TMM-Based Process Improvement in Small Global Organization

THESIS
submitted in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE
in
SOFTWARE ENGINEERING
by
Firas Alhassany
Student number 1055348
born in Al Basra, Iraq

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

Mercator GeoSystems B.V
Ondernemersweg 54R, 1422DZ
Uithoorn, The Netherlands
www.mercatorgeo.nl
Acknowledgements

I would like to thank my colleagues at Mercatorgeo, B.V. who helped me during my internship: Roeland v.d Spek, Richard Numan and Maghash Babo. Also I would like to thank Hans Geers, my supervisor at the TU Delft. His input was invaluable for the completion of this thesis.

Finally, I would like to place much gratitude to my family, who supported me continuously during my years at the university and work. Many thanks for your advice and support. Also thanks to my wife Feryal for her patience and support. Thank you!
Software Process Improvement in Small Global Organization

Author : Firas Alhassany  
Student ID : 1055348  
Email : f.alhassany@hotmail.com

Abstract
This thesis is about improvement of software test process using the Testability Maturity Model (TMM) model in a small global software organization. TMM is an improvement model based on the CMM model, but focuses on the improvement of the test process. By applying the TMM model, the test will be assessed every time an improvement will take place. The assessments’ results are displayed in a list of process areas and practices. Improving these process areas will increase the quality of the developed software. Claims concerning Testability Maturity Model are evaluated in this thesis by executing two case studies. The evaluation focuses on the usability of the improved test process in a small global software organization with different work cultures, and the quality of the produced software. Finally, this thesis will provide some solutions to problems which are found during the case studies.

Chair: Prof. Dr. Arie van Deursen, Faculty EEMCS, TU Delft

University supervisor: Hans Geers, Faculty EEMCS, TU Delft

Company supervisor: Roeland v.d Spek, Mercatorgeo system
# Table of Content

1. Introduction................................................................................................................................................... 5
   1.1 Background and problem statements........................................................................................................... 5
   1.2 Objectives, scope and methodology ........................................................................................................... 7
   1.3 Preparing to implement the improvement model........................................................................................... 9
   1.4 Overview of chapters ................................................................................................................................. 10
2. Mercator Geosystems ..................................................................................................................................... 12
   2.1 Organization overview................................................................................................................................. 12
   2.2 The state of software development at MG .................................................................................................... 14
   2.3 Problems in MG’s development lifecycle .................................................................................................. 17
   2.4 Work culture characterization .................................................................................................................... 19
   2.5 Concluding remarks ................................................................................................................................... 20
3. Analysis of MG’s Software Development Process ............................................................................................ 22
   3.1 The Capability Maturity Model (CMM) ....................................................................................................... 22
   3.2 MG’s development process assessment ...................................................................................................... 24
   3.3 Assessment results....................................................................................................................................... 29
   3.4 Testability Maturity Model (TMM) ............................................................................................................. 30
4. Analysis and improvement of MG’s test process .............................................................................................. 33
   4.1 Evaluation phase ......................................................................................................................................... 33
   4.2 Preparation phase ....................................................................................................................................... 39
   4.3 Implementation phase ................................................................................................................................... 44
   4.4 Concluding remarks .................................................................................................................................... 52
5. Case Study ....................................................................................................................................................... 54
   5.1 Z Factuur ....................................................................................................................................................... 54
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2</td>
<td>Z Factuur system description</td>
<td>55</td>
</tr>
<tr>
<td>5.3</td>
<td>Z Factuur processes</td>
<td>56</td>
</tr>
<tr>
<td>5.4</td>
<td>Z Factuur development</td>
<td>58</td>
</tr>
<tr>
<td>6</td>
<td>NotaBox</td>
<td>62</td>
</tr>
<tr>
<td>6.1</td>
<td>NotaBox system</td>
<td>62</td>
</tr>
<tr>
<td>7</td>
<td>Z Factuur BSP project</td>
<td>66</td>
</tr>
<tr>
<td>7.1</td>
<td>Requirement analysis</td>
<td>68</td>
</tr>
<tr>
<td>7.2</td>
<td>Design phase</td>
<td>72</td>
</tr>
<tr>
<td>7.3</td>
<td>Implementation phase</td>
<td>73</td>
</tr>
<tr>
<td>7.4</td>
<td>Testing</td>
<td>77</td>
</tr>
<tr>
<td>8</td>
<td>Z Factuur Account</td>
<td>83</td>
</tr>
<tr>
<td>8.1</td>
<td>Invoice presentation</td>
<td>83</td>
</tr>
<tr>
<td>8.2</td>
<td>Accounting Integration</td>
<td>84</td>
</tr>
<tr>
<td>9</td>
<td>Conclusions and future work</td>
<td>89</td>
</tr>
</tbody>
</table>
# Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Example of software Development Process</td>
<td>6</td>
</tr>
<tr>
<td>2-1</td>
<td>MG Organization</td>
<td>13</td>
</tr>
<tr>
<td>5-1</td>
<td>Primary processes and their relationship</td>
<td>55</td>
</tr>
<tr>
<td>5-2</td>
<td>Z Factuur Primary processes and their relationship</td>
<td>56</td>
</tr>
<tr>
<td>6-1</td>
<td>NotaBox system</td>
<td>62</td>
</tr>
<tr>
<td>6-2</td>
<td>Biller Service Provider (BSP)</td>
<td>64</td>
</tr>
<tr>
<td>7-4</td>
<td>Z Factuur BSP architecture</td>
<td>72</td>
</tr>
<tr>
<td>8-2</td>
<td>New invoice presentation</td>
<td>83</td>
</tr>
<tr>
<td>8-3</td>
<td>Accounting configuration</td>
<td>84</td>
</tr>
</tbody>
</table>
List of tables

Table 1: Requirements analysis process.................................................................25
Table 2: Design Process.........................................................................................26
Table 3: Implementation Procedures.................................................................27
Table 4: Testing Procedures..................................................................................28
Table 5: Acceptance testing activities...............................................................45
Table 6: System testing activities.........................................................................46
Table 7: Integration testing activities.................................................................47
Table 8: Unit testing activities.............................................................................48
Table 9: Reproduction of reported faults...........................................................49
Table 10: Maintenance unit testing activities.....................................................50
Table 11: Maintenance integrating testing.........................................................50
Table 12: Maintenance system testing procedure............................................51
Table 13: Fault reporting process......................................................................52
Table 14: Z Factuur Development team.............................................................66
Table 15: Z Factuur Development plan..............................................................67
Table 16: Subscription request test cases.........................................................71
Table 17: Summary of the test results..............................................................89
1. Introduction

1.1 Background and problem statements

In order to compete successfully in the software market, software developers need to improve both the quality of software and the ability to work within time and budget. The capacity to achieve improvements is largely dependent on the software development process and technology [1]. Technology and talented people are important elements of the software development, that have to be supported by a good development process. Problems with the process, rather than the technology, are often the cause of quality problems that arise during software development.

In this thesis we will describe an approach to quality problems within the software organization Mercator Geosystems. Before describing this problem, we will define the relevant terminology.

1.1.1 Software development process

A software development process is a set of activities that is performed toward a specific purpose [3]. An activity is a task or group of sub-activities with a specific purpose assigned to a team. Parts of the software development process are sub-processes like implementation and testing (figure 1-1).
To manage the software development process, process models were introduced. They specify which techniques and managerial steps are applied during the software development process [5]. Examples are the Waterfall, RUP, and the Agile models.

Whether an organization applies a specific process model or not, the quality of the produced software does not depend on that model alone but also on how well and maturely the process is applied. To evaluate the quality of the software process and sub-processes, quality improvement models are introduced. These models enable an organization to assess the software development process and detect weak areas. They lay out a road map for an improvement to enable the organization to produce high quality products [4].

In 1987 the ISO9000 was published. It was designed to help organizations ensure they meet the needs of customers and other stakeholders. In 1991 the Capability Maturity Model (CMM) [5] was introduced. This model focuses on the improvement of software development processes within organizations. A new version of CMM, the Capability Maturity Model Integrated (CMMI), was presented in 2002. The new model expanded the scope of CMM from the isolated software development process to encompass all of the processes within an enterprise [6].
Models to support the evaluation and improvement of specific sub-processes were introduced later. Examples of such models are the Testing Maturity Model (TMM) [7] for testing and the Requirement Maturity Model (RMM) for improvement of the requirement analysis process [8].

1.1.2 Problem statement

Mercator Geosystems B.V. (MG) is a small software house, established in 1997. It uses the Waterfall model for the management of the software development process. The development process is executed by teams at two locations: in the Netherlands and in India.

Despite the fact that the company applies modern technology and has talented people, MG produced faulty software detected by customers after the software release. This threatened the company’s reputation and increased the risk of losing customers. As a step to resolve this situation, MG’s management requested an investigation of the quality of the development process. MG further specified that CMM should be used as the improvement model.

1.2 Objectives, scope and methodology

The objective of this thesis is to report on the investigation and its resulting impact on the software development process at MG. CMM, and models derived from CMM, were applied during the investigation of the software development process. The investigation had two phases: a literature study about software development process improvement models, and a project to apply the theoretical work and to assess the results. The literature study resulted in a document with an overview of CMM and TMM that is based on CMM. It focused mainly on the test sub-process.
The study discussed the structure of the models and how the models can be applied to assess and improve the development software process.

Based on the literature study, an improvement strategy was developed to improve the software development process of Mercator Geosystems. The strategy divided the improvement activities into two steps. During the first step, CMM was used to assess the software process. This resulted in a list of the weak activities. The weak activities were classified in groups, related to the sub-process to which they belong. Each group would be improved separately using a model derived from CMM and tailored to that sub-process. This step would be repeated until all of the sub-processes had been improved.

The practical project, following the literature study, involved the application of the improvement strategy and investigation of the results of the improvement. To begin with, an assessment of the development process currently in use at MG was performed, resulting in a list of the weak areas. From this list the testing process was selected. TMM was applied to describe and improve the test sub-process.

In the next step, the improved test process was applied to various projects. The results of the experiments and the issues that arose are described in this document.

At the end of this document we will answer the following questions:

1. Can the combination of the CMM/TMM models be applied effectively in a small global software organization to enhance the software quality?

2. What influence do organizational culture, team size and team distribution have on the process improvement?
1.3 Preparing to implement the improvement model

The assessment and improvement activities we performed during the investigation were carried out in four phases.

**Assessment phase:** In this phase, the MG development process was assessed using the CMM model. In a period of two weeks (March 2008), we performed the following activities:

1. Collection of data on the MG software development processes, sub-processes and teams by interviewing various staff members in India and the Netherlands.
2. Analysis of software projects that were developed in the past using the MG software development.
3. Collection of data on how the work was performed by the teams at MG (India) and MG (Netherlands), and analysis of the impact of these methods on the improvement process.

**Preparation phase:** Four weeks (March-April 2008) were spent on making preparations for implementation of the improved process. The preparation fulfilled the process improvement goals defined by the TMM model and included the following activities:

1. Preparation of presentations on the goal of process improvement and the improvement models, which would be given to various MG teams.
2. Preparation of techniques required to carry out the creation of test cases and establishment of a test planning process.
3. Preparation of templates and documents required to support the new test process as specified in TMM.
Introduction phase: Three weeks (April 2008) were spent introducing the improved test process to the MG teams. The introduction included instruction on the process procedures, activities, techniques, tools, and the templates that support the application of the process during software development or maintenance.

Experimental phase: Over the course of eight months (May 2008 - January 2009), the improved test process was applied to several software development and maintenance projects. From these projects, two study cases were selected. For these cases, we monitored the process activities throughout the software development processes.

1.4 Overview of chapters

The remaining chapters in this thesis are organized as follows. Chapter 2 provides background on the software process in use at Mercator Geosystems, the organization where the literature study and practical project work were carried out. In Chapter 3 we will assess the MG software development process and discuss the assessment results. Chapter 4 describes the steps taken to prepare and improve the MG test process. Chapter’s 5 to 8 report on the experience of applying the improved process to two projects and include the evaluation of the improved process. Finally, in Chapter 9, conclusions are drawn and suggestions are made concerning how this work can be followed up.
2. Mercator Geosystems

In this chapter we will provide background information on the software company, Mercator Geosystems. The chapter begins with a short introduction of Mercator and its teams. In Section 2.2, we will describe MG’s software development process before the described improvements were implemented. Section 2.3 discusses problems we have identified in the original process. In the last section, we will discuss specific work culture attitudes that characterize the work at Mercator.

2.1 Organization overview

Mercator Geosystems [7] is a small software company that was established in 1997. It is located in the Netherlands in Uithoorn near Amsterdam. Due to the shortage of professional software developers in the Netherlands and too high labour costs, the company opened a new branch in India (Trivandrum) in 2001, making MG a global software company [8].

At present, MG employs 16 staff members with an IT education. 90% of the staff has at least three years experience in software development. The staff is engaged in the development, maintenance and testing of web-based applications and web services for customers in the financial and agricultural sector in the Netherlands. MG’s staff comprises three teams distributed across the two branches, as shown in figure 2-1.
Figure 2-1: MG Organization

Management team
MG’s management team consists of two members Figure 2-1. The first member is the company director. He is responsible for the acquisition of new projects, assignment of projects to the development team, and general management activities.

The second member is the Chief Technical Officer (CTO) at MG-India, who is responsible for the management of the projects carried out in India. He is also responsible for the deployment and supervision of the Indian staff members.

The two managers communicate daily via IP telephone. Email is used to exchange documents related to the projects. The director visits MG-India four to six times a year to be informed about the latest developments.

Development team
MG’s development team is divided over the Netherlands (two developers) and India (ten developers), and is supervised by the management team. The development team is responsible for software design, implementation, and preparation of the software for release. Tools that are used include various programming and scripting languages (Java and C#), development environment tools (C# Visual Studio and Eclipse [8]), and web-service and web-based components (like Xfire, Struts, and Spring).
**Test team**
MG has a small test team (two testers) in India, managed by the CTO. The activities of the testers are complementary to the developer’s activities. The testers are responsible for creating the test cases, testing the implemented code and reporting detected faults. They also conduct tests to identify faults in software that has already been delivered.

These tasks are performed in a basic test environment consisting of two test servers on which they install the source code. A single regression test tool called Badboy [12] is used. Faults detected during the development and tests are reported in a management tool called AceProject [10]. This tool is accessible by all members of MG’s teams. It is used extensively to assign development and repair tasks to the developers/testers. It is also used for the storage of project-specific documents in corresponding folders.

2.2 **The state of software development at MG**

During the development lifecycle, MG teams employ the Waterfall model as software development process. This process consists of five sub-processes, described below.

**Requirement Analysis Process (RAP)**

This process starts once an agreement is reached between a customer and MG. This process is managed by MG’s director, who is responsible for collecting information about the requested software and for establishing the requirements.

The requirement analysis process begins with the identification of the users of the requested system. Functional and non-functional requirements are formulated for each user. Requirements that pertain to all users are also collected. For complex components, screen shots are made and presented to the customer.
Once all the main requirements have been gathered, a requirements document is written, which is used in the next step of the development process. The quality and form of the requirements document can vary. In most cases the document is written at a high level using engineering terms. However, we have found many cases in MG’s history in which the projects were assigned without thorough analysis. In these cases the development team received a description of the customer’s expectations, some draft documents and emails from the customer.

**Design Process (DP)**

In the design process, a list of design goals for the requested software is drafted by the management team and one or two experienced developers. The draft describes the software architecture by a specification of the sub-systems in terms of responsibilities and by the dependencies of the sub-systems. After that, the design goals are translated into a draft design. In the design the system is divided into layers. Each layer performs a specific functionality in the requested software. Examples of such layers are the database layer and the business logic layer.

**Implementation Process (IP)**

MG aims to implement all requested software in India. Sometimes, factors like the vicinity of the customers, availability of resources, or experience of personnel with similar projects lead the management team to decide to implement a project in the Netherlands. During the implementation process, developers in both branches perform almost the same activities. These are: interpretation of the software requirements, and draft design based on the previous sub-processes. After that, the developers transform the requirements and design into source code using the development

---

1 Unless otherwise specified, the term ‘layer’ is used throughout this document to refer to a package of classes designed to serve a common purpose (e.g. database layer, user interface, business logic).
infrastructure. The implementation sub-process is finished when the software source code is ready for release.

**Test Process (TP)**
MG’s test process is only applied to software developed in India. The process is performed after completion of the implementation process. The test team creates test cases in an ad-hoc way to cover the “happy flow” scenario for the developed features. The test cases are applied to the developed application. If the test team detects faults, the faults are reported to the developers via the management tool. Once the faults have been corrected by the developers, the test team re-tests the software and confirms when the faults have been eliminated.

**Maintenance Process (MP)**
The maintenance process is applied when a customer reports a fault in the delivered software. MG has a special arrangement with its customers through which the company guarantees that serious faults are resolved within a short time period (usually 24 hours). Other faults will be corrected within a timeframe established in consultation with the customer.

The process begins with reproduction and classification of the reported faults by the test team. The customer is informed when the reported faults have been reproduced. After that, the development team is assigned to make the necessary corrections. Subsequently, the test team re-tests the software to confirm that the faults have been eliminated before the software is released again.
2.3 Problems in MG’s development lifecycle

During the process of extracting information about MG’s development process by conducting interviews with various staff members, we identified a number of problems in MG’s development process. We have classified these problems as either common to all sub-processes or related to a specific sub-process. We will discuss the common problems first.

1. In general, MG’s development process suffered from a lack of management. During software development, there was no project plan and tasks were frequently assigned or performed in an ad hoc manner. Frequently, developers were interrupted by assignments with a higher priority.

2. Lack of documentation: During the search in the archives for information on previously developed projects, we did not find documents such as project plans, software designs, or updated requirement documents.

3. Inefficient use of team skills: We noticed that MG makes a minimal use of developers’ capabilities to perform management tasks, requirement analysis, and design. Developers only performed programming activities.

4. Inefficient communication: During the process of software development, communication among MG teams was weak and depends entirely upon the two managers. Direct communication between developers in the two branches or between the test team and the customers occurred rarely.

In addition to the problems in the software development process, we found other problems that occurred during the requirement analysis and the test process.
The requirement analysis process
1. There were no activities that validated the requirements for consistency and completeness. A negotiation process with different stakeholders to achieve an acceptance level was also absent. The lack of these activities made the requirements for the requested software unclear to the developers and testers. It prevented them from performing the development tasks correctly.

2. Because the requirement analysis process depended strongly on just one actor (MG’s director), the resulting documents often suffered from a lack of details and are not updated regularly.

Problems in the test process
1. We observed that inadequate attention was given to the test process by both the management and the development team. In many cases, because the test process began too late and a great deal of time was required to test all the features, the test team was requested to test only important or risky scenarios before the software is released to the customer.

2. Lack of different test types and techniques: MG applied only one type of test “system testing”. This was performed during the last stage of development. Other kinds of tests, such as unit or integrated tests were not conducted. We have also noticed that the creation of test cases was not based on a specific approach, like the black box or the white box test. Test cases were created in a ‘happy flow’ manner, simply to validate whether or not the implemented feature was working.

3. We also noticed that the test team performs its work using only a basic test infrastructure. In the case of web-based applications, test cases were executed directly, using Internet browsers. For applications without a GUI interface, the test
work was shifted to the developers who debug the source code.

4. Absence of formal test reporting: In the current software process, faults were reported directly to the developers without documenting them in the management information system. This also applied to the release of (maintained) software: The test team did not deliver any test report to the customer.

2.4 Work culture characterization

During the analysis of the software process, we encountered specific work attitudes that influence the work process at MG. We believe these work attitudes have affected the improvement activities we aimed to introduce. Therefore it is important to describe them before discussing the improvement work.

We made the following observations with regard to the work attitudes:

1. We noticed that the team members in India preferred not to perform any documentation tasks. Tasks such as writing project plans, user manuals and requirement documents (if requested) were performed by the team in Uithoorn.

2. The company was structured such that the development team occupied a higher position in the organization than the test team. The implementation activities were considered to be the essential work, while other activities, such as testing and system configuration, were considered to be supportive.

3. During discussions with the MG management team, we noticed that the team in India demonstrated a particular group behaviour in which any attempts to change the current work procedures by extending the environment or adding new activities to the development process were resisted. MG also experienced this in the past when they tried to include activities
such as code review or writing of test reports and user manuals with the software release; all of these attempts have failed.

4. During the software development lifecycle, the MG teams in India preferred to avoid all direct communication with the customers and even with the management and development teams in Uithoorn. This kind of communication occurred via the CTO.

5. We have also noticed that the team in India made a strict distinction between development and test activities. Developers are only requested to perform programming related activities; they are not asked to perform test activities.

2.5 Concluding remarks

The state of software development at MG currently required improvement, both in terms of increasing the quality of the produced software as well as increasing the effectiveness of the process by achieving optimal results with a given effort. In the next chapter we will introduce the CMM improvement model. This model will be applied to systematically assess MG’s current software process. Further improvements will be recommended based on the results of the assessment.
3. Analysis of MG's Software Development Process

In this chapter, as a first step towards improvement, we will describe the assessment of the MG’s software development process using the Capability Maturity Model. The CMM is introduced in section 3.1, along with a discussion of its levels and structure. We will show how it can be applied to improve the development process within a software organization. In section 3.2 we apply the CMM model to assess MG’s software development process. This assessment will identify MG’s current maturity level. In section 3.3 we present the response of the management team to the results of the assessment. The obstacles to improvement of the development process will also be discussed in this section. We then present a strategy that divides the improvement of the software development process into sub-process improvements based on CMM-type models. To improve the test sub-process the Testing Maturity Model (TMM) is applied.

This model is presented in section 3.4, along with an explanation of how we will employ the TMM to improve MG’s test process.

3.1 The Capability Maturity Model (CMM)

In 1987, Watts Humphrey presented a model to measure the software development processes in terms of maturity of the process and the quality of the product, the Capability Maturity Model (CMM) [4]. CMM is used to assess the maturity of the software processes in an organization, and to identify key practices that will improve the maturity of the process. The model divides the software process maturity of an organization into five levels, ranging from ad hoc immature software processes to optimal mature processes [3].
Except for the first level, each level includes a number of attributes or Key Process Areas (KPA) [4]. These identify the issues that must be addressed to achieve a certain level of maturity. The KPAs are described in terms of key practices that contribute to attainments of the goal of a KPA. The CMM five maturity levels are: [3,4]

**Level 1.** Initial: at this level the software process is characterized as ad hoc, and occasionally even chaotic, only few processes are defined. The level has little or no common practices. The SEI has estimated in 1997 that 76 percent of all IS departments were at Level 1 [3].

**Level 2.** Repeatable: implies that basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications and conversely not to repeat failures. Key process areas in level 2 are the requirements management, software project planning, project tracking and oversight, software subcontract management, software quality assurance and software configuration management.

**Level 3.** Defined: is attained when the software process for both management and engineering activities is documented, standardized, and integrated into a standard software process for the organization. All projects use an approved, tailored version of the organization’s standard process for developing and maintaining software.

**Level 4.** Managed: is defined when a collection of detailed measures of the software, the software process, and the product quality are applied. Both the software process and products are quantitatively understood and controlled.
Level 5. Optimizing: is reached when a continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies. The key process areas at this level are focusing on defect prevention, technology, and process-change management.

For the improvement of MG's development process, the CMM model provides the organization with a guide to determine which areas of the development process should be targeted through an assessment process. It describes the KPA's that should be included in the process to achieve a specific maturity level. Based on this assessment process, one can trace where, in the software development process, the improvement will be incorporated.

3.2 MG's development process assessment

In this section, we will assess MG’s maturity level by describing the process areas defined by CMM model. At first we will express MG’s development process in the form of sub-process, activities and tasks. The content of the description was collected by interviewing MG's various team members. After that, we will map the process attributes on the KPA's defined by CMM to identify the weaknesses and strengths of MG’s current software development process. The assessment activity is finished by presenting a conclusion about the process maturity level.

Requirements analysis

The first stage in software development is the requirements analysis process. According to MG’s manager, the process is aimed to collect the user and system requirements for the requested software. The process is executed in the Netherlands and performed by MG's manager as described in table 1.
Table 1: Requirements analysis process

<table>
<thead>
<tr>
<th>Input</th>
<th>The requirements process gets the customer agreement as input to start execution one of the following activities and tasks.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity</td>
<td>Based on the form and the type of the received request, one of the next two practices is applied. The first practice includes the following activities:</td>
</tr>
<tr>
<td>task</td>
<td>MG manager will interview the customer and collects the information needed to develop the software.</td>
</tr>
<tr>
<td>task</td>
<td>The manager summarizes the collected information in a formal document that contains the customer requirements represented in use cases and explanations.</td>
</tr>
<tr>
<td>task</td>
<td>The created requirements document is sent to the customer.</td>
</tr>
<tr>
<td>task</td>
<td>If there is any disparity between the customer's expectations and the document contents, the document is sent back to MG for correction.</td>
</tr>
<tr>
<td>Activity</td>
<td>In this case, the customer provides MG with the requirements which can be classified into four types: - (1) A formal document with an adequate description of the required software, it includes the use cases and the customer’s expectations. (2) A simple document containing the software requirements. (3) A requirement document written in complex technical language. (4) Requirements submitted through an oral conversion.</td>
</tr>
<tr>
<td>task</td>
<td>MG's manager assigns the project to a development team in India or the Netherlands and presents the requirements through emails, IP phone Skype, or by direct presentation.</td>
</tr>
<tr>
<td>Output</td>
<td>The requirement analysis phase is terminated by complete understanding of the software and user requirements.</td>
</tr>
</tbody>
</table>
Design
Second phase in the development process is the design. This process intends to produce a high level design for the requested software. It is performed either by one experienced developer supervised by MG manager, or by the manager himself. The process is basic and contains only one activity with 3 tasks resulting in a draft software design as described in the table 2.

<table>
<thead>
<tr>
<th>Input</th>
<th>As input the process received the requirements document of the previous process.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity</td>
<td>The development team uses the software requirement as input and starts the designing through performing the following activities:</td>
</tr>
<tr>
<td>Task</td>
<td>One or more experienced developers create a high level software design. The design presents the software in layers i.e. database layer, business logic layer, and interface.</td>
</tr>
<tr>
<td>Task</td>
<td>The developers discuss the design with the MG manager. Eventually an adaptation of the design takes place.</td>
</tr>
<tr>
<td>Task</td>
<td>The management team divides the tasks per layer for each member in the development team and discusses individually the used techniques and the time needed to achieve these tasks.</td>
</tr>
<tr>
<td>Output</td>
<td>Draft software design in terms of layers and functionality.</td>
</tr>
</tbody>
</table>

Table 2: Design Process

Implementation
Implementation is an essential process in MG development life cycle where teams spend most effort and time. The process is aimed to translate the high level design, and the requirements document into program code. The process contains of one activity with 5 tasks, as described in table 3.
As input the process received the software requirements and the draft design.

**Activity**
The implementation process consists of 5 tasks carried out by the development team.

**Task**
The developers prepare the infrastructure needed for the implementation. This infrastructure includes configuration of the development framework (Eclipse, Visual Studio), database server (SQL server), and availability of the necessary literature as tutorials, lectures and eventually courses.

**Task**
Based on the task division in the design phase, the team divides the tasks related to class/module levels.

**Task**
The developers implement the tasks by converting the requirements into program code.

**Task**
The team saves and merges the implemented code into MG backup server (CVS) periodically. This activity is repeated until the closure of all tasks. During this activity, problems that appear are solved by the developers.

**Task**
The team runs the first version of the software and interacts with the system. When incidents occur, the developers will fix them. Then, the team delivers the software to the test team.

**Output**
Software source code.

**Table 3: Implementation Procedures**
Testing
The last phase in MG’s development life cycle is the test process. It will check whether the developed software satisfies the customer's requirements. Testing is performed and applied for GUI-based software. Other kinds of software are not tested, but debugged by the developers before release. In the following table, we have described the test process procedures and activities.

<table>
<thead>
<tr>
<th>Sub-process</th>
<th>Activity</th>
<th>Task</th>
<th>Task</th>
<th>Task</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>The developed software and the requirements.</td>
<td>The local manager in India assigns one or more testers to test the developed software.</td>
<td>The development team installs the software on a local machine or a server.</td>
<td>Developers explain the developed features to test team. Based on this, the testers create test cases.</td>
<td>During the software verification, the test team reports the detected incidents to the development team for fixing.</td>
</tr>
<tr>
<td>Output</td>
<td>Tested software.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3.3 Assessment results

From the process analysis described in the previous section, we have noticed that MG’s development process matched rarely with the KPA's defined by CMM levels. The process is suffering from a lack of many essential activities like planning, documentation, and management. Tasks were performed in an ad hoc way and depended strongly on the presence of the managers. Therefore MG development process can be rated as a process at the first level of CMM.

The CMM model offered a good opportunity to improve MG’s development process. But improving MG’s development process to a higher level in CMM was not trivial; the application of CMM required organization-wide changes in the process, culture, and structure of the organization. The management of company was aware that its team and capacities were small and could only change slowly. Making big changes in the improvement process could cause resistance and confusion. MG was concerned that an attempt to make a comprehensive change might lead to failure, as had happened in the past. Therefore we have proposed to take small steps to improve the sub-processes instead of the whole development process. Models used to improve these sub-processes should to be compatible with the CMM and have the same structure. Examples are the DMM [10] to enhance the design process or the RMM [13] to improve the requirement analysis process.

This proposed solution was accepted by MG’s management team. The management agreed with the improvement strategy and has decided to improve the test sub-process as a first priority. The objectives were to improve the software quality by reducing the amount of faults detected by the customers and to reduce the time
spent in faults’ reproduction and fixing. For MG, the test process and the test team were the weakest part of the development process and cause the most complaints. The proposed model to improve the test process is the Testing Maturity model (TMM) [2].

This model is based on CMM and focuses only on the test process. In the next section, we will give an introduction to the TMM model, its levels, and the steps that were be applied for improving the test process to a higher maturity level.

### 3.4 Testability Maturity Model (TMM)

TMM is a new improvement model focused only on the test process [5]. The framework is developed by the Illinois Institute of Technology as a guideline for test process improvement. It is positioned as a complementary model to CMM.

Like the CMM, TMM has an assessment procedure that allows an organization to evaluate the current state and to identify the process’ strengths and weaknesses. TMM has the same level structure as the CMM. It divides the maturity of the test process of an organization into five levels and highlights the relationships between the test process, procedures and activities on one side and the key players in an organization, like the managers, testers, and customers, on the other.

TMM first level is *Initial*. At this level testing is a chaotic, undefined process and is considered as part of debugging. It includes no maturity goals.

At level 2, *Definition*, testing is separated from debugging and defined as a phase that follows coding. Level 2 is a planned activity. Test planning however may occur after coding for reasons related to the immaturity of the test process.

The third level is *Integration*. Testing at this level is no longer a
phase that follows coding but becomes integrated in the software development cycle.

At the fourth level, Management and Measurement, testing is a thoroughly defined, well-founded, and measurable process.

The last maturity level in TMM is Optimization where testing becomes a completely defined process and one that is capable of controlling the costs and the testing effectiveness.

Each of the above described levels with the exception of the Initial has a set of maturity goals. They identify testing improvement goals that must be addressed in order to achieve maturity at that level. These maturity goals can be adapted by implementing the Activities, Tasks, and Responsibilities (ATR) at each TMM level.

Based on the above description, our goal in this thesis work became to improve MG low level test process to the second level in TMM. To achieve that, we followed the steps as defined by W. E. Deming [16] for the process improvement. At first, we assessed the status of the test process. After that, we implemented the improvement goals we wanted to achieve. For Level 2 we implement the following two goals:

1. Institutionalize basic testing techniques and methods.
2. Initialize a testing planning process

In the next chapter, we will describe the implementation these two goals. We will begin with the development and test staff with the knowledge by providing the techniques, tools, and training to execute the new process. Then a new test process was introduced to the teams and executed using a case study.
4. Analysis and improvement of MG’s test process

In this chapter we will explain the activities we have performed to improve MG’s current test process using the TMM model. The chapter is organized into three main sections. In the first section we will describe the steps we have applied to assess MG’s test process. The purpose of this assessment was to determine the maturity level of MG’s test process. This activity was performed in India and took about one week. In Section 4.2 we will describe the sessions and presentations we held in India to implement the TMM maturity goals required to improve the test process. In section 4.3 we will describe the improved test process and how it can be applied during the software development lifecycle.

4.1 Evaluation phase

During this phase we evaluated MG’s test process. The purpose of this evaluation was to determine the maturity level of MG’s test process and to identify the maturity goals needed to achieve improvement. For this purpose we used the maturity goals and the ART (Activities, Responsibilities and Tasks) defined in the TMM model to evaluate the test process together with the data collected from staff interviews and the inspection of documents.

The scope of the evaluation not only included MG’s test team and process but also aspects of the development and management processes and teams that influenced on the test process. For each team, we have reported a number of observations about the process and the team itself. These observations illustrated the shortcomings of the process related to the maturity goals defined using the TMM model.
Questions were asked and information was gathered about the processes during direct interviews. We also inspected archived documents stored in Ace Project, MG’s management system. Upon completion of the evaluation phase, we reached a conclusion about the maturity level of MG’s test process and established the maturity goals that are needed to achieve improvement.

4.1.1 Evaluation of the management team and process

The first process and team we evaluated was the management. We interviewed the management team and compared the activities they performed with the ARTs defined in the TMM model, and we extracted information about the testing activities applied by the management team. The evaluation of the management team and process resulted in two separate sets of observations, presented below.

Observations concerning the management process

1. There was no improvement plan for the test team and test process. The management team did not have any plan to improve the skills and knowledge of the test team, to expand the team to include other testers, or to add new tools to the test infrastructure.

2. There was no documented classification of faults. Faults were basically assigned to one of three categories: low, middle or high. This classification was based on the priority with respect to the time to delivery and not on the degree of severity of the fault.

3. There was no documented definition of the test process and the test team.
4. There were no test policies that address issues such as the conditions for starting and terminating the test process, the acceptable performance per function or the minimum time spent on testing during software development.

**Observations concerning the management team**

1. The test process had a low priority role in the manager’s daily activities. The manager did not include the testing in his daily plan for project management. Test activities were assigned in an ad hoc manner.

2. There was no knowledge about testing as an engineering process or the importance of this process to the development process.

3. There were no scheduled meetings during which the manager reviewed the work of the test team and discussed the reports on a project’s test status, test plans, and the identified issues.

**4.1.2 Evaluation of the development team and process**

The second process we evaluated was the development process. We conducted this evaluation to obtain information about the test activities applied by the development team during the software development lifecycle. Because asking questions to all the team members would have taken too much time, we selected just three developers and interviewed them individually. The three selected developers were the most experienced members in the team. The process evaluation resulted in two sets of observations, presented below.
Observations concerning the development process

1. MG’s (current) development process did not apply any test activities during development. The process did not include activities to test the software design or implementation. In some cases, the developers validated the functionality of components or classes during debugging.

2. The development infrastructure did not contain any tools to support the test activities.

Observations concerning the development team

1. During the evaluation, it was noted that there was a strict separation between the development and testing activities. This was part of the workplace culture at MG-India, where developers only worked on implementation activities. The developers at MG believed that testing is an activity that should only be performed by the test team.

2. The team did not have any knowledge of the advantages of testing or how to perform tests during the development phases.

3. There was no testing policy that compelled the developers to deliver tested code.

4.1.3 Evaluation of the test team and process

The last process we evaluated was the test process. This evaluation was conducted by holding direct sessions with the test team and covered areas like the testers’ experience and skills, educational background, and knowledge of programming techniques. The evaluation resulted in a number of observations about the test team and the process, summarized separately below.
Observations concerning the test process

1. The test process was not associated with all levels in the development processes. It was performed as the last activity and in an ad hoc manner before delivery of the software. The process was not applied during other phases, such as the requirement analysis to check the overlap and validity of the use cases or the implementation phase to create unit and integration tests.

2. There were no tools to support the test process; particularly conspicuous was the absence of tools suited to the software produced by MG, for testing web services or web-based applications for example. The infrastructure contained just one test tool, Badboy, which was used to test HTML pages. Another tool, Ace Project, was used to document reported faults for management purposes.

3. The test process included only one level of testing, namely system test [9]. Other levels, such as performance or acceptance tests by end users, were not applied.

4. There was no documentation or schedule, and the test process included no templates for the test plan, the test cases or for reporting faults.

Observations concerning the test team

1. There were no test case creation techniques. The testers did not have a specific method for the creation of test cases. The test cases were created at random for the purpose of checking whether or not a specific function worked.

2. Neither of the testers had programming skills in languages like Java or C#. This resulted in the delivery of software without any testing of the internal structure.
3. During the discussion with the testers, it was our impression that the testers had a secondary role in the current MG software development process. Testers were not treated in the same manner as developers, and they were often involved in tasks unrelated to testing, such as software installation for the other team members or network configuration.

4.1.4 Conclusion

We have examined the MG test process and the techniques applied within the company to develop and maintain software. During the evaluation phase, we studied the test process and processes that testing is dependent on, and a number of issues were noted. These issues identify a number of shortcomings in the test process. They showed that MG’s test process was very simple and was characterized as (initial) immature.

MG’s test process was performed in an ad hoc manner and was in many cases ignored by the management and development teams. Furthermore, the tasks, activities, and responsibilities, of the various teams rarely matched the maturity goals defined in the TMM model.

To elevate this process to the second level in the TMM model, the maturity goals defined for this level should be implemented. To achieve this, it was necessary to prepare MG’s teams and to build the infrastructure to support a more structured process. We will describe the steps we have taken to implement TMM level 2 maturity goals and to prepare MG’s teams and infrastructure for the new test process based on these goals.
4.2 Preparation phase

In this section we describe the implementation of the maturity goals defined by the second level of TMM, namely institutionalize of basic testing techniques and the initialization of the testing planning process. The implementation of these two maturity goals was achieved by providing MG teams with various training sessions and presentations. They were aimed at enabling the teams to perform the test process activities defined by the maturity goals and overcoming the lack of knowledge pertaining to test techniques, test methods and tools, project management and planning.

4.2.1 Testing and the improvement model

During the first session, we conducted a general presentation for all the MG team members in India. The presentation was aimed at illustrating that testing is an engineering process and at presenting the goal of process improvement. During the first part of the presentation we discussed the following subjects – including the definition of testing, the role of the test process in the software development process, test levels (unit, integration, system), and test types (security, performance, acceptance) [9] – associated with the development phase, the role of the testers, developers, managers and customers in the testing process, and the approaches used to create test cases.

The second part of the presentation was focused on software process improvement models, especially CMM and TMM. During this part we discussed points like what an improvement model is and why it is used, and the history of CMM and TMM models. This included the scope of the improvement and the effect on the test
process and the development life cycle. It also included the expected impact the new test process would have on software quality in terms of reduction of the number of faults found by the client. At the end of the presentation we explained that cooperation with the improvement team (MG Director van de Spek and MSc candidate Alhassany) and commitment to the new process activities were important factors for achieving successful implementation of the process improvements.

4.2.2 Organizational management

The second group of presentations during the preparation phase were held in two sessions. They focused on the management problems identified during the evaluation phase that needed correction to achieve the maturity goal of test planning.

Recommendations to the management team

1. In order to be sure that the new test process would receive the required support, a number of recommendations were presented to the management team.

2. It is the managers’ responsibility to ensure that the test team’s working methods would be based on the specified test process and to prevent any attempts to get around the specified process.

3. The managers should seek to motivate the testers by involving them in the early stages of development and keep them updated on the project’s development.

4. Communication with customers would give testers a better understanding of the customers’ expectations. Therefore the managers should encourage the test team to communicate with the customers.

5. Due to the time consuming nature of the test process, the managers should provide the test team with the time required to
perform all the activities described in the test plan.

6. Furthermore, it is the responsibility of the managers to ensure that all testing activities were included in the project plan. The time and resources required to perform them should be assigned along with a schedule of meetings with the test team to discuss issues like the test results, test plan, reported faults, etc.

**Test planning**

Planning is an essential process area in level 2 of TMM. Therefore, we introduced the MG teams to the concept of holding planning sessions. During the introduction, we discussed points like the advantages of a test plan, the essential items to include in the plan, its scope, the start and finish date of testing, test plan hierarchy, and the plan’s components. All of these components were based on a standard test plan template provided by the IEEE Computer Society [15]. The session was followed by a small workshop to create a test plan template. The template should be used to plan testing activities in the new test process.

**Test techniques and tools**

During the last sessions, we introduced the test and development teams to the test techniques and tools. These were aimed at implementation of the level 2 maturity goals for initialization of new test process techniques and introduction of the techniques required to create the test cases. Various testing tools were presented during the second session.

**Test case creation techniques**

During this session we presented two test case creation techniques. These techniques were intended to enable the testers/developers to develop efficient test cases that can detect the faults at any level of
testing. The presentation was divided into two sessions. During the first session we introduced three types of test case creation techniques based on the black box approach [16]: random testing, boundary value analysis (BVA) and equivalence class partitioning testing [2]. These three types were executed on previously implemented software.

The test team created a number of test cases based on the presented types, and then we reviewed and discussed the test cases and carried them out on the software to be tested.

The second session covered the white box approach [2]. We presented the approach to one of the developers and then worked together to implement a unit test class in one of the projects currently under development. A number of test cases were created in this unit test class to validate the methods in the selected class and to cover the code structure.

The execution resulted in the detection of 11 faults. Subsequently, in a separate session, we introduced the process of unit test class and test case creation to the rest of the development team.

**Test tools**

In order to make test work easier and more efficient, we introduced MG’s teams to a number of tools. The purpose of presenting these tools was to enable the team to perform the test work more easily
and to help the testers/developers perform other levels of testing.

**JUnit Framework**

The first tool we presented to the development team was JUnit [17], a plug-in framework added to the Java development environment Eclipse. It should be used by the developers to create a unit test class for each testable unit in the application. During the presentation of this tool, we discussed the following points: the creation of test classes for each testable unit in the software under the test, execution of the created test cases per unit test class, and the execution of one or more related test classes to perform an integration test.

**SoapUI**

The second application presented to the development and test team was soapUI [18], a tool that could be used for testing web services without a GUI. It enables testers to test web services directly, without the need to implement a test client or an interface that interacts with the service. The presentation included topics such as how to create a connection with the web service using the Web Service Description Language (WSDL) [19] and the creation of test cases per service call running the created test cases and register the detected faults.

**PMD**

The third tool we presented was PMD [20], an Eclipse [8] plug-in tool used to detect try/catches or dead code. The tool is able to detect overcomplicated expressions, such as unnecessary statements or methods implemented in a complex manner. The most important function of this tool, however, is to detect duplicated code. The presentation covered the installation of the tool on the development machines, the application of the PMD tool to one of the running
projects, and review of the detected faults, warnings and duplicate code. The reported issues were then discussed with the team.

*Ace Project*

The last tool presented to the teams was Ace Project [21]. The team already used the application on a regular basis, but we redefined a number of functions to accommodate tasks for the new test process. These included functions such as how test-related documents are saved and updated within a project, how faults and defects are reported for each project, how tasks are assigned to the development team members, how deadlines are assigned to the tasks and activities, and how to determine the time expected to be required to complete tasks and the actual time spent per task.

### 4.3 Implementation phase

In the last section we discuss the new test process presented to MG’s teams, which was designed to match the maturity goals defined for the second level of the TMM model. The new test process was defined as a sequence of sub-processes. Each subprocess is presented as activities and tasks required to complete the tasks and fulfil the responsibilities of the actors performing these sub-processes. The test process is designed for application in new software development projects as well as for software maintenance. In the next section we will present the new test process as well as another process for reporting faults detected during the software test.
4.3.1 Test Procedure for software development

This procedure is applied when MG is developing new software. It is divided into four levels, namely Acceptance, System, Integration, and Unit. Each level includes practices and activities detailed in the following sections.

Acceptance test level

The acceptance test process is the highest level of testing. It is associated with a project during the development phase and is followed by the system test. To carry out this process the teams must employ the following activities and perform the following tasks.

<table>
<thead>
<tr>
<th>Input</th>
<th>User and software requirements.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity</td>
<td>The first activity applies to the start of new process and contains the following tasks</td>
</tr>
<tr>
<td>Task</td>
<td>The management team assigns one or more testers in the first stage of the project starting and provides them with the documents related to the project.</td>
</tr>
<tr>
<td>Sub-process Activity</td>
<td>The second activity performs only after finishing all the development processes and the system testing</td>
</tr>
<tr>
<td>Task</td>
<td>The test team and the customer prepare the acceptance test plan. The plan includes the time and the team size to perform the testing activities.</td>
</tr>
<tr>
<td>Task</td>
<td>The test team will cooperate with the customer to create the test cases and the software installation.</td>
</tr>
<tr>
<td>Task</td>
<td>The customer executes the test cases and reports the incidents to the test and management teams.</td>
</tr>
</tbody>
</table>

Output | Acceptance test plan, test cases after that the test results. |

Table 5: Acceptance testing activities
**System test level**

The second level of the test process is the system test. It starts in parallel with the requirement analysis process and contains the following two activities:

<table>
<thead>
<tr>
<th>Input</th>
<th>Software Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity</td>
<td>The first activity starts after finishing the requirement analysis process and includes the following tasks:</td>
</tr>
<tr>
<td>Task</td>
<td>The test team analyzes the software requirements for overlap of use cases and discusses the issues with the project manager or customer. After changing or updating the requirements, the task is repeated.</td>
</tr>
<tr>
<td>Task</td>
<td>Based on the size of the project and development team, the test team plans the testing activities associated with the development processes.</td>
</tr>
<tr>
<td>Task</td>
<td>Based on the documented software requirements, the tester develops draft test cases for system testing. These test cases are created using black box techniques.</td>
</tr>
<tr>
<td>Activity</td>
<td>The second activity of the system test starts after completion of the implementation and the integration test processes. The activity includes the following tasks:</td>
</tr>
<tr>
<td>Task</td>
<td>The test team carries out the created test cases on the software being tested.</td>
</tr>
<tr>
<td>Task</td>
<td>If faults are detected, the test team reports these faults via the management tool and notifies the project manager.</td>
</tr>
<tr>
<td>Task</td>
<td>If the development team confirms these faults and corrects them, the test team re-tests for the faults and tests any other areas that may have been affected by the corrective modifications.</td>
</tr>
<tr>
<td>Task</td>
<td>The team prepares a final report that includes the results of the testing and the faults detected during the testing.</td>
</tr>
<tr>
<td>Output</td>
<td>System test report and faults list.</td>
</tr>
</tbody>
</table>

Table 6 : System testing activities
**Integration Test level**

During the integration test process, the developers take on the role of testers. This encompasses the following two activities.

<table>
<thead>
<tr>
<th>Input</th>
<th>Software Design.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity</td>
<td>The first activity begins during the design process and includes two tasks.</td>
</tr>
<tr>
<td>Task</td>
<td>The test team plans tests for the components addressed in the first activity.</td>
</tr>
<tr>
<td>Task</td>
<td>The test team discusses the software design (flow charts, class diagrams, etc.) and addresses the testable components.</td>
</tr>
<tr>
<td>Sub-process</td>
<td>The second task of the integration test begins upon completion of the unit testing.</td>
</tr>
<tr>
<td>Task</td>
<td>The developers execute the test cases created to validate the testable components. This is performed by executing grouped unit test classes related to the same component.</td>
</tr>
<tr>
<td>Task</td>
<td>If faults are detected, the team reports these faults via the management tool and notifies the project manager.</td>
</tr>
<tr>
<td>Task</td>
<td>If the faults are confirmed, they are corrected and re-tested.</td>
</tr>
<tr>
<td>Task</td>
<td>The team reports the results of the test and the faults detected during the test to the management team.</td>
</tr>
<tr>
<td>Output</td>
<td>Integration test report and fault list.</td>
</tr>
</tbody>
</table>

*Table 7: Integration testing activities*
**Unit test level**

The last level of testing in the new process is the unit test. This process starts in parallel with the implementation and is performed by the development team as specified below.

<table>
<thead>
<tr>
<th>Input</th>
<th>Software code.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Activity</strong></td>
<td>The test team performs the following tasks.</td>
</tr>
<tr>
<td><strong>Task</strong></td>
<td>A unit test plan is created based on the implementation plan and the division of tasks. The plan includes the testable components to which the unit test will be applied.</td>
</tr>
<tr>
<td><strong>Task</strong></td>
<td>For each class or module created, the developer will develop a test class aimed to validate the logic of the class being tested.</td>
</tr>
<tr>
<td><strong>Sub-process</strong></td>
<td>The unit test classes contain test cases created using the presented techniques (white box). It should at least check the boundary exceptions of each method.</td>
</tr>
<tr>
<td><strong>Task</strong></td>
<td>The developers execute the unit test classes and report the detected faults.</td>
</tr>
<tr>
<td><strong>Task</strong></td>
<td>If faults are detected, the team reports these faults via the management tool and notifies the project manager. If the faults are confirmed by the development and management team, they are corrected.</td>
</tr>
<tr>
<td><strong>Task</strong></td>
<td>The test team reports the results of test and the detected faults to the management team.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td>Tested classes and fault list.</td>
</tr>
</tbody>
</table>

**Table 8: Unit testing activities**
4.3.2 Test procedure for software maintenance

The second sub-process in the test process is applied to software maintenance. The procedure is executed through a number of activities and tasks, as discussed in the following description.

Reproduction of the faults

The purpose of the first process performed during maintenance is to reproduce the reported faults. The process is performed by means of the following activity.

<table>
<thead>
<tr>
<th>Sub-process</th>
<th>Input</th>
<th>Software code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity</td>
<td>Before MG begins correcting the reported faults, the test team performs the following tasks:</td>
<td></td>
</tr>
<tr>
<td>Task</td>
<td>The team runs the application that is believed to cause the reported faults. The application is run in an environment similar to the one used by the customer.</td>
<td></td>
</tr>
<tr>
<td>Task</td>
<td>The test team attempts to repeat the situation in which the faults occur by following the same scenarios reported by the customer or through other scenarios.</td>
<td></td>
</tr>
<tr>
<td>Task</td>
<td>If the faults are reproduced and verified by the test team, a confirmation will be sent to the customer, including an estimate of the time expected to be necessary to correct them. If the faults cannot be reproduced, the customer will be asked to send more information about the scenario that led to the faults. This may be in the form of screen shots, information about the environment configuration, etc.</td>
<td></td>
</tr>
<tr>
<td>Task</td>
<td>The team reports the faults via Ace Project [20].</td>
<td></td>
</tr>
<tr>
<td>Task</td>
<td>The team plans a test of the reported faults and prepares the test cases based on the software requirements.</td>
<td></td>
</tr>
<tr>
<td>Output</td>
<td>Reported faults, test plan, and test cases covering the reported faults.</td>
<td></td>
</tr>
</tbody>
</table>

Table 9: Reproduction of reported faults
Unit test level
The unit test process is applied when a developer has fixed reported faults.

<table>
<thead>
<tr>
<th>Input</th>
<th>Software code.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Activity</strong></td>
<td>Before MG begins correcting the reported faults, the test team performs the following activities:</td>
</tr>
<tr>
<td>Sub-process</td>
<td>Task</td>
</tr>
<tr>
<td></td>
<td>If the defective classes or components are associated with unit test classes, the tester runs these classes or extends the test cases to validate the correction. If not, the tester creates new unit test classes, including the test cases to validate the correction.</td>
</tr>
<tr>
<td></td>
<td>Task</td>
</tr>
<tr>
<td></td>
<td>The tester runs the created/modified test classes and checks whether any fault occurs.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td>Tested classes and faults list.</td>
</tr>
</tbody>
</table>

Table 10: Maintenance unit testing activities

Integration testing
The second level of testing in the maintenance process is the test process. This process is intended to ensure that changes made to correct a fault do not affect other classes. The process includes one activity, as described below.

<table>
<thead>
<tr>
<th>Input</th>
<th>Software code.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Activity</strong></td>
<td>This practice includes the following activities:</td>
</tr>
<tr>
<td>Sub-process</td>
<td>Task</td>
</tr>
<tr>
<td></td>
<td>The team runs the test cases related to the component. If any fault occurs, it is reported to the team.</td>
</tr>
<tr>
<td></td>
<td>Task</td>
</tr>
<tr>
<td></td>
<td>The team then re-tests the corrected faults until all reported issues have been fixed.</td>
</tr>
<tr>
<td></td>
<td>Task</td>
</tr>
<tr>
<td></td>
<td>The tester concludes the test process by delivering the final test report, including the fault list and the tested software.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td>Tested components and fault list.</td>
</tr>
</tbody>
</table>

Table 11: Maintenance integrating testing
**System testing**

The last process of the maintenance procedure is the system test, which is intended to validate the software based on the requirements. It is performed by the test team and starts once the faults have been confirmed.

<table>
<thead>
<tr>
<th>Input</th>
<th>Installed software.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sub-process</strong></td>
<td><strong>Activity</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Task</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Task</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Task</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Task</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Task</strong></td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td><strong>Test report, tested software, and fault list.</strong></td>
</tr>
</tbody>
</table>

Table 12: Maintenance system testing procedure

**4.3.3 Faults reporting process**

The last process we presented is the fault reporting process. It is a management process that documents the status of the software faults from reporting to the correction. The process is performed by the test team and executed using the Ace Project tool [21].

<table>
<thead>
<tr>
<th>Input</th>
<th>Software code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sub-process</strong></td>
<td><strong>Activity</strong></td>
</tr>
</tbody>
</table>

51
The team runs the application that is believed to cause the reported faults. The application is run in an environment similar to the one used by the customer.

If the fault is reproduced, its status is changed to TO DO, and if not, the status is changed to NOT REPRODUCED. An acknowledgment is sent to the customer.

If the developer starts fixing a TO DO fault, the fault status becomes IN PROGRESS.

If the developer completes the correction, the fault status becomes TO REVIEW.

If the tester starts testing the corrected fault, the status becomes TO VERIFY.

If the tester has verified that the fault is correctly fixed, the status is changed to CLOSED. Otherwise, it is changed to REOPENING.

<table>
<thead>
<tr>
<th>Task</th>
<th>Documented fault</th>
</tr>
</thead>
</table>

Table 13: Fault reporting process

4.4 Concluding remarks

During this chapter we have described the activities we performed to elevate MG’s test process to the second level of TMM. These activities have resulted in providing MG’s teams with the knowledge and techniques required to perform a sequence of activities as explained in the test process description. In the next chapter we will apply the new test process to case studies and verify whether the claims about the TMM test process are justified.
5. Case Study

To verify the test process that we have described in chapter 4, we will apply it to a case study. The project we have chosen as case study is Z Factuur. In this chapter, we will give a short introduction to Z Factuur. We will discuss the Z Factuur architecture, users and the processes that are supported by this system. In the next chapters, we will describe how Z Factuur is extended by new features and how is tested with the new test process.

5.1 Z Factuur

Z Factuur is a digital invoice application developed in 2007 by MG. It is developed for Z Solution B.V. for the use in the agriculture market. In this market, supplier and farmers are the main stakeholders. Suppliers are companies providing tools, seeds and labours, while farmers are the customers of these suppliers. Z Factuur enables the digital transfer of invoices between suppliers and the customers.

Z Factuur supports the suppliers by the automation of the invoice sending process. Suppliers provide Z Factuur with the data they want to send to their customers. Z Factuur will process the data and create invoices and forward them to the customers. Z Factuur will report the suppliers about the created invoices and whether exceptions occurred during processing. This process results in a cost-savings for the suppliers on prints, paper and postage.

For the customers, Z Factuur provides an immediate access to an overview of the received invoices from different suppliers. A customer, can view, download or print of one or more selected invoices. Further, Z Factuur enables customers to upload invoices they receive from non Z Factuur suppliers. For instance, invoices received for the internet or the electricity provider.
These uploaded invoices will be processed by Z Factuur and presented to the customer like other invoices sent by a supplier. This provides the customer for a better management of invoices they have received.

5.2 Z Factuur system description

The figure 5-1 presented below provides a schematic overview of the chain involved and of their relationships within the Z Factuur system.

![Figure 5-1: Primary processes and their relationship](image)

**Supplier**

Supplier is the source of the invoices. It is combination of an organization and the software to support the storage and sending the invoice data to Z Factuur.

1. **Customer**

Customer is the destination of the invoice. It is a customer of one or more suppliers and has been subscribed in Z Factuur.

1. **Z Factuur**

This is a system which receives data from suppliers and stores it. It will process the received data, convert it to invoices and present the invoices to the subscribed customers. For unsubscribed customers, Z Factuur will send printed invoices by mail.
5.3 Z Factuur processes

In this section, we will describe the main processes of the Z Factuur system. These primary processes have a mutually subsequent relationship as shown in the figure 5-2: no process can commence before the previous process has been completed. The main processes are:

![Diagram of Z Factuur Primary processes and their relationship]

**Figure 5-2: Z Factuur Primary processes and their relationship**

**Configuration of connection**

The first primary process in the Z Factuur system is the configuration of the supplier connection with Z Factuur. This connection needs only to be configured once per supplier. It is necessary for a supplier to be able to send the invoices data for the creation of the invoices.

To support the connection process, Z Factuur used the Enterprise Service Bus (ESB). The bus defines interfaces for the receiving invoices sent by suppliers. The bus offers a web service, FTP and email interfaces.

The configuration of connection process is established as follows. A client application to connect the supplier to Z Factuur has been developed. Examples of developed client applications are Java and dot.Net connectors or an FTP client application. The choice of the technique used to develop the client application depends on the technique used by the supplier internally.
In the second step, a supplier profile is created. This profile consists of a set of data about the supplier required for the creation of the invoices. The data include the supplier name, the logo displayed in the invoice and the supplier contact information. These data can be entered by the suppliers through a web portal provided by Z Factuur. After completion of this process, the supplier is ready to transfer data to Z Factuur.

Customer subscription

In the second primary process, a customer will submit a subscribe request to be able to receive digital invoices of suppliers. This subscribe request is sent through the Z Factuur web portal. Z Factuur will validate the subscribed customer by asking the customer to provide a unique characteristic that allows the suppliers to recognize the subscriber as customer. An example of unique characteristic is a postcode, first and the last name or a client id. Once the customer is validated, Z Factuur will send information to the customer how to log into Z Factuur and to receive invoices digitally on his account on Z Factuur.

In a similar way the customer can cancel a subscription by sending an unsubscribe request. The customer will not receive digital invoices any more.

Data processing presentation

The last process is the data processing and presentation. This process occurs each time a supplier sends data to Z Factuur. Z Factuur will process that data as follows.

First it will store the received data in the Z Factuur database. Then it will process the stored data by extracting the information required to create the digital invoices. Examples of extracted data are information about the customer, deliveries and amounts and the payment date.
Z Factuur will transform the extracted data into predefined invoices. For the creation, Z Factuur uses the data saved in the supplier profile.

The invoices will be presented digitally to the customers in the inbox of the customers. The invoices are presented in HTML and PDF format. The customer can view the invoices and eventually print the invoice.

### 5.4 Z Factuur development

The Z Factuur application has been developed by MG in the period between November 2006 and February 2007. Like many projects developed by MG, Z Factuur is an undocumented project. In the project repository of Z Factuur, we were not able to find documents related to the project like a development plan, a software function description, or a database design.

To get the information about Z Factuur, we interviewed the director R v.d Spek and reviewed the source code of Z Factuur. After one week of research we drafted the following conclusions:-

Z Factuur is a project developed by three developers, R v.d Spek and two developers from MG India.

In the requirement analysis phase, the functional description of the project has been presented orally by R v.d Spek. Developers in India were only provided with screenshots about how Z Factuur web should look like. There were no functional description documents. More information about the features supported by Z Factuur was gathered through the further research with Z Factuur application code.
In the software design phase, the architecture has been created for Z Factuur in an ad hoc way. It was not documented. During the interview with R v.d Spek, we have created a summary document about the Z Factuur components. It consists of four components.

- The Z Factuur Enterprise bus
  This is an open source Enterprise Service Bus (ESB) that defines the communication between suppliers and Z Factuur. The Z Factuur Enterprise bus provides different interfaces for receiving invoices data. This can be a web service, but also an SMTP (email) interface or other possibilities. The Z Factuur Enterprise will store the received invoices into the Z Factuur database.

- The Z Factuur web portal
  Z Factuur entity is the central part of the system which executes the process of receiving the data from Data Entry Entity and from Z Factuur Enterprise bus, and the presentation of the invoices to the customers.

- Z Factuur invoice database
  The database responsible for storing the invoices data.

- Data Entry Entity
  This is a web application developed by R. v.d Spek in 2006. The application is used to process non digital invoices by converting them to digital one’s. Non-digital invoices comprise of faxed invoices as well as invoices in the form of PDF documents. For the Z Factuur system, Data Entry Entity is used to process invoices uploaded by the customers. Invoices will be digitalized and presented to the customer.
The Z Factuur design has been implemented using Java. During the development of Z Factuur, test activities were NOT applied. The developed application was only “checked” by the developers. They only checked that implemented features were working. When the Z Factuur was released to Z Solution, no kind of acceptance test was applied.

Finally Z Factuur was introduced by Z Solution to its customers in April 2007. Until March 2008, the number of customers subscribed to Z Factuur was 997 while the number of suppliers was 4.

In January 2008, Z Solution asked for an extension of Z Factuur with a new process for sending the invoices to the NotaBox system.

In the next chapter, we will give a short introduction of Notabox system. In chapter 7, we will discuss the development of the connection between Z Factuur and NotaBox and how we applied the new test process to that connection.
6. NotaBox

NotaBox is a digital invoice system introduced by Rabo Bank. It enables companies to send invoices and giro forms online to their customer via the Rabo Internet Banking system. Consumers can view their invoices in the Rabo Internet Banking system and pay them.

For companies NotaBox offers cost saving on prints, paper and postage together with the efficient administration. It results into faster payment and less payment reminders send to the customers.

For customers, NotaBox responds to consumer demand for speed, convenience and safety.

6.1 NotaBox system

![NotaBox System Diagram]

Figure 6-1: NotaBox system

Figure 6-1 provides an overview of the subsystems and their relationships within NotaBox. We have discussed the supplier and customer subsystems of Z Factuur in the chapter 5. Other subsystems in NotaBox are:
Rabo Bank Portal

The Rabo Bank portal acts as service provider for customers and provides a link to the Biller Service Provider. The portal is part of Rabo Internet banking. It manages the invoices sent by suppliers to its customers and allows customers to subscribe to the suppliers of their choice.

Biller Service Provider (BSP)

The BSP acts as a service provider for suppliers and provides a link to the Rabo Bank portal. It manages subscribe requests from customers by having them validated by the suppliers. The BSP manages the invoices that suppliers distribute via NotaBox to customers. Suppliers who want to be connected to the Rabo Bank portal needs to develop their own BSP.

Subsystems in NotaBox act as a chain along which information is exchanged. The chain to an individual customers is created when a subscribe request to receive supplier's invoices is sent by a customer. Rabo Bank portal forwards the request to the BSP, who in turn forward it to the supplier for validation. The validation response is sent back to Rabo Bank portal. The portal reports the supplier's response to the customer. The process will result in an active subscription that is able to receive invoices from their suppliers and to make payments on them.

In the next chapter we will discuss the integration of Z Factuur into Rabo bank NotaBox. This will be done by developing a Biller Service Provider to link Z Factuur to the Rabo Bank portal as shown in the figure 6-2.
In the next chapter, the development of the Z Factuur BSP will be discussed. We will discuss also how we have applied the new test process during the project and what the results of the testing work are.
7. Z Factuur BSP project

In this chapter we will describe the software development of the Z Factuur BSP system. Z Factuur BSP will connect Z Factuur to the NotaBox. This project is chosen by MG as case study to evaluate the new test process. Therefore we will give a detailed description to the development activities.

Z Factuur BSP development started at 15/04/2008. MG’s director assigned a team consisting of a project manager, two developers and a tester in India. The team members were part of the team that developed the Z Factuur system in 2006. They were familiar with the development techniques, the design and the implementation. The team included also a tester/test coordinator who worked initially in India and later in the Netherlands. Information about the team including the member’s responsibility is presented in table 15.

<table>
<thead>
<tr>
<th>Authority</th>
<th>Name</th>
<th>Responsibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project manager</td>
<td>Magash Babu</td>
<td>Responsible for of the management of the project including management, monitoring etc.</td>
</tr>
<tr>
<td>Developer</td>
<td>S.Deepesh</td>
<td>Implementation tasks related to the Z Factuur BSP project</td>
</tr>
<tr>
<td>Developer</td>
<td>Anish C.P</td>
<td>Implementation of tasks related to the modification of the Z Factuur project.</td>
</tr>
<tr>
<td>Tester</td>
<td>Adarsh Babu</td>
<td>Execute test plan(s), develop and execute test cases, analyze requirements to verify it can be tested, document and monitor test issues and track to closure.</td>
</tr>
<tr>
<td>Tester/Test coordinator</td>
<td>F.Alhassany</td>
<td>Responsible for ensuring that new test process is followed, Monitor and control test work products and test results, Review test work products to ensure that they are complete.</td>
</tr>
</tbody>
</table>

Table 15: Z Factuur Development team

At the start MG director gave in India a short presentation of the new system to the development team. He described the purpose of Z Factuur BSP, the features to be supported by Z Factuur BSP and the users of that application. He discussed also the consequences of a delay in the project release and the risks by the delivery of faulty software.
MG’s director set the start and the end date of the project and the acceptance criteria for Z Factuur BSP. He provided the team with an English version of requirements document of NotaBox. This document described the functional and technical design of NotaBox components together with the service level agreements for the development of these components.

In the next section, the development of the Z Factuur BSP is detailed. In each phase, we will describe the development activities together with the tasks, and activities that have defined in chapter 4.

**Planning the Z Factuur development activities**

We have created the development plan. It defined the steps that should be done, the tasks of the team and the time needed to develop the Z Factuur BSP application. The plan was created in collaboration with the members of the development team. Time estimations for the separate tasks were based on the experience with similar projects and the information from the NotaBox requirement documents. The estimations were collected and resulted in a plan presented in table 16.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Activity</th>
<th>Duration</th>
<th>Responsible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirement analysis</td>
<td>Study of the requirements document, create test plan.</td>
<td>16/04 to 22/04/2008</td>
<td>Testers /Developers</td>
</tr>
<tr>
<td></td>
<td>Create test cases for the subscription and invoice sending features.</td>
<td>22/04 to 15/05/2008</td>
<td>Test team</td>
</tr>
<tr>
<td>Design</td>
<td>Create design for the BSP Z Factuur application</td>
<td>22/04 to 30/04/2008</td>
<td>Developers</td>
</tr>
<tr>
<td>Implementation</td>
<td>Implement the components; create the unit and integration test of the new implemented classes related to the BSP.</td>
<td>01/05 to 01/06/2008</td>
<td>Developers</td>
</tr>
<tr>
<td>Testing</td>
<td>Create system test cases for the Z Factuur BSP.</td>
<td>15/05 to 15/06/2008</td>
<td>Test team</td>
</tr>
<tr>
<td></td>
<td>Execute acceptance test and final test report</td>
<td>27/06 to 15/07/2008</td>
<td>Test team</td>
</tr>
</tbody>
</table>

Table 15 : Z Factuur Development plan
7.1 Requirement analysis

The requirement analysis phase began on 16/04/2008 and took four weeks. During this phase, three activities were performed.

1. Analysis of the NotaBox requirement document

The activity was performed in India by all the team members and took one week. It aimed to understand the Z Factuur BSP application defined in the requirement document. We began the activity by reading NotaBox requirements. The focus was on the architecture of NotaBox, the component interactions in the system, the interfaces between the components, and the interface we should develop to connect Z Factuur to the Rabo Bank portal. For that interface, we have studied the functional and non-functional features that should be supported. We conclude that the desired application should act as an interface to support the communication between Rabo bank portal and Z Factuur. The interface should implement two features: processing of subscription requests from customers, and sending of invoices to customers. In the test case creation activity, we will give more details about these features.
During the requirement analysis activity, we detected a few unclear terms in the NotaBox requirement. We have collected the unclear requirements (Figure 7-1) and asked the MG manager for further explanation.

**Question 1 (By F.Alhassany)**

As per the DigitalNota schema, there are four invoice types. They are Info, InvoiceDirectDebit, InvoiceAcceptgiro, InvoiceCredit.

*My Questions are*

1. Is the Info field refers to Remainder (Herinnering) in ZFactuur?
   (answer: R.v.d Spek)
   Negative, the info field is a notification message, like an attachment without invoice, like a notice, an email or something of the kind

2. From my understanding, an invoice is said to be InvoiceDirectDebit if the field A203817DatumIncasso of the invoice has been given a value, otherwise it is InvoiceAcceptgiro. Then, when can we assume that an invoice is InvoiceCredit?
   (answer: R.v.d Spek)
   Negative, if the field A203838CodeBetaling equals 3, then it is also a direct debit

**Question 2 (By F.Alhassany)**

Please manifest the selection of Amount field for ZFactuurBSP.

(answer: R.v.d Spek)

The amount can be found on AdedTotalen where the field A204655Bedragqualifier equals 86 (or where this there only is one "totalen" record)

**Question 3 (By F.Alhassany)**

As per the DigitalNota schema, there is DocumentId and OriginalDocumentId. What is the difference between this? Currently DocumentId is calculated in the following way.

DocumentId = current Time in Millis + Math.random() + AgroswitchId + MessageReference

(answer: R.v.d Spek)

That is something that I will need to check. Anyway, if you want to generate a true random sequence, it is better to use the java.util.UUID class.

Calling UUID.randomUUID().toString() Generates a unique string.
2. The creation of test cases

The creation of test cases was executed by the two testers in 3 weeks. The test cases were created for the verification the implemented features of the Z Factuur BSP. We have used the textual description of use cases Subscribing and Invoice sending from the NotaBox requirement document to create the use cases. The use cases described the flow of events for the normal execution by the Z Factuur BSP. It contained also alternative flows from an optional or exceptional situation. Figures 7-3 and 7-4 show the normal and exceptional use case of the Subscription Request feature.

- The End-user subscribes to a document stream of a Biller (subscribe request) or cancels a subscription to a document stream of a Biller (unsubscribe request).
- The BP processes the request and prepares a subscribe request for the End-user to be sent to the Biller (via the BSP).
- At a certain point the BP gathers all available subscribe requests and sends them to the BSP, including signature. This is a file with requests and cancellations (SRQ_MUT) and a signature file (SRQSIG).
- The BSP checks the files he has received for integrity (signature), XML structure and to see whether the header contains the correct information (supported version, no duplicate and correct number of records), issued an ACK or NACK to the batch he has received and prepares an acknowledgement for this batch to be sent.
- At a certain point the BSP gathers all available acknowledgements and sends them to the BP, including signature. This is a file with ACKs and NACKs (SRQACK) and a signature file (SRQSIG).
- When an ACK is issued in step 4, the BSP at a certain point starts processing the records in the batch. The BSP will have to validate the subscribe requests, whether or not via the Biller. Based on the results the subscribe responses are created and prepared for sending.
- At a certain point the BSP gathers all available subscribe responses and sends them to the BP, including signature. This is a file containing subscription confirmations, rejections and cancellations (SRSMUT) and a signature file (SRSSIG).
- Similar to step 4 the BP now checks the files he has received and prepares an acknowledgement for this batch to be sent.
- At a certain point the BP gathers all available acknowledgements and sends them to the BSP, including signature. This is a file containing ACKs and NACKs (SRSACK) and a signature file (SRSSIG).
- When an ACK is issued in step 8, the BP at a certain point starts processing the records in the batch.

![Figure 7-2: Subscription request use case](image)

1. A subscribe request is received for an unknown/inactive Biller or document stream, a new subscription is registered, but with Answer = REJECTED and ReasonCode = 321 “Biller unknown” or 322 “Stream unknown”. The record is placed immediately in the subscribe response cue and it is not sent to the Biller.
2. If a cancellation request is received for an unknown subscription, then Answer = REJECTED. The record is placed immediately in the subscribe response cue and it is not sent to the Biller.
3. If the subscribe response cannot be placed in the cue, an error is logged. If the subscription cannot be created in the administration, an error is logged.

![Figure 7-3: Exception in subscription request use case](image)
To generate test cases from the fully detailed use cases, we used a two-step procedure. In the first step the use case description was analyzed to identify each combination of main and alternate flows. This combination was called a test scenario. An example from the subscription is the following list.

- BSP receives SRQMUT (number 3 of in the use case)
- BSP sends ACK to BP (number 5 of the use case)
- BSP sends SRSMUT to BP (number 7 of the use case)

After the compilation of the test scenario, the next step was the translation to one or more test cases. Examples of test cases created for the above scenario are shown in Table 17.

<table>
<thead>
<tr>
<th>Test ID</th>
<th>Descriptions</th>
<th>Expected Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BSP receives a SRQMUT of 1 new valid subscription request and a valid cancellation request.</td>
<td>BSP send an ACK and SRSMUT (including 1 acceptance and rejected) back to the BP</td>
</tr>
<tr>
<td>2</td>
<td>BSP receive a SRQMUT of 6 new valid subscription requests.</td>
<td>BSP send an ACK and SRSMUT including 6 acceptance back to the BP</td>
</tr>
<tr>
<td>3</td>
<td>BSP receive a SRQMUT of 1 already requested and 3 are new valid requests.</td>
<td>BSP send an ACK and SRSMUT out of the 4 request 1 request should be rejected and other 3 needs to be accepted.</td>
</tr>
</tbody>
</table>

Table 16: Subscription request test cases

These two steps were repeated until all the possible flows were covered. The test cases creation activity resulted in 33 test cases for the subscription feature and 29 test cases for send invoice feature.
7.2 Design phase

In software design phase, the development team defined the components and the interfaces of the project. The phase was carried out in India by the developers and takes one week.

Due the lack no documentation, the developers used the source code of Z Factuur project and the NotaBox architecture as input. This input was used to define the architecture and its components. The design phase resulted in a high level design diagram (figure 7-4) representation of Z Factuur BSP.

![Figure 7-4: Z Factuur BSP architecture](image)

In this design, BSP Z Factuur is represented as a standalone system consisting of two components.

1. **Z Factuur BSP**

   Z Factuur BSP component is defined as a web service called by the Rabo bank portal to process subscription requests, and to send the invoices to Rabo Bank portal.

2. **ZfactuurBSPInvoiceWS**

   This is a web service that validates the invoices, sent from the Z Factuur end through Z FactuurMule2. It stores the invoice data into
the Z Factuur BSP database to prepare them for sending them later to the Rabo bank portal.

The developers redefined the functionality of two Z Factuur components. These changes were required to support the work of the Z Factuur BSP.

3. **Z FactuurMule2**
   
   It is responsible for sending invoices to the customers of the Z Factuur, was modified to be able to send the invoices details of the subscribed customers to the Z ZfactuurBSPInvoiceWS component.

4. **SearchMemberService**
   
   It is a component used to validate the users log in the Z Factuur application. The component was extended with a method to validate the customers ids received from the subscription/unsubscription requests from Rabo Bank.

   In the next phase we will explain the how the components have been implemented and tested.

### 7.3 Implementation phase

In the implementation phase the Z Factuur BSP design was transformed to software. The phase took four weeks and performed four activities.

**The implementation of Z Factuur and Z Factuur BSP components**

In a period of 2 weeks, the developers prepared the development environment consisted of Eclipse 3.2, MyEclipse 5.1 (java-1.6) development platform and SQL 2006 database server.

In Eclipse, they created a new project consisting of three packages.

- Handler package contains classes handle the subscription,
cancellation requests, and the sending of invoices.

- Business logic package implemented classes with methods shared by classes from the handler package. An example is the class common validates XML messages.

- Persistence package contained classes responsible for saving and updating data in the BSP Z Factuur database. Implemented using Hibernate technique.

**The modification of SearchMemberService and Z FactuurMule2**

This task was carried out by one developer in 8 days. SearchMemberService was extended with a new feature to validate the customer’s id’s and to check whether Z Factuur database contains the id. The Z FactuurMule2 component was extended with a feature for invoices received by the Z Factuur. The feature checks whether an invoice is directed to the NotaBox subscribed customers. If so, the invoice is send to the BSP.

**The creation of unit tests**

The activity took 5 days and was performed by one developer. The JUnit tool was used to construct unit tests for each class of the Z Factuur BSP component. The test classes were collected in a separate package that does contain the Z Factuur BSP component package structure. Each unit test class contained test cases to validate the methods of the class under test. An example is the SetInvoiceDetails method part of the DocumentSummaryBatchHandler class. For this method testSetInvoiceDetails is shown in the figure 7-5.
To test the method, the developer used two test cases. In the first case the method will receive a null object while in the other case is “happy flow” where the method will receive a valid object. When a test failed, the developer would fix the fault directly without logging it in Aceproject. If the test case passed, the other methods were tested.

The developer repeated the unit test creation steps for all implemented classes in the Z Factuur BSP components. Because the lack on time, the other new or modified components of Z Factuur and Z Factuur BSP were NOT unit tested. Even after reporting this omission to the project manager and the MG director, the activity was still ignored by the developers.

The unit test activities resulted in 73 unit test classes for the Z Factuur BSP component. During the execution of the tests 43 methods failed and were fixed.

```java
public void testSetInvoiceDetails()
{
    try {
        dsbh.setInvoiceDetailsAndNotify(null, null);// test case 1 fails
    } catch (NullPointerException e) {
        // we should get here
    }
    // test case 2
    dt.setAlttext(""s");
    dt.setAmount(0.00);
    dt.setCreditaccountno(""s");
    Date d= new Date();
    d.setYear(0000);
    d.setMonth(1);
    d.setDate(31);
    dt.setDuedate(d);
    dsbh.setInvoiceDetailsAndNotify(dt, ds);
}
```

Figure 7-5 : Java class unit test
The integration of the implemented components

The last activity was the integration of the Z Factuur BSP components with Z Factuur. It was executed by the two developers in 8 hours.

To integrate the components, the developers deployed the two Z Factuur BSP components and the Z Factuur on separate test servers.

They test the integrated components using a test application called the NotaBox dummy. The test application was implemented by the developers and contained two classes, the TestreceiverAction and TestrequestSender. They acted as the Notabox system and used to create subscription requests or to receive of the subscription responses.

The developers used TestrequestSender to test the integration of Z Factuur BSP with SearchMemberService. Developers send subscription request using TestrequestSender and receiving the response back using TestreceiverAction.

To test the integration of Z FactuurMule2 with the Z FactuurBSPInvoiceWS invoices were inserted in Z Factuur to check whether Z FactuurMule2 would send them to Z FactuurBSPInvoiceWS.

Because the lack of time the developers did NOT apply any other kind of integration test on the developed components. At last, the development team delivered Z Factuur BSP to the test team. In the next section we will discuss the activities performed during the test phase.
7.4 Testing

The last phase of Z Factuur BSP development was the test phase. It consisted of a system and an acceptance test. We will describe both activities and tasks we performed together with the test results.

System test

The system testing aimed to verify that the Z Factuur BSP application met the specification of the NotaBox document. It was conducted by the test team in a period of 1 month. We started the system test by configuring the Z Factuur BSP test environment. It is consisted of three test servers. One was used to deploy the Z Factuur BSP application, the other to run the Z Factuur, and the last to run a NotaBox dummy.

The execution of the Z Factuur BSP system test performed as follows.

As input, we used the test cases created during the requirement analysis phase. Each test case converted to a physical test script in an xml format includes a test data.

For example, figure 7-5 is a subscription request (SRQMUT) consisting of one subscription request for a valid customer (id is 4199780628) and a cancellation request for a valid customer (id is 4199180622).
Then we send the *SRQMUT* to Z Factuur BSP by using the Testrequestsender of the Dummy NotaBox. Z Factuur BSP verified first whether the *SRQMUT* had a valid format and sends an acknowledgement (*ACK*) and subscription response (*SRSMUT*) back to the Dummy NotaBox (figure 7-5).
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<Acknowledgements xmlns="http://www.sdndesk.nl/digitalenota/v02.0/">
  <Header>
    <BatchId>ACK-BSP5-RABOBANK-20080522135423</BatchId>
    <InterfaceId>ACK</InterfaceId>
    <Version>02.0</Version>
    <SourceSystemId>BSP5</SourceSystemId>
    <TargetSystemId>RABOBANK</TargetSystemId>
    <CreateDateTime>2008-05-22T13:54:23.457+02:00</CreateDateTime>
    <MessageCount>1</MessageCount>
  </Header>
  <Acknowledgement>
    <BatchId>SRQ-RABOBANK-BSP5-20080520104649</BatchId>
    <AckTimeStamp>2008-05-22T13:54:23.470+02:00</AckTimeStamp>
    <Result>OK</Result>
  </Acknowledgement>
</Acknowledgements>

<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<Header>
  <BatchId>SRS-BSP5-RABOBANK-20080522135426</BatchId>
  <InterfaceId>SRS</InterfaceId>
  <Version>02.0</Version>
  <SourceSystemId>BSP5</SourceSystemId>
  <TargetSystemId>RABOBANK</TargetSystemId>
  <CreateDateTime>2008-05-22T13:54:26.222+02:00</CreateDateTime>
  <MessageCount>6</MessageCount>
</Header>

<SubscribeResponse>
  <UserId>222166933737</UserId>
  <SenderId>S21</SenderId>
  <StreamId>S21-1</StreamId>
  <SSCC>7098114034</SSCC>

  <Response>
    <Answer>ACCEPTED</Answer>
  </Response>
</SubscribeResponse>

<UnSubscribeResponse>
  <UserId>222166933737</UserId>
  <SenderId>S21</SenderId>
  <StreamId>S21-1</StreamId>
  <SSCC>4199180622</SSCC>

  <Response>
    <Answer>ACCEPTED</Answer>
  </Response>
</UnSubscribeResponse>
</SubscribeResponses>

Figure 7-7 : Subscription acknowledgement (ACK)
We will check whether the generated acknowledgement (ACK) and response (SRSMUT) complied with the expectations of the test case. If no fault was discovered, the test case passed and we started with the next test case.

To execute all test cases, we have repeated the three steps. The execution of all test cases was completed on 06/06/2008. The time we spend to execute each test case was about 45 minutes. This time included the creation and running of the test scripts.

During the test cases activity we have detected 12 faults. Table 18 presents a selection of faults. A complete list of faults is added as Appendix.

<table>
<thead>
<tr>
<th>Fault ID</th>
<th>Priority</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>critical</td>
<td>Documents are sending to users without the URL of the original invoice.</td>
</tr>
<tr>
<td>2</td>
<td>major</td>
<td>If we send subscribe requests with unknown SenderId then the link table (tbl_subscriptioninfo) on the Z Factuur database recorded as SUBSCRIBED (Subscription). This is wrong</td>
</tr>
<tr>
<td>3</td>
<td>major</td>
<td>While Subscribe Response is sending if any SSCC (sender ID) is null then no response is sent. That is if any of the request contains SSCC as NULL (empty) then the Subscribe response will not be sent. Only the batch acknowledgment will be send.</td>
</tr>
<tr>
<td>4</td>
<td>critical</td>
<td>If we send an unsubscribe request then the cancellation is not executed even though the userId exists (Validation problem)</td>
</tr>
</tbody>
</table>

Detected faults were reported in the Ace project. The project manager assigned these faults to the developers for fix. After the receipt of the corrected software, we re-executed the test case steps for fixed fault. If the incident did not occur again, we recommended closure otherwise; we mentioned the development team the fault remained open. The retest work of all the fixed faults was finished in 11/06/2008.

In the last week of the system from 11/06/2008 through 17/06/2009, we performed regression tests to ensure all faults had been resolved and other parts of the application were not impacted. This was done by re-executing of all the test cases. We have finished the regression test without finding any faults in the application. In 18/06/2008 the final test report has been created and
reported to MG director. In that day, the Z Factuur BSP application became ready for the acceptance test.

**Acceptance Test**

The acceptance test aimed to verify Z Factuur BSP in a production environment and to check whether Z Factuur BSP meets the acceptance criteria defined by Rabo Bank. The acceptance test started in 20/06/2008 and took 3 weeks. It is performed in the by Sibel Sahen tester/ test coordinator in Rabo Bank and Firas Alhassany of MG.

In this phase, we were involved in two activities

**Preparation the acceptance test**

Acceptance test was prepared during meeting between the two testers in 19/06/2008. In that meeting we have discussed the acceptance test plan, the test procedure, test environment and the acceptance terms.

Rabo was strict about the conditions of the acceptance test. The acceptance test should be successfully executed in the last week of June. Otherwise, other suppliers were invited for the same job.

The procedure that we should follow during the test was. At first Rabo bank will create a set test case to demonstrate that Z Factuur BSP complied with all requirements. Sahen would send every test case by email. The test case was in an xml soap message contains test data. We will execute this test script on the Z Factuur BSP and send a response back to Rabo. If no faults were discovered, the test case “passed.” Otherwise the Rabo coordinator would report the faults to MG’s tester; with a specific time to repair the fault. MG should correct and retest the reported fault. If the repair was incorrect, the test was suspended and rescheduled to another date.
**Execution of the test cases.**

Acceptance test execution started at 20/06/2008. Rabo test coordinator performed four test cases. She started validating the subscription request feature by sending a test case contains 5 subscription requests. We have executed the test case locally on the Z Factuur BSP using the Dummy NotaBox and sent her the ACK and Response back. The second test case was to validate sending invoices to subscribed customers. Then she sent a test case to validate the cancellation feature. The final test was to check whether Z Factuur BSP would send invoices to unsubscribed customers after canceling the subscription. All these test cases were correctly passed. The time required to perform the 4 test cases was 12 days. The reason of the delay was the internal faults in Rabo Bank testing environment.

On 12/07/2008 Rabo test coordinator provide us with a test report. It summarized the results of the acceptance test and confirmed that Z Factuur BSP software has met the acceptance criteria defined by Rabo Bank NotaBox.

In 15/07/2008 Z Factuur was online connected with NotaBox. Since that date until the end of 2008, no incident was reported about the connection.
8. Z Factuur Account

In this chapter, we will discuss another project where Z Factuur was extended with new two features.

8.1 Invoice presentation

The invoice presentation was an existent feature in Z Factuur. It is responsible of the presentation of the invoice information to the customers like the name of the invoice sender and the total amount of tax. The feature was extended to present additional information. It will show the number of items, item price and description, discount type, the type of applied tax, the total discount amount with or without tax, and the invoice total amount. The presentation feature should not only present the data provided by the sender But also able to gather and calculate data from the provided data. The new invoice presentation would look like figure 8.2.
8.2 Accounting Integration

The accounting integration feature will enable Z Factuur customers to integrate their accounting software with Z Factuur application. A customer should configure Z Factuur for the accounting software with the user interface that is shown figure 8-3. Z Factuur will transform the invoices to the XML format that is supported by the chosen accounting software. The customer needs only to download the invoices from Z Factuur and import them into the accounting software he/she uses.

Figure 8-2: Accounting configuration

The development of these two features was started by assigning of 3 developers and 2 testers one from India and other in the Netherland. They were provided with the following documents.

- Technical requirements document for the integrating accounting future. The document includes screen shots of the invoice web page for the invoice presentation.
- Exact Globe as an example of accounting software that should be supported by the new accounting feature.
In the next section, we will explain the development phases of these two features.

8.2.1 Requirement analysis

In the requirement analysis we created the development plan and studied the technical and the screen shots documents. Based on these documents, we created a set of 32 test cases to validate the new accounting feature and 21 test cases to validate the new invoice presentation. We studied the Exact software and created a manual describing the steps required to configure Z Factuur with the ledger, supplier, and the creditor numbers and how to import invoices in Exact.

8.2.2 Design

This phase was performed in ad-hoc without creating documentation. The team decided to implement the integration new models as a separated application and using the same architecture as Z Factuur described in chapter 7. For the invoice presentation functionality, the existing architecture of Z Factuur will not be changed as the request is only extension the current feature.

8.2.3 Implementation

The implementation of the invoice presentation feature started with modifying classes in the business logic layer. A new class InvoiceDataCal was added. The class contained methods responsible for gathering and calculating data from the invoice data provided by the sender.

For the accounting integration feature, the developers created a new
package called the ZFactuurAccounting. It included classes with common methods responsible for presentation of the invoices and exportation of the invoices into XML format.

Finally, the source code of the accounting feature was integrated with Z Factuur and deployed on the test server. The application was checked in an ad-hoc way whether faults occurring.

Because of the developers were often interrupted by requests from MG managers to perform ad-hoc tasks related to other projects. Creation of unit test was escaped in this project and it becomes difficult to complete the unit test in the planned time.

8.2.4 Testing the Z Factuur Accounting

The testing the Z Factuur accounting application started with the system test. The new version of Z Factuur was installed on a test server accessed by two test machines. For both machines, various internet browsers were installed like Fire fox, Safari and IE, together with the web performance tool, Http Watch to estimate the application responses.

The test set created during the requirement analysis phase was executed. During the test case execution, we have detected 97 faults. They were assigned to the developers. After they were fixed, we repeated the execution of the test cases. Now it appeared that 14 faults were not correctly fixed.

After completion of the system test, we planned an acceptance test with ZET solution, they company that would introduce Z Factuur to the customers. They had assigned two testers to perform the acceptance test, an accountant and an information analyst of in the team that specified the accounting feature. They investigated whether the developed features corresponded with the customer’s expectation.
They began with exploring the new invoice presentation and whether the calculated values were consistent with the data indicated in the invoice. They configured the accounting functionality using the installation manual we created in the requirement analysis phase. Then they exported and imported into the Exact accounting application.

During this test session, I was reporting the observations made by the team. There were 13 observations and 3 faults in which the Z Factuur did not meet the customer's expectations. Later I discussed the observations with MG manager. All observations were sent to the developers in India for fix. The fixture took 3 weeks followed by two weeks of re-test the system. The new version of Z Factuur was released and presented in February 2009 to ZET solution.

Despite the 8 weeks of delay of the software release and the high number of faults detected during the test. We think Z Factuur was not ready to be presented to the customers without the effort done by the test process. The project shows how important the test process is to help the development team to discover weaknesses caused by lack on the requirements. We think that development team did not spent the necessary time for the technical analysis, especially, on how to retrieve and calculate data from the received invoices. This lack was discovered by the test team through the creation of test cases matching the customer expectations and validating the implemented functions.
9. Conclusions and future work

In this chapter we will discuss the result of applying the new test process on the case studies. The result is summarized in table 17. The table contains the development phases. In every phase, the test tasks applied during this phase are presented. For each task we will give the time spent to execute the task and the number of detected faults.

<table>
<thead>
<tr>
<th>Development phases</th>
<th>Z Factuur Accounting</th>
<th>Z Factuur BSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (h)</td>
<td>%faults</td>
<td>Time (h)</td>
</tr>
<tr>
<td>Requirement analysis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Requirement analysis and discuss unclear requirements</td>
<td>14 0</td>
<td>70 3</td>
</tr>
<tr>
<td>Create test cases</td>
<td>60 0</td>
<td>82 0</td>
</tr>
<tr>
<td>Create test plan</td>
<td>8 8</td>
<td></td>
</tr>
<tr>
<td>Design</td>
<td>No test applied</td>
<td></td>
</tr>
<tr>
<td>Implementation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unit test</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Create unit test classes</td>
<td>0 0</td>
<td>46 38</td>
</tr>
<tr>
<td>Report and re-test faults</td>
<td>0 0</td>
<td>24 0</td>
</tr>
<tr>
<td>Integration test</td>
<td>No test applied</td>
<td></td>
</tr>
<tr>
<td>System test</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execute system test cases</td>
<td>320 79</td>
<td>124 14</td>
</tr>
<tr>
<td>Re-test detected faults</td>
<td>169 27 (re-open)</td>
<td>32 0</td>
</tr>
<tr>
<td>Acceptance test</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Plan acceptance test</td>
<td>4 8</td>
<td></td>
</tr>
<tr>
<td>Execute test cases</td>
<td>4 13</td>
<td>50 0</td>
</tr>
<tr>
<td>Re-test detected faults</td>
<td>250 0</td>
<td>0 0</td>
</tr>
<tr>
<td>Deadline and release</td>
<td>No (8 weeks delay)</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 17: Summary of the test results
From the above table we can see the following points.

1. In both case studies, the development team had ignored applying the integration test. The developed components in both cases were integrated without any kind of test. We think that integration test should be applied to get optimal results of the test process.

2. Unit test was partially applied. The developers create the unit test only in Z Factuur BSP project.

3. In the case on Z Factuur Accounting, we can see the high number of detected faults in the system test. This was because the unclear requirement document which is used as input for to create test cases in for the system test.

Finally, we will answer the two questions proposed in the first chapter.

1. Can the combination of the CMM/TMM models be applied effectively in a small global software organization to enhance the software quality?

   Yes it can. We think that improvement in the test sub-process using the TMM lead to a partially improvement in MG development process as whole. Which lead to enhance in the software quality.

2. What influences do organizational culture, team size and team distribution have on the process improvement?

   In our case, the organizational culture has effect the test process improvement, an example is that development team in India is not used to apply any kind of test during the development process. Changing this habit need a lot of pressure from the management.

   For the team size and distribution, in MG case we have noticed that process improvement needs a lot of coordination work between the distributed teams. The coordination activities and tasks are not defined in CMM or TMM models.
Literature and references


[10] Subrahmanym Amallaraju and others, Professional java server programming


[12] Laurie Williams Dright Ho, Unit testing in Eclipse using Junit, Department of Computer Science North Carolina State University, 2008.


[17] Rabo bank NotaBox service www.rabobank.com/NotaBox.jsp

