mature enough. If not, the interface modifications between development iterations and changes in design will render most automation efforts fruitless. [3]

Automation generally involves coding in some shape or form. Although the programming language of the test scripts may be off according to developers, it is still code, albeit written by testers. This makes automation a software development activity, and not a testing activity. Every test script is really a feature, and every aspect of the underlying system — the system under test — is data. [14, 26]

The costs of a typical automated testing implementation effort — like many other software development related efforts — breaks down as follows: [3, 26]

- The cost of developing and planning the automation.
- The cost of operating the automated tests.
- The cost of maintaining the automation as the product changes.
- The cost of any other new tasks necessitated by the automation (documentation, managing test run results, testing the tests, meetings for coordination).
The tacit costs pertaining to most process alteration activities (management commitment, office politics, change resistance/inertia)

As these new tasks make a significant dent on the development team’s work load, one might be tempted to attempt to share these tasks in such a way that all testers work with automation part of the time. This has however in most groups been proven to be far less effective than dedicating team members to automation full time. As a kind of software development, it requires some amount of development talent — meaning testers without coding experience often lack the proper skill set for automation. One way or another, resource allocation is an issue that has to be cleared as a part of the overall automation strategy. [3, 35]

As automation is an activity that requires a lot of resources to introduce and does not give immediate results, project management and testers have to keep the project horizon in mind when working with the automation process. One more realistic method of calculating the cost of automation is to take the opportunity cost approach — what else could the project resources be spent on as opposed to allocating them to automation efforts? How many defects are being discovered later — or not at all — because of team members occupying themselves with coding automated test scripts? As already established, discovering defects later generally involve higher costs of repair. [14, 16, 19]

2.3.3 Best practices - what to automate, when and how

A software development team wanting to incorporate automation into their development process — given that there already is some kind of defined testing process and that the product is mature enough — should focus on the automation approaches that give the highest return of investment. Tests with high repeatability, broad scope, and high information content are strong contenders in this category. [14]

Generally, this would entail building up an automated smoke test suite that is run on new builds of the software, quickly painting the developers a rough picture regarding the state of the basic functionalities of the system under test. As smoke tests, they are intrinsically superficial and wide in their approach, meaning they are relatively cheap to develop and less susceptible to break than more thorough tests. Such a test suite may be executed without supervision as well, meaning it can be left to run overnight or during a lunch break, and the tester can return later to check the results. As an added bonuses, this test suite can be used for compatibility purposes — for example verifying correct web application rendering and correct behaviour under a different web browser. [8, 14, 27]

The test suite can also serve as a basis for a shallow regression test suite, where the suite is to be constantly tacked on with both pre-planned test cases as well as test scripts which previously have been used to reproduce and expose defects.
With time, the regression test suite will grow and provide a lot of information to the system’s stakeholders. However, in shorter projects the cost effectiveness of collecting and integrating a regression test suite is questionable. [17, 19]

When testing simple user interactions — like submitting data into a form and observing the output — resources can be saved even more by employing what is called the data-driven approach. Many capture/replay GUI testing tools include the functionalities of parametrizing the tests, and connecting the tests to a data source (e.g. a spreadsheet or CSV-file). This way, the same simple test can both be run in different ways using different input parameters as well as be used for testing every row of form input specified in the data source. Using the data-driven approach, the tester might seem like only one test is recorded, but it can lay the ground work for hundreds of test cases. [14, 16, 27]

That being said, some types of tests do require automation to be economically sound to perform. Few testing teams would rally 50 co-workers for an afternoon of load testing the latest build of a web application under development. Rather, these 50 users’ HTTP requests would be simulated using an appropriate load testing tool by a lone tester, and give significantly more information as to the state of the product. [27]

By tapping into the by far strongest property of automation — repeatability — the development team is given the possibility of saving man hours of testing already tested code, as the risks and costs involved in releasing faulty software are too high not to re-test the code. However, although automation reduces the human error factor, the flexibility and creativity coming with testing manually should not be overlooked. After all, most faults are still found in manual testing contexts. [16, 26, 27, 35]

2.4 Testing process maturity framework

2.4.1 Introduction

There are a number of process maturity frameworks in development in the academic world. The purpose of such frameworks is to serve as guidelines and reference material in aiding a company’s process improvement on a general level. Two of the most widely used process improvement frameworks are the Capability Maturity Model (CMM) and its successor Capability Maturity Model Integration (CMMI). Unfortunately, only limited attention is given to testing in these frameworks. As a result of this, the testing community has created its own improvement models.

In this thesis, the testing process maturity framework Test Maturity Model Integration (TMMi) will be explored and used as a basis for improving upon
Ipendo Systems’ testing activities. The TMMi is a detailed model for test process improvement and was developed to be compatible with and complementary to the CMMI. Source for the entire section 2.4 is the official TMMi documentation issued by the TMMi Foundation [12], unless stated otherwise.

Like the CMMI model’s levelled approach, the TMMi also employs maturity levels for process evaluation and improvement, where each level specifies process areas together with goals and practices to further gradual test process maturity growth, test engineering productivity, and — over time — product quality. Within the TMMi, testing evolves from disorderly, ad-hoc processes with unclear resource allocations and tools (level 1) to a mature, well defined, and controlled collection of processes with defect prevention as its main purpose (level 5).

Important to note is that the TMMi model does not describe specific use of tools during test execution, including tools for automated GUI testing, as only manual testing is described as the only test execution related activity. Rather, tool usage at this stage is implied at the reader or testing manager’s own discretion.

As no defined testing process exists in the official software development methodology at Ipendo Systems, and one of the goals of this thesis is to develop such a process, it will be assumed without further investigation that Ipendo Systems is currently at level 1 of TMMi. Therefore, after a quick overview of levels 1 through 5, only level 2 will be explored in detail as it constitutes the next step in test process improvement within the close future.

### 2.4.2 Overview of the TMMi model

Figure 2.2 at the following page gives a quick look at the different levels of the TMMi model and their respective process areas.

**Level 1 — Initial**

At entry level, testing is a chaotic activity in an unstable environment, often considered a part of debugging. Companies that despite this enjoy some success in the testing department rely solely on the competence and heroics of few people in the organization, as opposed to the use of proven and defined processes.

Tests are developed in an ad-hoc way after coding is completed, and testing and debugging are interwoven as a do-it-all bug isolation and repairing solution. The main objective of this activity is to show that the system under test runs without major failures.

When released, products lack information regarding their level of quality and risks. Once out in the open, the products are not stable, do not fully fulfil their requirements, and/or are too slow.
Maturity level 1 organizations are plagued by a tendency to overcommit, abandonment of processes in a time of crisis, and an inability to repeat former success scenarios. Furthermore, products tend to not be released on time, budgets are overrun, and delivered product quality is not up to par.

**Level 2 — Managed**

At TMMi level 2, testing processes are managed and are separate from debugging activities. Further discipline, coming from a more rigid base in the developmental approach, helps in keeping with practices even in times of stress. However, testing is often still seen as a distinct project phase and the natural follower of the coding phase.
CHAPTER 2. THEORETICAL FRAMEWORK

A company-wide test strategy is established for improving the test process and giving it structure. Project based test plans are also defined using product risk assessments as input, which in turn are derived from the product requirements. Test plans define what testing is required, when, how, and by whom.

Testing activities and their output are monitored and their execution is tweaked as necessary. Test design techniques are applied for developing appropriate test cases from specifications.

Testing’s main purpose at TMMi level 2 is to verify that the product conforms to its requirements — it is not until the upper tiers that fault prevention is introduced. A lot of quality issues at TMMi level 2 arise as a result of defects being discovered far too late in the developmental cycle — mistakes in requirements and defects in architectural design go unnoticed and propagate right into the code. Lack of formal review programs is a typical occurrence, as post-coding, dynamic testing still by most stakeholders is considered the primary — and perhaps sole — testing activity.

Level 3 — Defined

As the transition to TMMi level 3 is made, testing slowly comes out of its closet existence as a phase that follows coding and instead is integrated into the software development life cycle. Test planning is now done at an earlier project stage while being documented in a master test plan. A test organization and a specific test training program exist, and test process improvement is a full part of the test organization’s practices.

As the importance of reviews is realized in quality control, formal review programs are implemented, although not necessarily in conjunction with the already steadfast dynamic test processes. Also, non-functional aspects of the products are given more attention as test designs and techniques now widen to encompass aspects such as usability and/or performance as well, depending on business objectives.

Compared to TMMi maturity level 2, the scope of standards, process descriptions, and procedures is far more stringent on level 3. On level 2, these may differ by quite a bit in each specific instance or project, whereas they at level 3 are tailored from the organization’s set of standard processes to suit a particular project or organizational unit. Level 3 brings higher consistency — granting only differences allowed by the tailoring guidelines. As some of these processes are built upon ground work done at level 2, these process areas have to be revisited.

Level 4 — Measured

After having achieved TMMi level 2 and 3, a technical, managerial, and staffing infrastructure has been put into motion capable of thorough testing and continu-
ous test process improvement. In TMMi level 4 organizations, testing is perceived as evaluation, consisting of all life cycle activities concerned with investigating the state of quality of products and related work items and artefacts.

An organization-wide test measurement program will be started that will evaluate the level of the testing processes, to measure productivity, and to monitor the effects of changes. These measures, with their appropriate metrics, are incorporated into the company’s business intelligence system, making fact-based decision the standard in testing and quality related matters as well.

This measurement program is also to be used for quantitative product quality evaluations processes, whereby quality needs, attributes, and metrics are specified, gathered, and applied. Product quality is understood in measurable terms and is managed with respect to defined objectives throughout development.

Formal reviews and inspections are now considered part of the test process and are used to measure and evaluate quality early in a product’s life cycle and to formally control quality gates. The output of static testing is used in conjunction with dynamic testing in order to optimize the test approach, achieving higher effectiveness and efficiency.

**Level 5 — Optimization**

Successfully implementing all previous test improvement goals at level 1 through 4 of TMMi has resulted in an organizational structure capable of supporting a fully defined and measured process.

At TMMi level 5, the organization moves on to continually improving the test process based on statistic applications and quantifiable measures. Statistical sampling, measurements on confidence levels, trustworthiness, and reliability drive the test process.

A permanent test process improvement group is formally established consisting of members specially trained for this purpose. Support for such a group formally begins at TMMi level 3 when the concept of the test organization is introduced, while at TMMi level 4 and 5 the group’s responsibilities grow as more high level practices are introduced — such as identifying test (process) assets and developing and maintaining the test (process) asset library.

Defect prevention as also introduced as an own process area, as common causes of faults are identified in all stages of the development life cycle and measures are taken as to prevent the underlying mistakes from occurring in the future. The above mentioned statistically supported quality control techniques and test process optimization activities both aid the organization in further perfecting its
test processes and sustaining the expert level in testing knowledge that over time has been attained.

2.4.3 Process areas of level 2

As previously stated, transitioning to TMMi level 2 forces an organization to firmly embrace testing as its own — managed — process. Testing has to be clearly separate from other activities such as debugging, and be formulated and defined with a purpose.

Process areas of TMMi level 2 are:

- Test policy and strategy
- Test planning
- Test monitoring and control
- Test design and execution
- Test environment

Each of these will be discussed in more detail below.

Test policy and strategy

The first step in improving a company’s test processes comes by defining a test policy, which is to clearly state the organization’s overall test objectives, goals, and strategic views regarding testing. This is necessary in order to attain a common view of testing and its goals, compatible with the organization’s overall business and/or quality policy, that is required to be able to align test (process improvement) activities. The test policy should address testing activities for both new development as well as maintenance projects.

Within the test policy, the objectives for test process improvement should be stated. These objectives are subsequently translated into key performance indicators with appropriate metrics. These indicators should show the value of testing and test process improvement to the product’s stakeholders, as they are to communicate the expected and so far achieved levels of test performance in line with the direction defined by the test policy.

With the test policy in place, it is to serve as a basis for a test strategy. The purpose of such an artefact is to cover the generic test requirements in an organization, meaning it is to discuss generic product risks and present a process for addressing those risks — all in accordance with the test policy. Commencing work on a test strategy entails performing a generic product risk assessment on the products under development within the organization.
CHAPTER 2. THEORETICAL FRAMEWORK

The test strategy is to serve as a starting point for the testing activities within projects. A typical test strategy will include a description of test levels to be applied (see section 2.2.1) together with objectives, responsibilities, main tasks, and entry/exit criteria for each level.

When a test strategy is defined and followed, overlap and confusion regarding the test levels are less likely to occur. Also, as the efforts are better aligned, fewer gaps are likely to remain, resulting in an overall more streamlined process.

Test planning

With the defined test strategy as a basis, an overall study is conducted for each product to be tested with regards to its requirements, development process, and project organization. This study together with a product risk assessment results in a product specific test approach. Its goal is to specify which requirements of the products are to be tested, to what extent, how, and when — giving more attention and therefore test coverage to higher risk components as needed.

At this level of planning, resource consumption makes an entrance as a variable. Each test approach shall be supplied with a cost estimate — giving the reader a clear picture of the work involved.

The cost estimates, the product risk assessment, and the test approach together with other project management specific information (e.g. milestones, work products, scheduling, staffing needs, environmental requirements) together constitute the test plan.

The test plan, which at TMMi level 2 can be defined on a per-testing level basis for every product (TMMi level 3 defines the concept of the master test plan), is also to specify where, if applicable, the test plan deviates from the overall test strategy, and why. Finally, commitment is to be obtained to the test plans from the product’s stakeholders.

Test monitoring and control

The progress of testing and the quality of the products under development should be both controlled and monitored. The progress of testing is monitored by comparing the status of the actual test work products, tasks, effort, cost, and schedule to what is specified in the test plan, whereas product quality is monitored by factors such as risks mitigated, number of defects found, number of open fault reports, and status against test exit criteria or requirements.

In order to obtain the raw data required, test logs and failure reports are reviewed for validity before the above mentioned performance indicators are calculated. Test summary reports should be written and distributed on a periodic
and event-driven basis as a means to perpetuate a common understanding on test progress and product quality.

Whenever the measured test progress or product quality deviate from expectations, appropriate actions should be taken. These actions may entail re-planning, which may include revising the original test plan or adding alternative mitigation tactics for unforeseen risks. Corrective actions that bring alterations to the original committed plan should all be agreed upon by its stakeholders.

The driving force behind gathering and handling of measurements is test project risk management — which is performed in this process area in order to identify and solve major problems with the test plan as early as possible. It is also important to have a broad scope in the sense that problems beyond the responsibility of testing also should be able to be identified. For instance, organizational budget cuts, delay of developmental tasks, or changed product requirements can all affect the test process considerably. By building on the risks already documented in the test plan, tests project risks are monitored and controlled, and corrective actions can be initiated as needed.

**Test design and execution**

Having a structured approach to testing also entails employing test design techniques when designing test cases, where the basis for the design techniques come from requirements and design specifications. The test cases and their respective required running conditions are documented in a *test specification*.

Each test case is to be specified together with an identifier, a description of input values, execution preconditions, expected results, and execution post conditions. As the product under development matures, the test cases are translated into *manual test scripts*, describing specific test actions and checks in a step-by-step executable sequence. Specific test data required to be able to execute these scripts is also developed at this stage.

The test design and execution activities are to follow the test approach as specified in the test plan. Specific test design techniques are to be chosen based on test level, project resources, and the product risk assessment included in the test plan.

As tests are executed, any product failures are to be documented and logged using a bug tracking system and forwarded to the stakeholders as per established communication channels and protocols. A basic bug classification scheme is established for bug tracking, and a procedure is defined for handling the bug tracking life cycle process from discovery to closure.
CHAPTER 2. THEORETICAL FRAMEWORK

Test environment

A managed, controlled, and dedicated test environment is indispensable for any testing. It is also needed for obtaining test results comparable to what had happened under real-life conditions. This is especially true for testing at higher levels, i.e. the system level in the traditional subdivision of test levels (see section 2.2.1). Also, regardless of test level, the reproducibility of the test results should not be inhibited by undesired or unknown modifications or deviations in the test environment.

Specification of the test environment’s requirements should be performed early in the project, where the resulting document is to be reviewed for correctness, suitability, feasibility, and accurate mimicry of the real-life operational environment. Specifying the test environment requirements early has the advantage of providing more time in acquiring and/or developing the test environment and any needed auxiliary components such as stubs, drivers, or simulators. The type of environment required will naturally depend on the system under test and its specified testing approach.

One important issue that needs to be covered is whether or not several test environment should be employed — for example one for each test level. If resources are scarce, the same test environment could be shared between testers and developers. This would require rather strict management and control over the test environment so as to avoid a negative impact on overall progress due to miscommunication over the state of the test environment.

As a part of this process area, requirements regarding generic test data as well as its creation and management are also addressed. Whereas specific test data is defined and/or developed during the test analysis and design stage, generic test data is often defined and created as a separate activity. Generic test data is re-used by many testers and provides overall background data that is integral to performing basic system functions.

Finally, test environment management also includes controlling access to the test environment, providing login details, managing test (execution) data, employing configuration management, and providing information and technical support on any progress disturbing issues in test execution.
Chapter 3

Methodologies, products, and tools

3.1 Solutions Experts’ development methodology

Development of software at the IT consultancy oriented business area at Ipendo Systems is done using an own project framework heavily based on the ubiquitous Rational Unified Process, henceforth referred to simply as RUP. As management at Ipendo Systems wishes that the specifics of their development methodology remain within the company, only RUP — and none of Ipendo Systems’ alterations — will be described on a superficial level below.

RUP was originally developed by a company named Rational Software, which was acquired by IBM in early 2003. For this section, sources are both from Rational Software from before the acquisition [18], as well as more recent documentation from IBM [13].

RUP is a software development framework with I find to have both elements of recent iterative approaches (e.g. SCRUM, eXtreme Programming) as well as more traditional waterfall based methodologies. RUP’s main building blocks are categorized into nine disciplines together with specifications of roles, work products, and tasks (i.e. who does what and how). These nine disciplines in turn consist of six engineering disciplines — business modelling, requirements, analysis and design, implementation, test, deployment — and three supporting disciplines — configuration and change management, project management, environment.

As for the time dimension, RUP declares four distinct life cycle phases to a project, where each phase is segmented into one or more iterations. Each phase has key objectives, milestones, and entry/exit criteria denoting when the phase is
completed and the next can commence. An example of the plotting of discipline effort over project phases can be seen in figure 3.1 below.

![Figure 3.1: Discipline intensity over project phases in RUP [18]](image)

**Phase 1 — Inception**

In the inception phase, the main goals are to explore the system to be developed to the extent that the initial cost estimates can be validated. Here, a business case is established together with basic use case models, project plan and constraints, as well as a rough risk assessment.

**Phase 2 — Elaboration**

In the elaboration phase, the project starts to take shape. The use case models are further expanded upon until most actors and descriptions are developed, the architecture is realized, the business case and risk assessment are revisited, an overall project plan is written, and preliminary prototypes are developed to prove mitigation of identified technical risks. The key objective during the elaboration phase is a settled upon system architecture.

**Phase 3 — Construction**

This is where the system is actually coded. Main focus in this phase is developing the system’s individual components and features. The number of iterations in the
construction phase varies with the extent of the use cases, staffing, and scheduling. Some time during this project phase, an initial release of the system is made.

**Phase 4 — Transition**

Key objective in the final phase is to "transit" the system from being in development to its operational production environment. This includes checking the system against quality criteria established during the inception phase, performing acceptance testing, as well as training end users and maintainers. If all goals are met, the product is officially released and the development cycle is finished.

### 3.2 Overview of Microsoft Sharepoint 2010

Microsoft Sharepoint 2010 is a business collaboration platform with focus on interaction between people, content and information, and business intelligence data. Sharepoint 2010 includes a vast array of out-of-the box capabilities, but can also be heavily customized to address specific needs and integrated into other (Microsoft) products and solutions. It is often marketed as an enterprise intranet platform, but can be deployed onto the Internet as well.

Main sources for section 3.2, unless otherwise stated, are Microsoft’s official Sharepoint introduction documents called the *Sharepoint 2010 Evaluation Guide* [21] and the *Sharepoint 2010 Walkthrough Guide* [22].

According to Microsoft, the key selling points of Sharepoint 2010 are (as taken verbatim from [22]):

- **It delivers the best productivity experience** by letting people work together in ways that are most effective for them. Whether through the PC, browser, or mobile phone, SharePoint 2010 offers an intuitive and familiar user experience and enables people to collaborate effectively within the their current work context. These capabilities are significantly enhanced by the way SharePoint 2010 and Microsoft Office 2010 work together, enabling users to be more productive while using products and tools they are familiar with.

- **It cuts costs with a unified infrastructure** that offers enterprise-scale manageability and availability. Whether deployed on-premises or as hosted services, SharePoint 2010 lowers total cost of ownership by offering an integrated set of features and by allowing organizations to consolidate their business-productivity solutions on top of SharePoint Server. This leads to a reduction in costs related to maintenance, training and infrastructure management.

- **It rapidly responds to business needs** with dynamic and easily deployed solutions. Whether it’s an end user, a power user or a professional developer, SharePoint 2010 offers the tools and capabilities to design and create
business solutions that can be integrated with existing enterprise data, tools, and processes.

The main functionalities of the Sharepoint platform — of which there are six — will be described in further detail below.

Sites
Sites in Sharepoint 2010 are seen as a collection of individual web pages which in turn keep content and other functionalities. Users can create and edit pages within sites themselves using the Ribbon and a rich text editor. The pages may also be embedded with media files, Silverlight applets, or Web Parts — a pluggable user interface component. A set of pre-built Web Parts are included from the start, where their functionalities include RSS feed monitoring, business data access, calculation of key performance indicators, and searches.

The Ribbon user interface — prevalent throughout Microsoft’s Office suite since Office 2010 — is present in Sharepoint 2010 as well. Easily accessible from any state, the Ribbon grants easy access to the most common interactions with the system via an ever-present bar at the top of the page — all the while providing a consistent and familiar user experience. Like most components in Sharepoint 2010, the Ribbon is completely customizable and extensible. Figure 3.2 below provides an example of the Ribbon in its default state. [25]

![Figure 3.2: The Ribbon user interface in Sharepoint 2010](image-url)
CHAPTER 3. METHODOLOGIES, PRODUCTS, AND TOOLS

Files and documents from Microsoft Word, Excel, Powerpoint, and OneNote can all be accessed online in Sharepoint 2010 using the *Office Web Applications* without having the actual applications themselves installed. These documents can be viewed or edited by several users concurrently.

Aside from Internet Explorer, Sharepoint 2010 also supports Firefox and Safari. A number of native smart phones’ browsers also enjoy support for some functionalities, enabling users to view and edit documents and search for content while on the go.

**Communities**

In order to allow people and companies to work together to share knowledge and ideas, Sharepoint 2010 offers a number of social networking features. Users can design their own personal Sharepoint site and include a public profile displaying contact information and recent activities. A number of feedback options such as tagging, rating, commenting, and the ever-ubiquitous ”like button” enable users to discover content and partake in other’s thoughts of content.

Aside from social networking features, a number of alternative site types come ready to use in Sharepoint 2010. Wiki style sites aid users in knowledge management, users can keep their personal blog linked to their public profile, and forums can constitute the prime location for discussing company topics.

**Content**

As Sharepoint 2010 is meant to be the central hub for managing a company’s content, several functionalities have been included in order to handle as many types of content as possible.

Documents can be classified according to their content type and specific metadata properties, and subsequently grouped together with other documents into sets belonging to a single case or project. Metadata management and a degree of cohesion enable users to quickly navigate from within Sharepoint or through Office applications to find the content they need without knowing its saved location. Content types also support inheritance, meaning site owners can define specific content types for specific parts of the company.

The included infrastructure and editing tools for handling and publishing web content help people quickly and efficiently create, manage, and publish content pages with control over branding, page layout, and publishing approval. Also included is a web analytics package for gathering data on user activity, content inventory, and search engine use. This data can then be compiled into reports on the usage of specific Sharepoint sites or an entire Sharepoint web application.
Search
The search function in Sharepoint 2010 provides the user with intranet search as well as people search. Results are presented based on relevance, which is an aggregate of variables such as fuzzy URL matching, tags emanating from the social network area, phrase matching, and extracted metadata.

Insights
Insights is the business intelligence area of Sharepoint. The functionalities here allow users to gather, store, and analyse raw business data and turn it into usable reports without involving developers or server administrators. For example, key performance indicators regarding the progress of a project can quickly be calculated and visualized using built-in scorecards and chart Web Parts and published onto the company intranet.

Composite applications
Most companies need customized solutions that meet their specific needs and corporate culture. Sharepoint 2010 includes tools for creating custom solutions using out-of-the-box components without having to write custom code. Sites can be connected with external data sources or data-centric applications created in Microsoft Access, and custom workflows can be designed and exported for use in other Sharepoint sites.

The way Solutions Experts develops their Sharepoint based applications consists of combining custom code and Sharepoint’s out-of-the-box functionalities in order to fulfill their customers’ expectations and requirements. Other software components — such as extensions or add-ins to various Microsoft Office applications in order to seamlessly integrate the customers’ current workflow and processes into the Sharepoint based application — may also be included in the delivered product.

Section 3.3 below details the application chosen by management at Ipendo Systems for investigation of testability using TestComplete and LoadComplete.

3.3 System under test
3.3.1 Overview
The Sharepoint based application to be tested as a part of this thesis is a document management system. As management at Ipendo Systems wishes to keep specific details about this project secret, the system, henceforth known as the DMS, together with its functionalities to be tested will only be given a brief overview. Included in the testing are ancillary support components in the form of plugins to Microsoft Word and Outlook.
The main purpose of the DMS is to act as the customer’s central hub for storing and handling the documents pertaining to the intellectual property of their own clients. As the IP business is relatively artefact intensive, the DMS as of this writing houses well over a million separate pieces of documentation.

Documents belonging or related to the same IP are grouped into a case, which is assigned a unique identifier in the form of an integer with or without capital letters on the end. Each document also has the person responsible stored as metadata in the form of the person’s company e-mail address.

Because of the gargantuan mass of documents, the customer company wished to be able to upload documents into the DMS without assigning them to cases, as this would take some of the work load off the consultants working with the actual documents. This way, any employee can aid in the relatively mundane task of uploading documents and simply forward them to the appropriate person, who will then take over case assignment and any other handling from there.

### 3.3.2 Uploading and assigning to a case

A number of options exist for the user wishing to upload a document to the DMS.

- **Scanning** — a folder on the server running the DMS is shared with read/write access for employee user accounts. This allows a DMS user to scan physical documents using a special scanner, where the scanned files are immediately saved to this network folder. Every two minutes, the DMS server checks this folder and uploads any documents inside to the DMS, after which the shared folder is emptied. Documents uploaded using the shared network folder will need to be assigned to cases from within the DMS.

- **Outlook plugin** — e-mails and their attachments may be uploaded to the DMS using a right click context menu supplied by a plugin component in Outlook. The user can choose whether to specify a case for the file(s) being uploaded.

- **Word plugin** — if a user is working with a document in Word, it can be uploaded to the DMS using a plugin component residing in its own tab in Word’s Ribbon interface. Like with the Outlook plugin, the user is free to choose whether the document is assigned to a case or not.

The DMS main page consists of a customized Ribbon interface and four separate Web Parts — from top to bottom: ”Scanned Documents”, ”My Mailbox”, ”Active Cases”, and finally, ”My Relevant Documents”. The last mentioned Web Part provides a simple overview of recently handled documents regardless if they have been assigned to cases or not. Therefore, any newly uploaded document will be visible at the top of the ”My Relevant Documents” listing from the get-go. The listing shows up to the 40 latest handled documents during the last two weeks by default.
For practical reasons, the three remaining Web Parts will be described in detail in order of usage in the process of uploading and handling documents.

Aside from showing up in ”My Relevant Documents”, freshly uploaded documents without case numbers also end up at the top of the main page in the ”Scanned Documents” listing. From here, the user can edit the documents directly using the pre-supplied functionality in Sharepoint, or assign them to a case. The listing is sortable and filterable for ease of use, as well as paginated so as to not dominate the front page if a large number of documents have been uploaded in a short time span.

Once the user selects which documents should be assigned to a particular case, a window appears where the case number along with other types metadata need to be given in order to complete the process. The textbox for the case number has an auto-complete feature, where a dropdown menu appears below the textbox and suggests case numbers based on what the user already wrote.

The auto-complete feature utilizes JavaScript for display and the message-oriented asynchronous communication service BizTalk for the query itself. BizTalk is also developed by Microsoft but is not a part of Sharepoint.

Once a correct case number has been given, the user can fetch all metadata concerning the client at the click of a button. As any case only belongs to one client, the user does not have to go through the tedium of inputting this data him- or herself.

Aside from ”standard” types of metadata such as date/time, there is also the field named ”Responsible” meant to store the e-mail address of the user responsible for the document in question. It can be changed at this stage if the current user is not meant to be the one responsible once the document has been assigned to a case.

3.3.3 Handling case assigned documents

Once case assignment is completed, the document is no longer listed in ”Scanned Documents”, but is listed in the Web Part immediately under it — ”My Mailbox”. The listing in ”My Relevant Documents” remains as the document has in fact been recently handled. Any documents that have been assigned to cases already during the upload process will start out being listed in both Web Parts directly, as case assignment via ”Scanned Documents” is not needed.

”My Mailbox” serves as a detailed inbox for documents recently assigned to cases and that await further action. The document listing in ”My Mailbox” is paginated, sortable, and filterable in a similar fashion to ”Scanned Documents”. 
CHAPTER 3. METHODOLOGIES, PRODUCTS, AND TOOLS

However, "My Mailbox" has the possibility of filtering based on the person responsible. While the currently logged in user’s documents of responsibility are displayed by default, a user can quickly check other users’ recently added case documents.

Worthy of note is also that documents will have had their file names edited while making their way to "My Mailbox". Now, documents are referred to on the form of [CASE NUMBER]_[DATE]_[FILE NAME], where the "Date" field denotes the day the document was uploaded into the DMS. The listing in "My Relevant Documents" is also updated with this name change.

Every document in "My Mailbox" has three buttons to the right of its name. Each button’s functions is detailed below.

- **Received** removes the document from "My Mailbox", denoting that no further action is necessary at the current time. However, the document is still listed in "My Relevant Documents".

- **Forward** is meant for changing the person responsible for a particular document. Upon click of this button, a window appears where the "Responsible" field can be changed. The listing in "My Relevant Documents" remains after forwarding, and the document is only visible in "My Mailbox" for the new user after the operation.

- **Add to active cases** — below "My Mailbox" is the aptly named "Active Cases" Web Part, whose purpose is to serve as a to-do list of cases the user is currently working on. When the button is clicked, the document is no longer listed in "My Mailbox", but its case is now a new entry in "Active Cases", complete with links to the original document and head case page. Similarly to the functions above, the listing in "My Relevant Documents" remains even after this operation has been completed.

After completing this process, the document in question has ended up where it should belong and is kept there until needed. However, for users wishing to find documents not recently worked on or belonging to other users, there is a central type of functionality still to be described in the DMS.

### 3.3.4 Searching

Aside from utilizing the standard built-in search capability in Sharepoint, the DMS also has a custom search function called "Quick Case Filter".

As Sharepoint’s standard search more or less traverses the entire web application in its quest for relevant hits, "Quick Case Filter" provides a more lightweight alternative for situations when users simply want to find a certain case page or a particular client. Here, the database is queried with specific search parameters like case number or client name.
Another reason for the custom search function is visualization of results. In Sharepoint’s standard search, hits are shown as links to pages — similar to an everyday web search using Google — without more useful information than file name, URL, and some parsed content if found. Using "Quick Case Filter", any documents belonging to the particular client or case number searched for are presented as entries in a Sharepoint list Web Part displaying file name and type, document content type, and other metadata. This view is sortable and filterable as well, displaying almost 4 times as many hits in the same screen space as the standard Sharepoint search.

3.4 Test tools

Both TestComplete and LoadComplete are developed by Smartbear Software, based in the United States. Source for section 3.4 is Smartbear's official product documentation for TestComplete 8 [34] and LoadComplete 2 [33]. Screenshots were captured by the author of this thesis unless stated otherwise.

3.4.1 TestComplete

Overview

TestComplete is a GUI testing tool of the capture/playback variety, allowing the tester to record interactions with a system and condensing them to so called keyword tests. Employing keyword tests entails mapping the system under test’s graphical components into an object-based hierarchy, where commands applied to these objects simulate user actions. An example of a recorded keyword test is presented in figure 3.3 below.

<table>
<thead>
<tr>
<th>Item</th>
<th>Operation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run TestedApp</td>
<td>Orders</td>
<td>...</td>
</tr>
<tr>
<td>Orders</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MainForm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MainMenu</td>
<td>Click</td>
<td>'File</td>
</tr>
<tr>
<td>dlgOpen</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Edit</td>
<td>Keys</td>
<td>'C:s\Documents and Settings...</td>
</tr>
<tr>
<td>EtoOpen</td>
<td>ClickButton</td>
<td></td>
</tr>
<tr>
<td>MainForm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OrderForm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ProductNames</td>
<td>ClickItem</td>
<td>'Family\Album'</td>
</tr>
<tr>
<td>Customer</td>
<td>Click</td>
<td>91, 10, ...</td>
</tr>
</tbody>
</table>

Figure 3.3: Keywords and commands in TestComplete [29]
According to Smartbear, one of the main advantages of using keyword tests is that no programming skills are required. Tests can be put together by dragging and dropping operations, and mapping of graphical components can be done automatically while recording system interactions.

The object-based architecture behind the keyword tests also is supposed to make the tests more resistant to change, as specification of exact screen coordinates for interactions is not needed. The graphical components of the system under test are found regardless of location on the screen.

For more advanced testers, the keyword tests can be converted into automated test scripts in a range of languages such as JavaScript, C# Script, or VBscript. The test scripts can then be edited and extended with custom code according to the tester’s skills and abilities.

Keyword tests in TestComplete may also be parametrized and/or connected to data sources. This opens up possibilities of applying the data-driven approach, allowing the same test to be re-used with different inputs and outputs.

**Test logs**

As keyword tests are recorded, TestComplete automatically takes screenshots of any graphical controls manipulated during the recording. This is a part of so the "Test Visualizer", which helps in following and understanding the steps of a keyword test after the fact. Interactions such as clicks or keyboard input are also highlighted in these screenshots for clarity.

An example of this can be seen in figure 3.4 below, where a Windows Active Directory security prompt is filled in. First, the username textbox is clicked and filled out, after which the Tab key is pressed on the keyboard so as to transition the cursor to the password textbox. Finally, the password field is filled out as well.

![Figure 3.4: The Test Visualizer in TestComplete](image-url)
Tests can be organized into test runs, specifying which tests shall be run how many times and with which input parameters. Options can also be set regarding how TestComplete should recover and move on when tests return fails or warnings based on test run settings and parent-child hierarchies of test items.

After each test run, the results are condensed into a test log summary. An example of such a summary is presented in figure 3.5 below. While the summary is meant to visualize the test run on a mere superficial level, each run of each test can be accessed and investigated down to keyword-command detail level in order to retrace TestComplete’s steps. Fresh Test Visualizer screenshots are also presented at this level of the logs, giving the user a clear image of what interactions have been done with the system under test during that specific test run.

![Figure 3.5: Test run summary in TestComplete](image-url)
3.4.2 LoadComplete

Overview

While TestComplete does have some support for simulating HTTP traffic and measuring response times, SmartBear decided to break out and extend the load/performance testing part of TestComplete into its own product — LoadComplete.

Creating tests for LoadComplete starts out by recording HTTP requests and responses and condensing it into what LoadComplete calls a user scenario. In its simplest form, this would involve logging onto the web application under test, clicking some hyperlinks while waiting in between clicks so as to simulate the user reading text or thinking, and finally logging out.

A user scenario is then constructed from the observed HTTP frames being sent and received. Constructing an actual test in LoadComplete occurs first when this user scenario is combined with one or more load profiles denoting how the user scenario is to be applied over time so as to simulate load. Load profiles in turn specify a number of users, if recorded, randomized, or no think times shall be used, the users’ respective browsers and internet connection speeds, and with what intensity this group of users attempt to access the system under test.

Intensity can be specified in a number of ways. The simplest is for 100% of the users to interact with the system continuously, more or less spamming the system with traffic until enough data is gathered. Intensity can also be stepped up incrementally — e.g. 10% of the total number of users for the first 5 minutes, 20% for the next 5 minutes and so on — or plotted freely over time as visualized by figure 3.6 below.

![Figure 3.6: Plotting load profile intensity in LoadComplete](image)

Figure 3.6: Plotting load profile intensity in LoadComplete [32]
Test logs

Like in TestComplete, test runs are condensed into a summary, visualizing the most important data gathered during the test. Deciding if a test passes or fails is done by comparing against values defined before the test, e.g. highest acceptable page load time for any request during the test. An example of a test run summary is seen in figure 3.7 below.

![Figure 3.7: Test run summary in LoadComplete [28]](image)

If delving deeper into a LoadComplete test log, the tester can secure more specific data with regards to system response time from particular user requests. This way, the tester can receive a picture of typical system responsivity and user experience under tested load, and draw his or her own conclusions where the system needs improvement. The data gathered can also be used to check conformance to performance related non-functional system requirements.

3.5 Applying the test tools

3.5.1 General thoughts

After using TestComplete to record, play back, and manage tests over a time frame corresponding to about six weeks of full time work, my general impression
is that it gets the job done. While unstable at times — TestComplete hangs or crashes completely about once a week — compatibility with Internet Explorer and Sharepoint seems to be rather high.

Also LoadComplete, with which I worked with for a total of about a week full time, has some very good functionality built into it. However, some intricacies of the Sharepoint platform prohibit LoadComplete from rising above the status of merely serving as a load simulation program.

Although not measured with any sort of statistical significance, in general, the time required to automatize a test case into a usable test script seems to be around 10 to 20 times the time required to perform the test manually — excluding test maintenance. The exact coefficient depends on the type of test and how much additional programming is required after the initial recording. The fact that the tests themselves need to be tested also has an impact on the total time — inconclusive results caused by bad fails take time to circumvent properly and cannot fully be mitigated in the test case design stage.

As the display of the sequence of events in keyword tests (refer to figure 3.3 for a reminder) has an emphasis on actual interactions — somewhat logical considering the nature of the tool — execution of custom functions or snippets severely lack readability in comparison. While this clarity deficiency in part may be mitigated by commenting frivolously, refactoring of the custom code can still become quite time consuming. In particular, name changes of the keyword tests or their variables — despite the relative simplicity of this operation — required a considerable amount of time to propagate fully as there are no warnings for inconsistencies.

3.5.2 TestComplete and Outlook

Recording a simple script

One of the first tests I attempted to record was a simple uploading of an e-mail to the DMS from Outlook. The test consisted of right clicking an e-mail in the inbox, navigating to the custom menu component, and selecting to move or copy the e-mail to the DMS. See figure 3.8 on the next page for a screenshot of the context menu in Outlook.

If done correctly, a window would show where additional metadata — e.g. case number — could be assigned, after which the e-mail would show in the "My Mailbox" Web Part described earlier. The problem was that I never got that far in the recording of the test.

Non-standard GUI controls

This task, however mundane it may sound, proved to be virtually impossible to catch in an automated test script. When in recording mode and clicking any-
where in the custom popup menu, nothing happened except the whole menu dis-
appearing. Seemingly, the click did not register. Playback of the recorded script
generated the same result, except the DMS popup menu did not open at all —
the hovering of the mouse pointer over its menu entry was apparently not caught
either.

What followed was a two week long exercise in futility. After contact with the
SmartBear support department, I tried to "force" opening of the popup menu
by purposefully recording a click on its menu entry before interacting with the
popup. Nothing.

The next step was including Outlook in the TestComplete listing for applica-
tions where text reading is to be applied, followed by coding a script to activate
uploading into the DMS using only simulated keyboard presses — i.e. arrow
presses and the Enter key. Still nothing.

The root cause for this, according to SmartBear Nordic sales representative
Ole Vestergaard, is that Outlook curiously does not utilize the standard Windows
controls prevalent in practically every other Microsoft Office piece of software.
This leads TestComplete to not being able to distinguish between the custom
popup and its parent menu, believing them to be the same control. Therefore, an
interaction in the custom popup is caught by a non-existent menu entry in the parent menu. The end result is the same as if right clicking to open the context menu, followed by clicking outside of the menu’s bounds in order to close it.

At this point in time, I abandoned all efforts of recording tests in Outlook. I had a significant backlog of other tests to record — I had not yet started with the DMS itself — and time was slowly running short on the testing phase.

### 3.5.3 TestComplete and Word

**Parametrization**

Microsoft Word, on the other hand, fully incorporates Microsoft’s standard controls — with which TestComplete has proven adequate compatibility. The test recorded in Word was of the relatively simple nature of creating a new document with some text in it, and uploading it to the DMS using the Solutions Experts plugin. As a part of the uploading process, the document was also assigned to a case as well as assigned a content type for classification.

It was here that some of the power inherent in the concept of parametrization was tapped into. At the stage where the content type is chosen using a dropdown menu, I incorporated the use of an input parameter in the form of an integer in the keyword test. The value of the integer now specified the number of the entry in the dropdown menu to be selected — meaning the same keyword test could now be used to upload all of the different content types. As the Word plugin has nine content types to choose from for classification of documents, one test had become nine tests within a matter of minutes.

The Word test was one of the first tests I recorded that brought the value of programming skills to light. While recording and playing back keyword tests in itself can be done practically without coding experience, it greatly benefits from a programming background. It may not be as advanced as incorporating an ancillary script to enhance a keyword test, but as little as being able to use standard programming syntax such as `while`, `goto`, or `if`-clauses, which can be dragged and dropped into keyword tests, can strengthen existing tests or help the tester circumvent unexpected problems that may arise during recording.

**The value of programming skills**

One example of the above is that during the course of uploading a Word document to the DMS, the user might be asked to provide login credentials before the upload is allowed. This is done using a standard Windows security dialog as seen in figure 3.4. However, regardless if the "Remember my credentials" checkbox was ticked or not, the security dialog would sometimes show again after submitting the login data. The number of times it showed was quite random — anything from once to
five times — which made the test return quite a lot of bad fails as it was assumed that the dialog appeared only once, as it did during the recording of the test.

The solution was to wrap the keywords and commands concerning the security dialog in a while-clause, stating that the specific code snippet — performing the login — would be performed again and again until the security dialog would stop returning. Without my relatively superficial programming experience, it would be doubtful if this issue would have been solved otherwise. Now, the test was fully repeatable, and did not produce bad fails at this point in the script any longer.

Naturally, the value of applying programming skills while developing automated tests does not only apply to TestComplete and Word — the issue described above simply emphasizes the fact more than any other issues encountered during this study.

3.5.4 TestComplete and the DMS

Keeping tests to the point

After finishing a recording in TestComplete I recommended to immediately purge the test of irrelevant interactions. It is quite easy to forget that the recording engine catches everything one does, so actions like minimizing a window that shouldn’t be in view or lowering the volume using the keyboard all are caught and included in the final keyword test. Even simply placing the mouse pointer on a hyperlink is recorded as its own action — "HoverMouse" — and will be replayed exactly as was recorded.

All these unneeded commands should be removed from the keyword test as readability is impaired. The point could also be made that a test containing a lot of HoverMouse actions is more susceptible to break than one purged of such commands — the more objects are mapped and interacted with, the higher the likelihood of one of them changing at some point in the future — rendering a broken keyword test that fails with the error message "object not found".

The equals defect

One of the tests required me to save some metadata from a document in the "Scanned Documents" Web Part before manipulating the document and assigning it to a case. Finally, the keyword test was to compare the metadata in the document to the metadata saved before the operation to make sure that only the appropriate fields had changed, and changed correctly.

However, one of the fields — Connection ID — proved problematic. Connection ID is a Sharepoint specific attribute given as a unique identifier of a connection with a Web Part, and is represented by a combination of capital letters and numbers. Upon comparison of the respective connection ID field values from before and
after the operation, where the values should be equal, TestComplete deemed the test a failure with an error message on the form of "Expected value: 'ABCD12EF', Observed value: 'ABCD12EF', Comparison method: Equals, Conclusion: Fail".

As the field values seemed equal, in an effort to further isolate the problem I quickly recorded a simple test where the connection ID field value is fetched from the top document in "Scanned Documents", after which the saved value is compared to the field value it was just saved from. In other words, the value is checked to see if it equals itself, and such a comparison should always return true. However, the test failed with the same type of error message as before.

I have not invested any extensive amount of time in isolating the source of this problem, as the connection ID field has been the only field to suffer from this problem so far. Therefore, it is unclear whether Sharepoint or TestComplete is the culprit. My workaround was to instead compare connection ID:s using the contains boolean operator, essentially checking if the entirety of one text string is represented within another text string. With the specific case of connection ID:s, using the contains operator should not be able to cause bad fails, as the field values always are of the same length.

The lesson learned from encountering what I refer to as "the equals defect" is that anyone working with TestComplete has to keep in mind that these types of defect may occur, and that they can be easily mitigated awaiting future bug fixes or patches. As popular wisdom dictates, knowing is half the battle.

Auto-completion of case numbers
As previously described in section 3.3.2, one custom functionality added by Ipendo was the possibility of an auto-complete feature when assigning documents to a case.

After recording the interaction with TestComplete and incorporating checks for the existence of the JavaScript dropdown menu and its contents, the resulting keyword test would fail upon playback because the dropdown would not show.

As it turns out, the default command that TestComplete applies when entering text into a textbox is the "SetText" command, which is roughly equivalent to copying and pasting in the text string in question. However, this apparently did not qualify for triggering the dropdown.

The solution was to edit the keyword test, where the command applied to the textbox was changed from "SetText" to "Keys". The new command simulates proper key presses on the keyboard. Now, the dropdown was triggered and found by TestComplete during playback.
CHAPTER 3. METHODOLOGIES, PRODUCTS, AND TOOLS

However, TestComplete was not able to read the contents of the dropdown correctly. While the display of the dropdown on screen is a clear representation of hits for the case number in question — every hit separated by a carriage return — TestComplete reads the dropdown’s entire contents as one continuous string without white space. In the interest of obtaining as conclusive results as possible, I decided that using the contains operator for verification of the dropdown’s contents would suffice.

After all, if the test would fail at this point in the script, either the dropdown menu did not show at all (error message: "Object does not exist"), or the dropdown showed correctly but displayed no or incorrect hits (error message: "Object does not contain the expected string"). Ergo, the risk of bad fails because of this specific issue has been mitigated by sufficient falsifiability.

Mapping of objects

The behind-the-scenes object mapping taking place whenever tests are recorded seems to work well in the sense that it proves rather robust. Although some tests had to be tweaked during my time with TestComplete due to regression defects or irregular system performance, object mapping was never at fault for broken tests. Minor changes in the GUI occur all the time in the DMS as document listings in Web Parts are handled intensively — and the tests remained intact. A different tool — for example basing its GUI testing approach on screen shot comparisons — would not have been able to handle the DMS, as recorded tests would be obsolete within minutes.

One minor issue is that the object depth of the mapping in the DMS becomes so deep that it becomes virtually impossible for a human being to decipher which controls have actually been interacted with in the test logs — the number of child levels often surpasses 20, and the objects’ names make little sense. This is due to the underlying Sharepoint handling the HTML object hierarchy, which developers have little control over. Thankfully, TestComplete supplies a highlighting feature as well as additional screenshots in order to visualize the steps taken in earlier test runs.

A caution should be added about instances of programs during recording. If several instances of for example Internet Explorer is open, the automatic mapping may define DMS controls as specific child elements of the third instance of the browser, as opposed to the active browser window in general. After a reboot of the test machine, TestComplete would give an error stating that the control mapped earlier cannot be found, as only one instance of Internet Explorer is open. This issue can easily be mitigated by making sure to only have one instance of each application open at any given time. Consistency and repeatability are easily achieved by starting each test script out by opening the applications and closing them at the end.
For compatibility tests of alternate browsers, each object interacted with in the tests has to be re-mapped in the new browser. Once this is done, test scripts can easily be converted by editing the browser instance in every step of the script. Although this might seem rather time consuming, it still is radically quicker than re-recording entire test suites.

One type of functionality proved completely incompatible with TestComplete, however. When searching using the "Quick Case Filter", TestComplete mapped the control containing the results of the search as its own unique control every time the search was used, making verification of specific search results an impossibility. Smoke verification of the "Quick Case Filter" instead had to rely upon checking the contents of other, more static, controls on the search results page.

In part, the overall stability can be chalked up to the fact that vanilla Sharepoint has a rather static GUI and predictable component hierarchy. The DMS in particular does not delve that deep into a dynamic presentation — there is always the Ribbon at the top, and pages consist mostly of Web Parts, which in turn essentially are tables in some form or another. Although DMS users complain of a somewhat "boring" user interface, it is a thankful task to test it.

It should also be added that the fact that the DMS' GUI was mature enough when I stepped in contributed tremendously to test stability. The DMS project has been running for a long time and is now at a post-RUP maintenance stage with some new functionality added occasionally and no GUI overhauls planned.

Waiting

Although unresponsiveness is supposed to be handled by TestComplete — "Wait" commands are automatically added to keyword tests where needed, denoting that an object on screen may need additional time before being fully accessible and accepting of input. During playback TestComplete will wait until the object is ready within a time limit specified in the test project’s options. If the object is not ready within the allotted time frame, the test times out and fails.

However, it occurred on multiple occasions that TestComplete stayed in "Wait" mode even after the object in question had loaded completely, suggesting some sort of miscommunication of states between TestComplete and the browser or the SUT. The solution in this case was to switch the "Wait" command to a hard coded "Delay", forcing TestComplete to wait for the specified amount of time regardless of the state of responsiveness of the object at hand. This way, the risk of state misinterpretation is completely avoided.

It should be noted that I am not advocating replacing every "Wait" command with "Delay". Although the occurrence of this issue seemed to be contained to popup windows, it cannot be ruled out that it could apply to other types of
components as well. As this does not seem to happen very often, and the risk has a relatively low impact — the test fails — no precautionary measure should be needed. Switching commands where ever the issue is discovered should suffice.

**Test environment control**

Without proper control of the state of the test environment, elements left over from previous tests may interfere in new test runs and cause bad passes or fails. For example, if a test puts a certain case in the "Active Cases" Web Part, and a following test in the suite relies on "Active Cases" being empty before starting, the credibility of the results of the second test might be affected.

Also, the test machine should be a dedicated environment so as to ensure a higher degree of control of the system state. If the environment is shared with developers, their interactions with the system may interfere during the execution of a test suite. At minimum, a dedicated account must be made for pure testing purposes.

An anecdote to exemplify: one day I noticed that the test of the "Forward" functionality in "My Mailbox" consistently failed in the middle of execution. Upon further investigation, the stage where the script failed was at specification of the e-mail address of the person to forward the document to — the DMS gave the error message that the e-mail address did not exist in the system. As this had not happened before, it probably meant that either a regression defect had been found, or something else had changed. Upon asking a developer for input, he answered that he had removed that e-mail address from the system for some reason and had seen no reason to tell me, as he did not know it was a vital part of the test script. Upon reinstating the e-mail address in the user directory, the test ran without a hitch again.

This was caused by an error of miscommunication, and had been avoidable by either keeping me in the loop constantly about changes in the system, where the majority of this information would be irrelevant for black box testing, or by employing a dedicated testing environment with strict control.

That being said, some things cannot be fully controlled. While uploading many documents at the same time using the scanner functionality, they can be grouped together using similar file names. The end result is that the documents all end up on the "Scanned Documents" Web Part, where the first document in order is labelled with an additional ".1" at the end and displayed at the top, and the rest are indented in the listing together with correct numbering (".2", ".3" and so on). However, due to unstable handling in *forEach*-clauses in custom Sharepoint code, the order of the documents cannot be guaranteed to be sorted in order, but rather show up unpredictably in "Scanned Documents".
CHAPTER 3. METHODOLOGIES, PRODUCTS, AND TOOLS

While the indentation was present in all documents except the top one — as should be — this had the ramification that the test script for this functionality cannot (1) verify correct order, and (2) verify that all documents are present with correct file names. Checking for every possible order of documents is too complex to be worth the effort. Instead, only the top documents are checked for a valid file name without the final numbering using the contains operator, as well as checking for the existence of the indentation cell where applicable. In conclusion, where the possibilities of test environment control end, the tester will just have to make do.

3.5.5 LoadComplete and the DMS

A note on expectations

As quoted by developers at Solutions Experts, Sharepoint 2010 is "not famous for being the most responsive platform" to build applications on top of. As a disclaimer to load testing of the DMS: although not supported by facts, the developers speculate that the massively exceeded recommendation for the number of documents handled by a single Sharepoint server might have an additional impact on the already somewhat slow DMS. Microsoft, on the other hand, as expected assumes no responsibility for lacking performance when their recommended maximum number of documents per Sharepoint server is annihilated by a factor of one hundred.

Logging in

LoadComplete features an automatic login function, where credentials supplied by the tester beforehand are applied during recording of the user scenario as well as subsequent playback of tests. However, when reviewing the log of sent and received requests and responses, all of the responses gave the HTTP status code 401 — denoting unauthorized access. Ergo, the login attempt had failed.

The next step in solving this issue was recording a new user scenario where logging in manually was the first part of the scenario. Here, I noticed that Internet Explorer automatically logs on to the DMS using the same user as is currently logged in on the test machine — and the issue persisted despite forcing a logout beforehand.

This was possible due to the DMS being connected to the same user directory service as the Windows machine itself. It is a standard practice to employ this integration, as a user of the operational DMS will not have to keep track of yet another set of login credentials, but will rather have immediate access upon logging onto his or her Windows account. As Internet Explorer seemed to log the user in behind the scenes using something else than normal HTTP traffic — such communication would have been caught and replayable by LoadComplete — recording a LoadComplete user scenario for any Sharepoint application where
the test machine was on the same Windows domain seemed to be an exercise in futility.

On a whim, I tried recording the same user scenario with an alternate browser. This proved to be the solution to the problem, as the first thing to show after entering the DMS URL was a login screen. Recording using this login method resulted in tests where the simulated user was actually logged in and able to interact with the system — as should be. The only difference I noticed to using Internet Explorer was that the headers of the HTTP requests stated that it came from a Firefox browser, but it is doubtful if this could have any bearing whatsoever on test results.

After this discovery, all user scenarios for performance testing of the DMS were recorded using Firefox.

The nature of hard coded scripts
First off, I find the lack of commenting in the user scenarios to be a somewhat odd omission. The only means I found was a "Description" field in the user scenario settings, which gives the possibility of providing a brief overview of what steps are taken in the recording. Being able to add additional comment rows between HTTP requests to highlight what has been recorded would greatly improve the detail of the test cases’ documentation, as interpreting the requests through the URL:s accessed can be relatively cryptic. This is especially true for applications built on top of pre-existing platforms — for example DMS on top of Sharepoint — where developers have relatively little transparency regarding the inner workings of the underlying platform.

After solving the login problem described above, I scraped together some simple, working tests where the simulated users log on to the DMS, check some document properties, use the quick case search, use the Sharepoint standard search, and log out. The next step was incorporating data manipulation, where I recorded a user scenario featuring a simulated user logging on to the DMS, deleting the top document in "My Relevant Documents", and finally logging out.

While this interaction would have been fully repeatable in TestComplete thanks to the integrated object mapping — the top document will always be accessed regardless of which one it is — in LoadComplete, the hard coded URL:s and requests resulted in the test engine trying to access the exact same document as was deleted during the recording of the script. As the document did not exist any longer, this resulted in a log filled with HTTP response code 500 — "Internal Server Error". The recording was useless.
While it would be possible to integrate data manipulation, e.g. the standard CRUD operations (create, read, update, delete), into the user scenarios using variables, parametrization, or more individual user scenarios, the load generated on the server hardware would not necessarily be more realistic compared to real world conditions. If the goal is just to observe behaviour under load, tests containing data manipulation might be unnecessary. This issue will be further discussed in chapter 4.

Information gathered

Unfortunately, LoadComplete is incapable of gathering any information of value other than min/max/mean times of page access, as details concerning specific requests say nothing about custom code performance — the execution path has to go through Sharepoint first and that is where the transparency ends.

Applying simpler user scenarios in different combinations of simulated users might suffice in generating load on the server. In the mean time, the tester can observe and measure the performance of specific code snippets or database queries using developmental debugging tools. Fortunately, the so called Developer Dashboard tool for Sharepoint development does exactly this, and is delivered out-of-the-box together with the server software for Sharepoint 2010 [23].

Delivered tests

The tests delivered to Solutions Experts at the end of the LoadComplete evaluation period were based on a simple user scenario where a user logs in, surveys the front page, clicks some document properties pages, searches using the "Quick Case Filter", and finally does a search using the standard Sharepoint search functionality.

This user scenario was then applied in different amounts of simulated users under constant load under intervals of time ranging from 10 to 30 minutes in order to investigate DMS behaviour under stress.
Chapter 4

Solution

4.1 Best practice for the tools

4.1.1 Controlling the test environment

As described in the test environment section at level 2 of the TMMi Model, the test environment should in the ideal case be one dedicated to the sole purpose of testing. For situations where resources are scarce however, using an environment shared between developers and testers might be acceptable as long as it is strictly managed and controlled.

In the shared environment used at Solutions Experts, miscommunication and lacking control over the state of the system caused problems several times during this study. If the resources were more than abundant, my recommendation would be to have separate development and testing environments, but considering the massive amount of documents needed to replicate the customer’s operational environment, it is difficult to justify the costs involved.

With cost coming in as a factor, there is really only one option left — to continue to use a shared environment. However, I recommend at a minimum that two dedicated user accounts are created for testing purposes — one is not enough for testing use cases involving several human actors, e.g. the ”Forward”-button — where the state of these user accounts are strictly controlled by the person responsible for testing.

4.1.2 How to apply TestComplete

My recommendations for using TestComplete at Solutions Experts can be condensed into the points below.
CHAPTER 4. SOLUTION

General compatibility

The fact that Microsoft Outlook is using non-standard graphical controls which — according to my findings — provide lackluster compatibility with the TestComplete object mapping leads me to discourage any attempts at recording tests in the application at the moment.

For anyone wishing to solve the issue, there are two possibilities. Either, a stubbed version of the custom popup menu in the Outlook plugin is sent to SmartBear for debugging, or the popup menu is rebuilt to show its entries directly in Outlook’s right-click context menu. I advise starting out with the latter option, as this constitutes the quicker route. The issue whether TestComplete is compatible enough with the non-standard GUI controls in Outlook for it to be worth the time invested in automating its tests still needs further investigation.

As for recording tests in Microsoft Word and the DMS, TestComplete had no problems whatsoever mapping and handling the graphical controls. Although some hiccups did show along the way — as was expected — my overall impression was that TestComplete constitutes a fully adequate test tool for these applications.

Test the tests

A rather time consuming issue throughout this study was the fact that recorded tests need to be tested themselves. Some problems in this area arise from lacking repeatability in parts of the system (see section 3.5.4 on testing "Quick Case Filter"), while others have to do with random oddities impossible to foresee before commencing with a test recording (e.g. the equals defect, also in section 3.5.4).

Aside from purging tests from recorded trivial interactions such as "Hover-Mouse" or accidental clicks, executing a newly recorded test script a couple of times to observe its flow of events is vital to mitigate inconclusive results and verify consistency. Played back scripts do not always perform identically to expectations, and often need further refinement before being declared an adequate manifestation of a manual test case.

Repeatability and parametrization

There is little doubt that one of the key strengths with test automation — and TestComplete in particular — is repeatability. Keywords and commands can be converted to flexible scripts, or copied and pasted to create new tests without resorting to recording anew. The option of using variables, input parameters, and programming syntax creates an extensive toolkit for a Solutions Experts tester.

Employing the above features increase complexity, making test maintenance and refactoring more difficult and time consuming. However, when used efficiently, the power to create many tests out of one grossly outweighs the increased complexity.
Therefore, my guideline for this aspect of using TestComplete can be boiled down to the following:

- In general: the more possible uses a test has, the better. Parametrize and look into the data-driven approach where applicable.
- Re-use test code when possible — it saves time.
- Comment and document frivolously for superior readability.
- Keep tests simple.

**Approach to automation and its costs**

Achieving cost effectiveness in automated testing is an issue without a clear answer. It is in part a matter of picking the right tool and applying it effectively — as little resources as possible should be wasted trying to figure out compatibility issues with the system under test, or how to work around instability in the tool itself. Ideally, project resources marked for automated testing are to be spent recording, executing, and maintaining tests as well as reviewing results. As other studies have pointed out, a best practice is to treat automated testing as a form of software development, and letting developers do the automation results in more effective tests.

What can be said, however, is that designing and implementing a combined smoke and regression test suite to be used in conjunction with manual testing is one of the approaches with the highest potential for a return on the invested project resources. Although superficial, the information provided by the automated test suite is of high value to all stakeholders of the project, and can be used as a basis for deciding if and where further testing is needed.

In conclusion, my recommendation is the same as my earlier conclusion on the topic — if it is judged that a combined smoke and regression test suite will not be profitable in terms of invested effort and project variables, the project’s testing efforts are best spent in testing manually only.

**4.1.3 How to apply LoadComplete**

**General thoughts**

As for the application of LoadComplete with Sharepoint applications, it is to be assumed that the issue with logging onto the application will persist regardless of if it is the DMS or a different site being tested — the DMS uses no custom login functionality. Therefore, using Firefox while recording user scenarios is recommended for the time being.
CHAPTER 4. SOLUTION

The results from the trial phase with LoadComplete concluded that the information gathered and presented in the logs, as visually appealing it may be, does not supply sufficient information for the tool to be used on its own. Therefore, I recommend to always activate the Developer Dashboard for the testing environment when performing tests concerning non-functional requirements so that satisfactory metrics can be gathered for benchmarking or tuning.

User scenario complexity

The question remains whether it is worth investing the time to improve upon the depth of user scenarios by adding data manipulation and variables, or if simple simulated users suffice for generating load.

After observing the response times involved in Sharepoint applications such as the DMS and seeing response times climbing up towards half a minute for accessing the DMS front page while under light non-manipulative load, the findings points to simple user scenarios generating “enough” for general load testing purposes. In the specific case of the DMS, accessing the front page generates such an amount of database queries to cause considerable activity on the database layer either way.

Attempting to implement elaborate load tests featuring data manipulation would be defendable when for example targeting concurrency issues in the database layer of the Sharepoint server. However, execution of code in this area of the DMS, according to developers at Solutions Experts, typically consists of approximately 95 % Sharepoint platform code and 5 % custom code developed in-house.

Even in the most extreme case, say Solutions Experts developers, where a hypothetical extensive custom cache functionality is specifically targeted for concurrency testing under load, no more than 20 % custom code is executed.

Main focus while load testing Sharepoint based applications must be to verify the behaviour and tune the performance of custom code while under load. In cases where the quota of custom to platform code is relatively humble, development team members should consider if their testing efforts are better applied using other testing techniques — the surrounding Sharepoint code is not available for developers to review or edit anyway, and testing it is therefore a waste of resources.

If relatively little custom code is used throughout the application or only minor performance gains are achieved after tuning and performance still is lacking, the next step should be considering a server hardware upgrade.
4.2 Propagating changes through traceability

The solution to handling traceability so that changes propagate correctly between living documents is to instate the practice of keeping a requirement traceability matrix. As Solutions Experts utilize use cases as a part of their RUP based development approach, it follows that a table visualizing the connections between a project’s use cases and the test cases verifying functionality handled in these use cases would fulfil this purpose.

As a change is made in a use case, the person responsible is then to verify the consistency of the requirement traceability matrix ("Which test cases might be affected by this change?")", review these test cases ("Do the test cases still verify the functionality detailed by the use cases to a sufficient degree? Will the test cases still be executable in the new version of the system and return conclusive and relevant results? Is there a need for new test cases?"), and finally edit the test cases as needed.

An example of a requirement traceability matrix can be seen in figure 4.1 below. UC and TC represent use cases and test cases, while SRTS and FPTS stand for the purely hypothetical smoke/regression test suite and front page test suite, respectively. As a use case is changed, its row can be highlighted or colour coded to denote a pending review of the affected test cases.

![Figure 4.1: A hypothetical requirement traceability matrix based on my template](image)

4.3 Designing a test process

4.3.1 Introduction

Applying level 2 of the TMMi model as a makeshift check list, I outlined testing activities and artefacts to be incorporated into the existing development framework at Ipendo Systems. As already established, I unfortunately cannot share the
complete development process document because of its confidential nature. However, I will present the activities designed by me in their respective RUP phases, as well as concepts that need to be established outside of the existing development framework.

This may be taken as somewhat underwhelming as they cannot be seen as the parts of the larger developmental machinery that they are, but rather as individual cogs laying on a mechanic’s floor. They do however still serve their purpose when seen in motion inside the engine they drive.

To go along with the freshly proposed processes and activities, I also designed templates for internal documentation based on already existing templates and the TMMi level 2 specification. Included are templates for test plans, test suite specifications, and requirement traceability matrices.

Unless otherwise stated, all RUP activities detailed in sections 4.3.3 through 4.3.6 are a part of the testing discipline.

4.3.2 Top-level views, goals, and schemes

Organizational test policy
As Solutions Experts’ top-level goals on the topic of testing are not clearly defined, I suggest that an organization-wide test policy is to be declared. This is to define what testing and software quality means for Solutions Experts together with overall test objectives, goals, and strategic views, so that all organization members obtain a common view on testing their software and why.

As specified by the TMMi model, the test policy should also include objectives for test process improvements, which are to be translated into key performance indicators visualizing the value of testing as well as the progress of the test processes.

Organizational test strategy
With the veritable basis for all testing operations in place in the form of the organizational test policy, the next step is to identify and discuss generic test/product requirements and risks, as well as present methods for addressing these risks.

What also needs to be defined is a description of each test level to be addressed in projects together with main objectives, areas of responsibility, tasks, and entry/exit criteria for each level.

The finished test strategy will serve as a natural starting point for further test planning on project-specific level, all the while aligning and converging test efforts within the organization.
Defect classification and life cycle management

One important aspect of any testing effort is to document defects found under development and track them as outstanding issues to be investigated. As a part of this process, the defects are classified according to severity — e.g. ”major” and ”minor” might suffice, although a scale from 1 to 5 would provide more detail — as well as taken through a formalized life cycle.

This ensures that the issue goes from discovery to closure and that the issue is thoroughly investigated — possibly resulting in repair or refactoring, or, if there has been a misunderstanding, a dismissal of the issue. Having this classification system (partly) available to other stakeholders than the developers is usually appreciated as it provides transparency and information of value.

One way to employ a defect classification and life cycle system would be for Solutions Experts to use the already existing intranet Sharepoint site where other project artefacts are already stored. A Sharepoint list page can be created cataloguing defects found together with more detailed information, such as how to isolate and reproduce the defect, the system build it was detected in, and the developer responsible for the issue. The list can then be enhanced with a custom Sharepoint workflow for every list item, visualizing the status of the issue in the defect management life cycle — e.g. ”open”, ”under investigation”, ”fixed/closed”, and ”dismissed”.

4.3.3 Inception phase activities

Define a test approach

The test approach is the first part of the Test plan. Based on the generic risk analysis in the organizational Test strategy and the product-specific risk analysis conducted in the Project management discipline, decide — on a general level — which requirements of the product will be tested, to what degree, and how. The goal is to provide the best possible coverage to the parts of the system with the highest risk.

Input:

- Organizational Test strategy
- Risk analysis
- Draft requirement specification
- Draft iteration plan

Output:

- Test Approach part of the future Test plan
CHAPTER 4. SOLUTION

Actor:

- Test Manager

Actions:

- Based on the risk analyses, list the items and features to be tested and not to be tested.
- Select test design techniques for each item or feature to be tested. The resulting set of design techniques should be sufficiently detailed to support identification of major test tasks and estimation of time required to perform each task.
- Define the approach to review test work products.
- Define the approach for re-testing.
- Define the approach for regression testing.
- Identify the supporting test tools to be used.
- Identify significant constraints for the overall test approach based on other project variables.

4.3.4 Elaboration phase activities

As outlined by IBM, the activity of defining a test plan is originally meant to belong in the late inception phase. However, after discussion with Solutions Experts staff regarding compatibility with the rest of the development framework, it was decided to instead place it in the early elaboration phase.

Plan testing

Establish a well-founded plan for performing and managing the test activities of the project. The master test plan provides an overview of what testing is required, when, how, by whom, and at what cost. As the Test approach defined during the Inception phase is a part of the Master test plan, its contents need to be verified to be consistent with current requirements before integration into the Master test plan.

The more defined Iteration test plan is only established for the next-coming iteration of the project.

Input:

- Test approach
- Draft iteration plan
CHAPTER 4. SOLUTION

Output:

- Master test plan

Actor:

- Test Manager
- Developers (for effort estimation only)

Actions:

- Define entry, exit, suspension, and resumption criteria.
- Define and identify milestones and work products.
- Define test life cycle and tasks.
- Estimate total test effort and cost.
- Plan test schedule and staffing.
- Plan stakeholder involvement.
- Investigate environmental needs and requirements.
- Identify and analyze test project risks.
- Identify any non-compliance to the organizational Test strategy and its rationale.

Test suite specifications

Test suite specifications are based on the requirements, architectural documentation, and the test case design techniques outlined in the Test approach. The Test suite specification is a description of each Test suite and its respective coherent set of Test cases sharing a common goal. The Test cases in turn specify the steps of executing the test and the expected results along with any pre- and post-conditions. The connections between Use cases and their respective Test cases are recorded in a Requirement Traceability Matrix.

Input:

- Use Cases
- Supplementary Specification
- User interface model
- System Architecture Document (SAD)
CHAPTER 4. SOLUTION

Output:
- Test Suite specification
- Requirement Traceability Matrix

Actor:
- Test Manager

Actions:
- Design and prioritize Test cases. Mark any Test cases meant for automation along with the proposed tool.
- Identify and include necessary specific test data, pre- and post conditions.
- Maintain traceability with the requirements in the Requirement Traceability Matrix.
- Verify sufficient coverage of requirements from UC and Supplementary Specification as outlined in the Test plan.

4.3.5 Construction phase activities

Implementation discipline — Test implementation

Test cases scheduled for automation in the Test suite specification are coded and/or recorded using the selected tool(s), along with any supporting scripts needed for pre-execution setup or post-execution teardown. The system’s graphical interface must have matured to the point that none or few of the automated test scripts are expected to break in future builds of the system.

The automated test scripts should also be reviewed by someone else than the person coding the script so as to ensure correct implementation of the Test case and eliminate test result ambiguity, i.e. test scripts failing or passing for other reasons than originally intended.

Input:
- Test suite specification

Output:
- Automated test scripts

Actor:
- Tester
CHAPTER 4. SOLUTION

System test
The system test aims at testing the system according to the Test suite specification in an environment mimicking real-life conditions as closely as possible. The purpose is to verify that the system conforms to its functional requirements.

System testing is based on Test cases described in section (withheld) together with supplementary exploratory testing.

A manual test script outlined in a Test case should not be executed by the same person who developed the functions tested by the script. Automated test scripts may be executed by any stakeholder.

Input:
- Test suite specification

Output:
- Test logs
- Incident reports

Actor:
- Tester
- Developer

Actions:
- Build verification testing
- Regression testing
- Exploratory testing

Performance testing
Performance testing aims at testing the system according to the Test suite specifications in an environment mimicking real-life conditions as closely as possible. The purpose is to verify that the system conforms to its non-functional requirements.
CHAPTER 4. SOLUTION

Performance testing is based on Test cases described in section (withheld).

**Input:**
- Test suite specification

**Output:**
- Test logs
- Incident reports

**Actor:**
- Developers
- Testers

**Actions:**
- Benchmarking
- Tuning
- Lightweight code reviews to reduce complexity and improve responsiveness.

### 4.3.6 Other RUP phases

As only acceptance testing related activities belong in the testing discipline of the transition phase, no activities were outlined for this stage of development — acceptance testing is outside the scope of this thesis.

As for the maintenance phase, no activities were defined in any of the disciplines — neither by me nor Solutions Experts. This part of the development process is still a work in progress.
Chapter 5

Discussion

5.1 Evaluation

The test process

As the main part of this study was regarding compatibility between the testing tools and Sharepoint, and design of test process activities was approached as more of a secondary goal, the activities outlined in sections 4.3.3 through 4.3.6 were never meant to be officially implemented during my time at Ipendo Systems. Instead, management at Solutions Experts intend to investigate how well the proposed testing activities fit in the existing framework on their own, and implement the activities at their own discretion at a later date.

This has mainly to do with Ipendo Systems’ business and sales approach, where the customized nature of each solution makes it difficult to project how much resources can be allocated to testing in any given project. Unfortunately for this study, this makes it impossible to evaluate the solution in any other way than the feedback given as the complete testing activity package (including artefact templates) was handed in to management — the initial reaction from Solutions Experts management was that the solution seemed to sufficiently solve the issue of a lacking formalized testing approach.

Testing tools

As for the evaluation of the tools, management and developers at Solutions Experts were surprisingly satisfied with tool compatibility, and found that any issues discovered were not severe enough so as to discourage use of the tools.

As Ipendo Systems already has one combined license for TestComplete and LoadComplete — albeit for the other business area — what lies in the future may be sharing this license using a common virtual machine on the local network, enabling use from both business areas in parallel. If overall testing efforts in the
Solutions Experts business area are increased over time, management will consider purchasing more licenses in order to scale their testing capabilities with company growth.

5.2 Raising the TMMi level

Checking off criteria

Assuming that Solutions Experts implements the proposed solution without crucial changes and actually develops their software the way they specify in their methodology — such discrepancies were not the focus of this study — it could be said that they have successfully transitioned to level 2 of the TMMi Model.

Beyond TMMi Level 2

Although not explicitly outlined by the TMMi Model specification, it is inferred that it is meant to be applied at larger organisations. This is evident by for example the fact that level 3 of the TMMi Model outlines a dedicated test organization — it is doubtful if such a concept is economically feasible in companies smaller than 30-40 employees such as Ipendo Systems.

This makes it difficult for the company to transition into level 3 of the TMMi Model and beyond within the foreseeable future, as many of the concepts outlined in the model simply do not scale to smaller organizations. However, as Ipendo Systems has managed to stay in a growing state during the last couple of years — even through the dreaded financial crisis of 2008 — I have little doubt that the company will reach a point where it will make financial sense for them to continue improving upon their test processes if they so choose. When they do, the TMMi Model will serve as a solid basis for their continued testing efforts.

5.3 Dimensions of cost effectiveness

As previously concluded, automated testing is to be seen as its own discrete process — and a complement to manual testing — on the general process of verifying the software under development.

In order to calculate cost effectiveness in an automated testing effort, the alternatives have to be stated on a higher level than test case per test case basis. As the literature study already concluded that the stakes are too high not to test at all — there are many implications to releasing faulty software — the issue relies mainly on project planning and resource allocation within the given time frame, and, if stressed to isolate test automation, optimal test tool usage.
Let us for the sake of argument assume that all compatibility issues have been mitigated and that the chosen tool will be used optimally. In this scenario, careful planning based on effort estimation, risk analyses and project variables — technical as well as environmental and staff related — will serve as a basis for the decision whether to apply automated testing in the project, and to what extent. Making a rational and informed decision based on as much relevant information as possible greatly increases the chances of success in an automation effort in a project.

Because of the amount of variables involved, it is difficult to condense it all into recommendations for when to apply automated testing in a software development project and when to resort to testing only by hand — especially considering the fact that this study was mostly concerned with compatibility of tools as opposed to for example conducting a case study following a project from start to finish while gathering cost data and other metrics to base such a recommendation upon.

My raising awareness of the compatibility issues discovered will aid in achieving as optimal use as possible for project resources spent handling the tools. However, for the planning and resource allocation dimension of cost effectiveness I cannot — given the data at hand — formulate more specific recommendations or guidelines than the ones given in chapter 4.
Chapter 6

Conclusions

6.1 Future implications

Hopefully, future versions of TestComplete and LoadComplete will provide increased compatibility with Microsoft Outlook and Sharepoint 2010. SmartBear continuously releases patches and new versions of the tools, and it is quite possible that the issues raised in this thesis will be fixed in the near future.

6.2 Results and generalizability

Tool compatibility

Based on my findings, TestComplete proved sufficient compatibility with the underlying platform and its out-of-the-box components to be used as an automated functional GUI testing tool for Sharepoint. This is mainly based on the object mapping being able to find most objects, and if they can be uniquely identified and mapped, they can be interacted with and subsequently tested.

However, the generalizability of these findings to include any custom Sharepoint based application is questionable, as this matter largely depends on which third party technologies and scripting languages are applied. The compatibility with ancillary components other than the ones used in the DMS is therefore unknown.

In the cases of Microsoft Word and Outlook, the findings are rather clear. The superior object mapping in Word thanks to standard GUI controls should make most testing endeavours in the application run along smoothly. For Outlook the object mapping is erratic at best, even if we discount custom popup menus. Companies looking for an automated tool for testing their Outlook plugins should look elsewhere.
For LoadComplete, the fact that it does not tap into any Sharepoint API:s in its data gathering combined with the fact that the login issue is not caused by custom code, it can be concluded that my results are not specific for the DMS. My recommendations for using LoadComplete, although sparse, should be valid for any Sharepoint based application.

However, I must issue a rather warranted disclaimer as a result of the literature study; given all the possible problems that may or may not arise following lacking compatibility or stability in test tools in general, there is the risk of some of the issues only being reproducible on the specific machine I used during my research. The probability of this risk should however be minuscule.

**The traceability issue**

Using a requirement traceability matrix solves the issue of lacking transparency in artefact relations and inconsistencies with no more overhead than having one more artefact to keep updated. Given the extremely general nature — it is essentially a table — requirement traceability matrices should be applicable in most software developer teams’ methods of documenting and reviewing their work.

**The test process**

Given the non-specific nature of a quality framework such as the TMMi Model, applying it is a checklist onto an existing methodology should result in similar general guidelines regardless of starting conditions — especially considering the fact that I did not research any contrasting frameworks in order to investigate the TMMi Model critically. Where it will differ, however, is how the processes and activities are applied in actual projects — and this topic was outside the scope of this thesis.

Given how Solutions Experts’ testing process looked before the start of this study however, it can be said with relative significance that their testing process has been sufficiently pimped.
Bibliography


BIBLIOGRAPHY


Ipendo Systems in Linköping, Sweden, is a small IT consulting firm developing among other things custom portal applications based on Microsoft Sharepoint 2010. The purpose of this thesis is to investigate whether the test tools TestComplete and LoadComplete provide sufficient compatibility for functional and non-functional testing of Ipendo Systems’ Sharepoint based applications, as well as design testing related activities to be incorporated into their existing software development framework. A test process maturity framework was chosen and applied for the design of the testing activities, while the test tool investigation resulted in guidelines on how to apply the tools effectively and circumvent any issues discovered.