A repository for the management of artifacts in system engineering

By Edwin van Wasbeek

For Chess BV, I-business

University of Twente
Enschede - The Netherlands

A repository for the management of artifacts in system engineering

Student:
Ing. E. van Wasbeek

Committee:

Dr. Ir. K.G. van den Berg	University of Twente, department EEMCS-SE
Dr. Mr. Ir. Th.J.G. Thiadens	University of Twente, department BBT-IS&CM
M. Fillerup	Chess BV, department I-business
Dr. Ir. J.P. Zwart	Chess BV, department I-business

Version 1.0
Status: Final
Enschede, 28 August 2006

Internal project Chess: FOREST

University of Twente
Master of Business Information Technology
Specialization ICT & Innovation

PREFACE

In front of you lay my last assignment to complete my study Master of Business Information Technology at the University of Twente. Completing this study is a great achievement for me especially because of my dyslexia. This dyslexia made my study at the university allot harder for me.

This final project was executed at the software development department of Chess BV situated in Haarlem. Chess wanted me to investigate a new way of storing the design information. This question resulted in this final project with the subject to examine and design a repository to store software artifacts created during the software design activity.

First of all, I want to thank my four supervisors, Klaas van den Berg and Theo Thiadens of the University of Twente, and Mats Fillerup and Joris Zwart from Chess BV for their support during this research project.

I want to thank all the other employees of Chess BV for their hospitality and joyful working environment. My thanks especially go out to my colleagues in my room, who are Pascal, Rolf, and Ampica.

I specially want to thank Hans, Trinet, Marc, and Bas van de Peet for their hospitality. They let me stay at their house in Hoofddorp during my research project. I enjoyed the time spending with this family.

Great thanks go out to my parents for supporting and believing in me throughout the collage years. They believed that I could finish the university, even when I had some hard times. Without them I did not reach this point in my life. I also want to thank my little brother for correcting my English in this thesis.

Least, but not last I want to thank all my friends, especially my two roommates and the members of the executive board 2004-2005 of T.C. Ludica for supporting me.

Enschede, 28 August 2006

Ing. Edwin van Wasbeek

MANAGEMENT SUMMARY

The research took place at Chess BV in Haarlem. Software design at Chess (and other companies) is done in MS Word documents in which the systems decomposition is stated. MS Word documents are very simple in use but lack analysis checking features. Chess identifies this as a problem and that is why this research was started.

The problem for this research was defined as: "How to manage software design artifacts within Chess?". Design artifacts are parts of information about the system, like the structure or requirements of the system. The answer to this question led to a repository that supports the management of design artifacts, numerous of analysis checks on these artifacts, and the generation of documentation. To reach this goal, an explorative research in the literature was performed to provide a theoretical base. This theory is then used for analyzing the current situation and designing a new repository where the design artifacts are stored and managed in.

Nowadays, development projects are getting bigger and more complex. This means that large amounts of design artifacts are created and these artifacts need to be stored somewhere. This also means that when changes occur (and they do) these design artifacts need to be changed. All these problems demand facilities for storing and managing the design artifacts in a repository.

Nowadays MS Word documents and UML (Unified Modeling Language) diagrams are the primary repositories for storing and managing artifacts created in the software design activity. MS Word documents are very simplistic and do not support different analysis checks. UML diagrams however are very detailed in specifying a software design, but are complex and difficult to use. To make the use of these documents and diagrams easier, different tooling, like text editors and UML design tooling, is provided.

The repository suggested in this thesis tries to take the strengths of both storage facilities. The repository supports different analysis checks, but will be simple in use. It is also very flexible in how the design process is executed and does not have many constraints on how to use the repository. The XML (Extensible Markup Language) is used to represent the different design artifacts in the repository. XML is very flexible, human editable, and supports different database functions.

Important to understand is that this first version of the repository as it is presented has only the basic functionalities to represent the design process. Future development should add new features to the repository to make it a more advanced repository.

The repository is going to have a central role in the design process. It will be the medium where all stakeholders retrieve their information from. In general there are seven stakeholders each using the information stored in the repository in their own way. The requirements engineer, designer, programmer, and the maintenance programmer are the stakeholders that add, delete, or change information in the repository. The other three stakeholders are only retrieving information from the repository. The repository also support the generation of documentation. This means that tooling developed for the repository should be able to generate documents for each stakeholder.

The repository uses a very simple design concept. This design concept states designing as an activity of decomposing the system into sub-systems, creating requirements and relating these

requirements to these sub-systems. During each decomposition step the design process is stored by means of design motivations. These state why some particular design decisions is taken by the designer. This information can be very useful when other designers try to understand the design.

The user requirements are defined in the system, which is then decomposed into sub-systems. During this decomposition the user requirements are related to the appropriated sub-system. This sub-system is then responsible to do something with this requirement and eventually should implement this requirement. This implementation can be done by the sub-system itself, or by one of its sub-systems. During the decomposition of a system new design requirements may be created which need to be fulfilled by the system, like the system should fulfill the user requirements.

By defining the different relationships in the repository traceability is made possible. Traceability tries to follow the life of an artifact in the repository. This provides the mechanisms to execute different analysis checks and the generation of documentation. The analysis checks can be used to validate the designs or to illustrate what the impact is of a change. The generation of documentation can provide each stakeholder with its own document need to perform his task.

The conclusion of this research is that the repository supports the basic needs for defining the design process. Further development of the repository is needed to make it more mature, but the foundation is set. To make the repository usable and a success, tooling should be developed that will form the interface of the repository.

CONTENTS AT A GLANCE

1. Introduction 1
2. Research approach 9
3. System engineering 15
4. System engineering in practice 27
5. Design Artifact management 37
6. Traceability 43
7. Defining requirements 51
8. Testing 55
9. Requirements for the repository 63
10. Designing the repository 71
11. The implementation of the repository 83
12. Testing the repository 103
13. Operating the repository 111
14. Existing design storage comparisons 125
15. Conclusion and recommendations 129

Glossary 135
References 137
Appendix A. XML Repository structure tree 141
Appendix B. DTD of the repository 143
Appendix C. Example case Fiscal Tax 145

TABLE OF CONTENTS

1. Introduction 1
1.1. Description of Chess 1
1.1.1. Structure 1
1.1.2. Executive board 2
1.1.3. Service departments 2
1.1.4. Business-lines 3
1.1.5. Employees 3
1.1.6. Products and services 4
1.1.7. Market 4
1.1.8. Style 5
1.2. Reason for this research 5
1.3. Summary 7
2. Research approach 9
2.1. Problem statement 9
2.2. Goal statement 9
2.3. The scope of this thesis 9
2.4. Research model 10
2.5. Research questions 11
2.6. Limitations of this thesis 11
2.7. Overview of this thesis 12
2.8. Summary 13
3. System engineering 15
3.1. System definition 15
3.2. The system engineering process 15
3.2.1. System requirements definition 16
3.2.2. System design 16
3.2.3. Sub-system development 17
3.2.4. System integration 17
3.2.5. System installation 18
3.2.6. System evolution 18
3.2.7. System decommissioning 18
3.3. The software engineering process 18
3.3.1. Requirements engineering 19
3.3.2. Design 20
3.3.3. Design documentation 22
3.3.4. Implementation 24
3.3.5. Testing 24
3.3.6. Maintenance 25
3.4. Summary 25

4. System engineering in practice 27
4.1. General development approach 27
4.2. The system engineering process 28
4.2.1. System requirements definition 28
4.2.2. System design 28
4.2.3. Sub-system development 29
4.2.4. System integration 29
4.2.5. System installation 29
4.2.6. System evolution 29
4.2.7. System decommissioning 30
4.3. The software engineering process 30
4.3.1. Requirements engineering 30
4.3.2. Design 31
4.3.3. Design documentation 33
4.3.4. Implementation 33
4.3.5. Testing 34
4.3.6. Maintenance 34
4.4. Summary 35
5. Design Artifact management 37
5.1. Design artifacts 37
5.2. Management of requirements and design 37
5.2.1. Requirements management 37
5.2.2. Design management 38
5.2.3. Repository 39
5.3. Changes 39
5.3.1. Requirements changes 40
5.3.2. Design Changes 40
5.4. Summary 41
6. Traceability 43
6.1. Traceability definition 43
6.2. Importance of traceability 43
6.3. Stakeholders for traceability 44
6.4. Traceability directions 44
6.5. Relationships between requirements and design 46
6.6. Traceability analysis 48
6.7. Summary 48
7. Defining requirements 51
7.1. Definition 51
7.2. Types of requirements 51
7.2.1. Business requirements 51
7.2.2. Functional requirements 52
7.2.3. Non-functional requirements 52
7.3. Characteristics of testable requirements 52

7.4. Summary ... 53
8. Testing ... 55
8.1. The test process ... 55
8.1.1. Lifecycle ... 56
8.1.2. Techniques ... 57
8.1.3. Infrastructure ... 58
8.1.4. Organization ... 58
8.2. Requirements-based testing ... 58
8.3. Testing functional requirements ... 59
8.4. Testing non-functional requirements ... 60
8.5. Summary ... 61
9. Requirements for the repository ... 63
9.1. First version ... 63
9.2. Artifacts in the repository ... 63
9.3. Stakeholders ... 64
9.4. User requirements ... 64
9.4.1. Business requirements ... 65
9.4.2. Functional requirements ... 65
9.4.3. Non-functional requirements ... 66
9.4.4. Requirements matrices ... 68
9.5. Exclusions ... 69
9.6. Summary ... 70
10. Designing the repository ... 71
10.1. Repository approach ... 71
10.1.1. Example justification ... 72
10.1.2. Example ... 72
10.1.3. Reflection ... 72
10.2. Using the repository approach ... 73
10.3. Relationships ... 74
10.3.1. Dependency term and relationships ... 75
10.3.2. Requirement refinement ... 75
10.3.3. System refinement ... 75
10.3.4. Requirement assigning ... 76
10.3.5. Requirement implementation ... 76
10.3.6. Relationship explanation ... 76
10.4. Traceability in the repository ... 77
10.4.1. Types of traceability ... 77
10.4.2. Directions of traceability ... 78
10.4.3. Example of a trace in the repository ... 78
10.5. Types of analysis ... 79
10.5.1. Analysis and tooling ... 79
10.5.2. Analysis types supported by the repository ... 79
10.6. Summary ... 80

11. The implementation of the repository 83
11.1. Extensible Markup Language 83
11.2. Structure of the elements 83
11.2.1. System element 85
11.2.2. Subsystem element 87
11.2.3. Requirement element 88
11.2.4. Declaration element 89
11.2.5. Implements element 90
11.2.6. Implementation element 91
11.2.7. Mapping element 92
11.2.8. Source element 93
11.2.9. Target element 94
11.2.10. Motivation element 95
11.2.11. Description element 95
11.2.12. Title element 95
11.3. Implementation and mapping usage 95
11.4. Unique identifiers 97
11.5. File based or database repository 99
11.6. Document generation 99
11.6.1. Solution to the document problem 99
11.6.2. Information 100
11.6.3. Document users and types 100
11.7. Summary 101
12. Testing the repository 103
12.1. Tooling 103
12.2. Functional requirements 103
12.3. Non-functional requirements 107
12.4. Proof of concept 109
12.5. Summary 110
13. Operating the repository 111
13.1. Repository use 111
13.1.1. Defining the system “SY_Fiscal_Tax” 112
13.1.2. Defining the sub-systems “SS_Software” 116
13.1.3. Defining the sub-systems “SS_Documentation” 118
13.2. Rules for the use of the repository 119
13.3. Stakeholders 120
13.3.1. Requirements engineer 121
13.3.2. Designer 121
13.3.3. Project manager 122
13.3.4. Configuration manager 122
13.3.5. Programmer 122
13.3.6. Unit tester and integration tester 123
13.3.7. Maintenance programmer 123

13.4. Summary 123
14. Existing design storage comparisons 125
14.1. Unified Modeling Language diagrams 125
14.2. MS Word documents 126
14.3. Comparison 127
14.4. Summary 128
15. Conclusion and recommendations 129
15.1. Conclusion 129
15.2. Evaluating research questions 130
15.3. Recommendations and future work 131
15.4. Reflection 132
Glossary 135
References 137
Appendix A. XML Repository structure tree 141
Appendix B. DTD of the repository 143
Appendix C. Example case Fiscal Tax 145

LIST OF FIGURES

Figure 1.1 - Organization structure Chess ... 2
Figure 1.2 - Some disciplines involved in system engineering based on [58] ... 5
Figure 1.3 - Different documents and different users ... 6
Figure 2.1 - Research model ... 10
Figure 3.1 - Some disciplines involved in system engineering based on [58] ... 15
Figure 3.2 - The system engineering process [58] ... 16
Figure 3.3 - The system design process [58] ... 17
Figure 3.4 - The software engineering process based on [1][58][67] ... 19
Figure 3.5 - The requirements engineering activity [58] ... 19
Figure 3.6 - The software design activity [58] ... 21
Figure 3.7 - The testing activity [58] ... 25
Figure 4.1 - JSTD 016 system development process based on [63] ... 27
Figure 4.2 - Chess software design process ... 31
Figure 4.3 - System decomposition ... 33
Figure 6.1 - Directions of traceability relations ... 45
Figure 6.2 - Relationships in requirements and design ... 46
Figure 7.1 - The ISO 9126 quality attributes based on [49] ... 52
Figure 8.1 - The four pillars for structured testing [49] ... 55
Figure 8.2 - Hierarchy of the test sorts [49] ... 56
Figure 8.3 - TMap phases based on [49][65] ... 57
Figure 8.4 - Test-case form ... 60
Figure 9.1 - Repository stakeholders ... 64
Figure 10.1 - House build process ... 71
Figure 10.2 – Black-box of the design process ... 73
Figure 10.3 – The house build process and the artifacts ... 74
Figure 10.4 - Repository use and placement ... 74
Figure 10.5 - Mapping and implementation description ... 77
Figure 10.6 - An example trace of “Req 1” ... 78
Figure 11.1 - A simplified element tree of the repository ... 84
Figure 11.2 - Class diagram of the repository ... 85
Figure 11.3 - A basic System element ... 86
Figure 11.4 - A basic Subsystem element ... 87
Figure 11.5 - A basic Requirement element ... 88
Figure 11.6 - A basic Declaration element ... 89
Figure 11.7 - A basic Implements element ... 91
Figure 11.8 - A basic Implementation element ... 92
Figure 11.9 - A basic Mapping element ... 93
Figure 11.10 - A basic Source element ... 94
Figure 11.11 - A basic Target element ... 94
Figure 11.12 - A basic Motivation element ... 95

Figure 11.13 - A basic Description element 95
Figure 11.14 - A basic Title element 95
Figure 11.15 - A basic Target element 96
Figure 11.16 - Parent Requirement mapping with sub-requirements 96
Figure 11.17 - Parent requirement mapping 96
Figure 11.18 - Multiple sub-requirement mapping 97
Figure 11.19 - Example of the use of word identifiers 98
Figure 12.1 - Test-case for requirements [F1] and [F3] 104
Figure 12.2 - Test-case for requirements [F4] 105
Figure 12.3 - Test-case for requirements [F5] 105
Figure 12.4 - Test-case for requirement [F2] 106
Figure 12.5 - Test-case for requirements [F6] 107
Figure 13.1 - Repository use steps 111
Figure 13.2 - Defining the system 113
Figure 13.3 - Defining user requirements for the system 113
Figure 13.4 – Defining design requirements for the system 113
Figure 13.5 - Defining a sub-systems for the system 114
Figure 13.6 – Implementing the requirements on the new sub-systems 115
Figure 13.7 - Mapping the requirements on the new sub-systems 115
Figure 13.8 - Sub-systems design 116
Figure 13.9 - Defining design requirements for the sub-system 116
Figure 13.10 - Declaring a sub-system in the sub-system 117
Figure 13.11 – Mapping requirements on the new sub-systems 118
Figure 13.12 - Defining design requirements for the sub-system 118
Figure 13.13 - Defining declarations for the sub-system 119
Figure 13.14 - Repository users 120
Figure 13.15 - Repository use steps with the users 121
Figure 14.1 - UML diagrams based on [69] 125

LIST OF TABLES

Table 3.1 - User roles and attributes [4] 23
Table 3.2 - Views on the design documentation [4] 23
Table 5.1 - Requirements change causes based on [68] 40
Table 9.1 – Business, functional, and non-functional requirements matrix 68
Table 9.2 - Technical requirements and stakeholders matrix 69
Table 11.1 - Attributes of the System element 86
Table 11.2 - Elements of the System element 86
Table 11.3 - Attributes of the Subsystem element 87
Table 11.4 - Elements of the Subsystem element 88
Table 11.5 - Attributes of the Requirements element 89
Table 11.6 - Elements of the Requirements element 89
Table 11.7 - Attributes of the Declaration element 90
Table 11.8 - Elements of the Declaration element 90
Table 11.9 - Attributes of the Implements element 91
Table 11.10 - Elements of the Implements element 91
Table 11.11 - Attributes of the Implementation element 92
Table 11.12 - Elements of the Implementation element 92
Table 11.13 - Attributes of the Mapping element 93
Table 11.14 - Elements of the Mapping element 93
Table 11.15 - Attributes of the source element 94
Table 11.16 - Attributes of the Target element 94
Table 11.17 - User roles and attributes [4] 101
Table 12.1 - Non-functional requirements checklist 108
Table 12.2 - Quality attributes checklist 109
Table 14.1 - Design notation vs. criteria matrix 127

LIST OF ABBREVIATIONS

DTD	Document Type Definition
HRM	Human Resources Management
IDE	Integrated Development Environments
M2M	Machine-to-Machine
OEM	Original Equipment Manufacturer
OMG	Object Management Group
PR	Public Relations
RBT	Requirements Based Testing
SME	Small and Medium Sized Enterprises
UML	Unified Modeling Language
XML	Extensible Markup Language

1. INTRODUCTION

In this first introductionary chapter, a description of Chess BV is given to get a better impression and understanding of the company where this research project took place. For the ease of reading, Chess BV will be directed as Chess in this thesis.

The section 1.1 describes the different organizational parts of Chess. This part does not include the way system engineering is performed at Chess. This will be described in chapter 4. In section 1.2 the motivation for the execution of this research project will be elaborated. Finally, in the last section a summary of this chapter is given.

1.1. Description of Chess

Chess is a company specialized in problem solutions and services in high-end electronic products, M2M applications and critical internet applications. This is all realized from the two locations Haarlem and Best in the Netherlands. From these two locations, Chess clientele consists of many of the top 100 Dutch companies [10]. Some examples of Chess its customers are:

- Interpay;
- ING-Bank;
- Rabobank;
- ABN-AMRO;
- Shell;
- Philips;
- Siemens;
- Akzo Nobel;
- Unilever;
- Ahold.

In the following sub-sections, information about the company structure, the employees, the products and services, the targeted market and the style of Chess is provided. This information is retrieved from the Chess website [10], by informal interviews or conversations, and the summary of the policy plan 2006 [26]. This last source is classified and can not be made public for external use.

1.1.1. Structure

Chess is organized as a traditional organization structure [13] with a strict separation between policy and operation. The executive board and the business-lines are supported by three service departments as illustrated in Figure 1.1. Each of these departments are described in the following three sub-sections.

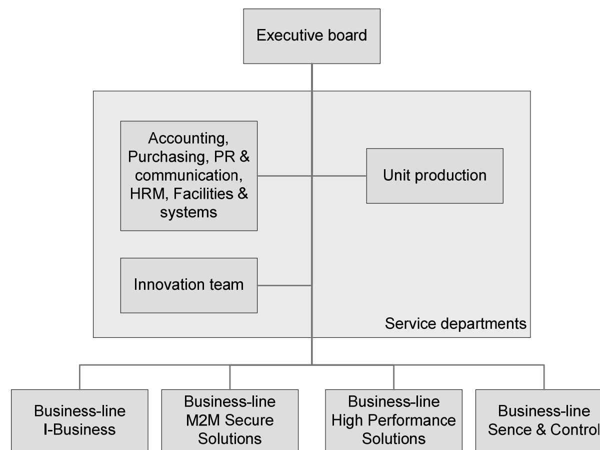

Figure 1.1 - Organization structure Chess

1.1.2. Executive board

The executive board controls the business-lines and the services departments by means of direct targeting and coordination. This is done by organizing management team consultations where representatives of all business-lines, service departments and the executive board, are present. In these consultations the current and the desired situation like planning and the financial status of projects are discussed.

The following enumeration summarizes some of the responsibilities of the executive board:

- Maintaining the relationships with the shareholders;
- Organizational and business development;
- Portfolio management;
- Treasury and financing;
- Maintaining internal and external communication.

1.1.3. Service departments

The service departments support and partially execute the company policy directed by the executive board. They also form the link between the executive board and the business-lines and support the business-lines in their execution of their tasks.

The following enumeration summarizes some of the responsibilities of the service departments:

- *Accounting*: supports information provision, finance, administration and justification;
- *PR and communication*: maintains the Chess style, image, reputation, website and public relations tools;
- *Human Resource Management (HRM)*: deals with laws, administration, recruitment and the selection of employees;
- *Purchasing*: does planning, purchasing, control, and the communication with the business-lines;
- *Facilities & systems*: manages the buildings, decoration, infrastructure, communication, hosting and desktop management;

- *Unit production*: is involved in the production and testing of the products designed by the business-lines;
- *Innovation team*: is responsible for the creation of new products or services and is in close contact with the business-lines.

1.1.4. Business-lines

The business-lines are responsible for the operation of the primary processes of Chess (that means, the business lines are the primary processes of Chess). These business-lines are market oriented, are supported by the service staff and operate on a 'profit & loss' principle. The business-lines operate as profit centers and are entirely responsible for and focused on their own profit [17]. They develop products, execute projects, and are responsible for customer relations.

The main activities of the business-lines are:

- Sales;
- Delivery;
- Product and service development;
- Business-line communication;
- Operational HRM;
- Operational PR;
- Planning and control.

As illustrated in Figure 1.1 there are four business-lines to meet market demand. These business-lines are:

- *I-Business*: aims at multi-channel software solutions and company critical systems;
- *M2M Secure Solutions*: The engineering process of M2M Secure Solutions characterizes itself by multidisciplinary development of casings, hardware, system software, embedded- and server application software;
- *High Performance Solutions*: aims at the development of innovative and complex embedded systems;
- *Sense & Control*: is responsible for the management and execution of different projects. Sense & Control also produces and manages the life-cycle of different systems and products.

1.1.5. Employees

The employees of Chess have a high technical and educational level but differ in their professional backgrounds. Most of the employees have a couple of years of technical and system engineering experience. Besides experienced engineers, Chess employs junior engineers, who have the potential to become highly skilled technical consultants. Chess provides specific training opportunities and workshops for her employees to cope with the ever changing market and knowledge demand.

To keep all employees of a business-line up to date with the current and future projects, Chess organizes informal meetings. These meetings are called 'Chess talks'. During these talks, every project is introduced and then discussed. These 'Chess talks' are great way of knowledge and information sharing among colleagues.

At the moment Chess consists of about 140 employees and the prediction is that at the end of 2006 this will grow to about 170 employees. The binding factor between the Chess employees is the profession of system engineering and joy of participating in a group of inspiring colleagues and challenging projects.

1.1.6. Products and services

Chess has different ways of delivering their products and services. Chess delivers electronic products directly 'off the shelf' or to the customer's specification. This is done in project form or as part of an existing solution. Moreover, Chess provides sophisticated software solutions as projects, as turn-key solutions or as part of M2M solutions. Chess also delivers production logistics, hosting, operations, and maintenance and support services for operational systems. Applications which are developed by Chess are:

- PinLinq (wireless PIN payment systems);
- Parking systems;
- Digital Rights Management (DRM);
- MiniTix (ChipKnip application for the internet);
- Shareholder Voting System (SVS);
- E-Travel (online vacation portal).

1.1.7. Market

Chess primarily aims on Dutch companies, generally with international markets. As described in the introduction of this chapter Chess supports some major international companies like ING, Rabobank, Siemens, and Philips (see the introduction of this section for more companies).

Besides the direct approach by the executive board and the account managers of the business-lines to attract new customers, much is expected of Chess its partners like ASML (www.asml.nl) and VDO Dayton (www.vdodayton.com), which have their own markers. Chess wants to be dominantly present in the following markets:

- The financial market;
- The entertainment or new media market;
- OEM product suppliers;
- Retail markets;
- Office program suppliers;
- Big SME's.

1.1.8. Style

Chess maintains a ‘high-tech’ image based on innovation and the use of the latest technologies. This gives Chess the edge to be the company for constituents and talented employees. Chess uses a ‘can do’ mentality when serving their customers. This is reached by a flexible organization and effectively arranging the processes within Chess.

How Chess is perceived by the labor market is very important for the company. Chess should be seen as a special and strong company where, besides healthy financial results, employees have a central position. In other words, Chess should be an enjoyment to work for.

The style Chess want to reflect to its customers and to its employees are described with phrases like:

- Vision;
- High-tech;
- Innovative;
- Result-oriented;
- Carefully and responsible;
- Communicative;
- Surprising (“obtaining an eleven instead of a ten”).

1.2. Reason for this research

System engineering includes different engineering branches that are needed for building a system [58]. These branches are software engineering, hardware engineering, electrical, and other engineering specialties. This is illustrated in Figure 1.2. Each engineering field is responsible for different parts of the system which are called sub-systems. During the system engineering process, these sub-systems are then (again) decomposed into sub-systems and so on.

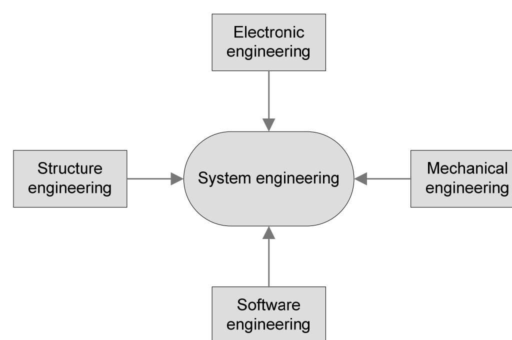

Figure 1.2 - Some disciplines involved in system engineering based on [58]

Each engineering field generates different types of documentation, which is used internally in an engineering field, but may also be used by other engineering fields. The use of the different documentations by multiple persons is illustrated in Figure 1.3. Communication between these engineering fields becomes then very important and can become very difficult in big projects when a large number of people are involved [38]. Especially with the documentation usage between

different engineering fields it is important to have up-to-date and correct documentation for all engineering fields. Otherwise the different sub-systems of the system may contain errors, may be unstable, or may not be compatible.

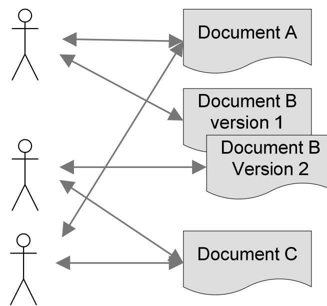

Figure 1.3 - Different documents and different users

Chess understands the importance of maintaining the system engineering artifacts. Although, there are at the moment no major problems in managing these artifacts at Chess, improving this activity should be considered. Improving the management of these artifacts may result in:

- Cost reductions;
- Efficiency;
- Reduction of the time to market;
- Better quality of the software;
- Better documentation.

Most solutions dealing with change management are based on the software engineering process [38]. This is just a small part of the field of the system engineering. There are at the moment several tools [39], like IBMs Rational Suit [30] and Goda Software’s AnalystPro [42] in the market which claim to help developers to manage the software artifacts.

Traditional change management methods attempt to formalize all activities concerning changes in projects [38]. Each activity must be requested, numbered, accepted, implemented and tested to prove it reached its goals. All those activities may be facilitated by specialized document flow tools, yet they still require a human to process the information in each phase. Formal methods are therefore expensive and tedious for their users.

These are some of the reasons why chess does not use these kinds of tools. The most important reason not to use these tools is that most of these tools do not fit Chess its needs. Chess does not primarily use UML [20] for designing the system which is used in most tooling. Instead Chess uses a natural language or text people use in daily life, supported by images. These images however can be UML oriented.

The documentation is written in one or more MS Word documents that are, for most projects, maintained by one person. This thesis could be an example of a design document that is used by Chess during system engineering.

The problem, or disadvantage, of keeping the information in these types of documents, is the way the information is stored and can be accessed. All artifacts are stored as textual items with no possibility to relate them to each other explicitly by doing this in text. There is no possibility to automatically check all relations and thus errors can not be found by automatic analysis. This makes maintaining these documents very hard. When changes are introduced, managing these documents becomes especially hard. It is difficult to track these changes and their impact on other decisions.

The description above is primarily focused on the situation at Chess, but this does not mean that Chess is the only one with this particular problem. It is the experience of Chess and also its customers and partners that software is build in the way described above. Most of these customers and partners do not use the different tools, but relay on MS word documents.

1.3. Summary

This chapter provided information about Chess and described the problem that was the reason for this research project. Chess is a diverse company which produces software and hardware products and provides services. Chess is situated in Haarlem and Best in the Netherlands and has some big companies as clients. Some of these companies are:

- Interpay;
- ING-Bank;
- Rabobank;
- Unilever;
- Ahold.

The organization structure of is traditional with on top the executive board supported by the service departments. The primary process is done by the business lines, which are:

- I-Business;
- M2M Secure Solution;
- High Performance Solution;
- Sense & Control.

Chess maintains a 'high-tech' image based on innovation and the use of the latest technologies. It tries to serve the client in every way possible by adapting to the clients whishes. This is reached by effectively arranging the processes and having highly skilled employees.

The reason for this research comes from the lack of good and useable tools for storing the system engineering artifacts. At the moment MS Word documents are used to store the design process. These types of documents are very limited in used and generate some problems when used and maintained. The main disadvantage is that the documentation is used as central storage and for communication purposes with the different stakeholders. Each of these stakeholders needs its own information which means that different documents need to be written. As a result, all documents needs to be rewritten when changes occur. An extra disadvantage is that it may

happen that different versions of documents are used by the different stakeholders. Eventually, the system may contain errors and may not satisfy the requirements of the customer. The conclusion is that the system engineering artifacts need to be managed

Chess understands the importance of maintaining the system engineering artifacts. Improving the management of these artifacts may result in:

- Cost reductions;
- Efficiency;
- Reduction of the time to market;
- Better quality of the software;
- Better documentation.

2. RESEARCH APPROACH

In the previous chapter the motivation was given for this research project. In this chapter the structure of this research project is given. The book of Verschuren and Doorewaard [66] is used to define this structure.

The sections 2.1 and 2.2 present the problem and the goal statements for this research project. These statements formulate the reason why this research project is taken place. In section 2.3. the reduction of the scope of this research thesis is explained. To answer the problem statement and achieve this projects goal, a research model is drawn in section 2.4 to visualize and structure the research approach. The research questions which are used to reach the projects goal are derived form the research model. These research questions are described in the section 2.5 and will be used to collect the necessary information needed for this research project. In section 2.6 some principles and limitations are given to narrow the research. In the final section the overview of this is presented.

2.1. Problem statement

The problem statement for this research project is:

How to manage system engineering artifacts within Chess?

2.2. Goal statement

As the result of the problem statement, the goal for this research is:

Develop a repository for the management of artifacts in system engineering.

The key terms in this goal statement are repository, management, artifacts, system engineering and will be discussed in detail in this thesis.

2.3. The scope of this thesis

The goal stated in the previous section is very general and large-scaled. It states a general goal for multiple and different research projects or thesis. That is why this thesis will only focus on a small part of this research goal. The scope of this research thesis needs to be narrowed down to have a positive, usable and reachable result. Trying to do the whole goal in one project will be very hard mainly duo the following two reasons:

- *Width of the research*: The goal of this thesis tries to develop a repository for the whole system engineering field. As described in section 1.2 of the previous chapter, the engineering field is very large. This research project would then become too extensive;

- *Time constraint*: The time for this research project is about six months which is not very long. Because of the size of the research project, it would be impossible to perform this research in the available time.

Because of these two reasons, the research projects scope will be reduced to only the software design activity of the software engineering process. This means that the research itself focuses on the management of software design artifacts. Defining what the software design artifacts are is a part of this thesis. The reduction of the scope automatically reduces the problem and goal area.

The problem statement of this thesis is:

How to manage software design artifacts within Chess?

The goal of this thesis is:

Develop a repository for the management of artifacts in software design.

The goal of the research project, which is stated in the previous section, will be incrementally reached in different research projects. This means that like this thesis, other sub-projects will be created to add piece by piece a solution to reach the research projects goal. The success of the research project depends on the success and results of this thesis. When the outcome, conclusions, and recommendations of this thesis are negative or unsuccessful, the assessment of the whole research will be negative and thus not applicable for further investigation.

2.4. Research model

A research model indicates how a research project is shaped and how the end result is delivered. Figure 2.1 demonstrates this research model for this research project and should be read from left to right. The model is developed with the diminishment of the scope of the research project in mind.

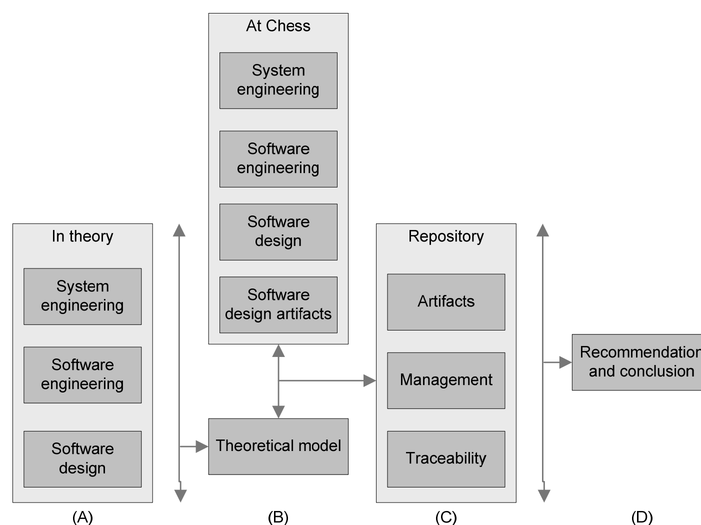

Figure 2.1 - Research model

Translating this model in to words looks like: (A) As a result of a theoretical exploration of the system engineering process, the software engineering process, and the software design process, (B) a theoretical model is formed which is used to analyze the current situation of Chess. (C) These results are then used as a foundation for the description and design of the repository. This includes a study on management, traceability and repositories and how this can be used for the software design repository. (D) Finally a recommendation and conclusion is given regarding the proposed repository.

2.5. Research questions

The following research questions are obtained from the research model in Figure 2.1. These research questions will be used to gather more information about the key terms mentioned in the goal statement. The questions are formulated with the diminishment of the scope in mind.

[R1] What is system engineering in theory and in practice?
[R2] What is software development in theory and in practice?
[R3] What is software design in theory and in practice?
[R4] What are the artifacts of software design in theory and in practice?
[R5] Why is management of software design necessary?
[R6] What is traceability?
[R7] How can traceability be used in management of software design?
[R8] What trace information is needed for the management of software design?
[R9] What is a repository?
[R10] What are the requirements for such a repository?
[R11] Which information needs to be stored in a repository?
[R12] How can trace information of software design be saved in a repository?
[R13] What is the design of the repository?
[R14] How to implement and test this repository?
[R15] How can trace information of software design saved in a repository be used for document generation?

2.6. Limitations of this thesis

Because of the constraints mentioned in section 2.3, the research project contains the following limitations:

- The focus for this research is, as described in section 2.3, only on software design activity of the software engineering process, and thus leaves all other activities of system and software engineering out of consideration. This is mainly done duo to the time constraint and interest field of Chess. The outcome of this research can be used as a basis for a repository to manage the artifacts in system engineering;
- The outcome of in this research project is to design and recommend a structure for a repository, but not to design and program the software or compiler to automate the use of the repository;

- Only the basic and necessary functionalities are implemented in the first version of the repository. This means that the requirements, design decisions and the relations between these items need to be stored in the repository. The requirements of this first version are described in section 9.

2.7. Overview of this thesis

In this section the overview or structure of this thesis is given. This thesis is based on the research questions stated in section 2.5. Each chapter answers one or more of these questions.

- *Chapter 1*: gives the introduction about chess. It described among other things the structure, markets, employees of Chess. It also provided the reason for this research project;
- *Chapter 2*: describes the structure and the approach of this research project. This section is part of this chapter;
- *Chapter 3*: describes the system and software engineering as it is stated in the literature. The section about software design gets special attention because this is the scope of this thesis;
- *Chapter 4*: uses the information gathered in the previous section to examine the system engineering process at Chess. The section structure is kept the same as that of chapter 3 to describe each part of the system engineering process;
- *Chapter 5*: describes what artifact management is and why it is needed. It provides what artifacts need to be managed and what the causes are for these artifacts to change;
- *Chapter 6*: explains the concept of traceability. It describes the importance, the stakeholders, the directions, the relations, and the analysis types of traceability. This chapter will later be used to define the different relations in the repository;
- *Chapter 7*: states the different stakeholders, requirements and quality attributes of the repository. The requirements and quality attributes will be used for testing and accepting the repository design;
- *Chapter 8*: provides the design of the repository. It describes the design approach which is used by chess. The information of this chapter will be used as the foundation in the repository. Also, the different traceability relations and analysis checks mentioned in chapter 6 that are used by the repository are described;
- *Chapter 9*: describes the implementation of the repository. It starts of with the motivation why XML is used as language for the repository. Then the structure with the different elements and attributes is provided;
- *Chapter 10*: defines the repository test-cases. The chapter begins with a description of the test process and how testing should be done. After this, the functional requirements and quality attributes are checked against the repository;
- *Chapter 11*: describes the operation of the repository. It provides a detailed description of how a designer could use the repository during his design activity;
- *Chapter 12*: In chapter 12 other repositories are examined and compared with the repository proposed in this thesis;

- *Chapter 13*: states the conclusion, evaluation of the research questions, recommendations and future work, and a reflection of this thesis;
- *Appendixes*: provide background information used by the different chapters.

2.8. Summary

This chapter provided information about the information about the goal and approach of this research project. The goal of this research project is:

Develop a repository for the management of artifacts in system engineering.

The goal of this thesis differs from the goal of the research project because of the time limitations and the width of the scope of this goal. The goal for this thesis is:

Develop a repository for the management of artifacts in software design.

To reach this goal a research model was provided. This research model starts with collecting information from the literature about system engineering, software engineering, and design activity. This literature study is used to analyze the situation at Chess. After this analysis information about artifacts, management, and traceability is gather which is used to design the repository.

From this model some research questions arise, which are used to research this thesis its goal. Also, some limitations of this research are provided. These limitations are:

- The limitation that this thesis is only focusing on the software design activity;
- Only a design of the repository will be provided (That means no tooling shall be written for the repository);
- Only the basic and necessary functions will be implemented in the repository.

3. SYSTEM ENGINEERING

This chapter describes the system engineering and the software engineering process. These descriptions are based on the book of Sommerville [58], and are sometimes supported by other references. The information provided in this chapter will be regarded as correct and will be used in chapter 4. to analyze the way Chess performs its system engineering process. Section 3.1 describes the definition of the term system. Second 3.2 describes system engineering process and section 3.3 describes the software engineering process. The last section will give a summary of the information given in this chapter.

3.1. System definition

Before describing what system engineering exactly is, it is important to know what a system is. The definition of a system is a collection of components organized to accomplish a specific function of set of functions [31]. To be more specific, a system is any organized assembly of personnel, resources and procedures united and regulated by interaction or interdependence to accomplish a set of specific functions [32]. The first definition is only mentioning components, where the second definition is concerned with persons, resources and procedures. This is much wider then only hardware or software components.

3.2. The system engineering process

System engineering is the activity of specifying, designing, implementing, validating, deploying, and maintaining systems as described in de design definition. System engineers are not just concerned with software, but also with hardware and the system's interactions with its users and its environment. System engineers must think about the services that the system provides the constraints under which the system must be build and operate, and the ways in which the system is used to fulfill its purpose.

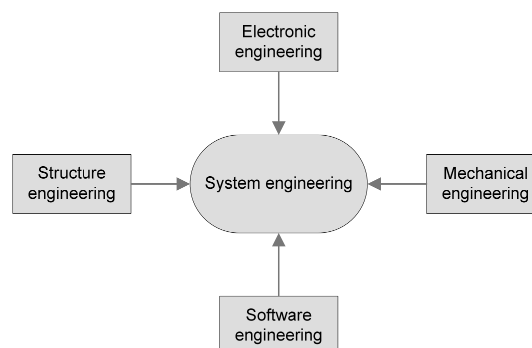

Figure 3.1 - Some disciplines involved in system engineering based on [58]

System engineering is an interdisciplinary activity involving teams drawn from various backgrounds. Figure 3.1 shows an example of four different disciplines involved in system engineering, but there are dozens of other engineering fields.

The system engineering process exists of seven activities which are illustrated in Figure 3.2. This model was an important influence on the model of the software engineering process that is described in section 3.3. Take note that the system engineering, like all other engineering processes are interactive and can contain loop-backs to a previous activity.

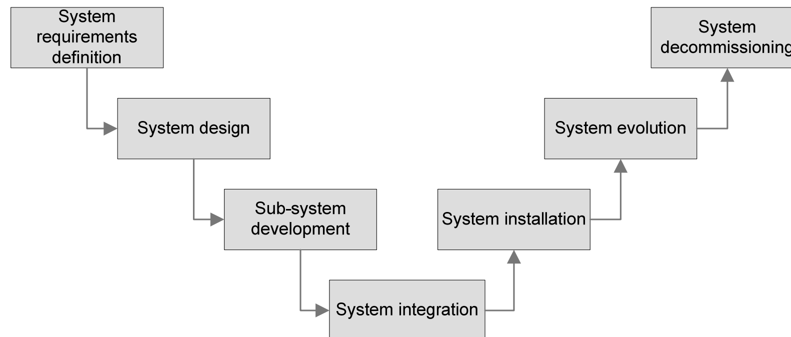

Figure 3.2 - The system engineering process [58]

In the following sub-sections, the seven activities of the system engineering process are described. Each sub-section relates to an activity from Figure 3.2.

3.2.1. System requirements definition

The system requirements definition specifies what the system should do (its functions), its essentials, and desirable system properties. An important part of the requirements definition phase is to establish a set of overall objectives that the system should meet. These should not necessary be expressed in terms of the system its functionality, but should define why the system is being produced for a particular environment.

Creating system requirements definitions involves consultations with the system its customers and end-users. The requirements definition phase usually concentrates on delivering three types of requirements:

- *Functional requirements*: are the basic function the system must provide at an abstract level;
- *Non-functional requirements*: or quality attributes are the non-functional system properties of the system by which its quality will be judged by some stakeholder or stakeholders;
- *Characteristics that the system must not exhibit*: specify what the system must not do.

3.2.2. System design

In the system design process, systems may be modeled as a set of sub-systems and components with relationships between these sub-systems and components. These are normally illustrated graphically in a system architecture model that gives the reader an overview of the system its organization.

At this high-level of detail, the system is decomposed into a set of interacting sub-systems. Each system should be represented in a similar way until it is decomposed into functional

components. The functional components, when viewed from the sub-system's perspective, provide a single function.

System design is concerned with how the system functionally is to be provided by the components of the system. The activities of this process, as illustrated in Figure 3.3, are:

- *Partition requirement*: is the activity of analyzing the requirements and organize them into related groups;
- *Identify sub-system*: is the activity of finding sub-systems that can individually or collectively meet the system requirements;
- *Assign requirements to sub-systems*: is the activity of allocating the requirements to sub-systems;
- *Specify sub-system functionality*: is the activity of defining functionalities provided by each sub-system;
- *Define sub-system interface*: is the activity of specifying the interfaces that are provided and required by each sub-system.

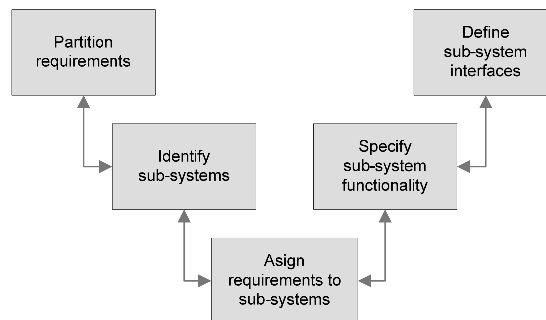

Figure 3.3 - The system design process [58]

The double-ended arrows in Figure 3.3 imply that there is a lot of feedback and iteration from one activity to another. When problems and new ideas arise, the previous activity or activities need to be redone.

3.2.3. Sub-system development

During the sub-system development, the sub-system identified in the system design activity, is build and implemented. This may involve starting another system engineering process for individual sub-systems or, if the sub-system is software, starting a new software engineering process. In this new process, all sub-systems and components are build according to the specifications of the design.

3.2.4. System integration

During the system integration process, the independently developed sub-systems and components put to gather to make up a complete system. This system integration can be done by using a 'big bang' approach, where all the sub-systems are integrated at the same time, or an incremental approach can be used, where the sub-systems are integrated one at a time.

Once the sub-systems and components are integrated in the entire system, extensive system testing is take place. The attentions of these tests are on testing the interfaces between the sub-systems and components and the behavior of the entire system. An acceptance test is executed by the customer to accept the system for use.

3.2.5. System installation

After the system is fully tested en found ready for use, the system needs to be installed at the customer. During the installation, all different engineering fields are brought together to install the system and finalize the system development process. This finalizing of the system development process means that each engineering filed helps and supports the customer in the use of the system. After this installation activity, the customer is ready to use the system in its organization.

3.2.6. System evolution

The system evolution process is responsible for keeping the system up-to date and bug free. During the life of systems which ends at the decommissioning of the system, changes and errors may occur. This means that requirements need to be changed or created, designs need to be redesigned and that the system needs to be rebuild. Possible reasons for a system change are:

- The hardware may change;
- The use of the system may change;
- The environment of the system may change;
- The software may be obsolete and need to be changed.

3.2.7. System decommissioning

System decommissioning is the activity of taking the system out of service after the end of its useful operational lifetime. This could for hardware systems be disassembling and recycling materials. Still valuable components like software or hardware can be reused for another system

3.3. The software engineering process

The software development process is an organized set of activities performed to translate user needs into software products [54]. It is the process of the initial birth of software to its death (Software life cycle).

One of the first introductions of software engineering models was the waterfall model. It captures the ideal process of software engineering in one model and is a result of the system engineering model shown in Figure 3.2. Both models have almost the same activities and the same order of these activities.

The waterfall model derives its name due to the cascading effect from one activity to the other as show in Figure 3.4. In this model each activity has a well defined starting and ending point, with identifiable deliveries to the next activity [1][58][67].

In the following sub-sections the software engineering model is discussed in detail. Like the system engineering process described in the previous section, the software engineering process is an interactive process with loop-backs to previous activities. Sub-section 3.3.3 is not really an

activity of the software engineering process, but focuses more on the documentation of the design activity.

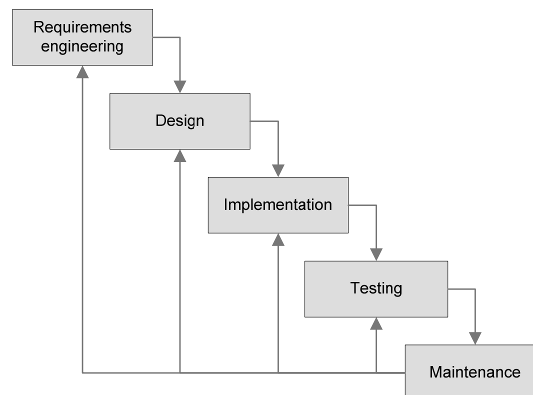

Figure 3.4 - The software engineering process based on [1][58][67]

3.3.1. Requirements engineering

Requirements engineering or software specification is the activity of understanding and defining what services are required from the system (that means, the requirements) and identifying the constraints on the system its operation and development. At the end of this activity a requirements document is created, which identifies the specification of the system.

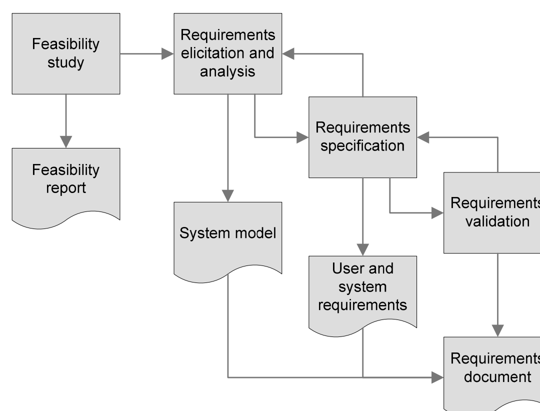

Figure 3.5 - The requirements engineering activity [58]

The requirements engineering activity is illustrated in Figure 3.5. The model suggests that the sub-activities in the requirements engineering activity are simply carried out in a strict sequence, but this is not the case. Throughout this activity, and even beyond this activity, existing requirements may change and new requirements may be introduced, which need to be specified and validated. Therefore the sub-activities of analysis, specification and validation are interleaved. The sub-activities of the requirements engineering are:

- *Feasibility study*: is making an estimate of whether the identified user needs may be satisfied using current software and hardware technologies. The study considers whether

the proposed system will be cost-effective from a business point of view and whether it can be developed within existing budgetary constraints. The result should decide whether to go ahead with a more detailed analysis;

- *Requirements elicitation and analysis*: is the process of deriving the system requirements through observation of existing systems, discussion with potential users and procurers, task analysis and much more. This may lead to the development of one or more system models and prototypes. These help the analyst understand the system to be specified*;*
- *Requirements specification*: is the activity of translating the information gathered during the analysis activity into a document that defines a set of requirements. Two types of requirements may be included in this document. User requirements are abstract statement of the system requirements for the customer and end-users of the system. System requirements are a more detailed description of the functionality to be provided by the system;
- *Requirements validation*: checks the requirements for realism, consistency and completeness. During this process, errors in the requirements documents are inevitably discovered. The requirements document must then be modified to correct these problems.

3.3.2. Design

A software design is a description of the structure of the software to be implemented, the data which is part of the system, the interfaces between system components and, sometimes, the algorithms used. The design activity is the activity of designing the architecture, components, interfaces, and other characteristics of a system or component [31]. This design activity, shown in Figure 3.6, involves adding formality and detail as the design is developed with constant backtracking to correct earlier designs. The design process may involve developing several models of the system at different levels of abstractions.

The input of the design phase is the requirements document created in the requirements engineering phase, describing 'what' the system should do. The outputs of the design activity are design documents, describing 'how' the system is to achieve the requirements [67].

Figure 3.6 may suggest that the activities of the design process are sequential, but in fact like all activities of system engineering, the activities are interleaved. The model described in this thesis, is a general model of the design process and may in practice differ. Some of these design activities are executed in the implementation activity or not even at all. The sub-activities of the design activity, shown in Figure 3.6, are:

- *Architecture design*: the systems is decomposed into sub-systems and the relationships between these sub-systems are identified and documented;
- *Abstract specification*: for each sub-system an abstract specification of its services and the constraints under which it must operate, is produced;
- *Interface design*: for each sub-system, its interface with other sub-systems is designed and documented. This interface specification must be unambiguous as it allows the sub-system to be used without the knowledge of the sub-system its operation;

- *Component design*: services are allocated to components and the interfaces of these components are designed;
- *Data structure design*: the data structures used in the system implementation are designed in detail and specified;
- *Algorithm design*: the algorithms used to provide services are designed in detail and specified.

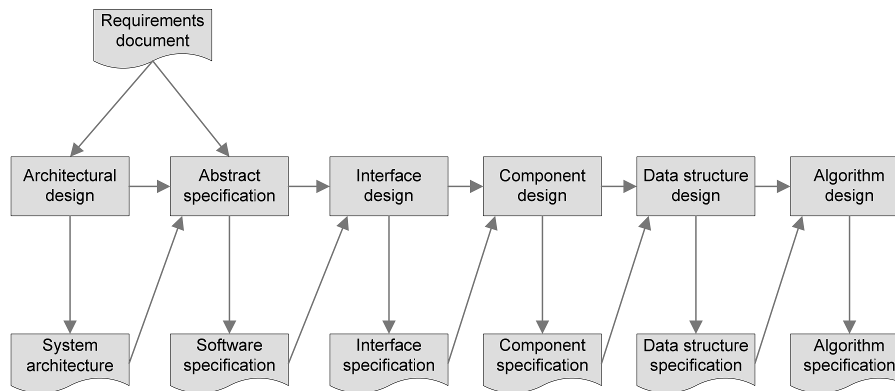

Figure 3.6 - The software design activity [58]

The most important thing to understand about designing is that there really is no universal design method. The design process is a creative one, which depends on the expertise of the designers as critical success factor. However, there are some guidelines and methods to help with the design process [67]. There are many ways to decompose a system in components. Some of these decompositions may not be as desirable as others.

Important features for the design of a system are maintenance and re-use. Maintenance describes the effort needed to maintain the system whereas re-use indicates if the system contains parts that can be reused in other or in the same system. These features can, in some sense, be used as a measure of the quality of the system (see also the quality attributes in section 8.4). Five guidelines that are related to these features are:

- *Abstraction*: means that the concentration is on the essential features and ignore (or abstract from) details that are not relevant at the level currently working on;
- *Modularity*: is the way sub-systems and components are created, grouped (cohesion) and related (coupling) to each other;
- *Information hiding*: is the principle that each sub-system or component has a secret, which it hides from other sub-systems or components (for example, a component that stores values a particular way, but other components do not know this way. It only knows that the component stores values);
- *Complexity*: refers to attributes of the software that affect the effort needed to construct or change a piece of software;
- *System structure*: indicates the way a set of components and their mutual dependencies are structured.

Knowing the properties of or the guidelines for a good system, the system needs to be decomposed into smaller and more detailed parts. There are many design methods available for decomposing a system. These design methods generally consist of a set of guidelines, heuristics and procedures on how to go about designing a system. Notations, generally in the form of graphical representations, are used to express the result of the design process. Together these notations provide a systematic means for organizing and structuring the design process and its products.

3.3.3. Design documentation

The documentation of the design activity serves different users, who have different needs [4]. A proper organization of the design documentation is therefore very important. Each user must be able to quickly find correct and up-to-date information in such a way that it can be used by the user.

The users using the design documentation can be grouped in one or more roles
There are seven different roles for the design documentation:

- *Project manager*: needs information to plan, control and manage the project. He must be able to identify each system component and understand its purpose and function. This is needed to make cost estimates and define work packages;
- *Configuration manager*: needs information to be able to assemble the various components into one system and to control changes;
- *Designer*: needs information about the function and the use of each component and its interfaces to other components;
- *Programmer*: must know the algorithms to be used, the data structures, and the kinds of interactions with other components;
- *Unit tester*: must have detailed information about components (such as algorithms used), required initialization, and data that is needed by the components;
- *Integration tester*: must know about relations between components, the functions, and use of the components involved;
- *Maintenance programmer*: must have an overview of the relations between components. He must know how the user requirements are realized by the various components. When the maintenance programmer needs to realize changes, he assumes the role of the designer.

Each of these user roles needs specific information about the sub-systems or components they are interested in. The IEEE Standard 1016 [4] identifies that each sub-system has ten relevant attributes that can give the user role the needed information about the sub-system or component and are minimally required in each project. These ten attributes are:

- *Identification*: the component’s unique name for reference purposes;
- *Type*: the kind of component, such as subsystem, procedure or file;
- *Purpose*: what is the specific purpose of this component;

- *Function*: what does the component accomplish;
- *Subordinates*: which components the present entity is composed of;
- *Dependencies*: a description of the relationships with other components;
- *Interface*: a description of the interaction with other components;
- *Resources*: the resources needed to let the component function;
- *Processing*: a description of algorithms used, way of initialization and dealing with exceptions;
- *Data*: a description of the representation, use, format and meaning of internal data.

When the seven user roles and the ten attributes are put together in a matrix, a clear view of what information (attribute) is needed by witch user role is created. The matrix is presented in Table 3.1.

Sub-system or component attributes	**User roles**						
	Project manager	**Configuration manager**	**Designer**	**Programmer**	**Unit tester**	**Integration tester**	**Maintenance programmer**
Identification	X	X	X	X	X	X	X
Type	X	X				X	X
Purpose	X	X					X
Function	X		X			X	
Subordinates	X						
Dependencies		X				X	X
Interface			X	X	X	X	
Resources	X	X				X	X
Processing				X	X		
Data				X	X		

Table 3.1 - User roles and attributes [4]

Design view	**Attributes**	**User roles**
Decomposition	Identification, type, purpose, function, subcomponents	Project manager
Dependencies	Identification, type, purpose, dependencies, resources	Configuration manager, maintenance programmer, integration tester
Interface	Identification, function, interfaces	Designer, integration tester
Detail	Identification, computation, data	Unit tester, programmer

Table 3.2 - Views on the design documentation [4]

As Table 3.1 illustrates, different information is needed for different user roles. It is not necessarily advantageous to incorporate all attributes into one document for all user roles to be used. This could lead to a surplus of information for each user role. However providing each user role with its own documentation is time consuming work and is hard to keep consistent and up to date.

IEEE 1016 groups the attributes in four clusters to compromise the documentation problem mention above. The clusters are arranged so that most user roles need information from only one cluster, while these clusters contain a minimum amount of superfluous information for that user role. Each cluster also contains its own view on the design. Table 3.2 shows the design views and the related attributes and user roles. The different views are explained in the following summarization.

- *Decomposition description*: describes the decomposition of the system into components. Using this description may follow the hierarchical decomposition and as such describes the various abstraction levels;
- *Dependencies description*: describes the coupling between components and sums up the resources needed. It is also possible to derive how parameters are passed and which common data are used. This information is helpful when planning changes to the system and when isolating errors or problems in resource usage;
- *Interface description*: describes how functions are used. This information constitutes a contract between different designers and between designers and the programmers;
- *Detail description*: describes the internal details of each component; which are needed by the programmers. This information is also useful when composing component tests.

3.3.4. Implementation

The result of the implementation activity is an executable program conform the technical specifications provided in the design activity. In this activity the focus lies on the components and their specifications. It is sometimes necessary to introduce an extra design process due to size between the component specification and the executable code.

Programming is a personal activity and there is no general process that is usually followed. Each programmer may build the sub-systems in its own way, although there are some programming standards [21], which can help standardizing the code.

3.3.5. Testing

Software validation or, more generally, verification and validation is intended to show that the system performs conform its specification and that the system meets the expectations of the customer buying the system. Verification is testing if the translation between subsequent processes is correct. Validation is checking if the project is still on the right track regarding the fulfillment of the user requirements.

The testing activity is not an activity following the implementation phase. Figure 3.4 may give the impression that testing is not necessary until after the implementation, but that is wrong. Testing happens in every activity of software engineering. The earlier errors are detected, the

cheaper and easier it is to correct them. Correcting these errors is represented by the loop-back arrows in Figure 3.4. The testing activity exists of three sub-activities as shown in Figure 3.7. These testing activities are:

- *Component (or unit) testing*: the individual components are tested, without other components, to ensure that they operate correctly;
- *System testing*: the components are integrated in the system and tested on the interaction between the components;
- *Acceptance testing*: the system is tested with data supplied by the customer rather than with simulated data.

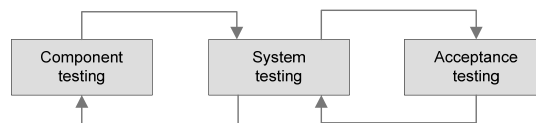

Figure 3.7 - The testing activity [58]

3.3.6. Maintenance

The maintenance activity is responsible for all activities needed to keep the system operational after it has been delivered to the user. Normally (although not necessary) the maintenance activity is the longest life cycle phase. Maintenance involves correcting errors which were not discovered in earlier stages of the life cycle, improving the implementation of them system its units and enhancing the system its services as new requirements are discovered.

As explicitly illustrated in Figure 3.4, the maintenance activity has loops back to previous activities. Sometimes when critical errors or enhancements are discovered, the software engineering process needs to be redone to preserve a correct working system.

3.4. Summary

This chapter described the system engineering process, which models the process of designing and building a system. A system is any organized assembly of personnel, resources and procedures united and regulated by interaction or interdependence to accomplish a set of specific functions [32]. This collection of different elements also returns in different engineering fields where system engineering is a part of. Examples of system engineering fields are structure-, electronic-, personnel-, mechanical-, and software engineering.

System engineering consists of multiple activities which provide a step by step model for designing and building a system. It is important to understand that the system engineering process is an interactive and incremental process with different loop-backs to previous activities. The activities of system engineering are:

- System requirements definition;
- System design;
- Sub-system development;

- System integration;
- System installation;
- System evolution;
- System decommissioning.

The focus of this thesis is the software engineering field. Software engineering is responsible for the software used in the system and looks like the system engineering field. Like the system engineering filed, the software engineering process is an interactive process with loop-backs. The software engineering field consists of:

- Requirements engineering;
- Design;
- Implementation;
- Testing;
- Maintenance.

The documentation of the design activity is very important for different stakeholders. These stakeholders are grouped in different user roles, which all have a specific interest in the design documentation. These stakeholders are:

- Project manager;
- Configuration manager;
- Designer;
- Programmer;
- Unit tester;
- Integration tester;
- Maintenance programmer.

Creating documentation for each stakeholder means, that for every user role a document should be provided which satisfy their specific interest. This is not realistic in practice and that is why the user roles are grouped in four views, which combine the user roles with similar interests. This provides four different documents which contain a minimum amount of superfluous information for that user role. This reduces the amount of work needed to produce all the documentation needed for the specific user roles.

4. SYSTEM ENGINEERING IN PRACTICE

This chapter describes the system engineering process performed at Chess. This is done to get a better understanding how Chess executes the system engineering process. Section 4.1 describes the chess development approach. The structure of the rest of this chapter is the same as that of chapter 3. This means that second 4.2 describes system engineering, section 4.3 describes software engineering and the last section a summarization of this chapter.

4.1. General development approach

Before describing the way Chess performs the system engineering process, it is important to know that there is not really a standard approach for Chess for building systems. Although there is no standard approach, Chess tries to use well known standards like the JSTD 016 [63] standard. Chess also relies on experience and own best practices.

Several factors are taken in to account when choosing the way the system engineering process is performed:

- *Customer demands*: the level of detail customers want for the progress reports and documentation;
- *Partner demands*: partners may use different development process;
- *Project size*: the project size may determine the need for specific activities, reports or documents;
- *Price arrangements*: when price is a factor, all non direct product related work is kept to a minimal to reduce cost;
- *Time*: When the time to market is short, only the critical parts of the project are performed.

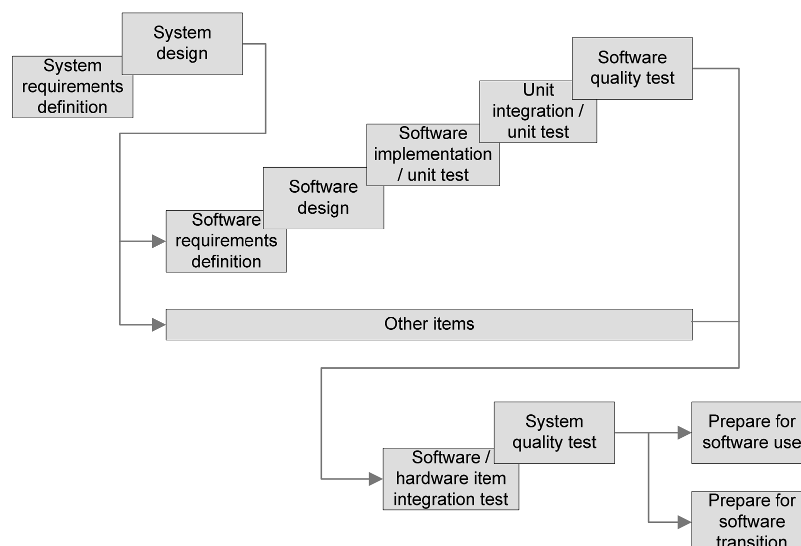

Figure 4.1 - JSTD 016 system development process based on [63]

Chess uses a very important statement, which states: “what the customers wants, Chess does”. The customers of Chess are highly professional companies (see the list of companies mentioned in section 1.1) with their own standards and procedures. Chess does not want to tell these companies what they should and should not do. In other words, Chess entirely adapts itself to the requirements, constraints, and wishes of the customer. When a customer wants a particular documentation standard or a particular design pattern, Chess will use it during the project.

When describing the system engineering process of Chess, it should be kept in mind that this approach is applicable for some, but not all projects. However, Chess uses the JSTD 016 [63] standard as guidance for most of its projects. This standard is illustrated in Figure 4.1. Take note that the different names used for the activities of the JSTD standard represent the same activities described in chapter 3.1.

4.2. The system engineering process

In the following sections the system engineering process of Chess is described. Because of the almost resemblance between the system and software engineering fields, and this thesis its focus is on the software design activity, the following sub-sections will be in less detail.

4.2.1. System requirements definition

To gather all necessary system requirements, Chess organizes workshops with the key users, shareholders and the architects. To gather all necessary requirements, these workshops may take several days. Besides the workshops, interviews or questionnaires are also used to collect the information needed from the different stakeholders.

In some projects, the requirements are already specified by the client. In that case Chess uses these requirements and rewrites these in a usable format. In other projects existing systems may need to be replaced by a new version. For these systems the requirements for the old system can be used as requirements for the new system. Additional requirements are then gathered by means of the interviews, workshops and questionnaires.

All requirements are documented in a MS Word documents. The requirements are presented in textual form supported by images, which may be drawn in MS Visio or other graphical tools. This thesis for example can be an example of how the requirements document could look like.

4.2.2. System design

During the system design activity, the system is designed into components or sub-systems and the requirements stated in the requirements document are related to the different sub-systems. The sub-systems are then grouped according to their engineering field and directed to the different architects. The sub-systems are then further handled by the different architects.

In the design activity the sub-systems are decomposed in functional components by the architect. New requirements for the current engineering field may arise, but it is also possible that new requirements for other engineering fields, like particular hardware requirements are created. These are then communicated to the appropriated engineering field.

In some projects another company provides the architect which is responsible for the architecture of the system and defines the sub-systems. Chess is then only responsible for the engineering field appointed to them and assumes an executive role.

Chess uses its own development departments for software engineering when needed. For hardware engineering, Chess sometimes uses its own hardware department to build special and unique hardware which can not be bought from other companies. External companies will be contacted when standard hardware is needed. An example of in-house hardware engineering is that Chess makes software for electronic parking ticketing system and also delivers the ticket machines.

4.2.3. Sub-system development

The system developers use the system design to further design their part of the system and finally transform this design into code or other entities. Each of the different engineering fields has its own entities which need to be produced. This activity can be seen as the actual building activity for all engineering fields.

During this activity all the sub-systems for each engineering field are extensively tested for errors and irregularities. When errors and irregularities are found, the developer tries to correct these and the sub-system is retested.

4.2.4. System integration

As a continuous interaction cycle, all sub-systems are composed into one system and are extensively tested. This is the incremental approach in stead of the ‘big bang’ as described in sub-section 3.2.4. The composition consists of all products developed by the different engineering fields, but also all sub-systems within the different engineering fields. This activity starts halve way into the sub-system development activity and ends when all sub-systems are integrated in one functioning. This creates a working system in an early stage of the project which gives customers and developers a clean picture of the system.

Like the subsystem development activity the system as a whole is also extensively tested. When errors and irregularities are found in a sub-system, that specific sub-system is rebuild and rechecked.

4.2.5. System installation

When the developers finished building, the installation of the system is done at the customer. This means that for the software engineering field, Chess is creating the installation disks, writing manuals, and importing the data needed for the system. For the system engineering field this means that the hardware is installed, procedures are executed and personnel is trained to use the system. The system is then finally ready for use by the customer.

4.2.6. System evolution

Chess is responsible for the maintenance of most of the systems build by Chess. This is done by a special department within Chess. This department maintains the system and corrects errors in a timely fashion. Critical cases get a higher priority and will be looked at first.

Chess also expand (that means, adding new features to) their own or other systems when customers want this. For most projects executed by Chess, follow-up projects are launched to expand the current systems. This indicates that Chess delivers good systems and that the customers are satisfied.

4.2.7. System decommissioning

Chess does not decommission systems, even when they are build by Chess themselves. This means that Chess is not responsible for any of the old systems present at the customer. Customers are responsible for the decommissioning of the system. They can decommission the system themselves or hire another external company.

When a new system is build and it replaces an old existing system, the old existing systems are evaluated and analyzed by Chess. This information is then used designing and building the new system. In some cases the old system is documented, which is then used as a guideline for the requirement engineering activity and design activity. Unfortunately most of the old systems are not well documented (that means the documentation is old, incomplete or not up-to-date) or the documentation does not even exist.

4.3. The software engineering process

In this section the software engineering process of Chess is described. This will, as described in the previous section, be done more elaborated then the previous section. Especially the design activity gets special attention, because that is where the focus of this research project is on.

4.3.1. Requirements engineering

After gathering and analyzing the system requirements, the software requirements are specified. This is done by the architect of Chess, who also will be responsible for the design of the software. The requirements are gathered in the same way as discussed in sub-section 4.2.1.

Chess classifies every requirement provided by the customer as a requirement. It does not really matter if the requirement is a technical or non-technical. Chess categorized requirements in three categories:

- *Business requirements*: are the requirements that are used by the executive board to identify the functionalities of the system. These are very general requirements witch indicate what the goal is of the system;
- *Functional requirements*: are the requirements that are used by the designer as input for the design process. The functional requirements specify what the system should do in terms of services. They are not in such detail that they can be implemented in the system. These requirements will be translated in technical requirements;
- *Technical requirements*: are the requirements that are used by the programmers to build the system. These requirements are very detailed and in most cases describe what the system should do (see the next section for an explanation of this).

Requirements that are created during the design activity are essentially design decisions but are also classified as requirements by Chess (see section 4.3.2. for more detail about the use of the term requirements by Chess). These requirements are not presented to the customers, but are for internal use only.

4.3.2. Design

As mentioned in sub-section 3.3.2, there is no universal design method for designing a system. This is also applicable for Chess, because every project requires a different approach. These differences in approaches were discussed in the introduction of this chapter. Another reason for the different approaches is the different designers and their working methods. Each of the Chess designers has their own approach for designing a system duo to different back-grounds and educations.

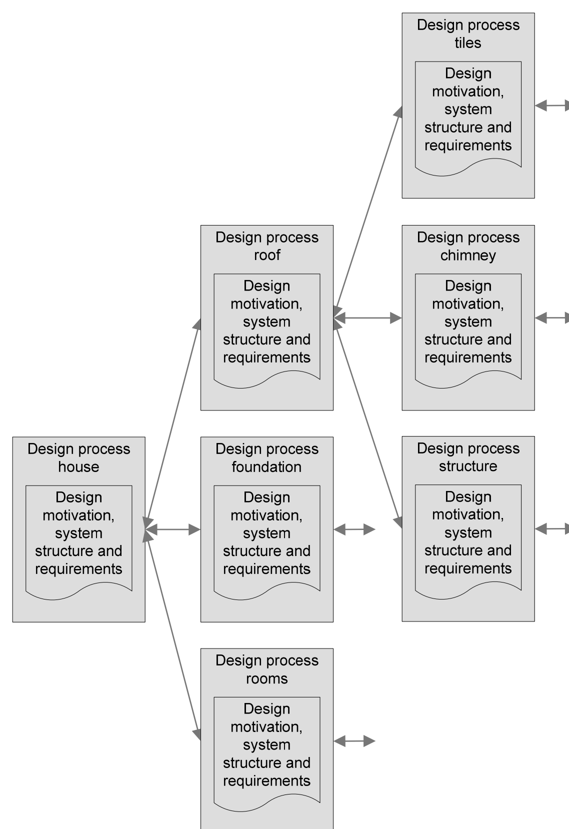

Figure 4.2 - Chess software design process

Chess uses a hand on approach for designing the system. The software system is divided into sub-systems till it can not be decomposed any further. During the design process of these sub-systems, design decisions are made, new design requirements may be specified, and new sub-systems or files are created. This design process is illustrated in Figure 4.2.

Like the traditional approach, the user requirements are gathered during the requirements engineering activity as described in previous sub-section (4.3.1). These requirements are used as input for the first design process which defines the complete system and describes what the system is and shall do or support.

During this design activity new design requirements may be created. These design requirements may be a refinement of a user requirement or just a new design requirement opposed to the design needed to let the system function correctly. Design requirements can be very technically and are necessary for a correct systems execution. These new requirements are primarily for internal use. The design requirements will be tested for implementation just as the user requirements are tested for implementation.

The new sub-systems are also defined in this design activity. All these decisions are documented as design decisions so later on in the system development project the designer can understand why some decisions are taken. This can especially be very helpful when the design need to be changed.

The user requirements and the new design requirements may form the input for the next decomposition step. New sub-systems are created and the whole process as described above is redone. Eventually files with program code or documentation files are created which may implement the user or design requirements.

The system is not always decomposed in to the deepest and lowest level. The designer may leave the contents of some sub-system not designed. The programmer is then responsible for the design of this sub-system. This gives the programmer more freedom, flexibility and understanding about the sub-system to be build.

According to Siddiqi & Shekaran [55] requirements only describe ‘what’ a system should do and not ‘how’. The design requirements created by Chess during the design activity describe “how” the system should do something. This means that, according to Siddiqi & Shekaran [55], the design requirements are not allowed or should not have the name “requirement”. Still, Chess uses the term requirement to indicate the design requirements. Although this may lead to confusion or even be wrong to some people, there are some arguments for using the term for design requirements. These arguments are:

- During the system engineering process system requirements are formulated. These are high level requirements expressing what the system should do. The next activity for each engineering field, like software engineering, is to define for each of these engineering fields the new requirements. When taking the requirements definition very literally, these software requirements are technical requirements from the system engineering’s point of view. The decisions to divide or decompose the system engineering process in different engineering fields can be directed as a ‘how’ question. This decision could be seen as a design decision, so the term requirements can also be used in the design activity (which is a decomposition of the engineering field) to express design requirements;
- Frezza, Levita & Chrysanthis [22] and Belev [5]indicate in their papers that these requirements exist. Although this is not widely accepted, there are some well known authors who believe design requirements exist and may be used;
- Using different terms for the same thing will result in miscommunication and chaos. Using the term requirements for both user and design requirements makes communication much easier. It is advisable to distinct the types of requirements in user and design requirements.

Finally, both chapters 10. and 13. describe the design process of Chess used. Chapter 10. uses this to clarify the concept of the repository. Chapter 13. describes how the repository should be used during the design activity.

4.3.3. Design documentation

Chess has only one design document for the entire design activity which is used by all stakeholders. The design document is written in MS Word and contains manly textual descriptions. Images drawn in MS Visio or other graphical tools are used to support and illustrate the text. The standard which is used for this design documentation is the JSTD 016 [63].

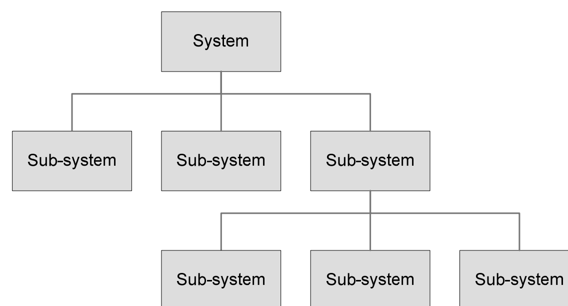

Figure 4.3 - System decomposition

The structure of the design documentation is shown in Figure 4.3. A typical design document starts with a description of the whole system. This is the architecture of the system. In most cases the architecture is the separation of front-end, back-end, project documentation and the database. This is then illustrated in an image to clarify the separation of the sub-systems.

From this point each sub-system is decomposed into new sub-systems. This is then described in detail and when needed supported by images. These descriptions describe what a sub-system needs to do and what the requirements are for this sub-system. Finally a decomposition level is reached that can translate the system into executable code or files.

The attributes and the user roles identified by IEEE Standard 1016 [4] and mentioned in sub-section 3.3.3 are not used as guide lines for writing the documentation. The attributes are not directly implemented in the design document. This is done unconsciously and more based on an intuition. When the designer thinks one of the attributes is needed to clarify the design, this attribute will then be used.

The user roles are very flexible for Chess. Each project has its own user roles. When the user roles are identified, a person can fulfill multiple user roles. In most of the Chess their projects, the testing roles are gathered in one testing role. The same can be said for the configuration manager and designer role. It even happens that some user roles are not even used in a project. This all depends on the factors stated in section 4.1.

4.3.4. Implementation

During the implementation activity, sub-systems are assigned to the programmers. They will build the sub-system. As discussed in section 4.3.2, the programmer may deliberately be

responsible for designing the contents of a sub-system. The programmer then tries to transform the design into executable code or files.

Chess uses advanced IDE's (Integrated Development Environments) to help the programmers with building the system incrementally and in an effective way. These IDE's are connected to a central main code repository. This means that every sub-system build is immediately incorporated in the complete system. This makes it possible for all system variables and relations to be checked and kept constant while the programmer builds his part of the system. This has the advantage that the system can be build incrementally. It also provides in the early stages of the project a working system which can be used to give an early preview of the system.

4.3.5. Testing

During the requirements engineering, the design and the implementation activity test-cases are developed for the different requirements and sub-systems. The requirements engineer, the designer and the programmer develop these test-cases in their activities. During the test activity these test-case are used to check whether the code is correct and all requirements are implemented in the system. As described in sub-section 3.3.5. The test activity is parallel to the other activities. During the requirements engineering phase tests are conducted to get valid requirements. The same sorts of tests are done during the design and implementation activity.

To make sure the code generated during the project is optimal, an external reviewing company is hired to test the code. This is done when a customer asks for these types of tests and is willing to pay for it. Then, every night during the development project, the code is uploaded to the Software Improvement Group [56]. This company analyses and checks the code for complexity, maintainability and other program characteristics and advises Chess when the code is not meeting their quality standards.

When the development project is two thirds underway, the programming and development activities are stopped and a period of extensive testing is executed. Chess uses one person who is dedicated to the testing of the system. When problems are encountered, the programmer responsible for the sub-system containing the error is assigned to fix this problem. When the same problem reoccurs, another more experienced programmer is assigned to the problem.

When Chess is satisfied with the code and the system, it is up to the customer to test the system. These tests are called the acceptance tests and are performed at the company of the customer. The bugs found during these tests are fixed by a Chess programmer. When both Chess and the client are satisfied with the system and no errors are found, the testing activity is finished and the system is accepted by the customer. Errors and bugs that are found after this point are processed in the maintenance activity.

4.3.6. Maintenance

Chess uses a separate maintenance department, which is responsible for fixing bugs and making changes to the existing Chess systems. When a new feature needs to be implemented, the size of this new implementation decides if a whole new development team will be created or that the maintenance department implements this new feature itself. Bug and change handling is done

by using the software tool Problem Tracker [50]. This program is also used during the test activity described in the previous section.

4.4. Summary

This chapter focused its attention on the way Chess executes the system engineering and software engineering process. As guidance for this research the information about system engineering provided in the previous chapter was used.

Important to know is that there is not really a standard approach for Chess for building systems. Chess tries to use well known standards like the JSTD 016 [63] standard but this may differ for each project. Chess also relies on experience and own best practices. Several factors are taken in to account when choosing the way the system engineering process is performed:

- Customer demands;
- Partner demands;
- Project size;
- Price arrangements;
- Time.

Chess does what the customer wants. This means that, when a customer wants a particular standard for building, testing, or documentation, Chess uses that standard. Chess entirely adapts itself to the requirements, constraints and wishes of the customer.

Chess participates in projects where different engineering fields are active. In some of these projects, Chess has a leader role and in other projects there are only executors. Chess has a software and hardware department. The software department is responsible for the software parts of the system. The hardware department builds exclusive and custom hardware parts for systems.

In general, the activities of gathering the requirements of the system, designing the system, building the system, testing the system, and maintaining the system are like that described in the previous chapter. However, the composition of these activities may differ from the ones described in the literature.

The difference with the literature is that during the design activity Chess creates design requirements. These requirements are constraints opposed to the designs of the system and describe how the system should function. According to Siddiqi & Shekaran [55] this is wrong. Requirements describe the "what", not the "how". Still, Chess uses the term design requirement.

Chess uses only one design document to clarify the design of the system. This document is a description of the system structure and the design requirements. The documents structure is the same as that of the structure of the system.

5. DESIGN ARTIFACT MANAGEMENT

This chapter provides a description of the management of artifacts during the design activity. During the development of a system, changes may happen and these changes need to be tracked [19] and actions need to be taken to handle with these changes. This is done by managing the different artifacts [38] during the design activity.

In the first section of this chapter a description of what design artifacts are is given. The second section describes the management of requirements and design. In section 5.3 the reasons why requirements and designs may change are presented. The last section summarizes this chapter.

5.1. Design artifacts

During the development process, software architects, designers, and developers identify certain information about the software. Examples of such information are use-cases, software architecture, object collaborations, and class descriptions. This information can be very abstract, such as the vision of the software, or very concrete, such as the source code of the software [28]. In this thesis, these pieces of information about the software are called design artifacts.

It is important to realize that there is a difference between a design artifact and its representation [28]. The design artifact determines the information about the software system, and the representation determines how the information is presented. Some design artifacts are represented in UML (Unified Modeling Language), some are represented by text or by tables, and some are represented in other ways. For example, the class life-cycle can be represented by a state chart diagram, an activity diagram, or in a state transition table. A useful specification of a software system is based on precisely defined design artifacts, rather than on diagrams.

5.2. Management of requirements and design

The purpose of artifact management is to establish a good communication between system users and system developers. This is to make sure that the requirements and design decisions have been addressed in the system and are up-to-date. The effectiveness of an organization at managing artifacts has a direct impact upon system quality and time to market. It is imperative to have access to up-to-date and accurate information about the status of all requirements and design of the system [68]. During the design activity requirements and designs management become important activities. These two management activities are described in the next two sub-sections. In sub-section 5.2.3 the term repository is clarified.

5.2.1. Requirements management

After defining the requirements in the requirements engineering activity, the requirements need to be managed or as Fiksel & Dunkle [19] call it; requirements need to be tracked. Requirements management, in system and software engineering, is the process of controlling the identification, allocation, and flow-down of requirements from the system level to the module or

part level, including interfaces, verification, modifications, and status monitoring [61]. It tries to understand and control the changes directed to the requirements [58].

Requirements management is the set of activities that concentrate on assuring that the requirements are met to the customer's satisfaction. The process begins at the project s inception and continues until the system is no longer needed. Thus, requirements management does not stop with the end of a project but continues into the operation and support of the product. The recruitment management activities are performed repeatedly in an iterative fashion and are summarized below:

- *Requirements capture*: is the process of expressing requirements in an explicit form that can be understood and interpreted by people and/or computer programs;
- *Requirements tracking*: is the process of monitoring the requirements during the course of a product development to assure compliance;
- *Requirements evaluation*: is defined as a procedure for determining the extent to which a product design is likely to satisfy a designated set of requirements.

Besides the activities mention above, requirements management includes other general management phases, such as organizing, implementing, sustaining, gathering, documenting, verifying, and managing changes.

5.2.2. Design management

Design management is a rather vague term which nowadays is used to explain many things. In most published papers or books a crisp definition of design management is missing or used incorrect. In this thesis design management is an uniform management facility for managing design data and design tooling in a multi-user, heterogeneous design environment [16][11].

Although design management is not extensively described in the literature, management of design data is becoming a very important topic. Nowadays different tools and thus different data types and documents are produced during the design activity. These need to be kept up to date, correct, and consistent. Design management is concerned with the following questions [5]:

- Is it the right design currently used?
- Can the design be made at the right cost?
- Is the design capable of being manufactured and delivered?
- Are the user specifications implemented in the design?

Konstantinov [36] indicates in his paper that design management is not only concerned with the design data, meaning the design documentation, but also with the design processes itself. The human cooperation and the stakeholders contribution to the development of the system needs to be captured and managed to understand the different trace information generated from the design data [35]. The design activity exists of multiple smaller sub design activities, which all need to be managed. The different decisions and information made and information gathered during the different activities need to be tracked and stored to form formal review-points. This type of

information can also be very helpful when the development project is finished and changes are proposed. This helps to understand the different decisions that where made during the developed of the system.

5.2.3. Repository

To be able to manage the design artifacts, place is needed to store these artifacts. This is done in a repository. A repository is a central place where data is stored and maintained. A repository can be a place where multiple databases or files are located for distribution over a network, or a repository can be a location that is directly accessible to the user without having to travel across a network [70]. Important characteristics of a repository are:

- A repository is used for saving information;
- The repository makes it possible to maintain this information.

5.3. Changes

The management of artifacts is needed because of changes occurring during projects. The cause of change may be either external to the enterprise (that means, unstable customer requirements, economical constraints), or internal (that means, detected bugs, replacement of development tools, migration of developers) [38]. Both large and small software projects undergo changes. Regardless of its source, a change introduced to one element of a project usually affects other parts. If this process is uncontrolled, changes may destabilize the project, leading to problems in keeping up with the schedule, budget, or even the correct functioning of the system.

As Davis & Leffingwell [14] describe in their paper, software development is still more an art then science. Everybody does artifact management its own way. Sometime this is done in a structured way but most of the time it is not.

Uncontrolled changes can have undesired effects. It frustrates developers unable to grasp what is being changed and how it affects other parts of the system [38]. The larger and more complex a project is, the harder change management gets. At some point, the increasing number of people involved in the project and its growing size creates a wall between project management and developers. Individual project members have only small bricks of the project under control. Even as a group, they have the knowledge about a larger part of the project, but not about the entire project.

The decisions taken at management level (which usually involve change acceptance) do not take into account the actual impact on the work to be done by developers. This causes underestimation of the effort needed to apply the change to the project, budget-wise or time-wise. The developers on the other hand, while usually specialized in their part of the project, are unable to see the overall dependencies of the change they introduce. Although there are development- and management processes which attempt to reduce this gap, they are mostly ‘paperwork-oriented’, bringing tiresome routines and moderate effects.

The following two sub-sections describe the different causes of requirements and design changes in more detail. This is done be distinguishing requirements and design changes and their causes.

5.3.1. Requirements changes

Wang & Lai [68] identify different reasons why requirements change and thus the design may change. These reasons are show in Table 5.1 with some percentages added to them to indicate the frequency of changes occurrences per project. The table illustrates that in general, incomplete requirements are the common reason for requirements to change. There are some reasons for these changes.

Most causes of requirements change have their origin at the beginning of the project when the system users meet the developers. At the beginning of a project users do not really know what they want or what is possible which may result in incorrect or incomplete requirements. During the project users may change their perspectives on the problem, which will result in new or changed requirements.

Causes	Percentage
Incomplete	24
Inconsistent	16
Postponed	15
Omitted	15
Infeasible	12
New occurred	10
Canceled by user	5
Changed by user	2
Other	1

Table 5.1 - Requirements change causes based on [68]

Misunderstanding caused by difference in experience and education make it also hard for developers [37]. Examples why misunderstanding could result in incomplete requirements are:

- The requirements engineer is not an expert in the application domain being addressed;
- Inadequate communication between the requirements engineer and the system's users;
- The system users may not give all information needed for the development of the system;
- Natural languages leave space for misinterpretations;
- Environments may change during the project and so do the requirements effected by the environment;
- There are no real requirements approaches or techniques to capture all user requirements.

5.3.2. Design Changes

Requirements form the basis for the design activity, which represents the systems structure. It should be clear that requirements changes are mainly the causes for design changes. When requirements change, the designs may change. These requirement changes can have devastating consequences for poorly designed designs, resulting in completely redoing the design activity.

Designs however can also change due to unrealistic design decisions made by the designer or because of poor designing. The design can simply not be realized or appears to be wrong, which

causes to force the designer to change the current design. Gurp & Bosch [25] identified four problems which cause design changes associated with the way software is currently developed. These four changes are:

- *Traceability of design decisions*: the notations commonly used to create software lack the expressiveness needed to express concepts used during design. Consequently, design decisions are difficult to track and to be reconstructed from the system and current designs;
- *Increasing maintenance cost*: during the systems evolution the maintenance tasks becomes increasingly difficult and effort consuming due to the fact that the complexity of the system keeps growing. This may cause developers to take sub-optimal design decisions either because they do not understand the architecture or because a more optimal decision would be too effort demanding;
- *Accumulation of design decisions*: design decisions accumulate and interact in such a way that whenever a decision needs to be revised, other design decisions may need to be reconsidered as well. A consequence of this problem is, that if circumstances change, developers may have to work with a system that is no longer optimal for the requirements;
- *Iterative methods*: the aim of the design phase is to create a design that can accommodate expected change requests. This conflicts with the iterative nature of many development methods (extreme programming, rapid prototyping, etc.). These methods require knowledge about these expected requirements in advance. This knowledge is in most cases not available, which means that the assumptions about these requirements are vague or incorrect.

5.4. Summary

This chapter clarified the importance of the management of design artifacts. Design artifacts are pieces of information about the software. Use-cases, software architecture, object collaborations, and class descriptions are examples of design artifacts. This information can be very abstract, such as the vision of the software, or very concrete, such as the source code of the software [28].

The purpose of artifact management is to establish a good communication between system users and system developers, to make sure that the requirements and design decisions have been addressed in the system and are up-to-date. Artifact management can be categorized in:

- Requirements management;
- Design management.

A repository is needed to store and manage this information. The management of artifacts is needed because of changes that occur during development projects. In this chapter two types changes where identified. These are:

- Requirements changes;
- Design Changes.

The cause of change may be either external to the enterprise (that means, unstable customer requirements, economical constraints), or internal (that means, detected bugs, replacement of development tools, migration of developers) [38].

6. TRACEABILITY

The objective of this chapter is to define traceability and the use of traceability in software design. In the first section the definition of traceability is provided. In the next two sections the importance and stakeholders of traceability are described. In the third section the directions of tracing are discussed. In sections four and five the traceability relations and how these can be used to analyze requirements and designs are describe. The last section provides a summary of this chapter.

6.1. Traceability definition

Traceability in general is the degree to which a relationship can be establish between two or more products of the development process, especially products having a predecessor-, successor, or master-subordinate relationship to one another [33].Traceability is the means by which the design community communicates the process of translating the user requirements, goals, and objectives into an operational system. Hughes & Martin [29] identify two different traceability categories:

- *Requirements traceability*: refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (that means, from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases) [23][64];
- *Design traceability*: reflects the decision making process of translating the requirements into physical manifestations or structures [29].

6.2. Importance of traceability

Traceability is becoming an important element in system engineering [59]. Hughes & Martin [29] identified two specific trends that are converging the focus to increased attention on effective (design) traceability management. These trends demand the use of traceability during system engineering process. These two trends are:

- Systems are getting bigger, more complex and more integrated with other systems;
- Escalating costs of the systems demand that appropriate mechanisms are in pace to ensure that they provide the level of performance and operational utility required by the user community.

To coop with the trends mentioned above, traceability was introduced. Traceability is about documenting the relationships between layers of information. For instance, the relationships between the system requirements and the software design. Broadly speaking, this has some benefits [15]. These benefits are:

- *Increased understanding of design*: many important questions about a project can be answered by understanding the relationships between the design layers. These questions range from “How does the system meet the customer’s requirements?”, “What is this component’s role?” to “What are the system requirements associated with this test step?”. Documenting these relationships provides greater reflection on the system and subjects the users thinking about the system;
- *Semi-automated analysis*: appropriate tool support for representing traceability relationships can make automated analysis of those relationships possible. When it comes to assessing the potential impact of change, your investment in documenting the relationships will pay off by letting the user calculate and present graphs of independent design artifacts at the push of a button.

6.3. Stakeholders for traceability

Traceability supports various stakeholders (that means, maintainer, project manager, or tester) in performing their tasks (that means, changing, controlling, or testing). The purpose of a tracing approach depends on the stakeholder and the task of the stakeholder that should be supported by the traceability [35]. Each stakeholder has his own view on traceability, but there is no agreement in the literature as to which entities and relationships are necessary to support the stakeholder and its task. The different stakeholders and tasks that can be supported by traceability are:

- *Customer*: traceability provides an overview of all the satisfied requirements to the customer. The traceability can be used to demonstrate the effect of a requirement change;
- *Project planner*: uses a tracing approach to perform impact analysis;
- *Project manager*: uses the traceability information to control the project’s progress;
- *Requirements engineer*: uses the traceability information to check correctness and consistency of the requirements;
- *Designer*: uses the traceability information to understand dependencies between the requirements. It also provides the designer the ability to check whether all requirements are considered by the design;
- *Verifier*: uses the traceability information to verify that system requirements have been allocated both to the design, to the code, and to the procedures for verification;
- *Validator*: uses traceability relationships between requirements and test plans to prove that the system meets the customer needs;
- *Maintainer*: uses the traceability information to decide how a required and accepted change will affect a system (that means, which components are directly affected and which other components will experience residual effects).

6.4. Traceability directions

The most common use of the term traceability is in the context of tracing requirements. Traceability can exist in different directions and in different places. All these types have different purposes and are important in the way the traceability information is used.

Gotel & Finkelstein [23] state that requirements traceability can be divided into two main parts as illustrated in Figure 6.1. The perspective of these parts is a certain document where between the traceability is viewed. These two parts of traceability are:

- *Pre-requirement specification traceability*: is concerned with those aspects of a requirement's life prior to its inclusion in the requirement document (requirement production). It is about tracing requirements to their underlying user needs;
- *Post-requirement specification traceability*: is concerned with those aspects of a requirement's life that result from its inclusion in the requirement document (requirement deployment). It is about tracing how each requirement is implemented in the design and the code.

Besides the distinction between the pre- and post requirements specification traceability, traceability can also be divided in forward and backward traceability. Forward and backward traceability are differentiated in Figure 6.1 [23]. Where pre and post requirements specification traceability points at the position of requirement, forward and backward traceability indicates the direction of the trace.

- *Forward traceability*: describes tracing entities to it realization on succeeding abstraction levels;
- *Backward traceability*: describes tracing entities to its source on preceding abstraction levels.

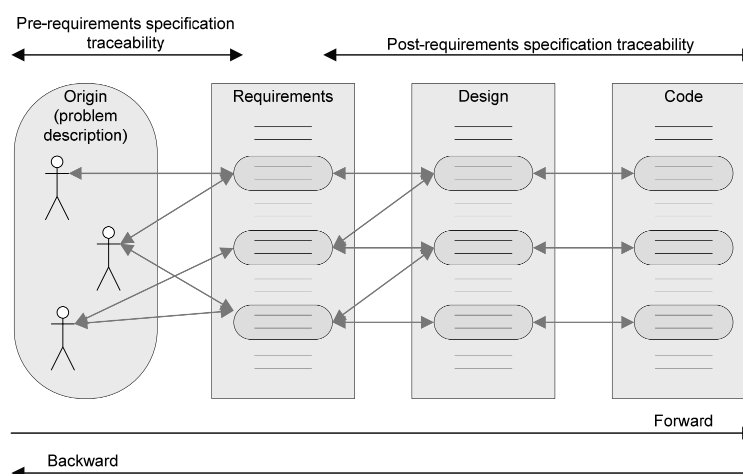

Figure 6.1 - Directions of traceability relations

It is generally accepted that traceability should be bidirectional, which means that it should be possible to trace forward and backward. Gotel & Finkelstein [23] found that most tools support post-requirements specification traceability. This is strange, because most of the problems created by poor requirements traceability were found to be due to a lack of or inadequate pre-requirements specification traceability.

6.5. Relationships between requirements and design

When relationships within the requirements and designs are properly constructed, the term traceability relationship is used. Thus, to support traceability, relationships need to be implemented within the requirements itself, within the design itself and between requirements and designs (and other artifacts like test-cases). These relationships provide a thread of origin from the implementation of the requirement to the requirements specification and vise versa, and serves as a validation that the design or the implementation indeed does what it was intended to do [24] .

Relations can be implemented in two ways; implicit and explicit.

- *Implicit*: relationships mean that a link is created manually. Examples of implicit relationships are relationships given by structure and relationships which use name tracing;
- *Explicit*: relationships come from external considerations supplied by the developers. For example, a relationship between a textual requirement and a use-case, that describes the requirement, is determined solely by the decision of the developer that such a relationship has meaning. There is no intrinsic relationship between the entities, only external decisions can establish the relationships [40]. Explicit relationships are used for all kinds of relationships and are supported in most tools, like requirement management tools and modeling tools.

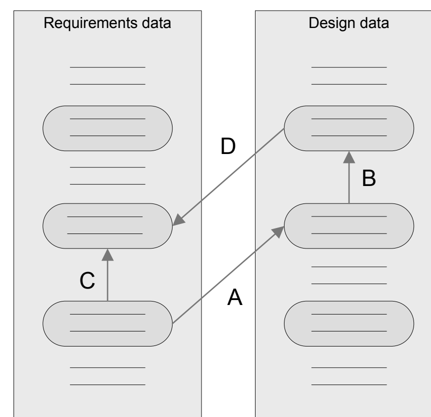

Figure 6.2 - Relationships in requirements and design

Frezza, Levita & Chrysanthis [22] identify and support four categories of traceability relationships, and their roles in tracing through the requirements and the designs. Two of these categories represent intra-dependencies, which mean dependencies within the requirements or designs. Similarly, two categories represent inter-dependencies, that is, the dependencies between designs and requirements.

To clarify the different dependencies requirements will be indicated as REQ and design objects indicated as DES for illustration purpose. A relation with the form [REQ -> DES] means that some

design object is depended upon some requirement object. These dependencies are shown in Figure 6.2. The requirements and the design objects are the circled lines in the data boxes.

- *Rational dependency*: this relationship has the form [REQ -> DES] and is arrow A in Figure 6.2. The design object is related to a particular (non-null) set of requirements. These kind of dependencies are normally called forward traceability links;
- *Technical dependency*: this relationship has the form [DES -> DES] and is arrow B in Figure 6.2. This is the dependence of one design object on another design object to perform/ meet its requirements. This link relates tot the design entities combined ability to do the right thing and typically encompasses the interface/ connections internal to the design. These links also include the configuration relations among the design entities;
- *Contextual dependency*: this relationship has the form [REQ -> REQ] and is arrow C in Figure 6.2. The purpose of the requirement objective is tied to another requirement object, and includes the configuration relations among the requirements data. This can include requirements derived (or implicitly stated) in the environment, such as where descriptions imply (or rely upon) assertive descriptions within the requirements data [34];
- *Implicative dependency*: this relationship has the form [DES -> REQ] and is arrow D in Figure 6.2. The assertion of a design object implies other (new) requirements/ constraints on the design. A typical example would be the design decisions affecting/ influencing the requirements definition. Similar to contextual dependencies, these form one class of links normally called reverse traceability links [52].

Relationships can exist in different levels and abstractions. The sort or type of the relationship describes which relationships are involved in a trace. In the literature, three general kinds of relationships can be identified [35]. These relationships are:

- *Relationships between entities on the same abstraction*: traceability relationships between entities on the same abstraction are called horizontal traceability [52] or vertical traceability [7]. The term used depends on how abstractions are arranged in a graphical representation. If they are arranged vertically, traceable relationships on the same level are called horizontal, otherwise vertical. Two kinds of relationships are distinguished by Knethen & Paech [35]; representation relationships and dependency relationships. The term representation relationship is used for a relationship between entities of different views that represent the same information (for example the same logical entity). The Dependency relationship term is used for a relationship between two entities that depend on each other and represent different logical entities on an abstraction;
- *Relationships between entities at different abstraction*: traceability relationships between entities on different abstraction are also called horizontal traceability or vertical traceability, just like relationships between entities on the same abstraction. Only here there are two different kinds of relationships between documentation entities on different abstractions: within-level refinement relationships and between-level refinement relationships. The term within-level refinement relationship is used for a relationship between entities at different abstractions on a certain abstraction level (for example

between two system use cases). The between-level refinement relationship term is used for a relationship between documentation entities at different abstractions on different abstraction levels (for example between a system and a software description entity);

- *Relationships between different versions of a document*: these types of relationships are also called evolutionary links [48], historical links [51], or horizontal traceability [12]. Tracing history is an extension of what usually is called version control, namely to trace all previous versions of a particular entity to recover its development history.

6.6. Traceability analysis

Once all relationships are incorporated and found correct, users or stakeholders may want to have the ability to see where the relationships come from, where they go to, and why they are applied in a particular case. Lee [39] and Knethen & Paech [35] identify different analysis types where traceability is used for:

- *Identifying inconsistencies*: tries to identify inconsistencies such as unrelated or unimplemented requirements or elements (orphans);
- *Relation following*: tries to follow specific requirements or element to other requirements or elements by following the relations between these requirements and elements. The visibility into existing relations from source to implementation;
- *Requirement verification*: tries to verify that requirements have been met. It provides information where the requirements is fulfilled, how it is fulfilled and who was responsible for this implementing this fulfillment;
- *Requirement performance verification*: once performance requirements have been allocated to system elements, the verification of those requirements is done by rolling up actuals and reporting on variances (this is the allocated weight versus the actual weight);
- *Dependency analysis*: tries to identify dependencies between the different requirements or elements;
- *Impact analysis*: the impact of changes in the repository need to be provided by means of a list of effected requirements and designs.

6.7. Summary

This chapter tried to explain what traceability is, why it is needed and how it can be used. Traceability in general is the means by which the design community communicates the process of translating user requirements, goals, and objectives into an operational system. Traceability can increase the understanding of designs and support semi-automated analysis [15]. There are two different traceability categories [15]:

- *Requirements traceability*: refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction ;
- *Design traceability*: reflects the decision making process of translating these requirements into physical manifestations or structures [29].

Traceability can be done before an artifact is specified (pre requirements trace) or after the specification (post-requirement trace) and exist in a forwards and a backwards direction. Traceability is about documenting the relationships between layers of information in an implicit or explicit way. There are four types of relationships identified by Frezza, Levita & Chrysanthis [22]:

- Rational dependency;
- Technical dependency;
- Contextual dependency;
- Implicative dependency.

Traceability supports various stakeholders (that means, maintainer, project manager, or tester) in performing their tasks (that means, changing, controlling, or testing). The purpose of a tracing approach depends on the stakeholder and the task of the stakeholder that should be supported by the traceability [35]. These stakeholders are:

- Customer;
- Project planner;
- Project manager;
- Requirements engineer;
- Designer;
- Verifier;
- Validator;
- Maintainer.

As stated earlier in this section, traceability supports semi-automated analysis checks. Lee [39] and Knethen & Paech [35] identify different analysis types where traceability is used for:

- Identifying inconsistencies;
- Relation following;
- Requirement verification;
- Requirement performance verification;
- Dependency analysis;
- Impact analysis.

7. DEFINING REQUIREMENTS

In this chapter the requirement types identified in section 3.2.1 are discussed. This chapter also discusses how these requirements should be stated. All this information will be used in chapter 9. when the requirements for the repository are provided. Section 7.1 provides the definition of a requirement. Section 7.2 describes the different requirement types. Section 7.3 provides the characteristics for creating testable requirements. The final section summarizes this chapter.

7.1. Definition

In this section the definition of a requirement according to the literature is given to have a clear understanding what a requirement is. A requirement is a condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents [31]. To be more clear a requirement only describe 'what' a system should do and not 'how' [55].

As described in section 4.3.2, design requirements are also called requirement, but these describe the 'how' part of the system. To prevent confusion, in this chapter the user requirements, that means the requirement describing the 'what' part of the system, are discussed.

7.2. Types of requirements

In the following sections, the requirements are categorized into three types [3][15]. These types will later on be used in chapter 9. to categorize the user requirements. Section 7.2.1 describes the business requirements. Section 7.2.2 describes the functional requirements and section 7.2.3 describes the non-functional requirements.

7.2.1. Business requirements

The business requirements state the general reason for building the system [3]. These requirements state what the repository in general shall support or accomplish, without describing the detail how this is realized. Business requirements should [9]:

- Set out the project scope and requirement;
- Outline the business opportunity for potential providers;
- Describe the context of the requirement;
- Give interested parties sufficient information to enable them to respond with expressions of interest.

Business requirements can be set out in the form of a business prospectus. This is particularly useful when there is a need to 'sell' the concept of an innovative requirement and gauge the likely interest of all stakeholders. The key target audience for the prospectus is the stakeholders of the project.

7.2.2. Functional requirements

Functional requirements provide statements of services that the system should provide, how the system should react to particular inputs, and how the system should behave in particular situations [58]. These requirements state the technical properties of the system, which are directly identifiable and implementable in the system. The functional requirements will eventually be used in testing the system for the satisfaction of the user requirements.

7.2.3. Non-functional requirements

Non-functional requirements specify criteria that can be used to judge the characteristics of a system, rather than specific behaviors or functions [70]. Non-functional requirements are sometimes called quality attributes. These attributes are concerned with the whole system and can be found in almost every system. ISO 9126 is an international standard from the International Organization for Standardization [33] for the evaluation of software [70]. The ISO 9126 standard defines six attributes, with minimal overlap, for software quality. These attributes and their characteristics are shown in Figure 7.1.

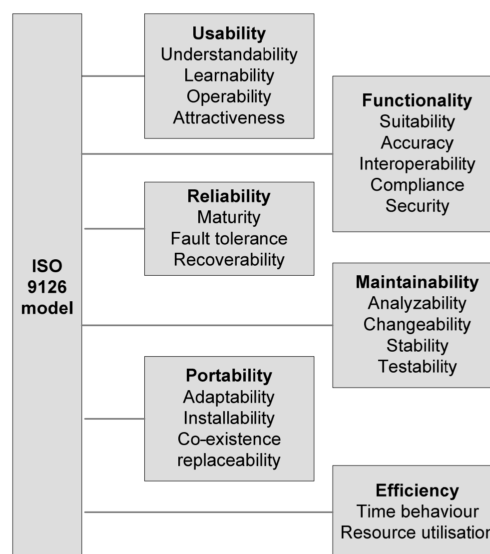

Figure 7.1 - The ISO 9126 quality attributes based on [49]

7.3. Characteristics of testable requirements

The user requirements are the criteria for the acceptance of the new system. According to Bender [53] the requirements of the repository need to be stated in such a way that the system can be tested if the systems satisfy the requirements. In order for the requirements to be considered testable, the requirements should have all or some of the following characteristics [53]:

- *Unambiguous:* all project members must get the same meaning from the requirements, otherwise they are ambiguous;
- *Correct:* the relationships between cause and effect are described correctly;

- *Complete:* all requirements are included. There are no omissions;
- *Non-redundant:* just as the data model provides a non-redundant set of data, the requirements should provide a non-redundant set of functions and events;
- *Lends itself to change control:* requirements, like all other deliverables of a project, should be placed under change control;
- *Traceable:* requirements must be traceable to each other, to the objectives, to the design, to the test cases and to the code;
- *Readable by all project team members:* the project stakeholders, including the users, developers and testers, must arrive at the same understanding of the requirements;
- *Written in a consistent style:* requirements should be written in a consistent style to make them easier to understand;
- *Processing rules reflect consistent standards:* processing rules should be written in a consistent manner to make them easier to understand;
- *Explicit:* requirements must never be implied;
- *Logically consistent:* there should be no logic errors in the relationships between causes and effects;
- *Lends itself to reusability:* good requirements can be reused on future projects;
- *Terse:* requirements should be written in a brief manner, with as few words as possible;
- *Annotated for criticality:* not all requirements are critical. Each requirement should note the degree of impact a defect in it would have on production. In this way, the priority of each requirement can be determined, and the proper amount of emphasis can be placed on developing and testing each requirement. We do not need zero defect in most systems. We need good enough testing;
- *Feasible:* if the software design is not capable of delivering the requirements, then the requirements are not feasible.

The key characteristics in the above list are deterministic and unambiguous. Every requirement should at least satisfy these key characteristics. Testing, by definition, is comparing the expected results to the observed results. This is not unique to software. Software testing is supposed to have that characteristic. Unfortunately, you almost never find requirements specifications written in sufficient detail to be able to determine the expected outcome.

7.4. Summary

This chapter tried to provide information about the different types of requirements and how these requirements should be stated. This information will be used in chapter 9. to define the requirements for the repository.

A requirement is a condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents [31].

Requirements can be divided in four categories, differencing in goal and detail. These requirements categories are:

- Business requirements;
- Non-functional requirement;
- Functional requirements;
- Quality attributes.

This chapter also provided characteristics for measurable requirements. Recruitments must be stated in such a way, that they can be checked for implementation in a system. A large list of characteristics was provided, but the key characteristics of this list are deterministic and unambiguous. Every requirement should at least satisfy these key characteristics.

8. TESTING

This chapter will describe the testing activity and forms the literary basis for chapter 12. In section 8.1 the test process according to TMap (Test Management Approach) is discussed. This section provides some background information for testing. In section 8.2 requirements based testing is described. Sections 8.3 and 8.4 describe how functional and non-functional requirements are tested. The final section gives a summary of this chapter.

8.1. The test process

TMap, the Test Management Approach, is a structured approach for testing information systems. Although there are different approaches and methods for structured testing, TMap is commonly used in the Dutch system engineering field and is well supported. TMap defines the use of four pillars for a structured test process. These pillars are shown in Figure 8.1.

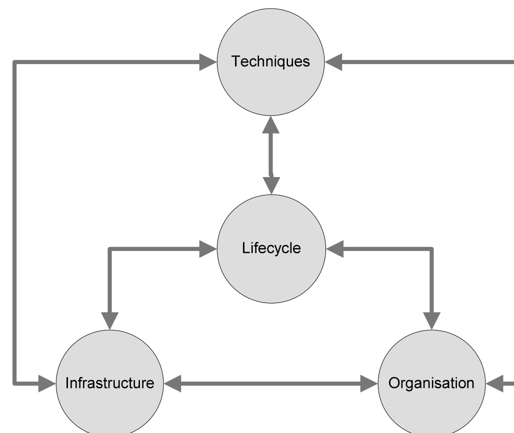

Figure 8.1 - The four pillars for structured testing [49]

Test process may differ from each other, but there are some characteristics that every good test process needs. These characteristics are [53][45]:

- *Testing must be timely*: Any system without a tight feedback loop is a fatally flawed system. Testing is that feedback loop for system engineering. Therefore, it must begin in the initial phase of the project, when objectives are being defined and the scope of the requirements is first drafted. In this way, testing is not perceived as a bottleneck operation at the end of the project. Testing should begin as early as possible and occur as often as possible;
- *Testing must be effective*: The approach to test-case design must have strictness or rigor to it. Testing should not rely solely on individual skills and experiences. Instead, it should be based on a repeatable test process that produces the same test cases for a given situation, regardless of the tester involved. The test-case design approach must provide high functional coverage of the requirements;

- *Testing must be efficient*: Testing activities must be heavily automated to allow them to be executed quickly. The test-case design approach should produce the minimum number of test cases to reduce the amount of time needed to execute tests, and to reduce the amount of time needed to manage the tests;
- *Testing must be manageable*: The test process must provide sufficient metrics to quantitatively identify the status of testing at any time. The results of the test effort must be predictable (that means, the outcome each time a test is successfully executed must be the same).

In general, the test process consists of a master test plan which specifies who and when what test is going to perform. This master test plan can be used for every test sort as illustrated in Figure 8.2. Each test sort uses multiple disciplines which mean that for each test sort the tasks, milestones and products need to be specified. It is possible to perform only a selection of these tests.

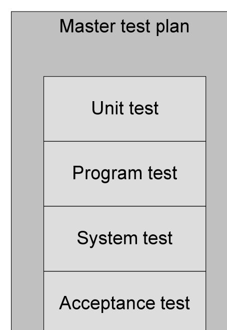

Figure 8.2 - Hierarchy of the test sorts [49]

TMap identifies two sorts of tests: white-box and black-box tests. A test sort is a group of test activities which are jointly organized and managed [49]. The white-box tests are based on the program code, program descriptions or the technical design. The focus of these tests is on the knowledge of the internal structure of the system. Examples of the tests that are considered to be white-box tests are: unit-, module- and program tests.

The black-box tests are based on the functional specifications and quality requirements. During this sort of test, the system is considered close to finalization and therefore will be close to the form it is going to be used by the users. The internal structure is not important and does not to be known. It tests what the system should do (that means what its outcome should be). The system- and acceptance test are part of the black-box tests.

8.1.1. Lifecycle

The test activities can be captured into a phase model shown in Figure 8.3, which can be used parallel to the different phases of the system engineering process. These phases are similar to the activities described in chapter 3. The lifecycle pillar tries to answer the “what” and “when” questions for the test process. The different phases are [49][65]:

- *Planning and maintenance*: are executed parallel to the other phases and initiate the test process. This phase is the basis for a controllable and qualitative test process. The phase begins with an information study and a goal statement. From the information gathered, a test strategy is formulated. All these documents are managed and reported during the test process;
- *Preparation*: In the preparation phase the specifications are specified which will form the basis for the test process. During this phase the test units and infrastructure are specified and the test techniques are appointed;
- *Specification*: During the specification phase all test cases are specified and the test infrastructure is prepared for use. The test scripts and screenplay are formulated and the test infrastructure is realized;
- *Execution*: The execution phase starts when the first testable components are available. During this phase the actual testing and re-testing are performed. To perform these tests, the prepared test-cases, scripts, and screenplays are used. At the end of each test the results are evaluated;
- *Completion*: During the completion phase the test process is finalized. The test process and objects are evaluated. To document the test process, a selection is made from all the testware which can be reused in future projects and changes. Finally an evaluation report is created to look back on the test process.

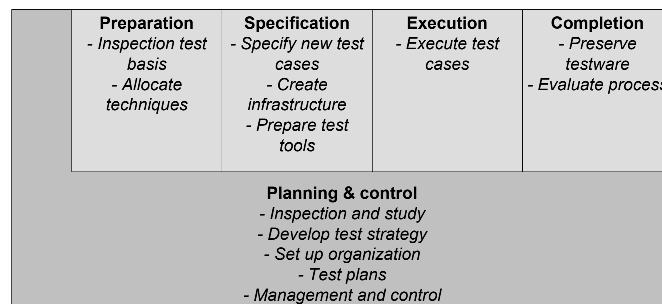

Figure 8.3 - TMap phases based on [49][65]

8.1.2. Techniques

TMap is supported by a big variety of techniques. These techniques provide proven, universal methods. In this pillar the "how" question is answered. There are different groups of test techniques. These groups of test techniques are:

- *Strategy provision*: Determining the explicit test strategy is a way to communicate with the constituent about the organization and the strategic choices for the test process;
- *Test point analysis*: The time needed to perform the tests needs to be budgeted for these tests. The functionality of the system is determined by a function point analysis, which is then used in the Test point analysis to determine the time needed for the tests;
- *Detailed intake of the test base*: The test base is absorbed and assessed to determine the quality of the test base;

- *Test specification techniques*: Each test case is described in terms of the current point of departure, change process and the prediction of the result;
- *Checklists*: Check lists are used to check if each item on the list is correctly implemented.

8.1.3. Infrastructure

The infrastructure for testing contains all facilities and resources that are needed to be able to test properly. In this pillar the "where" question is answered. There is a distinction between facilities that are needed to perform the tests (test environment), facilities that are needed to support the test activities (test tools) and the arrangement of working environment.

8.1.4. Organization

The organization of the test process is very important. Because of the involvement of different disciplines, the conflicting interests, the unpredictability, the complicated maintenance tasks and the time constraints make the organization of the test process a difficult job. In this pillar the "how" question of the test process is answered. The organization of structured testing requires attention to the following areas:

- *The operational test process*: It is imperative to have a flexible, but stable organization structure for the department directly responsible for the tests;
- *The structure of the test organization*: The arrangement and control of the structural test organization is subjected to different factors like maturity, size and infrastructure of the organization;
- Test management: This indicates how management is positioned in the test process and how this is realized;
- *Personnel and education*: At specific moments during the test process, the need for highly skilled personnel is needed to perform the test tasks with success. It is important to know what the experience level of the test personnel is and when they are needed;
- *Structure of the test process*: Very important is to know how the test process is structured. Are the activities and phases correct and are they executable?

8.2. Requirements-based testing

Testing if requirements are implemented in the system is called requirements-based testing (RBT) and is part of the black-box tests mentioned in the previous section [58]. Requirements-based testing is a systematic approach where each requirement is considered and for it, a set of tests in the form of test-cases are derived. The requirements-based testing process addresses two major issues [53]:

- *Validation*: The firs stage is validating that the requirements are correct, complete, unambiguous, and logically consistent. In sub-section 7.3, the characteristics of measurable requirements where given;

- *Test-cases*: The second stage is designing a necessary and sufficient (from a black box perspective) set of test cases from the requirements to ensure that the design and code fully meet the requirements.

Requirements-based testing is validating rather than defect testing. The aim of these tests is to demonstrate that the system has properly implemented its requirements. The requirements-based testing process is integrated throughout the development life cycle, but the real testing takes often place during the implementation activity. The testing process can be divided into the following eight activities [53][45]:

- *Define test completion criteria*: The test effort has specific, quantitative and qualitative goals. Testing is completed only when the goals have been reached (that means, testing is complete when all functional variations, fully sensitized for the detection of defects, and hundred percent of all statements have executed successfully in single run or set of runs with no code changes in between);
- *Design test cases*: Logical test cases are defined by five characteristics: the initial state of the system prior to executing the test, the data in the data base, the inputs, the expected outputs, and the final system state.
- *Build test cases*: There are two parts needed to build test cases from logical test cases: creating the necessary data and building the components to support testing (this means, build the navigation to get to the portion of the program being tested).
- *Execute tests*: Execute the test-case steps and document the results.
- *Verify test results*: Verify that the test results are as expected.
- *Verify test coverage*: Track the amount of functional coverage and code coverage achieved by the successful execution of the set of tests.
- *Manage and track defects*: Any defects detected during the testing process are tracked to be repaired. Statistics are maintained concerning the overall defect trends and status.
- *Manage the test library*: The test manager maintains the relationships between the test cases and the programs being tested. The test manager keeps track of what tests have or have not been executed, and whether the executed tests have passed or failed.

The “define test completion criteria”, “design test cases”, and “verify test coverage” activities are addressed by requirements-based testing. The remaining activities are addressed by test management tools that track the status of test executions. The first activity is discussed and dealt with in chapter 9. where the requirements where formulated according to the characteristics for testable requirements specified by Bender RBT Inc. [53]. Designing the use-cases for the repository will be discussed in the next section.

8.3. Testing functional requirements

In this section the test-cases that will test the implementation of the requirements provided in sub-section 9.4.2. A test-case restates the requirements as a series of causes and effects, or as Lewis [41] states, a test-case defines a step-by-step process whereby a test is executed. As a

result, the test-cases can be used to validate that the design is robust enough to satisfy the requirements. If the design cannot meet the requirements, then either the requirements are infeasible or the design needs rework.

As described in section 9.4 the implementation of the technical requirements means the implementation of the functional and business requirements. That is why only the technical requirements are tested.

Both Pol, Teunissen & Veenendaal [49] and Lewis [41] identify the different items which are needed to make a good test-case. A good test-case includes the objectives and conditions of the test, the steps needed to set up the test, the data inputs, the expected results and the actual results. Other information such as software, environment, version, test Id, and test type are sometime also provided. A test-case form is shown in Figure 8.4.

Date:		*Tested by:*	
System		*Environment:*	
Objective:		*Test ID:*	
Function:		*Req ID:*	
Version:		*Test type:*	
Condition to test:			
Data/Steps to perform:			
Expected results:			
Actual result: Passed [ ] Failed [ ]			

Figure 8.4 - Test-case form

8.4. Testing non-functional requirements

Quality attributes are properties of a system by which its quality will be judged by some stakeholder or stakeholders. These attributes are not directly identifiable in the system as the user requirements. Quality attributes are requirements that are concerned with the whole system and can be found in almost every system. These are requirements every system should satisfy, so that the system will be easier to use, to understand, or to maintain. The quality attributes can be divided into two groups which have each a different type of test approach [49].

- *Dynamic quality attributes*: When assessing the dynamic quality attributes, the properties of the system in a executing status is explored (that means, the system needs to be up and running);
- *Static quality attributes*: With static quality attributes, the intrinsic-factors of the system and documentation from a developer and administrator point of view is assessed.

Quality attributes, both static and dynamic can be assessed by means of a checklist. These checklists provide measures per quality attribute, which can have a positive or negative influence

on the concerning quality attribute [49]. These measures are not unique per quality attribute and may cause some overlap with other quality attributes. Each measure outcome decides the value of the quality attribute. Section 12.3 shows a partial checklist for the quality attributes of the repository and is a good example of a checklist.

8.5. Summary

This chapter tried to provide some information about the test process, requirements-based testing, and the testing of the quality attributes. The test process describes the TMAP approach, which consist of four pillars. Each of these pillars tackles a part of the test process. The four TMAP pillars are:

- Lifecycle;
- Techniques;
- Organization;
- Infrastructure.

Testing if the system satisfies the requirements is called requirements-based testing. This approach consists of validating the requirement, and building and executing test-cases. Validating the requirements is done according to the characteristics of testable requirement list provided in the previous chapter.

A test-case defines a step-by-step process whereby a test is executed [41]. As a result, the test cases can be used to validate that the design is robust enough to satisfy the requirements. A framework of a test-case was provided in section 8.3.

Quality attributes are properties of a system by which its quality will be judged by some stakeholder or stakeholders [49]. These attributes are tested by means of a checklist and are performed on the complete system.

9. REQUIREMENTS FOR THE REPOSITORY

This chapter provides the different requirements opposed by Chess to the repository. This means that the repository needs to satisfy these requirements to be accepted. Section 9.1 stresses out that the first version of the repository very basic is. Section 9.2 identifies the artifacts that need to be stored in the repository. The stakeholders of the repository are identified in section 9.3. The user requirements, the requirements opposed by Chess to the system, are discussed in section 9.4. Section 9.5 describes the exclusions of the first version of the repository. Finally, in the last section a summarization is given of this chapter.

9.1. First version

As mentioned earlier in this thesis, the first version of the repository will be a basic version, which has a minimal set of functionalities. This means that a minimal set of requirements are presented that need to be implemented in the repository. In the near future, the repository will be expanded and extra features will be added and thus new requirements will be created.

9.2. Artifacts in the repository

As described in section 5.1, the information produced during the design activity can be classified as design artifacts. Some of these design artifacts need to be stored in the repository. The design artifacts that need to be stored in the repository are:

- *Requirements*: are the different constraints opposed to the system. There are two types of requirements, namely user and design requirements. The user requirements are the result of the requirements engineering activity and will form the input for the design activity. Design requirements are the requirement created by the designer during the design activity. Both types of requirements are regarded the same in the repository;
- *System structure*: is the design of system with all its sub-systems and files. Thus each system and sub-system structure or decomposition needs to be stored in the repository;
- *Design motivations*: are the motivations for the decisions made during the decomposition of the system in smaller and detailed parts. The design motivation actually reflects the designers design process in terms of decisions made by the designer;
- *Relationships*: are the relationships between requirements, between designs, and between requirements and designs. The different requirements and sub-systems have some sort of relationships to each other. These relationships are stored to state the link between the different artifacts.

The design motivations are a very important concept for the repository. Current tooling and literature are very scarce on implementing or describing the design motivation concept. As Frezza, Levita & Chrysanthis [22] state most of the current design methods do not address the process or interaction between the development of and solution to the design problems. By storing the design

process by supplying the motivation for specific actions or design decisions, a better understanding can be created of the system during the development project and in the future when changes are proposed or maintenance is needed. This design motivation concept is described in more detail in chapter 10.

In section 6.5 the different relationships for traceability where described. The relationships form the links between requirements, between designs, and between requirements and designs in the repository. This makes it possible to generate trace information for (sub-)systems and requirements. The repository relationships are further described in section 10.3.

9.3. Stakeholders

The repository has different and multiple stakeholders. In section 3.3.3 different stakeholders where described for the design activity documentation. The repository also has these stakeholders

The requirements engineer, who is not stated as a stakeholder in section 3.3.3., is a stakeholder of the repository and thus added to the list of stakeholders. The requirements engineer will be responsible for adding the user requirements to the repository and maintaining these requirements in the repository. All the stakeholders for the repository are illustrated in Figure 9.1.

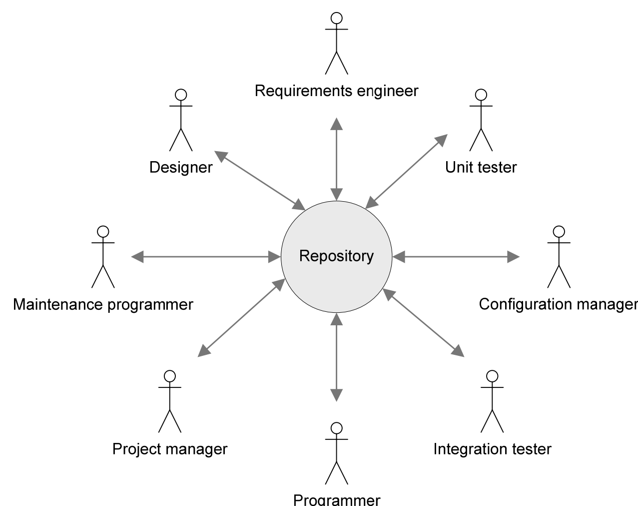

Figure 9.1 - Repository stakeholders

Each stakeholder will use the repository in its own way. This means that every stakeholder has its own requirements for the repository. In sub-section 9.4.2 the user requirements are related to the different stakeholders. Eventually, in section 13.3 the use of the repository per stakeholder is described.

9.4. User requirements

In this section the requirements for the repository are defined. As described in chapter 7. defining the testable requirements for the repository is very important. The requirements are an

important part of the requirements-based testing process which was described in chapter 8. and is used for designing the test-cases and checklists for the repository.

The requirements for the repository are described in the following sub-sections. The sub-sections are categorized in the three requirements types identified in chapter 7. , which are:

- Business requirements;
- Functional requirements;
- Non-functional requirements.

Take note that some of the following requirements are more tool-oriented then repository oriented. The reason these requirements are stated here is that the repository should support the tooling requirements, and thus is the requirement partially a repository requirement.

9.4.1. Business requirements

In this section the business requirements of the repository are formulated. Business requirements state what the repository in general shall support and are in not specific in detail. Also, these requirements will not be used in testing the repository because of their general nature. The business requirements for the repository are:

[B1] *The repository shall reduce project- and maintenance costs*: by using the repository, most of the costs generated during the project and maintenance activities shall reduce;

[B2] *The repository shall make the system expandable*: systems build with the use of the repository shall be easier to expand and it shall be easier to add new features and functionalities;

[B3] *The repository shall increase the systems life cycle*: by using the repository a system shall have a longer use;

[B4] *The repository shall increase quality of the system*: by using the repository, the system shall be more measurable and correct, which increases the quality of the system;

[B5] *The repository shall decrease the time needed to finalize a project*: by using the repository, time spend on a project shall decrease;

[B6] *The repository shall increase the quality of the documentation*: the documentation which shall be generated from the repository shall be up-to-date and correct.

9.4.2. Functional requirements

This section provides the functional requirements for the repository. The functional requirements state the technical properties of the repository and are directly identifiable and implementable in the repository. These requirements will be used in testing the repository. The functional requirements are:

[F1] *The repository shall allow storing partial (sub-)systems and requirements*: it is allowed to store an incomplete (sub-)system of a project in the repository. The need for more information to complete the (sub-)system can be a reason for doing this is. Partial designing is allowed as long as the hierarchy of the repository stays intact. Parentless requirements are allowed in the repository, but parentless designs are not allowed. The only time a design is allowed to be parentless, is when it is the root of the system. This technical requirement is essentially a tooling or compiler requirement, because the tooling must deal with this problem;

[F2] *The repository shall not enforce optional input*: optional information shall be optional in the repository, thus only important and critical information, needed to let the repository function correctly, shall be mandatory. The mandatory data needed in the repository needs to be at a minimal to make the repository more usable and flexible;

[F3] *The repository shall store the information as sets of hierarchical trees*: the design and requirement data stored in the repository shall form a set of hierarchical trees where the root nodes describes the outline at the highest level while the leaves go into more detail. All requirements and (sub-) systems form a set of trees where the root shall be formed by the main global requirement or system and the leaves are the implemental requirements and sub-systems. There are multiple requirement roots allowed, opposed to only one system as root;

[F4] *The system shall support the ability to add metadata information*: adding information like author, description, date, contributors, version, and motivation etc. to each item in the repository shall be supported. This information shall be optional and shall be used for documentation purpose. In the first version information about the author, the description and the motivation need to be supported;

[F5] *The repository shall support relationships*: relationships between requirements, between (sub-)systems and between requirements and (sub-)systems shall be supported. Linking these items to each other creates the hierarchical trees mentioned in requirement [F3] and shall make it possible to use traceability in the repository;

[F6] *The repository shall support file declarations of any type of file*: it shall be possible to declare images, documentation, guides and other files as part of a system. Requirements enforcing the implementation of these files shall be linked to these files and thus implementing this requirement.

9.4.3. Non-functional requirements

This section provides the non-functional requirements or quality attributes for the repository. First the user defined non-functional requirements opposed by Chess are discussed. After the user non-functional requirements, the general quality attributes for the repository are provided.

Non-functional requirements are requirements that are not identifiable in the repository, but in the repository as a whole. The non-functional requirements for the repository are:

[N1] *The repository shall be free or have a low price*: no or negligible costs shall be linked to the use of the repository. The problem with current tooling is that is very

expensive. When such a tool is used, a big part of the project budget is gone to the license costs;

[N2] *The repository shall be easy to setup*: the repository shall be easy to install and configure for use. The repository shall not take to much time to setup before using it;

[N3] *The repository shall be stand alone*: this means that using the repository shall not be restricted to a database or server. The repository shall be allowed to be used on a personal computer or workstation of a single user;

[N4] *The repository shall be human editable*: users shall be able to edit the data stored by hand in the repository in a text editor. Important is that the data shall be stored in such a way that the users understand what is stated in the repository;

[N5] *The repository shall support inconsistency checking*: the repository shall support and analyze the repository data for inconsistencies between requirements and (sub-)system. This means that all requirements need to be implemented otherwise there is an inconsistency in the system;

[N6] *The repository shall support impact analysis*: determining the impact of a change request shall be supported by the repository. It shall be able for tooling to determine, by providing a list of requirements, files, and (sub-)systems, what items will be affected by the change request;

[N7] *The repository shall support the generation of simple documentation*: different types of documentation shall be able to be generated from the data stored in the repository. The documents should be able to be generated are: requirements, system structure, consistency analysis, impact analysis, and requirements traceability documentation.

Quality attributes are the properties of the repository by which its quality will be judged by Chess. Only some of the attributes defined by ISO 9126 standard [33] are usable for the first version of the repository. The quality attributes that will be used to test the first version of the repository are:

[Q1] *Accuracy*: expresses the ability to correct errors in the repository to agreed results and impact;

[Q2] *Maturity*: expresses the ability to reduce disruptions created by errors in the repository;

[Q3] *Fault tolerance*: expresses the ability to maintain a specific level of operation in the repository incase of an error or a disruption in the repository;

[Q4] *Recoverability*: expresses the ability to repair the operation level and data of the repository in case of a malfunction, and the time and effort that is necessary to repair this malfunction;

[Q5] *Understandability*: expresses the effort needed to understand logical concepts of the repository and the usability of these concepts;

[Q6] *Learnability*: expresses the effort needed to learn the use of the repository;

[Q7] *Operability*: expresses the effort needed to use and control the repository;

[Q8] *Attractiveness*: expresses the effort needed to attract users for using the repository;

[Q9] *Analyzability*: expresses the effort needed to diagnose shortcomings, or error causes or the identification of modifiable parts of the repository;

[Q10] *Changeability*: expresses the effort needed for changing, or removing errors or environmental changes in the repository;

[Q11] *Testability*: expresses the effort needed to validate a changed repository;

[Q12] *Installability*: expresses the effort needed to install the repository in a specified environment.

9.4.4. Requirements matrices

In this section the functional requirements are related to the business requirements and stakeholders of the repository. In Table 9.1 the relationships between the business and the functional and non-functional requirements are illustrated. The functional and non-functional requirements are refinements of the business requirements. This means that when the functional or non-functional requirements are satisfied in the repository, the related business requirements are also satisfied.

Business requirement	**Functional and non-functional requirements**
[B1]	[F1][N1][N5][N6]
[B2]	[F1][F3][F4][F5][F6][N4][N5][N6]
[B3]	[F1][F4][N4][N5][N6]
[B4]	[F1][F3][F4][F5][N5][N6]
[B5]	[F1][N1][F2][N5][N6]
[B6]	[F2][F3][F4][F5][F6][N5][N6][N7]

Table 9.1 – Business, functional, and non-functional requirements matrix

Each of the summarized functional requirements has multiple stakeholders, which need to be identified. Identifying and registering the stakeholder of requirements is according to Gotel & Finkelstein [23][24] very important for pre-requirements specification tracing. Pre-requirement specification traceability is traceability is concerned with those aspects of a requirement's life prior to its inclusion in the requirement document (requirement production).

Table 9.2 illustrates the stakeholders identified in section 9.3 for the different functional and non-functional requirements. Most of the requirements are assigned to the designer. This should not be a surprise, because the repository will mainly be used during the design activity.

Functional and non-functional requirements	Stakeholders							
	Project manager	Configuration manager	Designer	Programmer	Unit tester	Integration tester	Maintenance programmer	Requirements engineer
[F1]		X	X	X			X	X
[F2]	X	X	X	X	X	X	X	X
[F3]	X	X	X	X	X	X	X	X
[F4]			X				X	
[F5]			X				X	
[F6]			X		X	X	X	X
[N1]	X	X	X	X	X	X	X	X
[N1]	X	X	X	X	X	X	X	X
[N3]		X	X	X			X	X
[N4]			X		X	X	X	X
[N5]			X	X	X	X	X	X
[N6]	X	X	X	X	X	X	X	X
[N7]	X	X	X	X	X	X	X	X

Table 9.2 - Technical requirements and stakeholders matrix

9.5. Exclusions

Knowing what the requirements are for the repository, it is also important to state what this first version of the repository is not going to support [58]. As Wieringa [69] describes, stating exclusions in two or three sentences is part of the process of expectation management. It is useful to write down some of the responsibilities that the product is not going to have. In the summary below the exclusions for the first version of the repository are listed.

- *Change registrations*; the repository will not save changes and keep track of change histories although there are some ideas on how to implement this in the future after the termination of this research project;
- *Building software*; building software for analyzing and checking the repository is not part of this research project as described in section 2.6. The repository is only a dumb storage place to store the different design artifacts;
- *Documentation generation*: documentation generation from the data stored in the repository is part of this research project, but this will be in a really simple and standard format. Another student will investigate documentation generation for the different documentation standards like JSTD 016 [63], after this research project is finalized.

9.6. Summary

This chapter provided the requirements for the repository. Important to understand is that these requirements specify the criteria for the first version of the repository. The first version of the repository will be a foundation for future work.

The design artifacts identified for storage in the repository are:

- Requirements;
- System structure;
- Design motivations;
- Relationships.

The design motivations are a very important concept for the repository. By storing the design process by supplying the motivation for specific actions, a better understanding can be created of the system during the development project and in the future when changes are proposed or maintenance is needed. This design motivation concept is described in more detail in chapter 10.

This chapter provided the business-, non-functional-, functional requirements and quality attributes. After identifying the requirements, the non-functional- and functional requirements where related to the business requirements. This means that when the functional requirements are implemented in the repository, the business requirements are also implemented in the system. Also, the functional requirements are related to the different stakeholders to identify which stakeholder benefits from the different requirements. The stakeholders of the different requirements are:

- Project manager;
- Configuration manager;
- Designer;
- Programmer;
- Unit tester;
- Integration tester;
- Maintenance programmer.
- Requirements engineer

At the end of this chapter some exclusions where provided. These exclusions will not be support in the first version of the repository. These exclusions are:

- Change or version registration;
- Building software for the repository;
- Detailed documentation generation.

10. DESIGNING THE REPOSITORY

In this chapter the design of the repository is provided. It will describe what the concept behind the repository is and how the different artifacts identified in section 9.2 are used. Section 10.1 clarifies the repository approach as it will be used in the repository. Section 10.2 discusses how this approach can be used in the repository. Section 10.3 provides a description about what relationships are supported by the repository. In section 10.4, traceability in the repository is discussed. Section 10.5 describes the analysis types that are supported by the repository. In the final section, an overview is given of this chapter.

10.1. Repository approach

The design concept or repository approach of the repository is clarified in this section with an example. This example describes roughly how a typical design activity of a house is executed. The house represents a system, which needs to be build. This process is also represented in Figure 10.1. At the end of this section it should be clear what the train of thought is behind the repository.

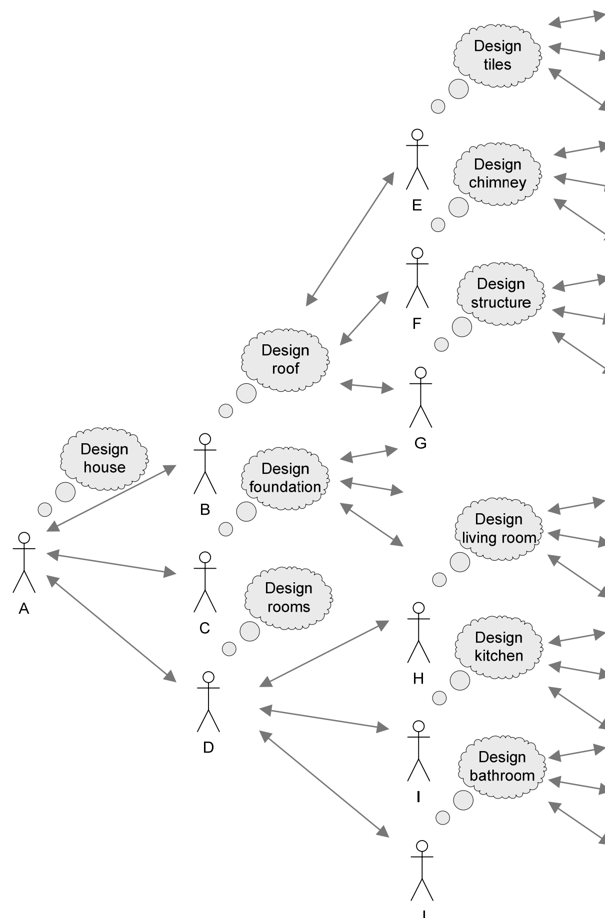

Figure 10.1 - House build process

10.1.1. Example justification

The example described in the next section is using a construction or building related theme instead of a software related theme. Although this is a complete different engineering field, the design processes of both engineering fields have similar activities and ideas. Construction examples are also easier to explain, because of the visual perception the reader can make. To finalize this justification, other authors like Vliet [67] and McConnell [44] both use building or construction examples in their books to clarify system engineering concepts.

10.1.2. Example

Imagine a person, called John, who wants to build a simple house. Building a house is a lot of work and very difficult. So, John needs people who can help him realize his idea. These people need to be specialized in building houses otherwise I will be a difficult task.

Before the house can be build, an architect is needed to design the house in brought outlines. The architect is visualized as person A in Figure 10.1. To know what kind of house John wants to build, the architect interviews John to capture the requirements for the house. This is the requirements definition phase as it is described in sections 3.2.1, 3.3.1, 4.2.1, and 4.3.1. The results of this activity are the user requirements for the house.

With these user requirements in mind, the architect is going to design the architecture of the house. This process is illustrated by means of the balloon hanging above person A. The architect designs or decomposes the house roughly into three parts (or in system engineering terms; sub-systems), mainly according to the user requirements and formulates new requirements for these parts. The parts identified by the architect for the decomposed house are:

- The foundation where the house is going to be build upon;
- The different rooms of the house;
- The roof.

The architect can decide to design the house further into more detail, but instead he hires persons B, C, and D to do this work. A reason to do this is because these people are experts in these parts, or the architect just does not want to design these parts. Each of these three new persons gets the appropriate requirements linked to their part and designs their part.

Each of these persons decomposes their part into new parts and creates new requirements. Again, this process is the same as the design process executed by the architect and is illustrated in Figure 10.2. The newly created parts are transfer to a new person (person E till J) who again performs the same process until eventually the design process is at such a detailed level that the actual physical building of the house is represented.

10.1.3. Reflection

The whole idea of this approach is that the house is decomposed step by step in smaller pieces by different people. Each person designs the house and formulates requirements, which then will be used as input for the next person who will design his part of the house. This design process is illustrated in Figure 10.2 as a black-box.

Figure 10.2 – Black-box of the design process

The user or design requirements are the input for the design process. With these requirements as input the designer decomposes the system or sub-systems into new sub-systems. During these decomposition new requirements, relationships, system structures, and design motivations are created. These artifacts form the output of the design process. Important to understand is that the input requirements also may be among the output requirements. This means that input requirements do not have to be implemented in the system currently designed.

The human factor is very important in this approach. Each person needs information to complete its task. The outline of the system can easily get lost when different persons work on it. That is why these sub-structures of the system need to be organized at a central position.

When all the designing is done correctly and the design of the house is correct, a person at a lower level, whom is responsible for a sub-part, does not need to know the requirements of the upper levels are. As long as he knows what is task is and what the requirements are for his task he can execute the task. In Figure 10.2, a person who is responsible for laying the tiles on the roof, only needs to know what tiles he should use, where they need to be placed and how they should be placed on the roof. He does not need to know how the roof is constructed let alone how the foundation of the house or the whole house is build.

10.2. Using the repository approach

The design process illustrated in Figure 10.1 can be translated into Figure 10.3 which shows the artifacts of each design process. Each design process exists of a system structure, requirements, design motivations, and the relation between these items. Figure 10.3 also shows that each design process is related to another design process. These different relations between the artifacts are elaborated in the next section.

Figure 10.4 gives an impression how the design process stored in the repository and how different stakeholder may use of the repository. All the necessary data is centrally stored in one place so all users can interact with this data. Some of these stakeholders are responsible for adding, changing, and deleting the data in the repository. Others are only retrieving information from the repository. The use of the repository by the different stakeholder is described in section 13.3. Saving the data centrally overcomes the problem that different versions and different information is used by the different users.

Documentation is also generated from the data stored in the repository instead of the documents being the repository as is the case at this moment for allot of projects. This means that it is not necessary to write the different documents by hand. All the necessary information is provided by the repository from where the documents are generated. Document generation is described in section 11.6.

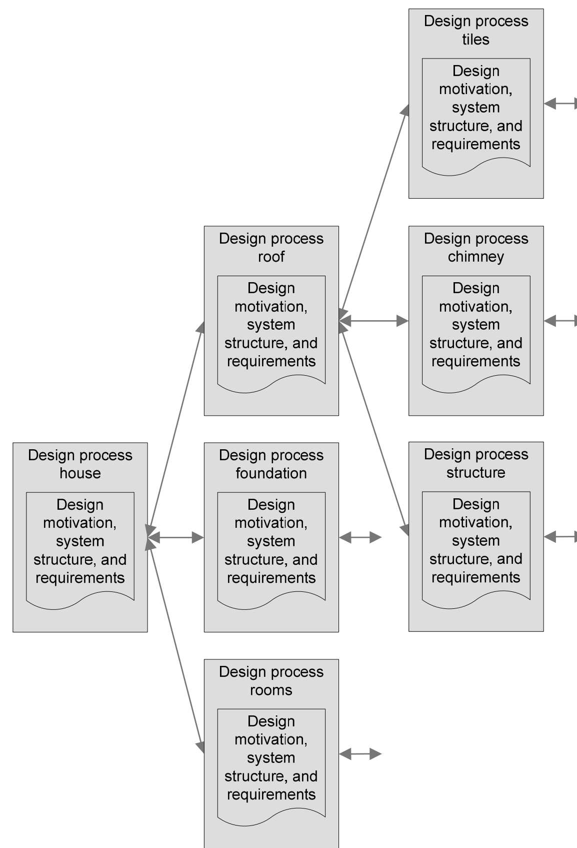

Figure 10.3 – The house build process and the artifacts

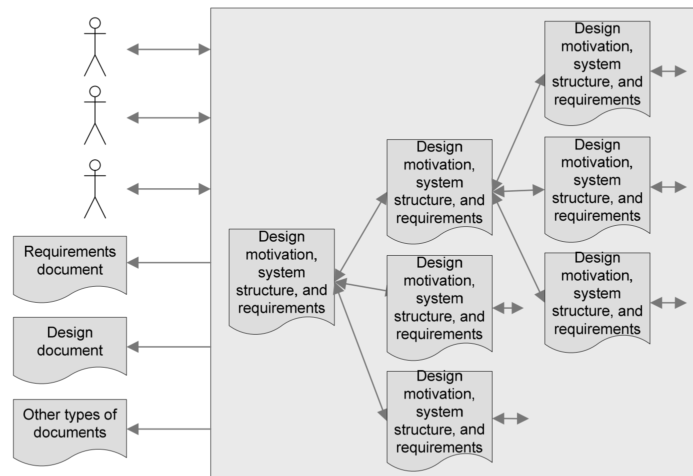

Figure 10.4 - Repository use and placement

10.3. Relationships

In section 6.5 different types of relationships that are found in the literature where discussed. These relationships should in some way be used in the repository.

Sub-section 10.3.1 clarifies the term dependency and excludes the dependency relationship. Sub-sections 10.3.2 till 10.3.4 will describe the different relationships of the repository. To clarify the different relationships, the following abstracts will be used:

- Requirements are identified as [REQ];
- System structures are identified as [SYS];
- Files are identified as [FIL].

Section 10.3.1 will explain why dependency relationships are not incorporated in the first version of the repository. The last sub-section tries to clarify the relationship usage.

10.3.1. Dependency term and relationships

Before describing the repository relationships, some comments need to be given about the information provided in section 6.5. Section 6.5 uses the term dependency for all types of relationships [22]. This is according to the author of this thesis not a correct usage of this term. The most suitable definition of dependency would be: "reliance on something for help or support" [46][62]. Not all the relationships are of this nature. That is why different terms are used for the relationships of the repository

Chess identifies different types of relationships where dependency relationships are one of. Although dependency relationships are part of the repository it will be used in an explicit way. Dependency relationships will be expressed by formulating a design requirement which states the dependency. An example of such a dependency requirement is "There shall be a dependency between…". The implementation of this dependency depends entirely on the designer who is responsible for this implementation. This relationship can also not be automatically checked.

10.3.2. Requirement refinement

The requirement refinement relationship has the form [REQ refines REQ]. It means that a child requirement or sub-requirement is a refinement of another parent requirement and describes in more detail the use of this requirement. This can be realized in the repository in two ways:

- *Implicit*: the relationship is defined by a parent-child structure. The sub-requirement is nested into the body of or under the parent requirement. An example is the nesting of a section in a chapter, where the section is part of the chapter;
- *Explicit*: the relationship between (sub-)systems is defined by literally stating the relationship between these items. This is done by 'mapping' a requirement onto another requirement in the repository. This mapping thus tells a user or computer that there is a relation between two requirements, which could not be seen without this mapping.

10.3.3. System refinement

The system refinement relationship has the form [SYS refines SYS]. This relationship is actually the same as the requirement refinement. It illustrates that the sub-system is a decomposition of the system. This can be created in two ways:

- *Implicit*: this is done in the same way as the implicit requirement refinement relationship, only the requirements are now (sub-)systems;

- *Explicit*: this is done in the same way as the explicit requirement refinement relationship, only the requirements are now (sub-)systems;

10.3.4. Requirement assigning

The requirement mapping relation has the form [REQ assigned to SYS]. This relationship means that a sub-system needs to do something with this particular requirement, but the designer does not specifies this. It is up to the designer of the sub-system to decide if a child of this sub-system is going to implement this requirement or that it is also passes though the requirement to be implemented by a child sub-system. Requirements assigning is also called "mapping" in the repository and is done implicitly.

10.3.5. Requirement implementation

Requirement implementation relationship can have different forms and is done implicitly. Implementing the requirement means that the item directed to is responsible for implementing the requirement. In other words, the item fulfills or satisfies the requirement.

Files can be declared as part of the system structure. Files can be images needed on a webpage, source code of a component, or a user guide that is needed in the system. That is why relationships between these files are needed in the repository.

The different forms of the requirement implementation relationships are:

- *The requirement is implemented by a file*: this relationship has the form [DES implements REQ]. This relationship means that a particular file implements a particular requirement. This file can be a file containing source-code responsible for the implementation of the requirement or an image defined as requirement. For example a requirement state that a webpage shall illustrate the company logo and the designer declared this logo as a file in the repository. Then the file implements the logo requirement;
- *The requirement is implemented by another requirement*: this relationship has the form [REQ implements REQ]. This relationship means that a particular requirement implements another particular requirement. This may look like a requirement refinement relationship;
- *The requirement is implemented by a design*: this relationship has the form [DES implements REQ]. This relationship means that a particular sub-system is constructed in such a way that it fulfills a particular requirement. The designer defines two sub-systems, named Front-Office and Back-Office as stated in a requirement. By creating these two sub-systems the requirement is implemented by these two sub-systems.

10.3.6. Relationship explanation

The goal of this sub-section is to clarify some questions which may arise when using mapping or implementation relationships. Figure 10.5 will be used as an illustration for the explanation of these two relationships. The following concepts will be explained:

- *The difference between the mapping and implementation relationship*: is that the implementation relationship indicates that the sub-system directed is responsible for the

implementation of a requirement. This sub-system can not assign the requirement to another sub-system. This is illustrated in Figure 10.5 with the arrow between "System" and "Sub-system 2". This relationship states that "Sub-system 2" implements the requirement "Req 2". The mapping relationship however only tells the sub-system to do something with the requirement as long as it is not implementing the requirement. "Sub-system 2" receives one requirements from "system" as a mapping relationship;

- *Mapping or implementing requirements onto system structures*: requirements may be mapped or implemented by a sub-system. This means that a sub-system needs to do something with the requirement. Stating the mapping or implementation relationship is used as input for the next design process. It clearly informs the designer of the next design process and what is expected of the next sub-system. Thus, the designer of "Sub-system 2" knows that he needs to design "Sub-system 2" in such a way that it fulfills "Req 2" and does something with "Req 3";
- *How does it work with mapping*: in Figure 10.5 "Sub-system 2" receives a mapping of "Req 3". The designer of this sub-design need to do something with this requirement. He decides to define a new sub-system named "Sub-system 3" and also creates two new design requirements "Req 4" and "Req 5". The designer then maps the two new requirement "Req 4" and "Req 5" and the requirement "Req 3" received from "System" to the a new sub-system. It is thus up to the designer of "Sub-system 3" to do something with the three requirements. By passing the requirements through each sub-system an opportunity for traceability is created. This is discussed in the next section.

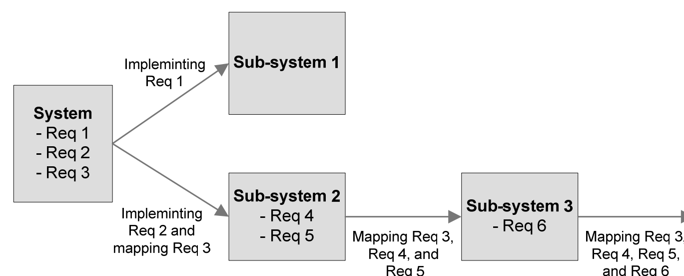

Figure 10.5 - Mapping and implementation description

10.4. Traceability in the repository

Traceability can easily be executed in the repository, because requirements, system structures and design motivations are linked to each other with relationships which form sets of hierarchical trees. Partially, the creation of traceability was discussed in sub-section 10.3.6. Section 10.4.1 describes the types of traceability supported by the repository. In section 10.4.2 the directions of traceability is described. In section 10.4.3 an example of a trace is given.

10.4.1. Types of traceability

In general traceability is related to just tracing user requirements to their implementation [23][24]. But as described in section 6.5 system structures and design requirements are also traceable. There are three types of traceability defined in the repository. These three types are:

- *Tracing of the user requirements*: using user requirements traceability gives the customer, but also the designer, an overview whether all the user requirements are implemented in a system and where and how these requirements are implemented.;
- *Tracing of the design requirements*: this is actually the same type of traceability as tracing user requirements, only just for the design requirements. This type of traceability can be very helpful for the designers to understand the design of the system;
- *Tracing of the system structures*: tracing the structure of the system gives the designer an overview of all the sub-systems of the system or sub-system. This can be very helpful for understanding what the components of a system are.

All traces can be made more understandable by reading the different design decisions added to each of the system structures. The design decisions provide information about the design process and thus why and how some decisions are made.

10.4.2. Directions of traceability

Forwards and backwards traceability as suggested by Gotel & Finkelstein [23] is supported in the repository. All requirements can be traced from their roots to its implementation and vise versa. During such a trace, the motivations for the different decisions can be reviewed to understand the trace route and the decisions made during the design process.

Only Post-requirements specification traceability is supported by the repository. Requirements are stored in the repository as they are used for testing. No prior information about the creation of the requirements is stored in the repository. Thus the origin of a requirement can not be retrieved from the repository.

10.4.3. Example of a trace in the repository

In this section an example of a trace is given. The example will trace a user requirement to its implementation, but the same principle of tracing is also usable for the design requirements and system structures.

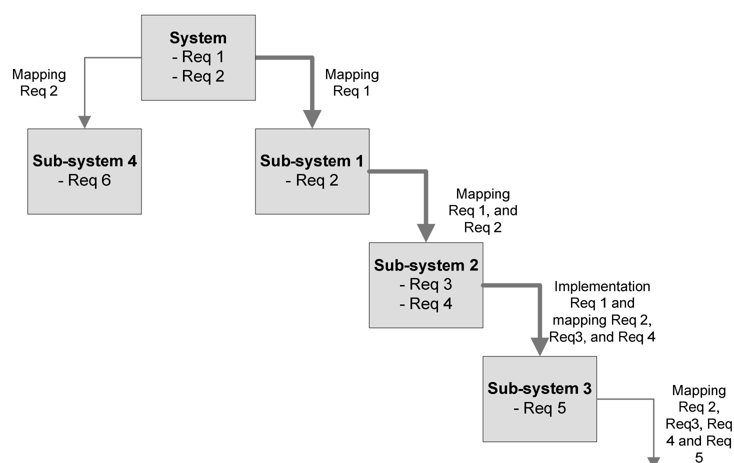

Figure 10.6 - An example trace of “Req 1”

The example in Figure 10.6 shows a trace (the arrows in bold) of "Req 1" from "system" to "Sub-system 3". This trace is made possible through the mapping and implementation relationships between the different sub-systems. When forward tracing is used, the trace of requirement "Req 1" starts in "System" where the requirement is defined and ends in "Sub-system 3, where the requirement is implemented. The end sub-system is reached through two other sub-systems. The trace in this example looks like: "System" -> "Sub-system 1" -> "Sub-system 2" -> "Sub-system 3". A backwards trace can be performed in the same ways as the forwards trace. Backwards traces are done in the same way only bottom-up. Again this is possible by the hierarchical structure of the data in the repository.

10.5. Types of analysis

This section provides the different analysis type for the repository. These analysis types are made possible because of the different relationships supported by the repository. Section 10.5.1 points out that these analysis types supported by the repository need tooling to be used automatically. Section 10.5.2 describes the different analysis types and how this is supported by the repository.

10.5.1. Analysis and tooling

Important to understand is that the repository will support these analysis types but does note execute them. Tooling should be written to execute these analyses automatically. The support of these analysis types by the repository will be realized by the hierarchical trees of data stored in the repository. These trees, for requirements and (sub-)systems, are linked to each other by different types of relationships mentioned earlier in this chapter. All analysis types use these trees to analyze the data in the repository.

Take note that it is beyond the scope of this research project to define the algorithms for the compiler or tooling to execute these traceability analysis. For this research project is it sufficient to define the relations which will be used to produce the traceability information.

10.5.2. Analysis types supported by the repository

The repository will support most of the different analysis types mentioned in section 6.6. Only the dependency analysis will not be supported in the first version of the repository, because dependency relationships are not supported. The reason that this will not be supported was given in sub-section 10.3.1. The analysis types that are supported by the repository are:

- *Identifying inconsistencies*: Identifying inconsistencies is called in the repository consistency checking. Consistency checking tries to identify the requirements which are not implemented. This means that the system does not fulfill these requirements and does not satisfy the customer's expectation. Besides the user requirements, which satisfy the customer, the design requirements also need to be implemented. If this is not the case, then the design is incorrect. To have no inconsistencies, all requirements need to be implemented by a (sub-)system, requirement, or file. The requirements can be traced to their implementation via the different relationships in the repository. The (sub-)systems in

the repository form a hierarchical tree where at the beginning in the root, the user requirements are defined. Through each sub-system the requirements are than redirected to the eventual sub-system, where the requirement is implemented;

- *Relation following*: Following specific requirements or element to other requirements or elements by following the relations between these requirements and elements is automatically supported by the repository due to the hierarchical structure. Because every sub-system implements or maps a requirement onto another sub-system, the flow of the requirement can easily be created;
- *Requirement verification*: Verifying that requirements have been met or fulfilled is really easy in the repository. The implementation of a requirement is directly stated in a (sub-)system. This (sub-)system where the requirement is implemented can be identified and is described or motivated, as well as the item which fulfils the requirement and the motivation for this implementation;
- *Impact analysis*: Requirements and designs may change as shown in chapter 5. The repository shall support the function to illustrate the impact of a design or requirement change on the system. This analysis is done by looking at the requirements and (sub-)systems effected by the change. Because of the hierarchical tree structure of both the requirements and the (sub-)systems, a simple and small change can lead to an explosion of effected (sub-)systems and requirements.

10.6. Summary

This chapter provided information about the design of the repository. But before the design of the repository was discussed the repository approach was explained to give a better understanding of the concept behind the repository.

The design process supported by the repository is al about decomposing a system into sub-system till it researches the lowest level of the decomposition. In every decomposition step requirements are related to the different sub-systems to specify what these sub-systems should accomplish. The design motivations are added to each decomposition step to provide some explanation why this decomposition step is done. All this information should be stored in a central place from where all users should receive their information and from where documentation can be generated.

To store all data in the repository, relationships need to be implemented between the different artifacts. The repository identifies four relationship types. These are:

- Requirement refinement;
- System refinement;
- Requirement assigning;
- Requirement implementation.

The difference between a mapping and an implement is important to understand. The implementation relationship is used when an sub-system or file satisfies the requirements which is implemented. A mapping relationship only passes the requirement through to another sub-system.

Because the repository forms a hierarchical tree of sub-systems with requirements and design motivations related to it, traceability can be used. The repository identifies three types of traceability which can be used in a forward and backward direction. These types are:

- Tracing of the user requirements;
- Tracing of the design requirements;
- Tracing of the system structures.

Because traceability is supported in both directions by the repository, the different analysis checks are supported. The analysis types that are supported by the repository are:

- Identifying inconsistencies;
- Relation following;
- Requirement verification;
- Impact analysis.

11. THE IMPLEMENTATION OF THE REPOSITORY

In this chapter the implementation of the repository is discussed. Central in this chapter is the structure of the repository. In sections 11.1 the foundation is laid for the repository in the form of defining the language for the repository. In section 11.2 the structure of repository and how these will be used is described. Section 11.3 describes how the mapping and implementation elements should be used. Section 11.4 describes how elements in the repository are made unique. In section 11.5 the decisions is made if the repository shall be file based or a database. In section 11.6 document generation from the data stored in the repository is discussed. Finally, the last section summarizes this chapter.

11.1. Extensible Markup Language

The first thing to define before implementing the repository is the structure or markup of the repository. Extensible markup language (XML) is a highly structured representation for storing data and is receiving growing support in the e-commerce community. That is why XML will be used to store the requirements and design data. According to Maletic, Collard & Simoes [43], XML and the techniques used with it are very usable for storing and linking items to other items. XML offers many immediate benefits [27][6]:

- *Plain-text*: XML is plain-text based;
- *Human editable*: Because of the plain-text properties, XML documents can be edited in a text editor or a program alike;
- *Database functionalities*: an XML-based repository offers features of database technology;
- *Extendable*: it is possible to create your own tags, or use tags created by others, that use the natural language of a specific domain, that have the specific attributes needed, and that make sense to the users;
- *Separation of data and representation*: the data and presentation formats are separated, which will make document generation very easy;
- *Parseability*: XML files are easy to parse;
- *Available tooling*: there is a wide range of tools and parsers available for parsing and editing XML.

The database functions and the extendibility are especially important features for the repository. This makes the XML repository behave like a database. The human editable property of XML is requirement [N4] that is opposed by Chess to the repository.

11.2. Structure of the elements

The repository exists of different elements as show in Figure 11.1. and shows the element tree of the repository, which is a simplistic data model. The class diagram of the repository is shown in

Figure 11.2. It clearly shows what and how the elements are related to each other. Appendix A shows Figure 11.1 in XML format as it will look like in the repository.

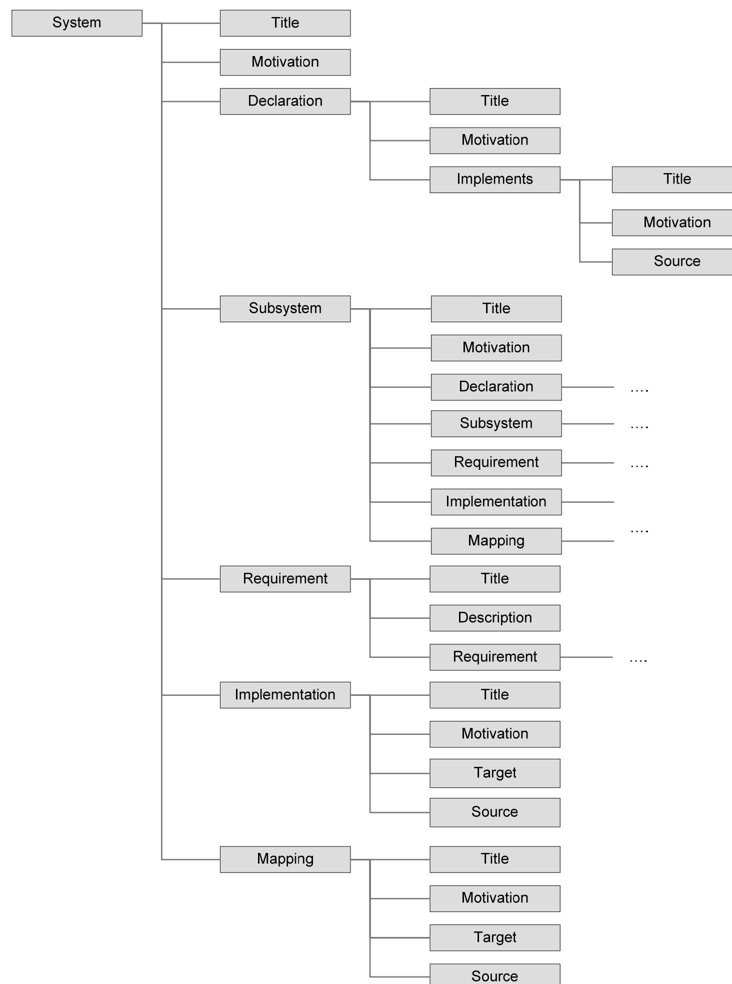

Figure 11.1 - A simplified element tree of the repository

The root of the repository is the System element and consists of seven sub elements. One of these elements is the Subsystem element, which consist of the same elements as the System element. This means that in a sub-system another subsystem can be defined. This creates the hierarchical tree as stated by requirement [F3] in sub-section 9.4.2.

The Requirement element which is a sub element of both the System and Subsystem element can also contain a Requirement element. This means that the requirement is a sub-requirement. Like the Subsystem element, this also creates a hierarchical tree. During the design process, more sub-requirements can be added to this tree.

A good way to describe the structure and syntax of an XML document, including the permissible values for much of that document's content, is the use of Document Type Definitions (DTD) [6]. The rules described in a DTD are called validity constraints, which ensure that any XML data is conform its associated DTD. This DTD can then be used in conjunction with a validation parser to validate the existing XML data or enforce validation during the creation of XML documents by a human author. Appendix B shows the DTD of the repository.

In the following sub-sections each element shown in Figure 11.1 is described in detail. For each element the attributes and its sub-elements is provided.

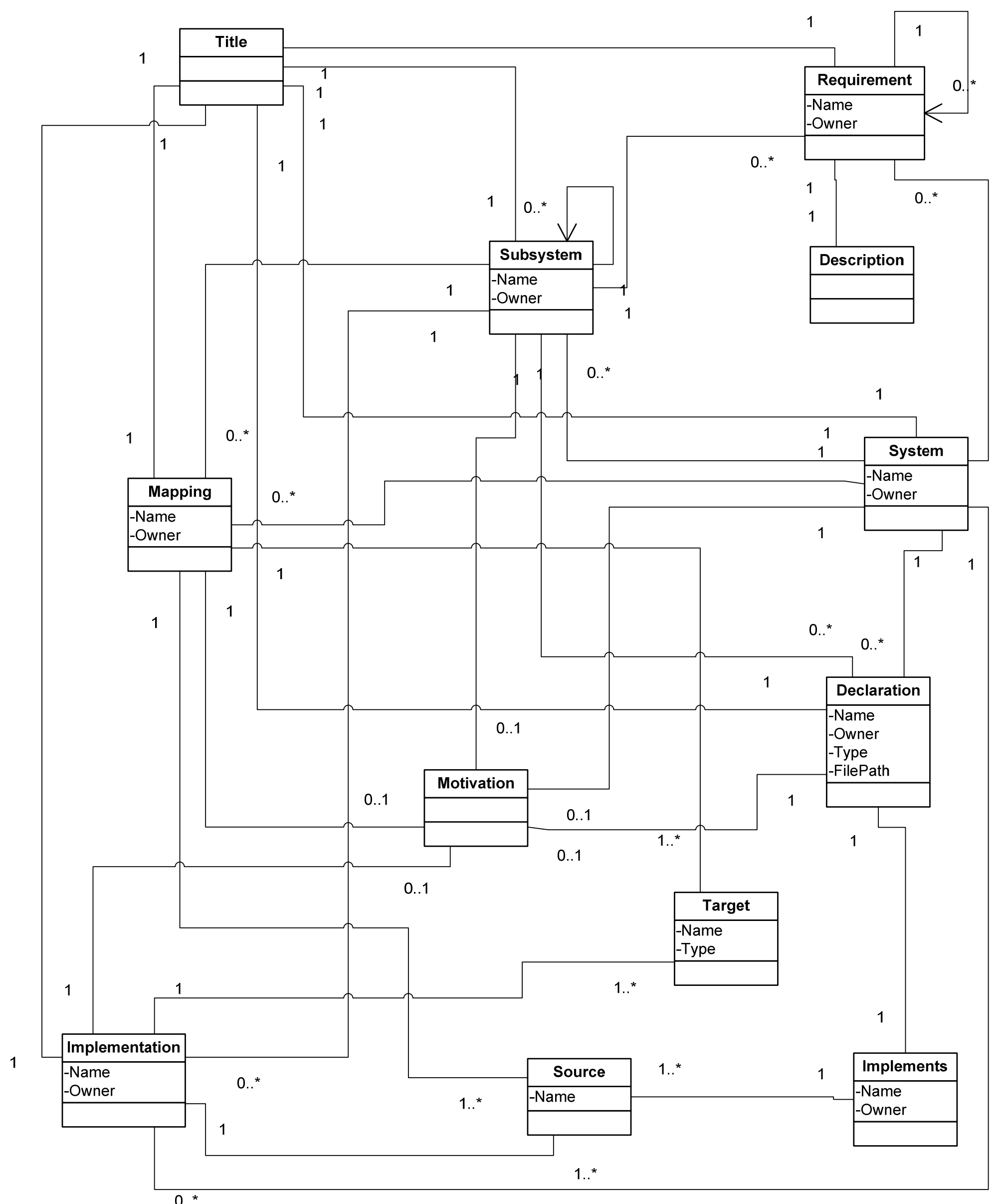

Figure 11.2 - Class diagram of the repository

11.2.1. System element

The System element is the root of the repository and declares the system to be build. The system represents the goal of the development project, the system which needs to be built. The System element has the structure listed in Figure 11.3 and consists of attributes and sub-elements summarized in Table 11.8 and Table 11.9.

The System element is always the first element in the repository and can only exist once. Using this element means the start of the design process and will often have the name of the system. A System element needs to have a name for reference purpose. This name needs to be unique within the repository. Each System element has also the option to define the owner or creator of

this system by using the Owner attribute. This is in most cases set to the designer its name that created the system.

```
<System Name="SY_Fiscal_Tax">
  <Title>Fiscal Tax</Title>
  <Motivation>The fiscal tax module is a free service to Bus Lease Netherlands drivers
  and fleet managers for requesting the catalogue value of their lease car for tax
  purposes.</Motivation>
  <Subsystem Name="SS_Software">
    …
  </Subsystem>
  …
  <Implementation Name="IM_Doc_Part">
    …
  </Implementation>
  …
  <Mapping Name="MA_Software">
    …
  </Mapping>
</System>
```

Figure 11.3 - A basic System element

Attributes	**Explanation**
Name	The name for the design.
Owner *(Optional)*	Indicates who the owner or creator of this system is.

Table 11.1 - Attributes of the System element

Elements	**Explanation**
Title	Indicates for documentation purpose what the title is of this system.
Motivation *(Optional)*	A description of what this system means and does.
Declaration *(Optional)*	This element declares external designs, or files which are used in the System element.
Subsystem *(Optional)*	This is a decomposition of the system it is currently designing. The sub-systems are children of the System element.
Requirement *(Optional)*	These are the requirements created in this design process. These can be the user requirements and design requirements.
Implementation *(Optional)*	Indicates if requirements are implemented by a sub-system or other requirements.
Mapping *(Optional)*	Indicates the relationship between a requirement or the system and its sub-systems or other requirements.

Table 11.2 - Elements of the System element

The System element consists of multiple elements. The Title element provides the Systems name, which will be shown in documentation or in the tooling. It is also possible to motivate the

system or describe it. This can be especially helpful in later stages of the project when changes occur and the design needs to be redesigned.

The System and the Subsystem elements which is described in the next sub-section, are the most complex elements in the repository, because it encapsulates allot of other elements. In the System element new declarations, sub-systems, requirements, and relationships are created. Every element of the System element, except for the Title and Motivation element, can exist zero, one or multiple times in the System element. These elements will be discussed in more detail in the next sub-sections.

When creating a sub-system in the system, requirements need to be mapped onto the new sub-systems by using the Mapping. The user requirements for the system are stated in the System element and will be related to the different sub-systems. The requirements created in the system do not automatically forward to all new sub-systems and need to be stated explicitly. The refinement relationship between de system and de sub-system, is automatically generated, because of the parent-child relationship. When requirements are implemented by a sub-system or declaration, the Implementation element is used.

11.2.2. Subsystem element

The Subsystem element is almost the same as the System element described in the previous sub-section. As shown in Figure 11.4, Table 11.3 and Table 11.4, the attributes and sub-elements of the Subsystem are the same as the System element.

```
<Subsystem Name="SS_Software">
  <Title>Software</Title>
  <Declaration Name="SS_Input_Page">
    …
  </Declaration>
  …
  <Requirement Name="RE_Configuration_File_Dependence">
    <Title>Configuration File Dependecy</Title>
    <Description>The different screens depend on the configuration file. This is needed
    form showing the right content</Description>
  </Requirement>
  <Mapping Name="MA_Configuration_File">
    …
  </Mapping>
  …
</Subsystem>
```

Figure 11.4 - A basic Subsystem element

Attributes	Explanation
Name	The name for the design.
Owner *(Optional)*	Indicates who the owner or creator of this sub-system is.

Table 11.3 - Attributes of the Subsystem element

The main difference with the System element is that a sub-system is not allowed to be the root of the system and can exist more then once. Sub-systems are the decomposition of the System element. Sub-systems can also be composed into new sub-systems.

Elements	Explanation
Title	Indicates for documentation purpose what the title is of this design.
Motivation *(Optional)*	A description of what this sub-system means and what this design does.
Declaration *(Optional)*	This element declares external designs, or files which are used in the System element.
Subsystem *(Optional)*	This is a decomposition of the system it is currently designing. The sub-systems are children of the System element.
Requirement *(Optional)*	These are the requirements created in this design process. These can be the user requirements and design requirements.
Implementation *(Optional)*	Indicates if requirements are implemented by a sub-system or other requirements.
Mapping *(Optional)*	Indicates the relationship between a requirement or the system and its sub-systems or other requirements.

Table 11.4 - Elements of the Subsystem element

11.2.3. Requirement element

The Requirement element creates or defines new requirements for the system. This element is part of the System and Subsystem element. The requirement element may look like the code listed in Figure 11.5. and can exist zero, one or multiple times in a System or Subsystem element.

Figure 11.5 shows that there are different attributes and sub-elements for this element describing specific properties of this element. These attributes are described in Table 11.5 and the sub-elements are described in Table 11.6.

```
<Requirement Name="RE_Error_Screen" Owner="User">
  <Title>Error screen</Title>
  <Description>The error screen shall look like figure 6.</Description>
  <Requirement Name="RE_Show_Detailed_Error" Owner="User">
    <Title>Show detailed error message</Title>
    <Description>Upon entering incorrect data, user is redirected to an error page which shows a detailed
    error description.</Description>
  </Requirement>
</Requirement>
```

Figure 11.5 - A basic Requirement element

Every Requirement element has a Name attribute, which gives the requirement an unique name in the repository. This name will be used to as a reference key when defining relationships from and to this requirement. The Owner attribute is optional and can be used to state who the owner or creator is of this requirement. For example, the Owner attribute may be set to “user” or “client” to indicate the requirements specified by the customer (also known as user requirements).

Although this attribute is optional, it can be very helpful for tracing user-requirements, because the requirements can be queried for the specific users.

The Requirement element has three elements. These elements are the Title element, the Description element, and the Requirement element. The Title element is used to give the requirement a title or a name, which can be used in the documentation and in future tooling. The Description element describes in more detail what the requirement is. This should be a crisp description of what the system shall do.

Attributes	Explanation
Name	The Name for the requirement.
Owner *(Optional)*	Indicates who the owner or creator of this requirement is.

Table 11.5 - Attributes of the Requirements element

The Requirement element can contain zero, one, or multiple Requirement elements in its body. This is the code listed on lines 4 till 7 in Figure 11.5. This means that the requirement contains sub-requirement which states that every sub-requirement is refinement of the parent requirement. Thus requirement “Show detailed error message” is a sub-requirement of requirement “Error screen”.

Elements	Explanation
Title	Indicates what the title is of this requirement.
Description	A description of what this requirement means and what the system shall do.
Requirement *(optional)*	These can be used to express sub-requirements.

Table 11.6 - Elements of the Requirements element

11.2.4. Declaration element

The Declaration element declares new sub-systems or files for the system. In other words, the Declaration element creates a new sub-system or specifies the existence of a file. Figure 11.6 states that a new declaration of a sub-system with the name “SS_Input_Page” is made. So, the designer created a new sub-system within his System element.

```
<Declaration Name="SS_Input_Page">
  <Title>Input page</Title>
  <Motivation>Page for the user input</Motivation>
  <Implements Name="IM_Input_Page">
    <Title>Input page implementation</Title>
    <Motivation>The implementation of the input screen requirement</Motivation>
    <Source Name="RE_Input_Screen" />
  </Implements>
</Declaration>
```

Figure 11.6 - A basic Declaration element

The attributes and elements for the declaration element are show in Table 11.7 and Table 11.8. Each Declaration element has a Name attribute to make this element unique. The Owner attribute can be used to specify the creator of this declaration. Files can be declared as part of the repository. This means that the place and type of the file declared should be present. The attribute Type and FilePath are only mandatory when a file is declared. The Type is by default the value “subsystem” and does not need to be specified when a sub-system is declared. The Filepath attribute is used to specify the path of a file which can be a code file, a user guide or an image file.

The Declaration element exists of three elements which are the Title element, the Motivation element and the Implements element. The Title element is used to give the declaration a title or a name. The Motivation element motivates the declaration decision and is optional. The Implements element can be used when the declaration also implements a requirement. For example in Figure 11.6 the requirement “RE_Input_Screen” states that the system need to have a input screen or page. The declaration of sub-system “SS_Input_Page” fulfils this requirement and thus implements this requirement. This implementation is stated with the Implements element. Take note that there is also an Implementation element, which is different in use. These two implementation elements will be discussed in the following two sub-sections.

Attributes	**Explanation**
Name	The name of the file or design declared.
Owner *(Optional)*	Indicates who the owner or creator of this declaration is.
Type *(Optional)*	Indicates what kind of declaration is given. Value for this attributes are: ▪ Subsystem; ▪ File. By default the type is Subsystem.
Filepath *(Optional)*	The path to the file that is declared.

Table 11.7 - Attributes of the Declaration element

Elements	**Explanation**
Title	Indicates for documentation purpose what the title is of this declaration.
Motivation *(Optional)*	A motivation of what this declaration means and does.
Implements *(Optional)*	Indicates if the declaration implements a requirement.

Table 11.8 - Elements of the Declaration element

Stating the implementation of a requirement in the Declaration element, keeps the overview of the repository simple. This means, that the implementation and the declaration of a sub-system or file are kept together. This solution also reduces the need of a separate Implementation statement in the sub-system body, which may decrease the overview of the repository.

11.2.5. Implements element

The Implements element declares the relation between requirements that are implemented by a file or a sub-system. This element may look like the code listed in Figure 11.7. The Implements element only exists in the Declaration element.

When declaring a design or file in the Declaration element, the designer can assign an implementation to this new declaration immediately, instead of having to specify the implementation as the Implementation element in the body of the System or Subsystem element.

```
<Declaration Name="SS_Input_Page">
   <Title>Input page</Title>
   <Motivation>Page for the user input</Motivation>
   <Implements Name="IM_Input_Page">
      <Title>Input page implementation</Title>
      <Motivation>The implementation of the input screen requirement</Motivation>
      <Source Name="RE_Input_Screen" />
   </Implements>
</Declaration>
```

Figure 11.7 - A basic Implements element

In Figure 11.7 a sub-system is declared with the name “SS_Input_Page”. The designer decides that this sub-system is responsible for the implementation of the requirement with the name “RE_Input_Screen”. This is done by stating this implementation within the Declaration element. The implementation of the requirement is named “IM_Input_Page”.

Attributes	Explanation
Name	The name of the file or component declared.
Owner *(Optional)*	Indicates who the owner or creator of this implementation is.

Table 11.9 - Attributes of the Implements element

Elements	Explanation
Title	Indicates for documentation purpose what the title is of this implementation.
Motivation *(Optional)*	A description of what this implementation means and does.
Source	Identifies the source with is implemented

Table 11.10 - Elements of the Implements element

The only attributes for this element are show in Table 11.9. The Name attribute specifies the name of the implementation and should be unique. The Owner attribute specifies the creator of this implementation and is optional. There are only three elements of the Implements element and these are shown in Table 11.10. The Title element is used to give the implementation a title or a name. The Motivation element motivates the implementation decision. The target element identifies the target which can be a requirement or a file.

11.2.6. Implementation element

The Implementation element declares the relationship between requirements that are implemented by new requirements, by files, or by sub-systems. This element may look like the code listed in Figure 11.8 and exists in both the System and Subsystem element.

In Figure 11.8 a sub-system is shown that is responsible for the implementation of two requirements. Requirements “RE_Software_Part” and “RE_Software_Infrastrcuture_Dependency” are implemented by sub-system “SS_Software” and are stated in the Implementation element with the name “IM_Software_Part”.

```
<Implementation Name=“IM_Software_Part” >
  <Title>Implementation of Software implementation”</Title>
  <Source Name=“RE_Software_Part” />
  <Source Name=“RE_Software_Infrastrcuture_Dependency” />
  <Target Name=“SS_Software” />
</Implementation>
```

Figure 11.8 - A basic Implementation element

Attributes	Explanation
Name	The name of the file or component declared.
Owner *(Optional)*	Indicates who the owner or creator of this implementation is.

Table 11.11 - Attributes of the Implementation element

The Implementation element has two attributes which are show in Table 11.11. The Name attribute specifies the name of the implementation and should be unique. The Owner attribute indicates who created the implementation.

The elements of the Implementation element are shown in Table 11.12. This element exists of four elements which are the Title, the Motivation, the Source, and the Target element. The Title element is used to give the implementation a title or a name, which is used in the documentation and in the future tools. The Motivation element motivates the implementation decision. The source element specifies a requirement. The Target elements can be a requirement, sub-system, or a file.

Elements	Explanation
Title	Indicates for documentation purpose what the title is of this implementation.
Motivation *(Optional)*	A description of what this implementation means and does.
Target	Identifies the target with is implemented
Source	Identifies the source with is implemented

Table 11.12 - Elements of the Implementation element

11.2.7. Mapping element

The Mapping element declares the relationship between requirements or sub-system. In other words; the Mapping element passes the requirement to another requirement or sub-system, but does not let the requirement or sub-system implement the requirement. Otherwise the Implementation element should be used. The Mapping element may look like the code listed in Figure 11.9.

Figure 11.9 states that the sub-systems “SS_Success_Page”, “SS_Error_Page” and “SS_Input_Page” are mapped on requirement “RE_Configuration_File_Dependence”. This means

that the requirement is passed on to these sub-systems and that the sub-system need to 'deal' with this requirement. This is done within the Mapping element with name "MA_Configuration_File".

```
<Mapping Name="MA_Configuration_File">
    <Title>Configuration File</Title>
    <Motivation>Mapping the configuration file requirement to the screens</Motivation>
    <Source Name="RE_Configuration_File_Dependence" />
    <Target Name="SS_Success_Page" />
    <Target Name="SS_Error_Page" />
    <Target Name="SS_Input_Page" />
</Mapping>
```

Figure 11.9 - A basic Mapping element

Attributes	Explanation
Name	The name of the file or component declared.
Owner *(Optional)*	Indicates who the owner or creator of this mapping is.

Table 11.13 - Attributes of the Mapping element

The Mapping element exists of two attributes and is show in Table 11.13. The Name attribute specifies the name of the mapping and should be unique. The Owner attribute indicates the creator of the mapping.

The elements of the Mapping element are shown in Table 11.14. This element exists of four sub-elements, which are the Title, the Motivation, the Source, and the Target element. The Title element is used to give the mapping a title or a name, which is used in the documentation and in the future tooling. The Motivation element motivates the mapping decision. The source element specifies the requirement that is mapped. The Target element identifies the target, which can be a requirement or a sub-system.

Elements	Explanation
Title	Indicates for documentation purpose what the title is of this design.
Motivation *(Optional)*	A description of what this mapping means and does.
Target	Identifies the target with is implemented
Source	Identifies the source with is implemented

Table 11.14 - Elements of the Mapping element

11.2.8. Source element

The Source element indicates the source name of a requirement and is used by the Implementation, Implements, and the Mapping element. The Source element consists of only one line of code and may look like the code highlighted in Figure 11.10. In this example the Source element is used in the body of the Mapping element.

Figure 11.10 also shows that there is only one attribute ad no elements for this element. This attribute is described in Table 11.15. The Name attribute specifies the name of the requirement that is used as a source.

```
<Mapping Name="MA_Configuration_File">
    <Title>Configuration File</Title>
    <Motivation>Mapping the configuration file requirement to the screens</Motivation>
    <Source Name="RE_Configuration_File_Dependence" />
    <Target Name="SS_Success_Page" />
    <Target Name="SS_Error_Page" />
    <Target Name="SS_Input_Page" />
</Mapping>
```

Figure 11.10 - A basic Source element

Attributes	Explanation
Name	The name for the requirement.

Table 11.15 - Attributes of the source element

11.2.9. Target element

The Target element indicates the target name of some requirement, sub-system, or file and is used by the Implementation element, Implements element, and the Mapping element. The Target element consists of only one line of code and may look like the code listed in Figure 11.11. In this example the Target element is used in the body of the Mapping element.

```
<Mapping Name="MA_Configuration_File">
    <Title>Configuration File</Title>
    <Motivation>Mapping the configuration file requirement to the screens</Motivation>
    <Source Name="RE_Configuration_File_Dependence" />
    <Target Name="SS_Success_Page" />
    <Target Name="SS_Input_Page" />
</Mapping>
```

Figure 11.11 - A basic Target element

Attribute	Explanation
Name	The name for the requirement.
Type (*optional*)	Described the type of the type of the target ▪ Requirement; ▪ Subsystem; ▪ File. Default the Subsystem value is assigned.

Table 11.16 - Attributes of the Target element

11.2.10. Motivation element

The Motivation element motivates the decision to create the current element, where this Motivation element is part of. The Motivation element consists of only one line of code and may look like the code listed in Figure 11.12. In this example the Motivation element is used in the body of the Mapping element. It describes why or what this mapping means or does. The Motivation element has no attributes or elements as children.

```
<Mapping Name="MA_Configuration_File">
    <Title>Configuration File</Title>
    <Motivation>Mapping the configuration file requirement to the screens</Motivation>
    <Source Name="RE_Configuration_File_Dependence" />
</Mapping>
```

Figure 11.12 - A basic Motivation element

11.2.11. Description element

The Description element gives a description of what the current Requirement element, where this Motivation element is part of, means. The Description element consists of only one line of code and may look like the code listed in Figure 11.13. The Motivation element has no attributes or elements as children and is a child of the Requirement element.

```
<Requirement Name="RE_Error_Screen" Owner="User">
  <Title>Error screen</Title>
  <Description>The error screen shall look like figure 6.</Description>
</Requirement>
```

Figure 11.13 - A basic Description element

11.2.12. Title element

The Title element gives the current element, where this Title element is part of, a title. The Title element consists of only one line of code and may look like the code listed in Figure 11.14. In this example the Title element is used in the body of the Requirement element. It describes what the name of this requirement is. The title element has no attributes or elements as children.

```
<Requirement Name="RE_Error_Screen" Owner="User">
  <Title>Error screen</Title>
  <Description>The error screen shall look like figure 6.</Description>
</Requirement>
```

Figure 11.14 - A basic Title element

11.3. Implementation and mapping usage

This section explains the usage of the Mapping, Implements, and Implementation elements. The usage of these elements is very straight foreword, but needs some clarification. Especially how

the elements should be used when more then one target or source is used, and the behavior of requirements with sub-requirement needs more clarification.
When a designer wants to map one requirement onto multiple sub-systems, one Mapping element is sufficient to express this relationship. The different sub-systems are then grouped as illustrated in Figure 11.15 under the Mapping element. For each sub-system a Target element is created within the Mapping element. In the example the requirement “RE_Configuration_File_Dependence” is mapped on the designs “SS_Success_Page”, “SS_Error_Page” and “SS_Input_Page” .

```
<Mapping Name="MA_Configuration_File">
    <Title>Configuration File</Title>
    <Motivation>Mapping the configuration file requirement to the screens</Motivation>
    <Source Name="RE_Configuration_File_Dependence" />
    <Target Name="SS_Success_Page" />
    <Target Name="SS_Error_Page" />
    <Target Name="SS_Input_Page" />
</Mapping>
```

Figure 11.15 - A basic Target element

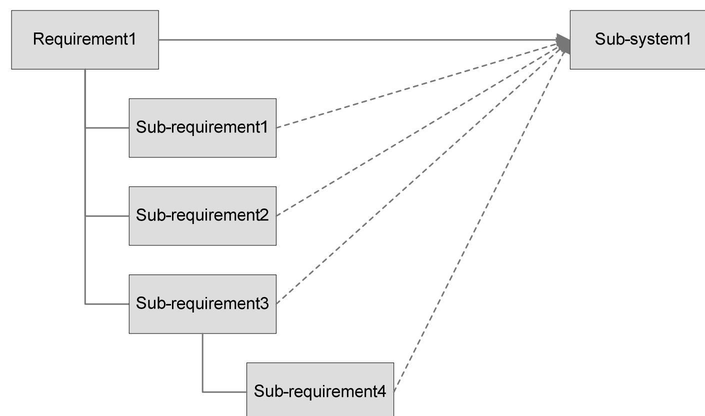

Figure 11.16 - Parent Requirement mapping with sub-requirements

```
<Mapping Name="Parent_Mapping">
    <Source Name="Requirement1" />
    <Target Name="Sub-system1" />
</Mapping>
```

Figure 11.17 - Parent requirement mapping

When mapping or implementing a requirement with sub requirements, all the sub requirements are also a part of this mapping or implementation. Thus, all sub requirements are passed on to the item mapped onto. This is illustrated in Figure 11.16. The straight arrow is the actual mapping which is expressed by the designer and is saved in the repository. The dotted lines represent the indirect mapping of all the sub-requirements. The structure of the requirement tree is kept intact and can be reused in sub-system1. The code for the mapping looks like Figure 11.17.

When the designer only wants a small selection of these sub requirements to be mapped, the selection needs to be entered separately. For example, the designer wants to map “Sub-requirements1” and “Sub-requirements2” on “Sub-system1”. These two requirements need then to be entered separately under the Mapping element as shown in Figure 11.18.

```
<Mapping Name="Partial_Mapping">
    <Source Name="Sub-requirement1" />
    <Source Name="Sub-requirement2" />
    <Target Name="Sub-system1" />
</Mapping>
```

Figure 11.18 - Multiple sub-requirement mapping

11.4. Unique identifiers

In the repository, every requirement, system, sub-system or file needs to have an unique identifier or name so it can be addressed in the repository, in tools, or in parsers. It needs to be unique to be able to refer to a specific item without causing pointers to more then one item.

Paskin [47] wrote an extensive paper on how to create unique identifiers. He offers different approaches and standards to create unique identifiers. In his paper he refers to two international standards which formulated requirements for generating unique identifiers. Both Thacker of the ISO/TC 46/SC 9 (ISO Technical Committee 46: Information and Documentation Standards) Secretariat [60] and the Internet Engineering Task Force (IETF) informational RFC 1737 [57] laid out some requirements for generating unique identifiers. These requirements are:

- *Uniqueness*: An identifier must be assigned to one object only and must never be reused;
- *International*: The identification system must be international in scope. It should have the same meaning everywhere;
- *Persistent*: Identifiers must have an unlimited lifespan even though the objects they identify may not.
- *Designed for ease of use in automated systems*: The identifier should have a prescribed syntax. This means that the identifier should preferably be numeric or capable of conversion to numeric form. It should preferably incorporate a check digit and the component elements should be computer parseable;
- *Capacity*: Identification system must have the numbering capacity to cover the volume of objects defined within the scope of the system. This means that the construction of the identifier should include an element to refresh the capacity of the system at periodic intervals (for instance, a year of assignment identifier or some sort of administrative sequence identifier);
- *Extensibility*: Any scheme for identifier must permit future extensions to the scheme.

The requirements mentioned above can be very helpful in creating unique identifiers, but Chess prefers words as identifiers apposed to numbers. The reason for this is that numbers do not

indicate the item it belongs to. This makes understanding the repository very hard when reviewing it by hand (this is a requirement!). Relationships will refer to a numbered item, which does not indicate what the item is. Using names can give a hint or information about the item, before even looking at the item. The problem is that this requirement makes the possibilities for creating unique identifiers very hard.

Bireck, Diamond, et al [6] describe in their book how XML names can be made unique. Although this is about the element names used in XML, it also can be used for making unique names as identifiers. XML provides a method for creating globally unique names (and this is literally, as in the World Wide Web), as long as the namespace is made universally known, and people stick to using it (although this can never be totally guaranteed).

The technique for making unique names is just simple applying a contextual prefix to every name. Figure 11.19 illustrates an XML structure which is used as an example. Here the item with the name "SubDesign1" can be directed as "MainDesign1.SubDesing1". This makes it possible to use identical names at different levels. But, according to Bireck, Diamond, et al [6] this only moves the problem of differentiating names in shared vocabularies to a different (new) level. It is still possible to create another item with the name "SubDesign1" in item "MainDesign1". This will result in two items with the same name.

```
<Item Name="MainDesign1">
   …
   <Item Name="SubDesign1">
      …
      <Item Name="Design1"> … </Item>
      …
   </Item>
   …
</Item>
```

Figure 11.19 - Example of the use of word identifiers

There is still a need for a "name prefix authority" to ensure non-overlapping name prefixes. This can generate very long names when directing an item from four levels above the item directed to. Because of the requirement Chess opposed to the repository, it is very hard to realize a method for creating unique identifiers in the repository.

To overcome this block or problem, a simple solution is provided for the first version of the repository. The names of the items stored in the repository need to be unique, but no special standards or techniques are provided. This means that it is the responsibility of the designer to create and use unique names. The tooling that is developed for the repository should be able to deal with this problem. For no, the designer needs to give each element a clear and unique name.

Naturally, this is no solution to the problem, but for now it will be sufficient to be able to use the first version of the repository. In later version of the repository a suitable solution will be created for this problem.

11.5. File based or database repository

In this section the decision will be made if the repository should be a file based- or a database repository. There are a number of factors that will influence the choice of repository type to be used in the repository [18]. These factors can be based on performance, reliability, cost, customer preferences, deployment environment and the use cases.

For the repository the choice is not important, because the repositories usage does not depend upon the choice whether the repository is file based or database. Both approaches are possible and usable for the repository. For the tooling however it is important to specify the type.

Although both types of repositories are possible, a choice needs to be made to be able to implement the repository. That is why the first version the repository will be file based. This choice also fulfils requirement [N3] that the repository needs to be stand alone although this is also possible with a database. The main reasons for making the repository file based are:

- Files based repositories do not need licenses which are needed for database repositories;
- Files based repositories are easier in use then database repositories;
- Files based repositories do not need installation procedures to used it. It is accessible in the existing file system. Databases however need to be installed and configured.

In later versions of the repository it can be advisable to use a database instead of the file based repository. Databases are advanced storage mechanisms which provide different functionalities. These functionalities can be very helpful in future development of the repository. But for now, a file based repository is sufficient for the task at hand.

11.6. Document generation

Document generation is an important feature that is supported by the repository. It is one of the requirements (requirement [N7]) Chess poses to the repository. In sub-section 11.6.1 the solution to the documentation problems are discussed. In sub-section 11.6.2 The needed information for documents generation is provided. Finally ion sub-section 11.6.3 the documents users and types are discussed.

11.6.1. Solution to the document problem

Documentation generation is one of the reasons for developing this repository. It should prevent the problems mentioned in section 1.2 and sub-section 3.3.3. In general these problems are:

- Inconsistent documentation;
- Documentation is not up-to-date;
- For each stakeholder different documents need to be written.

In some development projects documentation is the primary information source or repository. Each user needs its own document to perform his task, which is a costly, time consuming, and difficult task. This problem is partially solved by using four design views as suggested in sub-

section 3.3.3, which mean that groups of users share a document. But still, there are different documents that need updating.

With the suggested repository the information source is the repository. Documentation is generated from the data in the repository and only presents this data on paper. This means that when something changes in the design or requirements, new documents can be generated from the data stored in the repository rather then updating all existing documents. This prevents inconsistencies in the different documents and is more efficient. This also means that each user can have its own document with the data needed for his task.

11.6.2. Information

As stated earlier in this thesis, the first version of the repository shall be able to support the generation of simple documentation. This means that data needed to produce these documents, shall be present. The information that is present in the repository for the generation of the documents matches the attributes identified by the IEEE Standard 1016 [4] mentioned in sub-section 3.3.3. Keep in mind that the designer may decide to define this information and that this information is not present for the generation of the documentation. Designers are also able to add new elements or attributes to the structure of the repository to provided extra information. The attributes provided by the repository are:

- *Identification*: each requirement, file, or sub-system has a unique name for identification purpose;
- *Type*: each element in the repository has its own element, from which the type can be identified. The declarations element also have a type attribute which can be used to identify the type;
- *Purpose*: each element has a motivation or description element where the purpose of the element can be stated;
- *Function*: the Description or motivation element can be used;
- *Subordinates*: the Description or motivation element can be used;
- *Dependencies*: the relations between the requirements, (sub-)systems, and files;
- *Interface*: the Description or Motivation element can be used;
- *Resources*: the Description or Motivation element can be used;
- *Processing*: the Description or motivation element can be used;
- *Data*: again the Description or Motivation element can be used.

11.6.3. Document users and types

The documentation for each user identified in sub-section 3.3.3 can be generated from the information mentioned in the previous sub-section. The users and the different attributes are shown in Table 11.17, which was also shown in sub-section 3.3.3. Using the information provided by this matrix, documents for each user can be generated in real time from the data stored in the repository.

It is important to understand that documentation generation is something that needs to be done by a tool. However, it is possible to describe what types of documents can be generated from the repository. These documents are:

- Reports about the different analysis checks;
- Reports about the user and design requirements;
- Reports about the structure of the system;
- Traceability reports.

Sub-system or component attributes	**User roles**						
	Project manager	**Configuration manager**	**Designer**	**Programmer**	**Unit tester**	**Integration tester**	**Maintenance programmer**
Identification	X	X	X	X	X	X	X
Type	X	X				X	X
Purpose	X	X					X
Function	X		X			X	
Subordinates	X						
Dependencies		X				X	X
Interface			X	X	X	X	
Resources	X	X				X	X
Processing				X	X		
Data				X	X		

Table 11.17 - User roles and attributes [4]

11.7. Summary

This chapter provided information about the implementation of the repository. It mainly described the type of language used to store the artifact data, the structure of the repository, and the document generation features.

As the foundation for the repository XML is used as the language to store the artifact data. Important features of XML, which contributed to this choice, are:

- XML contains database functions;
- XML is extendible;
- XML has human editable properties.

The repository consists of twelve elements that are related to each other. These repository elements are:

- System;
- Subsystem;
- Requirement;
- Declaration;
- Implements;
- Implementation;
- Motivation;
- Description;
- Title;
- Mapping;
- Source;
- Target.

Each element has its own function and may be used in combination with other elements. Each element may also have some of the following attributes:

- Name;
- Owner;
- Type.

The repository uses a parent child structure to create the hierarchical tree that mentioned in earlier chapters. Other relationships are made explicit by using the appropriated elements.

After defining the structure of the repository some properties of the repository where discussed. These properties are:

- The first version of the repository shall use unique names, but creating these unique names is the responsibility of the designer. So, it is possible that the names in the repository are not unique. Tooling however should be able to check the uniqueness of the names in the repository;
- The first version of the repository shall be file based. The reason for this is mainly that file based is cheap and easy to set-up. In later versions of the repository however, a database can be more suitable for the repository.

Finally, document generation from the data stored in the repository was discussed. The repository provides the opportunity to generate up-to-date and consistent documentation for different stakeholders. The data stored in the repository should provide sufficient information for each stakeholder and his task.

12. TESTING THE REPOSITORY

In this chapter the test-cases and checklist for the repository are provided. As described in chapters 3. and 8. testing is an important part of the system engineering process. The test process was described in section 8.1. to give the reader an impression of how this activity is executed. This chapter does not describe this process, but only provides the test-cases and checklist to perform the tests.

Section 12.1 states that the repository itself is hard to test. Instead the tooling that is written for it should be tested. In second 12.2 test-cases for the functional requirements are provided. Section 12.3 provides a checklist for the non-functional requirements. In section 12.4 a proof of concept of the use of the repository is provided. In the final section a summary of this chapter is given.

12.1. Tooling

In section 9.4 the requirements for the repository where defined. This section also stated that some of these requirements are more tool oriented then repository oriented. This means that the requirement state that the repository should be able to support specific tool tasks or operations, which the repository by itself can not perform. These tool oriented requirements were stated as non-functional requirement in sub-section 9.4.3. Testing these requirement mean that there should be tooling available, which is not available yet. This makes testing these requirement very bit difficult.

For the functional requirement stated in sub-section 9.4.2 test-cases are created, which is described in the next section. Still, testing is even hard to do for these requirements without tooling, because the repository does and can not validate the data and the structure by itself. An advanced XML editing tool could use the DTD defined in Appendix B could be used as a validation method for the structure of the repository.

12.2. Functional requirements

In section 8.3 a description was provided about the characteristics of test-cases used in requirements-based testing. This section provides the test-cases for the functional requirements stated in sub-section 9.4.2 for the repository. The test-case for the repository are stated in Figure 12.1 till Figure 12.3. Some comments should be made about the test-cases it self and the expected results of these test-cases. These comments are:

- Some of these functional requirements are joined together in one test-case. This is illustrated in Figure 12.1, where requirements [F1] and [F3] are tested in the same test-case. This is possible, because the test actions needed to test the different requirements are similar for both requirements;
- Figure 12.1 also illustrates a very important response for tooling when errors or problems are detected. The expected result of test seven of this test-case shows that the action that

is tested is allowed, but a warning should be raised that this action could result in a problem or error. For test seven this is the case. A requirement is not implemented, which means that the system does not fulfill that particular requirement. This is allowed according to functional requirement [F1] , but a warning should be shown to help or remind the user of this problem;

- Another feature supported by the repository is adding own elements and attributes to the repository structure to provide more information about the design process or design itself. This feature is opposed by requirement [F4]. However, tooling may not be able to use these new elements or attributes. Here the difference between the repository testing and tool testing is very clear. To let tooling be able to use these new elements and attributes, the tooling need to be reprogrammed. Only then are the tools able to use these elements and attributes.

<table>
<tr><td>Date:</td><td></td><td>Tested by:</td><td></td></tr>
<tr><td>System</td><td>Repository</td><td>Environment:</td><td></td></tr>
<tr><td>Objective:</td><td></td><td>Test ID:</td><td>1</td></tr>
<tr><td>Function:</td><td></td><td>Req ID:</td><td>[F1][F3]</td></tr>
<tr><td>Version:</td><td></td><td>Test type:</td><td></td></tr>
<tr><td colspan="4">Condition to test:
[F1] The repository shall allow storing partial (sub-)systems and requirements.
[F3] The repository shall store the information as sets of hierarchical trees.</td></tr>
<tr><td colspan="4">Data/Steps to perform:
1. Add a parentless sub-system “SS_Parentless”;
2. Delete sub-system “SS_Software”;
3. Change sub-system “SS_Infrastructure”;
4. Change system “SY_Fiscall_Tax”;
5. Add anew requirement “RE_New” to sub-system “SS_Infrastructure”;
6. Delete a requirement “RE_Error_Screen”, but keeping the sub-requirement;
7. Do not implement requirement “RE_Confirmation_Screen”;
8. Do not implement all the requirements.</td></tr>
<tr><td colspan="4">Expected results:
1. This is not allowed and an error shall be raised;
2. This is allowed and the action shall be permitted;
3. This is allowed and the action shall be permitted;
4. This is allowed and the action shall be permitted;
5. This is allowed and the action shall be permitted;
6. This is allowed and the action shall be permitted;
7. This is allowed and the action shall be permitted, although a warning shall be raised;
8. This is allowed and the action shall be permitted, although a warning shall be raised.</td></tr>
<tr><td colspan="4">Actual result: Passed [ ] Failed [ ]</td></tr>
</table>

Figure 12.1 - Test-case for requirements [F1] and [F3]

Date:		*Tested by:*	
System	Repository	*Environment:*	
Objective:		*Test ID:*	2
Function:		*Req ID:*	[F4]
Version:		*Test type:*	
Condition to test: [F4] The system shall support the ability to add metadata information.			
Data/Steps to perform: 1. Add the author to an element; 2. Add a description to an element; 3. Add a date to an element; 4. Add a version number to an element.			
Expected results: 1. This is allowed and the action shall be permitted; 2. This is allowed and the action shall be permitted; 3. This is allowed and the action shall be permitted; 4. This is allowed and the action shall be permitted;			
Actual result: Passed [ ] Failed [ ]			

Figure 12.2 - Test-case for requirements [F4]

Date:		*Tested by:*	
System	Repository	*Environment:*	
Objective:		*Test ID:*	3
Function:		*Req ID:*	[F5]
Version:		*Test type:*	
Condition to test: [F5] The repository shall support relationships.			
Data/Steps to perform: 1. Add an relationship between a requirement and another requirement; 2. Add a relationship between a requirement and a (sub-)system; 3. Add a relationship between a (sub-)system and another sub-system; 4. Add a relationship between a requirement and a file;			
Expected results: 1. This is allowed and the action shall be permitted; 2. This is allowed and the action shall be permitted; 3. This is allowed and the action shall be permitted; 4. This is allowed and the action shall be permitted;			
Actual result: Passed [ ] Failed [ ]			

Figure 12.3 - Test-case for requirements [F5]

<table>
<tr><td>Date:</td><td></td><td>Tested by</td><td></td></tr>
<tr><td>System</td><td>Repository</td><td>Environment:</td><td></td></tr>
<tr><td>Objective:</td><td></td><td>Test ID:</td><td>4</td></tr>
<tr><td>Function:</td><td></td><td>Req ID:</td><td>[F2]</td></tr>
<tr><td>Version:</td><td></td><td>Test type:</td><td></td></tr>
<tr><td colspan="4">Condition to test:
[F2] The repository shall not enforce optional input.</td></tr>
<tr><td colspan="4">Data/Steps to perform:
1. Adding or changing sub-system “SS_Infrastructure”:
2. The Name attribute is not used;
3. The Owner attribute is not used;
4. The Title element is not used;
5. The Description element is not used;
6. The Subsystem element is not used;
7. The Mapping element is not used;
8. The Requirement element is not used;
9. The Implementation element is not used;
10. The Declaration element is not used.
11. Adding or changing requirement “RE_Confirmation_Screen”:
12. The Name attribute is not used;
13. The Owner attribute is not used;
14. The Title element is not used;
15. The Description element is not used;
16. The Requirement element is not used.</td></tr>
<tr><td colspan="4">Expected results:
1. This is allowed and the action shall be permitted;
2. This is not allowed and an error shall be raised;
3. This is allowed and the action shall be permitted;
4. This is not allowed and an error shall be raised;
5. This is allowed and the action shall be permitted;
6. This is allowed and the action shall be permitted;
7. This is allowed and the action shall be permitted;
8. This is allowed and the action shall be permitted;
9. This is allowed and the action shall be permitted;
10. This is allowed and the action shall be permitted;
11. This is not allowed and an error shall be raised;
12. This is not allowed and an error shall be raised;
13. This is not allowed and an error shall be raised;
14. This is not allowed and an error shall be raised;
15. This is not allowed and an error shall be raised;
16. This is allowed and the action shall be permitted;</td></tr>
<tr><td colspan="4">Actual result: Passed [ ] Failed [ ]</td></tr>
</table>

Figure 12.4 - Test-case for requirement [F2]

Date:		*Tested by:*	
System	Repository	*Environment:*	
Objective:		*Test ID:*	25
Function:		*Req ID:*	[F6]
Version:		*Test type:*	
Condition to test: [F6] The repository shall support file declarations of any type of file.			
Data/Steps to perform: 5. Declare an image file in the repository; 6. Declare an MS Word file in the repository; 7. Declare an UML file in the repository;			
Expected results: 5. This is allowed and the action shall be permitted; 6. This is allowed and the action shall be permitted; 7. This is allowed and the action shall be permitted;			
Actual result: Passed [ ] Failed [ ]			

Figure 12.5 - Test-case for requirements [F6]

12.3. Non-functional requirements

As described in section 9.4.3 the non-functional requirements of the system can be tested by means of a checklist. The checklist for the repository is provided in this section. The positive influences for the repository are indicated with the "+" sign and negative influences are indicated with the "–" sign.

Take note that this is only a short list. The list can be more extensive, but that is beyond the scope of this thesis. The most important items are provided in this checklist.

The non-functional requirements checklist questions are:

Attribute and question		**Influence**
[N1]	*The repository shall be free or have a low price*: no or negligible costs shall be linked to the use of the repository.	
	Are there any costs for installing the repository?	**-**
	Are there any cost for using the repository?	**-**
[N2]	*The repository shall be easy to setup*: the repository shall be easy to install and configure for use.	
	Are there procedures needed to install the repository?	**-**
	Are there other programs or servers needed for the operation of the repository?	**-**
[N3]	*The repository shall be stand alone*: this means that using the repository shall not be restricted to a database or server.	
	Are there other programs or servers needed for the operation of the repository?	**-**
[N4]	*The repository shall be human editable*: users shall be able to edit the data stored by hand in the repository in a text editor.	
	Can the repository be opened in a text editor?	**+**
	Can a user understand de structure of the repository	**+**

Attribute and question		Influence
[N5]	*The repository shall support inconsistency checking*: the repository shall support and analyze the repository data for inconsistencies between requirements and (sub-)system.	
	Is tooling able to detect inconsistencies in the repository?	+
[N6]	*The repository shall support impact analysis*: determining the impact of a change request shall be supported by the repository.	
	Is tooling able to perform a impact analysis in the repository?	+
[N7]	*The repository shall support the generation of simple documentation*: different types of documentation shall be able to be generated from the data stored in the repository.	
	Is tooling able to produce documentation from the data stored in the repository?	+

Table 12.1 - Non-functional requirements checklist

The checklist questions for the repository quality attributes are:

Attribute and question		Influence
[Q1]	*Operability*: Expresses the effort needed to use and control the repository.	
	Is there an automatic back-up option for the repository?	+
	Uses the repository an international or company standard?	+
	Is documentation provided with data models and is this consistent and up-to-date?	+
[Q2]	*Maturity*: Expresses the ability to reduce disruptions created by errors in the repository.	
	Are there periodically executed consistency checks for the data stored in the repository?	+
	Uses the repository some sort of encryption?	-
	Is user input automatically refined with additional data?	+
[Q3]	*Fault tolerance*: Expresses the ability to maintain a specific level of operation in the repository incase of an error or a disruption.	
	Are there roll-back facilities in the repository?	+
	Is the data storage of the repository kept in two different places?	+
[Q4]	*Recoverability*: Expresses the ability to repair the operation level and data of the repository in case of a malfunction, and the time and effort that is necessary to repair this malfunction.	
	Are there processes that check the repository for inconsistencies ("watchdogs")?	+
	Are the recovery procedures periodically tested?	+
[Q5]	*Understandability*: Expresses the effort needed to understand logical concepts of the repository and the usability of these concepts.	
	Is the structure of the repository easy to understand?	+
	Is the user documentation or guide easy to understand?	+
[Q6]	*Learnability*: Expresses the effort needed to learn the use of the repository.	
	Is the user documentation or guide easy to understand?	+

Attribute and question		Influence
	Is the structure of the repository easy to understand?	+
[Q7]	*Operability*: Expresses the effort needed to use and control the repository.	
	Is the user documentation or guide easy to understand?	+
	Is the structure of the repository easy to understand?	+
[Q8]	*Attractiveness*: Expresses the effort needed to attract users for using the repository.	
	Does the repository encourage the user to use the repository?	+
	Is the structure of the repository easy to understand?	+
[Q9]	*Analyzability*: Expresses the effort needed to diagnose shortcomings, or error causes or the identification of modifiable parts of the repository.	
	Is the structure of the repository well substantiated?	+
	Is documentation provided with data models and is this consistent and up-to-date?	+
[Q10]	*Changeability*: Expresses the effort needed for changing, or removing errors or environmental changes in the repository.	
	Uses the repository an international or company standard?	+
	Is the structure of the repository well substantiated?	+
[Q11]	*Testability*: Expresses the effort needed to validate a changed repository.	
	Are there processes that check the repository for inconsistencies (“watchdogs”)?	+
	Are there roll-back facilities in the repository?	+
[Q12]	*Installability*: Expresses the effort needed for the repository to be installed in a specified environment.	
	Is the user documentation or guide easy to understand?	+
	Is the repository platform or operating system independent?	+
	Does the installation and setup of the repository take a long time?	-

Table 12.2 - Quality attributes checklist

12.4. Proof of concept

To proof that the repository can be used for real design problems, an existing Chess project was partially defined in the repository. This is shown in Appendix C. The project used for this proof of concept is an existing Chess project. The goal of this project was to build a website where customers of a car lease company can retrieve their fiscal tax value of their car.

The size of this development project was very small and thus idea to be used as a proof of concept. However, the project was but too large to enter entirely the entire design in the repository by hand. That is why the project is only partially entered in the repository.

Originally the website was designed in a MS word document with the websites its structure as document structure. The structure of such a document was described in sub-section 4.3.3. The data in the original design document and the data in the repository both represent the design process of this website.

12.5. Summary

This chapter provided the test-cases for the functional requirements and a checklist for the quality attributes that where specified in chapter 9. A proof of concept of the repository is shown in Appendix C., which was used for creating the test-cases.

Important to understand is that most of the requirements are tool oriented and that testing the repository without a tool is very hard. This is because the repository does and can not validate its own data. However advanced XML editing tools can use a DTD to validate the structure of the repository.

In this chapter five test-cases where defined for the six functional requirements. This means that one test-case is used to tests two requirements. This is allowed. Other important outcomes that should be understood about the repository are:

- Actions that may causes a bad design are allowed, but should in tooling be reported to the user;
- Adding user-defined elements and attributes is allowed in the repository, but may cause problems when tooling do not recognize these elements and attributes.

The quality attribute checklist provides for every quality attribute defined in chapter 9. two or three questions. The answer to these questions may have a positive or negative effect for the repository and determine the quality of the repository.

13. OPERATING THE REPOSITORY

In this chapter a description is given about how the repository should be used in general and per stakeholder. In section 13.1 the process of creation of a system and two sub-systems is step by step described. In second 13.3 a description of the different stakeholders and their use of the repository is provided. Finally, in the last section a summarization of this chapter is given.

13.1. Repository use

In this section the use of the repository is step by step described. During this description the example case of "Fiscal Tax" shown in Appendix C is used to clarify the different steps. The steps are illustrated with code snipes shown in the different figures. The lines of code that are applied to the step are highlighted in grey. The repository is in this example used by manual editing, because there is no tool available yet.

It should be clear that there is no specific order in which the repository can be used. The main constraint is that the system is defined as root and that the hierarchical structure of sub-systems is not broken (that means, there is exactly one System element defined as root and the Subsystem elements can not act as roots).

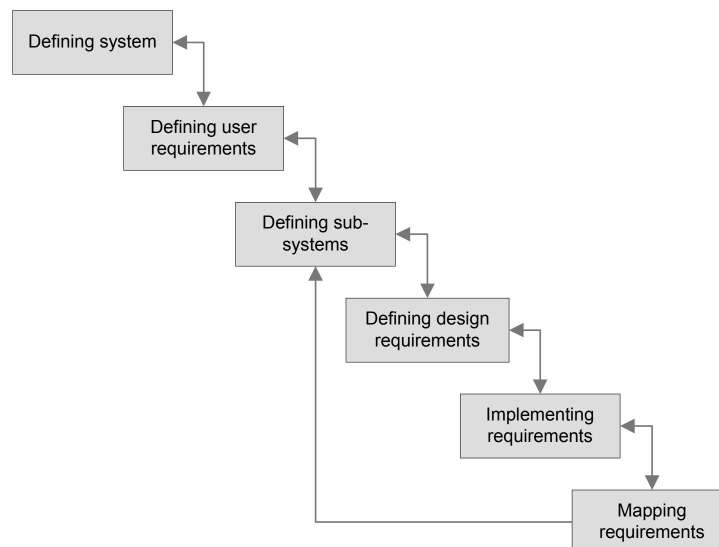

Figure 13.1 - Repository use steps

The different repository use steps are shown in Figure 13.1 and mean:

- *Defining system*: Creates the root of the repository and describes the whole system;
- *Defining user requirements*: Creates the user requirements;
- *Defining sub-systems*: Creates sub-systems for the current system or sub-system
- *Defining design requirements*: Creates the design requirements for the current system or sub-system;

- *Implementing requirements*: Creates the implementation relationships of a requirement in the current system or sub-system;
- *Mapping requirements*: Creates the relationships of a requirement and other requirements or sub-systems in the current system or sub-system.

The steps are not linear, but have feedback loops back to the previous steps. This is needed when:

- Changes need to be implemented;
- The system is partially designed and needs to be finished;
- Design mistakes are made and need to be fixed;
- A new design process needs to be executed.

After designing the system, the design process is redone for the each defined sub-system. This is illustrated as the arrow between the "Mapping requirements" step and "Defining sub-system" step.

In the "Fiscal Tax" example case each name of the different elements uses a prefix to state what type of element the name is. This can be very useful when a designer wants, for example, to implement an requirement with the name "Documentation", but there is also a sub-system with a name that looks almost similar (but it is not the same, because names should be unique), for example "Documents". It is then hard for the designer to know which of these two names the right requirement is. The prefixes should help the designer with this problem. Take note that other naming conventions or not using any prefix is also allowed. In the example case the following prefixes are used:

- *SY*: The item is a System element;
- *SS*: The item is a Subsystem element;
- *RE*: The item is a Requirement element;
- *IM*: The item is a Implementation element;
- *MA*: The item is a Mapping element;
- *FI*: The item is a file.

Explanations about the meaning of the different elements and attributes used in the following sub-sections can be found in chapter 11.

13.1.1. Defining the system "SY_Fiscal_Tax"

The use of the repository begins with the creation of the system as the root of the repository. This is done by using the System element as illustrated in Figure 13.2. The system is defined and is given an unique name. In the example this name is "SY_Fiscal_Tax". The designer does not set the Owner attribute for some particular reason. This is allowed, because the Owner attribute is optional.

```
<System Name="SY_Fiscal_Tax">
    <Title>Fiscal Tax</Title>
    <Motivation>The fiscal tax module is a free service to Bus Lease Netherlands
    drivers and fleet managers for requesting the catalogue value of their lease car for
    tax purposes.</Motivation>
</System>
```

Figure 13.2 - Defining the system

Next, the system is given the title "Fiscal Tax" by using the Title element. To motivate or describe the creation of the system, the Motivation element is used. In this case, a description of the "Fiscal Tax" system is given. The system title and motivation can be used in tooling and document generation.

```
<System Name="SY_Fiscal_Tax">
    …
    <Requirement Name="RE_Configurable_Screens" Owner="User">
        <Title>Configurable</Title>
        <Description>All text in the screens are configurable.</Description>
    </Requirement>
    <Requirement Name="RE_Configurable_Emails" Owner="User">
        <Title>Configurable emails</Title>
        <Description>All text in the email messages are configurable.</Description>
    </Requirement>
</System>
```

Figure 13.3 - Defining user requirements for the system

The next step after creating the system is to add the user requirements. This is done with the Requirement element and is illustrated in Figure 13.3. In this example two user requirements with names "RE_Configurable_Screens" and "RE_Configurable_Emails" are created and added to the repository. The requirements are given a title and a description. Both are mandatory for this type of element. The description formulates the requirement's meaning. The designer might have used the requirement characteristics mentioned in sub-section 7.3. The Owner attribute is set to "User" to distinguish these requirements from the other design requirements. This can be very helpful in the future when the designer wants to check the implementations of all the user requirements.

```
<System Name="SY_Fiscal_Tax">
    …
    <Requirement Name="RE_Documentation_Part" Owner="Designer">
        <Title>Documentation</Title>
        <Description>The system shall contain a documentation </Description>
    </Requirement>
    …
</System>
```

Figure 13.4 – Defining design requirements for the system

Next, the designer may create and formulate new design requirements. These are specified in the same way as the user requirements are specified. One design requirement of the example case is shown in Figure 13.3. The only difference with the user requirement is that the Owner attribute is set to a different stakeholder, in this case changed to “Designer”, and the requirement is technical. When the Owner attribute is not used no immediate difference between de user and design requirements can be identified. The new design requirement state that the system shall have documentation.

It is now time for the designer to decompose the system into sub-systems. This is represented in Figure 13.5. The designer creates three new sub-systems with the names “SS_Software”, “SS_Infrastructure” and “SS_Documentation”. The “SS_Documentation” sub-system is a result of requirement “RE_Documentation_Part”, which state there should be documentation of the system. This documentation will be provided in the sub-system. This means that the creation of this sub-system implements the documentation design requirement. The designer will state this implementation later.

```
<System Name="SY_Fiscal_Tax">
  …
  <Subsystem Name="SS_Software">
    <Title>Infrastructure</Title>
  </Subsystem>
  <Subsystem Name="SS_Infrastructure">
    <Title>Infrastructure</Title>
  </Subsystem>
  <Subsystem Name="SS_Infrastructure" Owner="Designer">
    <Title>Infrastructure</Title>
    <Motivation>The documentation of the system is placed here</Motivation>
  </Subsystem>
  …
</System>
```

Figure 13.5 - Defining a sub-systems for the system

Only for the documentation sub-system the owner is specified, which is set to “Designer”. The other sub-systems have no owner (that means, the Owner attribute is not set). The designer gives all the sub-systems a title, but does not specify the motivation for the sub-systems except for the third sub-systems. Specifying the motivation of a particular decision is up to the designer, but it is recommended. These motivations can become a great help in the future. The motivation for the third sub-system describes that this sub-system will contain all the documentation needed for the system. The designer does not design the sub-systems further in detail. This will be done in a later stage.

It is now time to relate the user and design requirements to the newly created sub-systems. Relating requirements can be done using the Implementation or Mapping element. When a sub-system is implementing a requirement, the designer indicates this by using the Implements element. The designer may deliberately designed the sub-system in such a way that it is

responsible for the implementation of a requirement. This is the case with the sub-system "SS_Documentation", which was created to fulfill the requirement with the name "RE_Documentation_Part". The designer then uses the Implements element as illustrated in Figure 13.6. The implementation is given the name "IM_Doc_Part". It relates requirement "RE_Documentation_Part" to sub-system "SS_Documentation".

```
<System Name="SY_Fiscal_Tax">
    …
    <Implementation Name="IM_Doc_Part">
        <Title>Implementation of Documentation Part</Title>
        <Source Name="RE_Documentation_Part" />
        <Target Name="SS_Documentation" />
    </Implementation>
    …
</System>
```

Figure 13.6 – Implementing the requirements on the new sub-systems

The designer decides to map or relate all the remaining requirements to the different and appropriated sub-systems. This is done with the Mapping element and is illustrated in Figure 13.7. In the example a mapping is made with the name "MA_Software", which maps the two user requirements "RE_Configurable_Screens" and "RE_Configurable_Emails" to sub-system "SS_Software". The designer let the designer of sub-system "SS_Software", which can be him self or some other designer, determine what to do with the requirements. The designer only tells the sub-system designers that they should do something with these requirements.

```
<System Name="SY_Fiscal_Tax">
    …
    <Mapping Name="MA_Software">
        <Title>Software requirements mapping</Title>
        <Motivation>All requirements are software related</Motivation>
        <Source Name="RE_Configurable_Screens" />
        <Source Name="RE_Configurable_Emails" />
        <Target Name="SS_Software" />
    </Mapping>
    …
</System>
```

Figure 13.7 - Mapping the requirements on the new sub-systems

At this point, the design process of the system is finished and the design process of the sub-systems can begin. As pointed out in the introduction of this section, the design process of the system may be redone. When designing the sub-systems, mistakes and other occurrences may arise. This forces to redesign the System element.

The sub-system design process can be done by other designers or by the same designer that designed the system. The design processes of each sub-system are somewhat similar. As described in the introduction of this section, the use of the repository is not linear and can be executed in different ways.

13.1.2. Defining the sub-systems "SS_Software"

The designer decides to design the sub-system with the name "SS_Software". Figure 13.8 illustrates this empty sub-system. The reason for this sub-systems existence is that the designer of the system decomposed the system in this sub-system. The system designer then decided that the sub-system currently designing implements the requirement "RE_Software_Part". It is then up to the sub-system designer to design this sub-system in such a way that it will meet the constraints of the requirement.

```
<System Name="SY_Fiscal_Tax">
    …
    <Subsystem Name="SS_Software">
        <Title>Infrastructure</Title>
    </Subsystem>
    …
</System>
```

Figure 13.8 - Sub-systems design

Just like the system design process, the designer creates new design requirements. For this sub-system the designer created a requirement called "RE_Configuration_File_Dependence". This is shown in Figure 13.9. As the name of the requirement may suggest, this requirement describes a dependency between some items. The requirement state that the different screens, which will be used in the system depend on the existence of a configuration file. This way of creating dependencies was explained in section 10.3.

```
<System Name="SY_Fiscal_Tax">
    …
    <Subsystem Name="SS_Software">
        …
        <Requirement Name="RE_Configuration_File_Dependence">
            <Title>Configuration File Dependecy</Title>
            <Description>The different screens depend on the configuration file. This is
            needed form showing the right content</Description>
        </Requirement>
        …
    </Subsystem>
    …
</System>
```

Figure 13.9 - Defining design requirements for the sub-system

The designer then defines new sub-systems, but this is not done in the way described in the previous section. The designer decides to only declare the new sub-systems by using the Declaration element and not use the Subsystem element. When in the future the subsystem needs to be designed, the Subsystem element will be added to this sub systems body. The declaration of the sub-system "SS_Input_Page" is shown in Figure 13.10. The declaration of the sub-system fulfills requirement "RE_Input_Screen", which is immediately implemented. This is done by the use of the Implements element. This element states that the requirement is implemented by the declared sub-system.

The requirement that is implemented by the new sub-system was created in the system and mapped onto the current sub-system. The designer is doing what he is told to do by the system design, which is doing "something" with this requirement. In this case, the designer implements the requirement.

```
<System Name="SY_Fiscal_Tax">
  …
  <Subsystem Name="SS_Software">
    …
    <Declaration Name="SS_Input_Page">
      <Title>Input page</Title>
      <Motivation>Page for the user input</Motivation>
      <Implements Name="IM_Input_Page">
        <Title>Input page implementation</Title>
        <Motivation>The implementation of the input screen
        requirement</Motivation>
        <Source Name="RE_Input_Screen" />
      </Implements>
    </Declaration>
    …
  </Subsystem>
  …
</System>
```

Figure 13.10 - Declaring a sub-system in the sub-system

Finally the designer needs to map all other requirements, the ones just created and the ones received from the (parent) system, to the newly declared sub-systems. Again the Mapping element is used to realize this.

In Figure 13.11 the designer maps the dependency requirement on the three sub-systems "SS_Success_Page", "SS_Error_Page" and "SS_Input_Page". This means that all these sub-systems depend on the configuration file and might not work properly if this file does not exist.

For now, the designer is finished with the design process of sub-system "SS_Software". But the sub-system needs further work, because the declared sub-systems need to be further designed. They are at this moment empty declarations which do nothing. This design process is done another time.

```
<System Name="SY_Fiscal_Tax">
   …
   <Subsystem Name="SS_Software">
     …
     <Mapping Name="MA_Configuration_File">
       <Title>Configuration File</Title>
       <Motivation>Mapping the configuration file requirement to the
       screens</Motivation>
       <Source Name="RE_Configuration_File_Dependence" />
       <Target Name="SS_Success_Page" />
       <Target Name="SS_Error_Page" />
       <Target Name="SS_Input_Page" />
     </Mapping>
     …
   </Subsystem>
   …
</System>
```

Figure 13.11 – Mapping requirements on the new sub-systems

13.1.3. Defining the sub-systems "SS_Documentation"

The next sub-system the designer decides to design is sub-system "SS_Documentation". This sub-system was created to implement the requirement "RE_Documentation_Part". This means that in this sub-system the different documents can be specified. This current sub-system design process differs a bit from the process described in the previous sub-section. It will be much sorter and only a few elements are used.

```
<System Name="SY_Fiscal_Tax">
   …
   <Subsystem Name="SS_Documentation">
     …
     <Requirement Name="RE_Requirement_Document" Owner="Designer">
       <Title>Requirements documentation</Title>
       <Description>The system shall contain a requirements document</Description>
     </Requirement>
     <Requirement Name="RE_System_Archtecture_Document" Owner="Designer">
       <Title>System architecture documentation</Title>
       <Description>The system shall contain a system documentation</Description>
     </Requirement>
     …
   </Subsystem>
   …
</System>
```

Figure 13.12 - Defining design requirements for the sub-system

The first step the designer has to do is defining new designer requirements. The designer decides to create requirements for each document that is needed. Figure 13.14 shows two of these requirements. The first requirement states that there shall be a requirements document. The second requirement states that there shall be system architecture documentation.

The next step is declaring the different documentation files. Declaring files is done by using the Declaration element. By doing this, the requirements in the repository can be linked to external files. Tooling may then be able to check for the existence of these file and is thus able to check if the requirement is really implemented in this sub-system. In Figure 13.13 the declaration of the file "Req.doc" is illustrated. The declaration is named DE_Requirements_Document" and indicates to the repository that the designer wants to use an external file. To indicate that the declaration is of a file and not a new sub-system which is located in an external file, the Type attribute is set to "File". The title and the motivation is then set by the designer. Again, the designer doesn't want to indicate who the owner of the declaration is.

Because the declaration of the file "Req.doc" implements the requirement "RE_Requirement_Document", the Declares element can be used. This is also illustrated in Figure 13.13. The implementation is named "IM_Requirements_Document" and states that the declaration implements requirement "RE_Requirement_Document". The designer also had the option to use the Implementation element as discussed in the previous sub-section and state the implementation.

At this point the design process for sub-system "SS_Documentation" is finished. The design process was short, but it still implemented different new requirements. These implementations, like the one of file "Req.doc", are the leaves of the hierarchical tree view.

```
<System Name="SY_Fiscal_Tax">
   …
   <Subsystem Name="SS_Documentation">
      …
   <Declaration Name="FI_Requirements_Document" Type="File" Filepath="./Req.doc">
      <Title>Requirements documentation</Title>
      <Motivation>The Requirements documentation of this project</Motivation>
      <Implements Name="IM_Requirements_Document">
         <Title>Implementation of req doc</Title>
         <Source Name="RE_Requirement_Document" />
      </Implements>
   </Declaration>
      …
   </Subsystem>
   …
</System>
```

Figure 13.13 - Defining declarations for the sub-system

13.2. Rules for the use of the repository

In this section the constraints on the use of the repository are described. The repository does not have many rules. As long as the repository has the structure as described in section 11.2 is

maintained everything is all right. It is allowed to define only the system structure by using the System en Sub-system elements or by defining requirements, but not implementing them.

Although the repository is very flexible there is one thing that is not allowed. There should only be one root system element. Sub-system elements are not allowed to have no parent otherwise the hierarchical tree is broken. Thus the repository has one System element as a root and Sub-systems always have a System or Subsystem element as its parent.

13.3. Stakeholders

In section 9.4.3 the way the different stakeholders use the repository are described. These stakeholders will use the repository by adding, changing, updating, and viewing data in the repository. The different repository stakeholders are illustrated in Figure 13.14.

Some of the stakeholders identified are the ones that execute the steps shown in Figure 13.1. and are responsible for the data stored in the repository. These stakeholders and their steps are shown in Figure 13.15. The stakeholders that are involved in the creation of the data in the repository are:

- The requirements engineer is only active in the first two steps.
- The designer is involved in all the steps;
- The maintenance programmer usage of the repository can differ from only the last four steps to all the steps
- The programmer is in some cases responsible for design a sub-system.

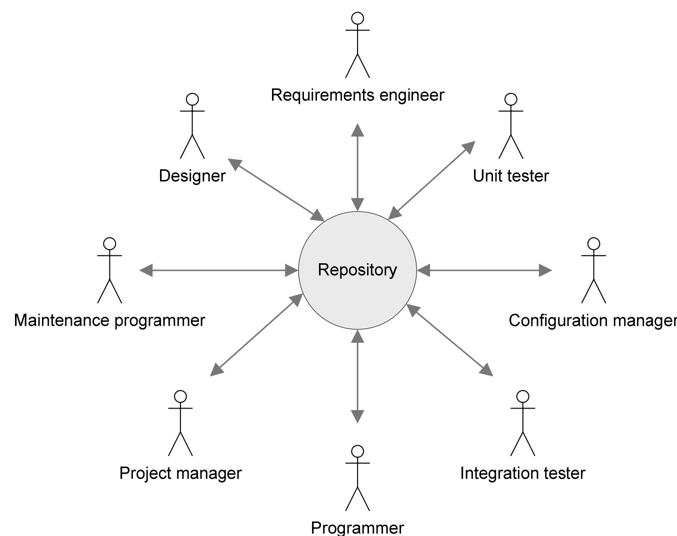

Figure 13.14 - Repository users

The other remaining stakeholders are only users of the data stored in the repository. They are not involved in any of the repository steps. These stakeholders only use the information to do their tasks or generate documentation.

The following sub-sections described the operation of the repository by the different users.

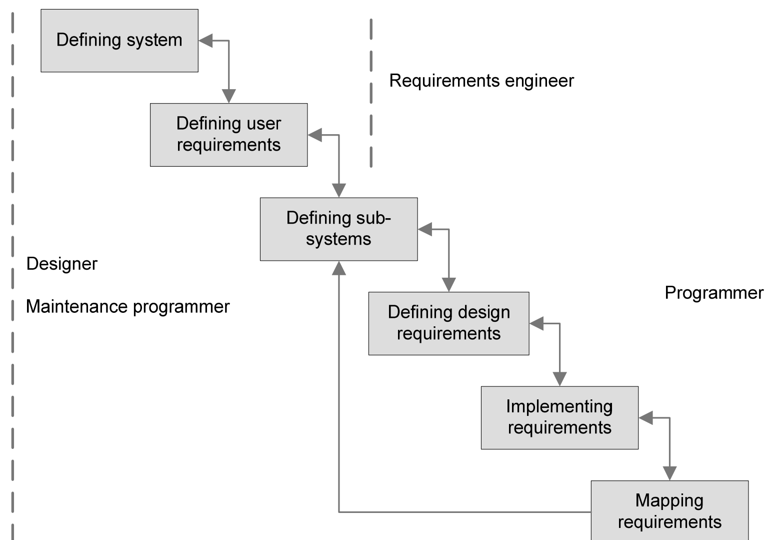

Figure 13.15 - Repository use steps with the users

13.3.1. Requirements engineer

The requirements engineer is in most cases the first user of the repository and is responsible for the creation of the repository. When a repository is created, the system is created by declaring the System element. The system will have the name and a description of the system to be built. From this starting point the system will be further designed by the designer.

The requirements engineer stores all the requirements, gathered and specified in the requirements definition activity, in the repository. The requirements engineer may change the requirements or add new requirements during the development project. When a requirement needs to be changed, an impact analysis may be executed to analyze the impact of the change. The result of this analysis check may decide if the requirement should be changed and provide the consequences of that change. The requirements engineer may also check the consistency of the repository to examine the status of all the user requirements.

The requirements engineer can generate documentation from the data stored in the repository. This can be a requirements document or trace documentation of particular requirements.

13.3.2. Designer

The designer is in most cases responsible for the system structure data stored in the repository. It is his job to design the system in such a way that the system implements all the user requirements. This is done by decomposing the system into sub-systems. During the design process new design requirements may be created which provide constraints on the design process. Both the user- and design requirements are then linked to the appropriated sub-system. This can be done as an implementation or as just as a mapping relationship.

To check if all the requirements are implemented in the system, the consistency check can be executed. All requirements, user- and design requirements that are not implemented are the result of this check. It is then up to the designer to implement these requirements to make the repository consistent.

When changes are announced, the designer may need to redesign the system to coop with these changes. To implement the changes, the designer can use the impact analysis to identify the areas that need to be redesigned.

The documentation generated from the repository for the designer is very versatile. The designer can generate documentation about the component structures, the design decisions, the results of the analysis checks, or the complete system structure.

13.3.3. Project manager

Project managers will use the repository for planning and estimating the cost of the system. To be able to plan and estimate the cost of the system, the project manager needs to know the structure of the system. The system and all the sub-systems in the repository form a hierarchical tree, and are well commented, so the project manager can easily understand the system. With this structure in mind, the project manager can estimate the cost and make the project planning.

The project manager can easily gather information for this planning and cost estimation activity when changes are proposed. This information is provided in the form of impact analysis check results. The project manager can then take the severity of the impact of a change into account to adjust the planning and cost estimation for the system.

13.3.4. Configuration manager

As described in the previous section, the complete structure of the system is stored in the repository and is well commented. The configuration manager can use this structure to assemble the various components into one system.

The configuration manager is responsible for controlling the changes in the system design and requirements stored in the repository. Again, changes in the system can be analyzed by the impact analysis check and he results of this check can be transformed in documentation. After this check it should be clear which requirements and system structures need to be changed to coop with the proposed change.

13.3.5. Programmer

Programmers will use the repository as a guide for their programming problems. They should transform the design of the system stated in the repository to executable code. In most cases, the programmer is assigned to a sub-system, for which he needs to program executable code.

Sometimes these sub-systems are not designed in detail by the designer. The designer has left the details to be filled in by the programmer on purpose. This is done to give the programmer some room for being creative when programming his sub-system. The programmer then uses the repository as a designer. He will then design the sub-system till it can be implemented in executable code.

The programmer can generate documentation of the sub-systems it structure and all its requirements. This information can then be used to understand the sub-system or to program it in executable code.

13.3.6. Unit tester and integration tester

Unit testers are responsible for testing the components or sub-systems. To be able to perform these tests, knowledge of the system is needed. This can be found in the repository, where the structure of the system and the related requirements is stored. Detailed test-cases can be created for each requirement, because of the well documented system structure.

The unit tester is also able to produce al sorts of documentation need to define the different test-cases and testing the system.

13.3.7. Maintenance programmer

Maintenance programmers are responsible for the design, programming and testing of bugs, errors and suggested changes. This means that the maintenance programmer assumes multiple roles as repository users, which are mentioned in the previous sections. See the previous sections for the use of the repository by the different user roles.

13.4. Summary

This chapter provided information about how the repository should be used in general and per stakeholder. Important to understand is that there is no specific way the repository should be used. This is the strength of the repository. It does not force a user to follow specific steps. In general the following steps are identified for the use of the repository:

- Defining system;
- Defining user requirements;
- Defining sub-systems;
- Defining design requirements;
- Implementing requirements;
- Mapping requirements.

The designer or requirements engineer defines the system to be built. This is done by using the System element and forms the start for both the design process and the repository. The next step is to define and add the user requirements to the repository. After is this done, the system is decomposed in sub-systems. These sub-systems form a new design process. But the design process of the system is not finished. The designer may create new design requirements to add constraints to the new sub-systems. The last thing the designer needs to do is to relate the different requirements to the newly created sub-systems. This can be done by means of the implementation or mapping relationship. After designing the system, the sub-systems are designed. This is done till there are no sub-systems left or the designer does not want to design any further.

The different stakeholders of the repository identified in an earlier chapter use the repository in different ways. In general, the requirements engineer, designer, and maintenance programmer are the stakeholders that add data to the repository. The other stakeholders are retrieving the data from the repository.

14. EXISTING DESIGN STORAGE COMPARISONS

In this chapter two design storage ways are compared with the in this thesis suggested way. Section 14.1 provides information about UML diagrams. In second 14.2 MS Word is discussed for saving design data. In section 14.3, a comparison is made between de UML diagrams, the MS Word documents, and the repository suggested in this thesis. The final section will summarize this chapter.

14.1. Unified Modeling Language diagrams

Most tools used in the software engineering market use the Unified Modeling Language (UML)-based modeling concept as a foundation. UML is a commonly used design or modeling language and not a method [20][69]. It is best known as a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a system [8].

UML exist of thirteen types of diagrams show in Figure 14.1. A diagram is a graphical representation of a set of element, most often rendered as a connected graph of vertices (things) and arcs (relationships). Diagrams are drawn to visualize a system from different perspectives, so a diagram is a projection into a system.

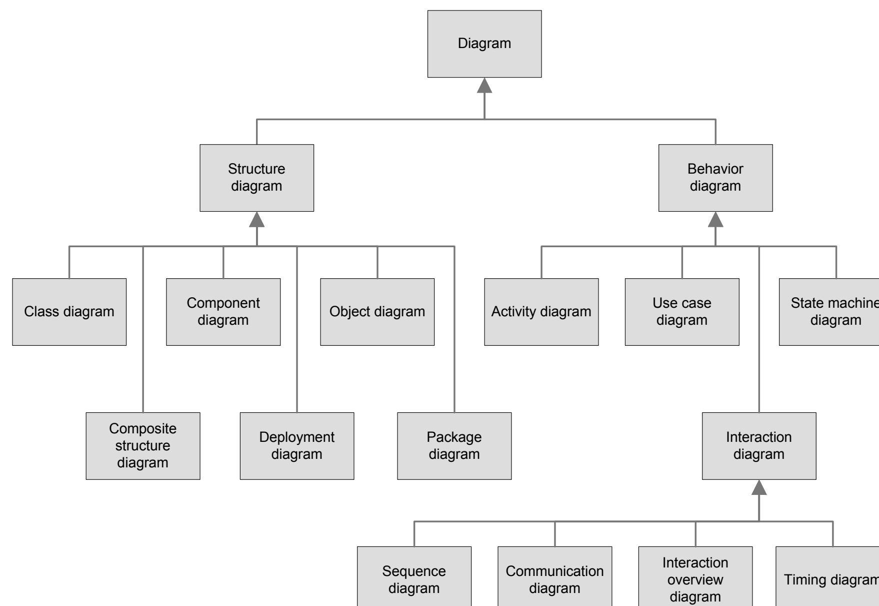

Figure 14.1 - UML diagrams based on [69]

The blocks named “Diagram”, “Structure Diagram”, “Behavior Diagram” and “Interaction diagram” are only for categorization purpose and are not actual UML diagrams. In the following summary the meaning of each diagram is given [2]:

- *Activity Diagram*: Depicts high-level business processes, including data flow, or to model the logic of complex logic within a system;

- *Class Diagram*: Shows a collection of static model elements such as classes and types, their contents, and their relationships;
- *Communication Diagram*: Shows instances of classes, their interrelationships, and the message flow between them. Communication diagrams typically focus on the structural organization of objects that send and receive messages;
- *Component Diagram*: Depicts the components that compose an application, system, or enterprise. The components, their interrelationships, interactions, and their public interfaces are depicted;
- *Composite Structure Diagram*: Depicts the internal structure of a classifier (such as a class, component, or use case), including the interaction points of the classifier to other parts of the system;
- *Deployment Diagram*: Shows the execution architecture of systems. This includes nodes, either hardware or software execution environments, as well as the middleware connecting them;
- *Interaction Overview Diagram*: A variant of an activity diagram which overviews the control flow within a system or business process. Each node/activity within the diagram can represent another interaction diagram;
- *Object Diagram*: Depicts objects and their relationships at a point in time, typically a special case of either a class diagram or a communication diagram;
- *Package Diagram*: Shows how model elements are organized into packages as well as the dependencies between packages;
- *Sequence Diagram*: Models the sequential logic, in effect the time ordering of messages between classifiers;
- *State Machine Diagram*: Describes the states an object or interaction may be in, as well as the transitions between states. Formerly referred to as a state diagram, state chart diagram, or a state-transition diagram;
- *Timing Diagram*: Depicts the change in state or condition of a classifier instance or role over time. Typically used to show the change in state of an object over time in response to external events;
- *Use Case Diagram*: Shows use cases, actors, and their interrelationships.

With these diagrams a system can be modeled at different levels of detail. The diagrams can also in tooling be linked to each other to provide a detailed representation of the different aspects of the system.

14.2. MS Word documents

MS Word is widely used for storing the artifacts of system engineering. As described in earlier chapters, Chess and other companies use MS Word instead of the advanced tools top store the design artifacts. Thus, MS Word can also be classified as a repository for storing design artifacts.

MS Word is an advanced text editor where text, diagrams, and images can be integrated in one document. Linking artifacts to other artifacts in MS Word documents is possible by using references, cross-references, and hyperlinks, but this is not ideal and time consuming to do.

14.3. Comparison

In the previous section the UML and MS Word documents were discussed. In this section this repository is compared with the repository proposed in this thesis. Although both items compared are repositories with the purpose to store design data, it is a dishonest comparison. There are several reasons why this is dishonest:

- UML diagrams exist for almost ten years and are in these years often revised. MS Word documents exists even longer;
- UML was developed by the Object Management Group which consists of different companies and system engineering gurus from all around the world which bundled their knowledge about system engineering;
- MS Word documents are not specifically designed for defining the system engineering artifacts. It is more often used for just storing text and images;
- Both UML diagrams and the MS Word documents have tooling, which support these repositories.

Comparing these repositories is like comparing MS Visio and AutoCAD. Both tools are used to draw images, but AutoCAD is much more sophisticated then MS Visio. In this example MS Visio can be seen as the MS Word document and AutoCAD as UML. Both are their extremes.

Although the great differences a comparison is made. Table 14.1 illustrates this comparison in a matrix.

Criteria	MS Word documents	UML diagrams	The suggested repository
Support querying		X	X
Support changing	X	X	X
Supports capturing requirements	X	X	X
Supports capturing designs	X	X	X
Supports traceability		X	X
Supports document generation		X	X
Supports automatic analysis		X	X
Is human editable	X	X	X

Table 14.1 - Design notation vs. criteria matrix

MS Word documents are plain text with possibly images to support the text. Therefore the use of these documents is limited by this. The different analysis checks suggested in section 6.6 are

hard to execute, let alone to automate these analysis checks. The use of MS Word documents as a repository is thus very simplistic and limited.

UML diagrams on the other hand are very complex. This is duo to the fact that UML repository is a product of a combination of different requirements opposed by different stakeholders. Each stakeholder wants his ideas implemented in the UML, which results in a big and complex set of diagrams. It is simply said a collection of different ideas combined in a particular manner.

The repository suggested in this thesis should be beneficial for both the users of MS Word documents who use these documents as a repository and do not like the disadvantages of these types of repositories, as well for the users who find UML too complex and difficult to use. For these users this repository should be a welcome addition. The repository tries to be simplistic, but offer all the necessary features to be used in the system engineering process.

To clarify this categorization, the example of MS Visio and AutoCAD is presented again. Both tools can draw 2D images, but when that is all that is needed, using MS Visio will be sufficient. If more is wanted then just making 2D images, like making 3D images with specific calculations and measurements, a more sophisticated tool like AutoCAD should be used. Users who want to draw 3D images, but do not want to use all the calculations and measurements, should use a tool that supports only drawing 3D images. This is also applicable for the choice which repository should be used. The repository suggested in this thesis tries to be that image program that only supports drawing 3D images.

14.4. Summary

In this chapter a comparison was made between the UML repository (both graphical and textual), the MS Word document repository, and the repository proposed in this thesis. This comparison is dishonest because of the following reasons:

- Time the repository exists;
- Development support;
- Purpose of the repository.

The UML diagrams are very complex, advanced, and detailed. The MS Word documents however are simplistic and do not support automated analysis checks. Still MS Word documents are widely used as a repository for storing design artifacts.

The repository proposed in this thesis can be placed between the MS Word documents and the UML repository. It tries to be simple in use, but support the different analysis checks. This repository should be a welcome addition for the designers who still use MS Word documents as primary repository for storing design artifacts.

15. CONCLUSION AND RECOMMENDATIONS

In this final chapter the conclusion, recommendations, and future work regarding the suggested repository are provided. In section 15.1 the conclusion of this research is given. In section 15.2 the research questions stated in section 2.5 are answered. Section 15.3 provides some recommendation and future development for the repository. Section 15.4 provides a reflection of the execution of this research project.

15.1. Conclusion

The repository suggested in this thesis is described as a solution for the problems that occur in the management of design artifacts as described in chapter 5. MS Word documents are used at Chess and other companies to store all the artifacts of system engineering. MS Word documents can only be used to record the artifacts and serve also as the documentation for the different stakeholders. The suggested repository should be a welcome addition to these stakeholders who want simplicity as provided by MS Word documents, but also want specific functions like automated analysis checks, simple document generation, and the use of traceability. The repository of this thesis provides these features.

The repository will be a central storage facility where all the design artifacts are stored in. This creates a central place where all the different stakeholders add and retrieve their information from. This means that all data used in the project is the same for every stakeholder.

There are only four stakeholders that will add, change, and delete data in the repository and use this data to perform their tasks. These stakeholders are the requirements engineer, designer, programmer, and the maintenance programmer. The other stakeholders mainly the testers and integrators are only using the data to perform their task.

The repository stores the design process as it is executed. The design process supported by the repository consists of defining requirements, system, sub-systems, and design decisions. The system and sub-systems form a hierarchical tree. The design decision and requirements are then related to these systems and sub-systems. This is process and how this is captures is described in chapters 10. and 13. The structure of this repository was described in chapter 11.

The different relationships stored in the repository make it possible to perform traceability, execute different automated analysis checks, and generate simple documentation. The generation of design documentation was described in chapter 11. and traceability and analysis checks were described in chapter 10. Important to understand is that the repository it self is not able to perform these traces, checks, and documentation generation. It only provides the mechanisms to perform these tasks. Tooling should be created to perform traceability and the actual analysis checks.

Finally, it should be stressed out that there is a long way to go before this tool is mature enough for industrial use. But it satisfies the basic software design repository needs. A lot of designing, development, and testing needs to be done for both the repository and the needed tooling before the repository is mature enough to be used as primary design tool. Eventually, the

use of best-practices will transform the repository and tooling into a mature system design platform.

15.2. Evaluating research questions

In this section the research questions stated in section 2.5 are systematically evaluated and answered. Each of these research questions can be answered with the information provided in this thesis. The research questions and their answers are:

[R1] *What is system engineering in theory and in practice?*
System engineering is the activity of specifying, designing, implementing, validating, deploying, and maintaining systems as described in de design definition [58]. It transforms an idea into a system. A system can exist of different engineering fields like hardware-, software-, electronic-, and mechanical engineering. The description of system engineering in theory and practice are similar, although adaptations of the steps involving system engineering may exist;

[R2] *What is software development in theory and in practice?*
Software engineering is a part of the system engineering. It is an organized set of activities performed to translate user needs into software products [54]. The description of software engineering in theory and practice are similar, although adaptations of the steps involving software engineering may exist;

[R3] *What is software design in theory and in practice?*
The software design activity involves adding formality and detail as the design is developed with constant backtracking to correct earlier designs. During this activity the user requirements are translated in a system model. The design activity is an yet to be explored territory. That means that this activity has different approaches;

[R4] *What are the artifacts of software design in theory and in practice?*
Artifacts are pieces of information about the software. This can be a version number or the structure of the system. Every piece of information about the software that is important is classified as an artifact;

[R5] *Why is management of software design necessary?*
Changes happen during the development of a system. These changes may be caused by requirements or design changes. These changes need to be processed in the software. To be able to do this the software design need to be redesigned. All these factors demand the management of the software design. When software management is neglected, the software design MAY contain errors, which also may be present in the software;

[R6] *What is traceability?*
The degree to which a relationship can be establish between two or more products of the development process, especially products having a predecessor-, successor-, or master-subordinate relationship to one another [31]. It tries to follow or trace these user requirements, goals, or objectives as there are translated into an operational system;

[R7] *How can traceability be used in management of software design?*
Traceability is very important for understanding the software design. It also make it possible to perform different analysis checks like impact- and inconsistency analysis checks. Traceability is used to make sure the software design satisfies all the user and design requirements;

[R8] *What trace information is needed for the management of software design?*
The traceability information need for the management of software design are the result of impact analysis, inconsistency analysis, dependency analysis, relation following analysis, requirement performance verification analysis, and requirement verification analysis;

[R9] *What is a repository?*
A repository is a central place where data is stored and maintained;

[R10] *What are the requirements for such a repository?*
Chapter 9. provided the requirements for the repository. These are too many to summarize in this section;

[R11] *Which information needs to be stored in a repository?*
The information that need to be stored in the repository are requirements, system structures, design motivations, and relationships. This information is also addressed a design artifacts of the repository;

[R12] *How can trace information of software design be saved in a repository?*
Traceability is made possible by the relationships stored in the repository. The system structure forms a hierarchical tree, which by it self is ideally for traceability. Both the design motivations and requirements are then explicitly related to the different (sub-)systems. These relationships in combination with the tree structure of the system structure provide the ability to perform traces in the repository;

[R13] *What is the design of the repository?*
The design of the repository is provided in chapter 10. This description is to big too present it in this section;

[R14] *How to implement and test this repository?*
The design of the repository is provided in chapters 11. 12. This descriptions are too big to present it in this section;

[R15] *How can trace information of software design saved in a repository be used for documents generation?*
For the generation of documentation the trace information can be used as described in research question [R7]. All the system structure, design motivation and requirement data in combination with the relationships can be used to generate different documents.

15.3. Recommendations and future work

In the first section of this chapter the conclusion was drawn that the repository suggested in this thesis satisfies the basic design needs. These basic needs are not more then just capturing the requirements, (sub-)systems, the relations between these items, and the motivation for these

design decisions in a repository. This first version of the repository also supports some analysis checks and document generation actions.

This first version is fine when the basic design actions need to be met, but to get more mature and usable extra features and expansions are needed. The following sub-sections suggest some expansions to the current repository:

- *Tools*: should be written to make it easier to use the repository and to automate analysis checks and document generation. Currently, a Chess employee is developing a tool for the repository;
- *System engineering scope*: the scope of the repository should be increased to the whole system engineering area. This was also the original research projects goal;
- *Pre- and post conditional requirements*: are conditions for specific requirement to state the situation when this requirement is correctly implemented in the system. This means that a condition before and after the situation described by a requirement must be right to satisfy the requirement;
- *Abstractions*: are pre-defined system structures that can be incorporated in the system. Examples of abstractions are patterns or standards. The repository should be able to refer or incorporate these abstractions;
- *Documentation*: generation should be done according to different standards. This may mean that some additional information needs to be stored in the repository to meet this documentation standards demands. Another graduate student begins in September 2006 with a research study;
- *Dependencies*: need to be made analyzable. At this moment dependencies are formulated as requirements, but the status of these dependencies can not be automatically checked. This means that the dependency relationships need to be saved in the repository as the implementation relationships.

15.4. Reflection

In this section a reflection on the research project is given. This is needed because the research project was not performed as it should be. The project took nine months, instead of six months and still this thesis is not at the highest quality. The main reasons for these problems are:

- At the beginning of the project, the goal and results of this research was not clear. It took a long time before everybody new what the goal of the project was. At the end of this research project there are still some misunderstandings about what the real goal and results are of this thesis;
- At the beginning and still at the end, all committee members and the author of this thesis have their own opinions about how the research project should be executed. This means that every member has its own whishes, constraints, and demands which need to be satisfied in this thesis. These whishes, constraints, and demands conflicted sometimes with each other and where be hard to satisfy;

- The research project was very practical and so was the author of this thesis. This made it very difficult to provide a more theoretical approach for this research project;
- Finally, the author of this research project had some difficulties to stay focused with what he was writing. Some parts of the thesis are more a sidetrack of what should be stated in the thesis. Also translating the ideas in English text grated some problems.

All the points mentioned above should be approached differently in the future. The main and important thing to focus on the next time is to get all participants the same on the projects goal and results. This means that everybody knows what the goal of the project is and what expected results are.

This process starts with the research him self. It should be clear to himself what the goal and results should be before trying to explain it to other members of the research project. This means that the researcher needs a clear understanding of the problem, the constraints, the expectations, and the requirements of the project. From this he tries to formulate a clear and understandable goal and results.

The next step is then to communicate the problem, the goal, and the results to the project remembers in such a way that they understand it. It is not possible to guarantee for hundred percent that everybody has a complete overview of the problem, the goal and the results, but possibilities for different interpretations should be kept to a minimal.

As a result of a clear view of the goal and the results the researcher can do its research more effectively and purposively. This means that the research is focused completely and only on the goal instead of investigating sidetracks, which are not needed to reach the goal and the results.

GLOSSARY

Artifact	An artifact is information about the software that are identified in the development process by development project members.
Business requirement	The business requirements state the general reason for building the system [3]. These requirements state what the repository in general shall support or accomplish, without describing the detail how this is realized. Business requirements should [9].
Consistency	Consistency is the correspondence among related aspects [62].
Dependency	Dependency is the reliance on something for help or support [46][62].
Design artifact	A design artifact is design information about the software that are identified in the development process by development project members.
Designing	The approach that engineering (and some other) disciplines use to specify how to create or do something [62].
Diagram	A diagram is a graphical representation of a set of element, most often rendered as a connected graph of vertices (things) and arcs (relationships) [69].
Functional requirements	Functional requirements provide statements of services that the system should provide, how the system should react to particular inputs, and how the system should behave in particular situations [58].
Implementation	Implementation is the process of translating a design into hardware components, software components, or both [31].
Mapping	Linking an artifact to another artifact by means of defining a relationship.
Non-functional requirements	Non-functional requirements or quality attributes specify criteria that can be used to judge the characteristics of a system, rather than specific behaviors or functions [70].
Open source	Refers to software that is created by a development community rather than a single vendor and the source code is free and available to anyone who would like to use it or modify it for their own purposes [62].
Parsing	Analyzing or separating (for example input) into more easily processed components [62].
Quality attribute	Non-functional requirements or quality attributes specify criteria that can be used to judge the characteristics of a system, rather than specific behaviors or functions [70].
Repository	A repository is a central place where data is stored and maintained [70].

Requirements Requirements describe 'what' that system should do and not 'how' the system should do it [55].

Software system A software system consists of a number of separate programs, configuration files, which are used to set up these programs, system documentation, which describes the structure of the system, and user documentation, which explains how to use the system and website for users to download recent product information [58].

Stakeholders A person who has a share or an interest in something [62].

System A system is a purposeful collection of interrelated components that work together to achieve some objective [32].

System engineering System engineering is the activity of specifying, designing, implementing, validating, deploying, and maintaining systems as described in de design definition.

Test-case A test-case defines a step-by-step process whereby a test is executed [41].

Testing Testing is the process of planning, preparing, executing and assessing, which aims at determining the characteristics of a information system and to show the difference between current and required status [49].

Trace A trace is a record of a relationship between two or more products of the development process [31].

Traceability The degree to which a relationship can be establish between two or more products of the development process, especially products having a predecessor-, successor-, or master-subordinate relationship to one another [31].

REFERENCES

[1] Als, A., & Greenidge, C. (2005). The waterfall model. Retrieved February 27, 2006, from http://scitec.uwichill.edu.bb/cmp/online/cs22l/waterfall_model.htm

[2] Ambler, S. W. (2006). Introduction to the diagrams of UML 2.0. Retrieved August 7, 2006, from http://www.agilemodeling.com/essays/umlDiagrams.htm

[3] Archer, J. E. (2003). Requirement tracking: a streamlined approach. *Proceedings of the 11th IEEE International Requirements Engineering Conference*, 305.

[4] Barnard, H. J., Metz, R. F., & Price, A. L. (1986). A recommended practice for describing software designs: IEEE standards project 1016. *IEEE Transactions on Software Engineering, 12*(2), 258-263.

[5] Belev, G. C. (1992). Design management-begin at the beginning. *Proceedings Annual Reliability and Maintainability Symposium,*98-100

[6] Bireck, M., Diamond, J., Duckett, J., Gudmundsson, O. G., Kobak, P., Lenz, E., et al. (2001). *Professional XML* (2 ed.): Wrox Press Ltd.

[7] Bohner, S. A. (1991). Software Change Impact Analysis for Design Evolution. Proceedings of the 8th International Conference on Software Maintenance and Re-engineering.

[8] Booch, G., Rumbaugh, J., & Jacobson, I. (1999). *The unified modeling language user guide*: Addison-Wesley.

[9] Business requirements - high level. Retrieved August 4, 2006, from http://www.ogc.gov.uk/documentation_and_templates_business_requirements_-_high_level.asp

[10] Chess; development of micro-electronics, software and business-critical IT applications. Retrieved 4 June, 2006, from http://www.chess.nl

[11] Cooke, D., Swan, G., Sirott, J., Kane, R., Stevens, P., Yang, J., et al. (1989). Design management in a workstation environment. *Architecture Track, Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences, 1*, 111-117.

[12] Corriveau, J.-P. (1996). Traceability Process for Large OO Projects. *IEEE Computer Society Press, 29*(9), 63-68.

[13] Daft, R. L. (2000). *Management* (5 ed.): The Dryden Press.

[14] Davis, A. M., & Leffingwell, D. A. (1996). Using requirements management to speeds delivery of higher quality applications. from http://citeseer.ist.psu.edu/cache/papers/cs/23414/http:zSzzSzstar2.vub.ac.bezSz~dvermeirzSzcourseszSzsoftware_engineeringzSz696wp.pdf/using-requirements-management-to.pdf

[15] Dick, J. (2005). Design Traceability, *IEEE software*, 22(6), 14-16.

[16] Drescher, P., Miller, J., & Schulz, G. (1990). Design management within a design environment. *Proceedings of the European Design Automation Conference*, 368-373.

[17] Drury, C. (2004). *Management and cost accounting* (6 ed.): Thomson.

[18] EAI Knowledge Base. Retrieved 13 April, 2006, from http://eai.ittoolbox.com/groups/technical-functional/tibco-l/899155#.

[19] Fiksel, J., & Dunkle, M. (1991). Principles of requirement management automation. *Combined Proceedings of the 1990 and 1991 Leesburg Workshops on Reliability and Maintainability Computer-Aided Engineering in Concurrent Engineering*, 231 – 236.

[20] Fowler, M., & Scott, K. (1997). *UML distilled - Applying the standard object modeling language*: Addison-Wesley.

[21] France, R., Koushik, P. & Cline, B. (1992). Programming Standards – General. Retrieved July 26, 2006, from http://www.dlib.vt.edu/projects/MarianJava/CodingStand.pdf

[22] Frezza, S. T., Levita, S. P., & Chrysanthis, P. K. (1996). Linking requirements and design data for automated functional evaluation. *Computers in Industry*.

[23] Gotel, O., & Finkelstein, A. (1994). An analysis of the requirements traceability problem. *IEEE International Conference on Requirements Engineering*, 94-101.

[24] Gotel, O., & Finkelstein, A. (1994). Modeling the contribution structure underlying requirements. *Proceedings of the first international conference on requirement engineering*, 94-101.

[25] Gurp, J. v. & Bosch, J. (2002). Design Erosion: Problems & Causes. *Journal of Systems & Software, 61*(2).

[26] Hodde, R. (2006). How to do Chess - Samenvatting Organisatie and Plan in 2006. (2).

[27] Holzner, S. (2000). *Inside XML* (1 ed.): New Riders Pub.

[28] Hrubý, P. (1998). A Pattern for Structuring UML-Based Repositories. Retrieved July 29, 2006, from http://www.phruby.com/publications/OOPSLA98_TechNote.pdf

[29] Hughes, T. & Martin, C. (1998). Design traceability of complex systems. *Fourth annual symposium on human interaction with complex systems*, 37-41.

[30] IBM Rational Software. Retrieved May 23, 2006, from http://www-306.ibm.com/software/rational/

[31] IEEE Std 610.12-1990 Standard Glossary of Software Engineering Terminology. (1990). Retrieved July 24, 2006, from http://www.swen.uwaterloo.ca/~bpekilis/public/SoftwareEngGlossary.pdf

[32] Institute for telecommunications science. Retrieved April 13, 2006, from http://www.its.bldrdoc.gov/fs-1037/dir-036/_5255.htm.

[33] ISO - International Organization for Standardization. Retrieved 31 May, 2006, from http://www.iso.org/iso/en/ISOOnline.frontpage

[34] Jackson, M., & Zave, P. (1993). Domain descriptions. Proceedings of the IEEE international symposium on requirement engineering, 56-64.

[35] Knethen, A. v., & Paech, B. (2002). A survey on tracing approached in practice and research. *Quasar*.

[36] Konstantinov, G. (1988). Emerging standards for design management systems. *Proceedings of the 1988 Computer Standards Conference, Computer Standards Evolution: Impact and Imperatives*, 16-21.

[37] Kotonya, G., & Sommerville, I. (1996). Requirements engineering with viewpoints. Leesburg Workshops on Reliability and Maintainability Computer-Aided Engineering in Concurrent Engineering, 231-236.

[38] Kowalczykiewicz, K., & Weiss, D. (2002). Traceability: Taming uncontrolled change in software development. *IV Krajowa Konferencja Izynierii Oprogramowania*.

[39] Lee, D. (1990). INCOSE requirements management tool survey. Retrieved April 5, 2006, from http://www.paper-review.com/tools/rms/INCOSERMToolSurvey.doc.

[40] Leffingwell, D., & Widrig, D. (2000). *Managing Software Requirements. A Unified Approach*: Addison Wesley.

[41] Lewis, W. E. (2005). *Software testing and continuous quality improvement* (2 ed.). Florida: CRC Press LLC.

[42] Lifecycle software requirements specification, analysis, design, tracking and project management software tool. Retrieved 23 May, 2006, from http://www.analysttool.com/

[43] Maletic, J. I., Collard, M. L., & Simoes, B. (2005). An XML based approach to support the evolution of model-to-model traceability links. *Automated Software Engineering archive - Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering table of contents*, 67-72.

[44] McConnell, S. (2004). Code complete. *Microsoft Press*.

[45] Mogyorodi, G. E. (2003). What Is Requirements-Based Testing? Retrieved July 5, 2006, from http://www.stsc.hill.af.mil/CrossTalk/2003/03/mogyorodi.pdf

[46] MSN Encarta. Retrieved March 29, 2006, from http://encarta.msn.com.

[47] Paskin, N. (1999). Toward unique identifiers. *Proceedings of the IEEE, 87*(7), 1208-1227.

[48] Pohl, K. (1996). *Process-Centered Requirements Engineering* (2 ed.). New York: John Wiley & Sons, Inc.

[49] Pol, M., Teunissen, R., & Veenendaal, E. v. (2000). *Testen volgens TMAP* (2 ed.): Uitgeverij Tutein Nolthenius.

[50] Problem Tracker - Web-based bug tracking, defect tracking software. Retrieved June 12, 2006, from http://www.problemtracker.com/

[51] Ramamoorthy, C. V., Usuda, Y., A. Prakash, & Tsai, W. T. (1990). The evolution support environment system. *IEEE Transactions on Software Engineering, 16*(11), 1225-1234.

[52] Ramesh, B., & Edwards, M. (1993). Issues in the development of a requirements traceability model. *Proceedings of the IEEE international symposium on requirement engineering*, 256-259.

[53] Requirements Based Testing Process Overview. (2003). Retrieved July 3, 2006, from http://www.benderrbt.com/Bender-Requirements%20Based%20Testing%20Process%20Overview.pdf

[54] Rigby, K. (2003). Managing standards. Retrieved March 2, 2006, from http://sparc.airtime.co.uk/users/wysywig/gloss.htm

[55] Siddiqi, J., & Shekaran, M. C. (1996). Requirements engineering: The emerging wisdom. *IEEE Computer Society Press, 13*(2).

[56] Software Change Management. Retrieved 12 June, 2006, from http://www.software-improvers.com/

[57] Sollins, K., & Masinter, L. (1994). Functional requirements for uniform resource names. from http://ds.internic.com/rfc/rfc1737.txt

[58] Sommerville, I. (2004). *Software engineering* (7 ed.): Pearson Education Limited.

[59] Spanoudakis, G., Zisman, A., Perez-Minana, E., & Krause, P. (2004). Rule-based generation of requirements traceability relations. *The Journal of Systems and Software 72*, 105-127.

[60] Thacker, J. (1997). Notes prepared for a meeting hosted by the National Information Standards Organization on June 18, 1997.

[61] Thayer, R. H., & Dorfman, M. (1997). *IEEE Software Requirements Engineering* (2 ed.). New York: IEEE Computer Society.

[62] The Free Dictionary. Retrieved March 29, 2006, from http://www.thefreedictionary.com.

[63] Trial-use standard for information technology software life cycle processes software development acquirer-supplier agreement. (1995): Institute of Electrical and Electronic Engineers, Inc.

[64] Turbit, N. (2005). Requirements traceability, *The project perfect white paper collection*.

[65] Vellekoop, T., & Klok, B. (2004). Gestructureerd testen SAP. Retrieved July 3, 2006, from http://2004.ngi.nl/docs/limburg/vellekoop_klok.ppt#298,6,Test%20Process%20Improvement

[66] Verschuren, P., & Doorewaard, H. (2005). *Het ontwerpen van een onderzoek* (3 ed.): Lemma BV.

[67] Vliet, H. v. (2000). *Software engineering: Principles and practice* (2 ed.): John Wiley & Sons, Ltd.

[68] Wang, Q., & Lai, X. (2001). Requirements management for incremental development model. *Proceedings Second Asia-Pacific Conference on Quality Software*, 295-301.

[69] Wieringa, R. J. (2003). *Design Methods for Reactive Systems*: Elsevier Science.

[70] Wikipedia - The free encyclopedia. Retrieved May 31, 2006, from http://en.wikipedia.org

Appendix A. XML REPOSITORY STRUCTURE TREE

In Figure A.1 a simple XML tree is given of the repository. This figure shows how the structure of the repository may look like with without any data, thus with only elements.

```
<System>
        <Title></Title>
        <Motivation></Motivation>
        <Declaration>
                <Title></Title>
                <Motivation></Motivation>
                <Implements>
                        <Title></Title>
                        <Motivation></Motivation>
                        <Source></Source>
                </Implements>
        </Declaration>
        <Subsystem>
                <Title></Title>
                <Motivation></Motivation>
                <Declaration>…</Declaration>
                <Subsystem>…</Subsystem>
                <Requirement>…</Requirement>
                <Implementation>…</Implementation>
                <Mapping>…</Mapping>
        </Subsystem>
        <Requirement>
                <Title></Title>
                <Description></Description>
                <Requirement>…</Requirement>
        </Requirement>
        <Implementation>
                <Title></Title>
                <Motivation></Motivation>
                <Source></Source>
                <Target></Target>
        </Implementation>
        <Mapping>
                <Title></Title>
                <Motivation></Motivation>
                <Source></Source>
                <Target></Target>
        </Mapping>
</System>
```

Figure A.1 - Simple XML tree example of the repository

Appendix B. DTD OF THE REPOSITORY

In Figure B.1 show the DTD of the repository. This schema can be used to validate XML documents. It specifies what and how many child elements or attributes are allowed in the body of another element.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT System (Title, Motivation?, Declaration*, Subsystem*, Requirement*,
Implementation*, Mapping*)>
<!ATTLIST System Name ID #REQUIRED>
<!ATTLIST System Owner CDATA #IMPLIED>
<!ELEMENT Subsystem (Title, Motivation?, Declaration*, Subsystem*, Requirement*,
Implementation*, Mapping*)>
<!ATTLIST Subsystem Name ID #REQUIRED>
<!ATTLIST Subsystem Owner CDATA #IMPLIED>
<!ELEMENT Requirement (Title, Description, Requirement*)>
<!ATTLIST Requirement Name ID #REQUIRED>
<!ATTLIST Requirement Owner CDATA #IMPLIED>
<!ELEMENT Declaration (Title, Motivation?, Implements?)>
<!ATTLIST Declaration Name ID #REQUIRED>
<!ATTLIST Declaration Owner CDATA #IMPLIED>
<!ATTLIST Declaration Type (Requirement | Subsystem | File) #IMPLIED>
<!ATTLIST Declaration Filepath CDATA #IMPLIED>
<!ELEMENT Mapping (Title, Motivation?, Source+, Target+)>
<!ATTLIST Mapping Name ID #REQUIRED>
<!ATTLIST Mapping Owner CDATA #IMPLIED>
<!ELEMENT Implementation (Title, Motivation?, Source+, Target+)>
<!ATTLIST Implementation Name ID #REQUIRED>
<!ATTLIST Implementation Owner CDATA #IMPLIED>
<!ELEMENT Implements (Title, Motivation?, Source+)>
<!ATTLIST Implements Name ID #REQUIRED>
<!ATTLIST Implements Owner CDATA #IMPLIED>
<!ELEMENT Target ()>
<!ATTLIST Target Name IDREF #REQUIRED>
<!ATTLIST Target Type (Requirement | Subsystem | File) #IMPLIED>
<!ELEMENT Source ()>
<!ATTLIST Source Name IDREF #REQUIRED>
<!ELEMENT Title (#CDATA)>
<!ELEMENT Motivation (#CDATA)>
<!ELEMENT Description (#CDATA)>
```

Figure B.1 - The DTD of the repository

To understand the DTD notations a short explanation is given in the following summary [6]:

- *!ELEMENT*: represent elements which are the main building blocks of XML;
- *!ATTLIST*: represents attributes which provide extra information about elements;
- *#CDATA*: also means character data and is text that will not be parsed by a parser;
- *?*: the element is optional or at most one element may exist;
- +: there is at least one element present;
- *: there are zero or more elements allowed;
- *#REQUIRED*: the attribute is required;
- *#IMPLIED*: the attribute is optional;
- *CDATA*: also means character data and is text that will not be parsed by a parser;
- *ID*: Unique ID data;
- *IDREF*: ID of another element.

Appendix C. EXAMPLE CASE FISCAL TAX

In Figure C.1 is the example case Fiscal Tax represented in the repository. All data found in the official Fiscal Tax documentation is entered by hand in the repository. This is done to check if all basic actions of the design process can be executed in the repository.


```
<?xml version='1.0' ?>
<!DOCTYPE System SYSTEM "file://./Forest.dtd">
<System Name="SY_Fiscal_Tax">
  <Title>Fiscal Tax</Title>
  <Motivation>The fiscal tax module is a free service to Bus Lease Netherlands drivers and fleet managers for
requesting the catalogue value of their lease car for tax purposes.</Motivation>
  <Subsystem Name="SS_Software">
    <Title>Software</Title>
    <Declaration Name="SS_Input_Page">
      <Title>Input page</Title>
      <Motivation>Page for the user input</Motivation>
      <Implements Name="IM_Input_Page">
        <Title>Input page implementation</Title>
        <Motivation>The implementation of the input screen requirement</Motivation>
        <Source Name="RE_Input_Screen" />
      </Implements>
    </Declaration>
    <Declaration Name="SS_Success_Page">
      <Title>Success page</Title>
      <Motivation>The page that confirms the success of the tax value abstraction.</Motivation>
      <Implements Name="IM_Success_Page">
        <Title>Success page implementation</Title>
        <Motivation>The implementation of the confirmation screen requirement</Motivation>
        <Source Name="RE_Confimation_Screen" />
      </Implements>
    </Declaration>
    <Declaration Name="SS_Error_Page">
      <Title>Error page</Title>
      <Motivation>The error page when a license plate is not found</Motivation>
      <Implements Name="IM_Error_Page">
        <Title>Error page implementation</Title>
        <Motivation>The implementation of the error screen requirement</Motivation>
        <Source Name="RE_Error_Screen" />
      </Implements>
    </Declaration>
    <Declaration Name="SS_Configuration_File">
      <Title>Configuration File</Title>
      <Motivation>The configuration file witch is used by the different screens</Motivation>
      <Implements Name="IM_Error_Page">
```

```
      <Title>Error page implementation</Title>
      <Motivation>The implementation of the error screen requirement</Motivation>
      <Source Name=“RE_Error_Screen” />
    </Implements>
  </Declaration>
  <Requirement Name=“RE_Configuration_File_Dependence”>
    <Title>Configuration File Dependecy</Title>
    <Description>The different screens depend on the configuration file. This is needed form showing the
right content</Description>
  </Requirement>
  <Mapping Name=“MA_Configuration_File”>
    <Title>Configuration File</Title>
    <Motivation>Mapping the configuration file requirement to the screens</Motivation>
    <Source Name=“RE_Configuration_File_Dependence” />
    <Target Name=“SS_Success_Page” />
    <Target Name=“SS_Error_Page” />
    <Target Name=“SS_Input_Page” />
  </Mapping>
  <Mapping Name=“MA_Go_Back”>
    <Title>Mapping of Go Back</Title>
    <Source Name=“RE_Go_Back” />
    <Target Name=“SS_Error_Page” />
    <Target Name=“SS_Success_Page” />
  </Mapping>
  <Mapping Name=“MA_Error”>
    <Title>Mapping of Error</Title>
    <Source Name=“RE_Error_Screen” />
    <Target Name=“SS_Error_Page” />
  </Mapping>
  <Mapping Name=“MA_Confirmation”>
    <Title>Mapping of Confirmation</Title>
    <Source Name=“RE_Confirmation_Screen” />
    <Source Name=“RE_Configurable_Emails” />
    <Target Name=“SS_Success_Page” />
  </Mapping>
  <Mapping Name=“MA_Input”>
    <Title>Mapping of input</Title>
    <Source Name=“RE_Input_Screen” />
    <Target Name=“SS_Input_Page” />
  </Mapping>
 </Subsystem>
 <Subsystem Name=“SS_Infrastructure”>
  <Title>Infrastructure</Title>
  <Declaration Name=“SS_Internet” Type=“Subsystem”>
    <Title>Internet</Title>
    <Motivation>The internet connection for this project</Motivation>
    <Implements Name=“IM_Internet”>
```

```
      <Title>Implementation of internet</Title>
      <Source Name=“RE_Interenet_Connection” />
    </Implements>
  </Declaration>
  <Declaration Name=“SS_Webserver” Type=“Subsystem”>
    <Title>Webserver</Title>
    <Motivation>The web server of this project</Motivation>
    <Implements Name=“IM_Webserver”>
      <Title>Implementation of web server</Title>
      <Source Name=“RE_Webserver” />
    </Implements>
  </Declaration>
  <Declaration Name=“SS_HTTPserver” Type=“Subsystem”>
    <Title>HTTPserver</Title>
    <Motivation>The HTTPserver for this project</Motivation>
    <Implements Name=“IM_HTTPserver”>
      <Title>Implementation of HTTP server</Title>
      <Source Name=“RE_HTTPserver” />
    </Implements>
  </Declaration>
  <Declaration Name=“SS_Databaseserver” Type=“Subsystem”>
    <Title>Databaseserver</Title>
    <Motivation>The database server for this project</Motivation>
    <Implements Name=“IM_Databaseserver”>
      <Title>Implementation of database server</Title>
      <Source Name=“RE_Databaseserver” />
    </Implements>
  </Declaration>
  <Declaration Name=“SS_Emailserver” Type=“Subsystem” Owner=“Designer”>
    <Title>Emailserver</Title>
    <Motivation>The EmailPserver for this project</Motivation>
    <Implements Name=“IM_Emailserver”>
      <Title>Implementation of email server</Title>
      <Source Name=“RE_Emailserver” />
    </Implements>
  </Declaration>
  <Requirement Name=“RE_Interenet_Connection” Owner=“Designer”>
    <Title>Internet connection</Title>
    <Description>The system shall have a internet connection</Description>
  </Requirement>
  <Requirement Name=“RE_Webserver” Owner=“Designer”>
    <Title>Webserver</Title>
    <Description>The system shall have a web server</Description>
  </Requirement>
  <Requirement Name=“RE_HTTPserver” Owner=“Designer”>
    <Title>HTTPserver</Title>
    <Description>The system shall have a HTTP server</Description>
```

```
    </Requirement>
    <Requirement Name=“RE_Databaseserver” Owner=“Designer”>
      <Title>Databaseserver</Title>
      <Description>The system shall have a database server</Description>
    </Requirement>
    <Requirement Name=“RE_Emailserver” Owner=“Designer”>
      <Title>Emailserver</Title>
      <Description>The system shall have a email server</Description>
    </Requirement>
    <Mapping Name=“MA_Webserver_Internet_Dependency”>
      <Title>Mapping of webserver and internet dependency</Title>
      <Source Name=“RE_Interenet_Connection” />
      <Target Name=“SS_Webserver” />
    </Mapping>
    <Mapping Name=“MA_Server_Webserver_Dependency”>
      <Title>Mapping of server and internet dependency</Title>
      <Source Name=“RE_HTTPserver” />
      <Source Name=“RE_Databaseserver” />
      <Source Name=“RE_Emailserver” />
      <Target Name=“SS_Webserver” />
    </Mapping>
    <Mapping Name=“MA_Infrastructure_Requirements_Refinement”>
      <Title>Infrastrcuture requirements refinement</Title>
      <Motivation>The requirement of the parent Subsystem is refined in new requirements</Motivation>
      <Source Name=“RE_Infrastrcuture_Part” />
      <Target Name=“RE_Interenet_Connection” Type=“Requirement” />
      <Target Name=“RE_Webserver” Type=“Requirement” />
      <Target Name=“RE_HTTPserver” Type=“Requirement” />
      <Target Name=“RE_Databaseserver” Type=“Requirement” />
      <Target Name=“RE_Emailserver” Type=“Requirement” />
    </Mapping>
  </Subsystem>
  <Subsystem Name=“SS_Documentation” Owner=”Desginer”>
    <Title>Documentation</Title>
    <Motivation>The documentation of the system is placed here</Motivation>
    <Declaration Name=“FI_Requirements_Document” Type=“File” Filepath=“./Req.doc”>
      <Title>Requirements documentation</Title>
      <Motivation>The Requirements documentation of this project</Motivation>
      <Implements Name=“IM_Requirements_Document”>
        <Title>Implementation of req doc</Title>
        <Source Name=“RE_Requirement_Document” />
      </Implements>
    </Declaration>
    <Declaration Name=“FI_Architecture_document” Type=“File” Filepath=“./Sa.doc”>
      <Title>System architecture documentation</Title>
      <Motivation>The system architecture documentation of this project</Motivation>
      <Implements Name=“IM_SA_Document”>
```

```
      <Title>System Archtecture Document</Title>
      <Source Name="RE_System_Archtecture_Document" />
    </Implements>
  </Declaration>
  <Declaration Name="FI_Database_Document" Type="File" Filepath="./Dbdd.doc">
    <Title>Database documentation</Title>
    <Motivation>The database documentation of this project</Motivation>
    <Implements Name="IM_DB_document">
      <Title>Implementation of DB doc</Title>
      <Source Name="RE_Database_Document" />
    </Implements>
  </Declaration>
  <Declaration Name="FI_Test_Document" Type="File" Filepath="./Test.doc">
    <Title>Test documentation</Title>
    <Motivation>The test documentation of this project</Motivation>
    <Implements Name="IM_Test_Document">
      <Title>Implementation of test doc</Title>
      <Source Name="RE_Test_Documentation" />
    </Implements>
  </Declaration>
  <Declaration Name="FI_User_Document" Type="File" Filepath="./User.doc">
    <Title>User documentation</Title>
    <Motivation>The user documentation of this project</Motivation>
    <Implements Name="IM_User_Document">
      <Title>Implementation of user doc</Title>
      <Source Name="RE_User_Documentation" />
    </Implements>
  </Declaration>
  <Requirement Name="RE_Requirement_Document" Owner="Designer">
    <Title>Requirements documentation</Title>
    <Description>The system shall contain a requirements document</Description>
  </Requirement>
  <Requirement Name="RE_System_Archtecture_Document" Owner="Designer">
    <Title>System architecture documentation</Title>
    <Description>The system shall contain a system documentation</Description>
  </Requirement>
  <Requirement Name="RE_Database_Document" Owner="Designer">
    <Title>Database documentation</Title>
    <Description>The system shall contain a database documentation</Description>
  </Requirement>
  <Requirement Name="RE_Test_Documentation" Owner="Designer">
    <Title>Test documentation</Title>
    <Description>The system shall contain a test documentation</Description>
  </Requirement>
  <Requirement Name="RE_User_Documentation" Owner="Designer">
    <Title>User documentation</Title>
    <Description>The system shall contain a user documentation</Description>
```

```
    </Requirement>
  </Subsystem>
  <Requirement Name="RE_Configurable_Screens" Owner="User">
    <Title>Configurable</Title>
    <Description>All text in the screens are configurable.</Description>
  </Requirement>
  <Requirement Name="RE_Configurable_Emails" Owner="User">
    <Title>Configurable emails</Title>
    <Description>All text in the email messages are configurable.</Description>
  </Requirement>
  <Requirement Name="RE_Input_Screen" Owner="User">
    <Title>Input screen</Title>
    <Description>The input screen should look like figure 2.</Description>
    <Requirement Name="RE_User_Input_Licence_Plate" Owner="User">
      <Title>User licence plate input</Title>
      <Description>In order to select the catalogue value, the user has to enter the licence
plate.</Description>
    </Requirement>
    <Requirement Name="RE_User_Input_Email_Address" Owner="User">
      <Title>User email address input</Title>
      <Description>In order to select the catalogue value, the user has to enter his email
address.</Description>
    </Requirement>
    <Requirement Name="RE_User_Input_Present_Driver" Owner="User">
      <Title>User is present driver</Title>
      <Description>In order to select the catalogue value, the user has to indicate hy is the current driver or
not.</Description>
    </Requirement>
    <Requirement Name="RE_User_Input_Store_Email" Owner="User">
      <Title>Allow storing user email</Title>
      <Description>In order to select the catalogue value, the user has to indicate ING is allowed to store his
email.</Description>
    </Requirement>
    <Requirement Name="RE_Store_User_Email" Owner="User">
      <Title>Store user email in DB</Title>
      <Description>If the user didn't object to ING adding the e-mail address to the address information and
the user is the current driver of the car, add the e-mail to database.</Description>
    </Requirement>
  </Requirement>
  <Requirement Name="RE_Confirmation_Screen" Owner="User">
    <Title>Confirmation screen</Title>
    <Description>The confirmation screen should look like figure 3.</Description>
    <Requirement Name="RE_Send_Email" Owner="User">
      <Title>Send Email</Title>
      <Description>the system shall send een email with value information. the email should look like figure
4.</Description>
      <Requirement Name="RE_Two_Different_Emails" Owner="User">
```

```
        <Title>Two different emails</Title>
        <Description>The system shall support two different types of emails. A one tax value email and a two tax value email.</Description>
      </Requirement>
    </Requirement>
  </Requirement>
  <Requirement Name="RE_Error_Screen" Owner="User">
    <Title>Error screen</Title>
    <Description>The error screen shall look like figure 6.</Description>
    <Requirement Name="RE_Show_Detailed_Error" Owner="User">
      <Title>Show detailed error message</Title>
      <Description>Upon entering incorrect data, user is redirected to an error page which shows a detailed error description.</Description>
    </Requirement>
  </Requirement>
  <Requirement Name="RE_Go_Back" Owner="User">
    <Title>Go Back</Title>
    <Description>The system shall have a "go back" link on the confirmation and erro page to go back to the input screen</Description>
  </Requirement>
  <Requirement Name="RE_Documentation_Part" Owner="Designer">
    <Title>Documentation</Title>
    <Description>The system shall contain a documentation</Description>
  </Requirement>
  <Requirement Name="RE_Software_Part" Owner="Designer">
    <Title>Documentation</Title>
    <Description>The system shall contain a software part</Description>
  </Requirement>
  <Requirement Name="RE_Infrastrcuture_Part" Owner="Designer">
    <Title>Documentation</Title>
    <Description>The system shall needs infrastructure to function</Description>
  </Requirement>
  <Requirement Name="RE_Software_Infrastrcuture_Dependency" Owner="Designer">
    <Title>Software infrastructure dependency</Title>
    <Description>The Software its execution depends on the infrastructure.</Description>
  </Requirement>
  <Implementation Name="IM_Software_Part" >
    <Title>Implementation of Software implementation"</Title>
    <Source Name="RE_Software_Part" />
    <Source Name="RE_Software_Infrastrcuture_Dependency" />
    <Target Name="SS_Software" />
  </Implementation>
  <Implementation Name="IM_Software_Infra_Dep" >
    <Title>Implementation of Software_Infrastrcuture_Dependency</Title>
    <Source Name="RE_Software_Infrastrcuture_Dependency" />
    <Target Name="SS_Infrastructure" />
  </Implementation>
```

```
  <Implementation Name="IM_Doc_Part">
    <Title>Implementation of Documentation Part</Title>
    <Source Name="RE_Documentation_Part" />
    <Target Name="SS_Documentation" />
  </Implementation>
  <Mapping Name="MA_Software">
    <Title>Software requirements mapping</Title>
    <Motivation>All requirements are software related</Motivation>
    <Source Name="RE_Configurable_Screens" />
    <Source Name="RE_Configurable_Emails" />
    <Source Name="RE_Input_Screen" />
    <Source Name="RE_Confirmation_Screen" />
    <Source Name="RE_Error_Screen" />
    <Source Name="RE_Go_Back" />
    <Target Name="SS_Software" />
  </Mapping>
  <Mapping Name="MA_Infrastrcuture">
    <Title>Infrastructure requirements mapping</Title>
    <Motivation>The infrastructure requirement need to be related</Motivation>
    <Source Name="RE_Infrastrcuture_Part" />
    <Target Name="SS_Infrastructure" />
  </Mapping>
</System>
```

Figure C.1 – Example case