Generating system documentation augmented with traceability information, using a central XML-based repository

Master’s Thesis

How the customer explained it
How the Project Leader understood it
How the Analyst designed it
How the Programmer wrote it
How the Business Consultant described it
How the project was documented
What operations were installed
How the customer was billed
How it was supported
What the customer really needed

Thomas U. Kraus
Generating system documentation augmented with traceability information, using a central XML-based repository

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

SOFTWARE ENGINEERING

by

Thomas U. Kraus
born in Geldrop, The Netherlands

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

Chess, iBusiness
Nieuwe Gracht 13
Haarlem, The Netherlands
http://www.chess.nl
Cover picture:
“Project swing”: simple requirements transform disastrously without proper documentation.

This picture, as well as several variants, can be found all over the Internet and no original author is known. In fact, in 1978 a predecessor of the picture was already part of American urban folklore. A description and three variants of the picture can be found in [6].

Since the code name of this graduation project is “Forest” and the main goal is to evaluate a method for better creating and maintaining documentation, this picture was found to be particularly suitable as a cover picture by the author.
Generating system documentation augmented with traceability information, using a central XML-based repository

Author: Thomas U. Kraus
Student id: 1041754
Email: t.u.kraus@student.tudelft.nl

Abstract

Many companies have difficulties with creating and maintaining proper documentation of the systems they build. In my Master’s graduation project, I investigated a new way of creating and maintaining documentation of any type of system. The project rests on two pillars: generating documentation from a central repository on one hand, and adding traceability support on the other. Since the project took place at the software development department of Chess, for validation, the approach was applied to creating software systems.

The prototype that was created and validated, consists of a plug-in for the Eclipse IDE, and a command line application. The installation of the plug-in enriches the Eclipse environment with functionality for creation and maintenance of documentation, as well as several types of analyses based on traceability of system artefacts. The command line application can be invoked from an automatic build script to generate documentation from, or perform analyses on the repository.

Thesis Committee:

Chair: Prof. dr. A. van Deursen, Faculty of EEMCS, TU Delft
University supervisor: Dr. H.G. Gross, Faculty of EEMCS, TU Delft
Company supervisor: M. Fillerup, Chess, ibusiness
Committee member: Ir. M. Hajian, Faculty of IDE, TU Delft
Preface

This Master’s Thesis describes my graduation project, which concludes my study of Computer Science at Delft Technical University. The graduation project was done at Chess in Haarlem, a leading company in the development and production of ICT innovations in home electronics, micro electronics, embedded software, and ICT applications.

A few years ago, my supervisor Mats Fillerup first thought of storing system documentation artefacts in a structured manner. As a hobby project, in evenings and weekends some testing and prototyping was done. By the end of 2005 a Master student from Twente University came to Chess to do his graduation project, who investigated the repository used for storing documentation artefacts. After that, I started with my graduation project, to investigate the whole chain of entering information in the repository, checking and tracing this information, and generating documentation.

Several people were involved in my graduation project, both in Delft and in Haarlem. First, I would like to thank dr. Hans-Gerhard Gross, my TU Delft supervisor, for his academic guidance and valuable comments. Second, I would like to thank Mats Fillerup, my direct supervisor at Chess, and original inventor of the Forest project. Next, thanks to dr. Joris Portegies Zwart, and drs. Jeroen Koomen, for their valuable input and contributions to the project. Joris acted as a co-supervisor and is the author of the document generator I used in my prototype. Jeroen created an earlier version of the prototype and participated in many discussions and meetings. Furthermore, thanks to Annegien Coenen, with whom I had the first contact about doing a graduation project at Chess, for quickly responding to my request with this interesting assignment. I would like to thank all people at Chess for their enthusiasm which made my stay very enjoyable and fruitful. Last, but certainly not least, I would like to thank my family for always being very supportive during my entire education.

Thomas U. Kraus
Delft, The Netherlands
June 26, 2007
Contents

Preface iii

Contents v

List of Figures vii

1 Introduction 1
   1.1 Context ................................................. 1
   1.2 The proposed solution: Forest .......................... 3
   1.3 Scope of this graduation project ....................... 5
   1.4 Research Questions ...................................... 6
   1.5 Summary ................................................. 7

2 Related Work 9
   2.1 Alternative software usable for system documentation ................................................. 9
   2.2 Scientific research ........................................ 10
   2.3 Summary ................................................. 11

3 Forest Concepts 13
   3.1 Forest concepts at a glance .................................. 13
   3.2 Forest repository concepts .................................. 14
   3.3 Additional elements .......................................... 18
   3.4 Changes with respect to previous versions .................. 20
   3.5 Summary ................................................. 21

4 Using Forest in practice 23
   4.1 Overview ................................................. 23
   4.2 Central repository .......................................... 23
   4.3 Entering information into the repository ..................... 24
   4.4 Checking for completeness and consistency .................. 24
   4.5 Generating documentation .................................... 25
## CONTENTS

4.6 Redocumenting existing systems ........................................... 26
4.7 Summary .............................................................................. 27

5 Forest tools ....................................................................... 29
5.1 Forest repository ................................................................. 29
5.2 Editors ............................................................................... 29
5.3 Model checker ................................................................. 30
5.4 Documentation generator ...................................................... 30
5.5 IDE integration ................................................................. 31
5.6 Summary .............................................................................. 31

6 Case Studies ..................................................................... 33
6.1 The first experiment: bootstrapping ........................................ 33
6.2 The second experiment: a ticketing system ......................... 38
6.3 Summary .............................................................................. 47

7 Summary, Conclusions, and Future Work ......................... 49
7.1 Summary .............................................................................. 49
7.2 Conclusions ....................................................................... 50
7.3 Future work ...................................................................... 51

Bibliography ....................................................................... 55

Glossary ............................................................................... 57
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>The position of the Forest documentation approach: in between writing free</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>text in MS Word and the use of formal languages</td>
<td></td>
</tr>
<tr>
<td>3.2</td>
<td>A system extending another, abstract system</td>
<td>16</td>
</tr>
<tr>
<td>3.3</td>
<td>A simple repository structure</td>
<td>17</td>
</tr>
<tr>
<td>3.4</td>
<td>A more extensive repository structure</td>
<td>18</td>
</tr>
<tr>
<td>6.1</td>
<td>High-level requirements and design of the Forest prototype</td>
<td>35</td>
</tr>
<tr>
<td>6.2</td>
<td>A problem found by the model checker</td>
<td>36</td>
</tr>
<tr>
<td>6.3</td>
<td>A high-level overview of the Theatre ticketing system’s requirements</td>
<td>41</td>
</tr>
<tr>
<td>6.4</td>
<td>By means of delegations, the top-level requirements of the Theatre ticketing</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>system are delegated to subsystems</td>
<td></td>
</tr>
<tr>
<td>6.5</td>
<td>The subscription process elaborated in terms of requirements, subsystems and</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>atoms</td>
<td></td>
</tr>
<tr>
<td>6.6</td>
<td>A trace of a translated requirement</td>
<td>44</td>
</tr>
<tr>
<td>6.7</td>
<td>A full trace of a requirement to its implementation</td>
<td>45</td>
</tr>
<tr>
<td>6.8</td>
<td>A full trace of a split requirement to its implementations</td>
<td>46</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This chapter introduces all aspects that form the context and motivations of my graduation project. First, the context of the project is introduced, followed by the proposed alternative for generating system documentation. Since this proposal involves a project that is too large for a single Master’s graduation project, a scope section defines the boundaries of my graduation project. Finally, a number of research questions is stated, which are answered in the following chapters of this thesis.

Chapter 2 describes related work: a number of existing software application I looked at as well as related scientific research. In Chapter 3, an overview of the key concepts of the proposed solution is presented. Chapter 4 explains the method to use these concepts in practice and in Chapter 5 an overview is given of the tools needed when working according to this method. See the glossary at the end of this thesis for definitions of terms and abbreviations used in this thesis. In this thesis I will use an *italics font style* to introduce definitions and a `code font style` to indicate the use of keywords.

1.1 Context

Many companies, including Chess, the company that hosted this graduation project, are facing difficulties with creation and management of documentation. During my literature study, I found that it is important to properly document all work that is done for projects, throughout the entire lifecycle of the respective system [12]. Without proper documentation, a number of problems can occur, which I will illustrate in this section.

First of all, understanding a system (for development, maintenance, assessment, etc.) without having any (proper) documentation about this system may take quite some time. Especially the rationale behind the implementation of the system can take a while to figure out. For example, in a software system, figuring out the rationale is not simply a matter of reading the source code, but understanding why the source code is the way it is. In a commercial setting, it is often desired to keep the time needed for a project to a minimum. Having proper and up to date documentation, in which the respective stakeholder can simply read the rationale instead of having to try to reconstruct it by looking at implementation details, could help in reducing the project’s time.
Second, when changes need to be made to a (complex) system, and the architecture of this system is not (well) known, the structure of the system will slowly start to degrade. This is also known as architectural erosion [17]. Maintenance costs will get higher while the architecture degrades further, until the moment comes that stakeholders decide to rebuild or replace the system [16]. Good documentation about the architecture of the system could help in making the right decisions when extending or changing the system. For example, when making changes to a software system, a developer could decide to refactor [15] part of the system to keep the architecture solid.

Third, when working on a system with a medium-sized or large team, creating proper documentation allows for efficient communication. For example, changes to be made in the requirements or design of a system can be told to a few people, but when working with large development teams, it is more efficient to write the changes in the documentation and have everybody read about these changes. This way, a common understanding of the system is being created [21]. Next to these more general problems, several more specific problems with current documentation, resulting from the development processes at Chess and its customers (Shell, ING, Rabobank, etc.), were identified as well:

**Great number of loosely coupled documents:** For each project, numerous documents need to be created. Currently, there is no frame holding all these documents together, and hence no alarm bells go off if a document is accidentally removed. For new projects, there is no indication that a certain type of document should be created. Furthermore, the consistency inside each of these documents, e.g. cross references, is currently not guaranteed either.

**No consistency checking of contents:** No automated checking possibilities of any kind are available to verify that the contents of a document make sense. To name a few consequences: Too many or too few requirements may be listed, e.g. when using an old version of a requirements document. References to artefacts may exist in a document, while the artefacts itself are no longer present, or not present yet. A requirement may be visible in the design but not in the implementation.

**Only manual reuse possible:** When doing several projects for the same customer, or in the same domain, it is common practice to reuse a set of requirements, a part of the design, or even a part of the implementation. Currently, reusing parts of documents can be done only by manually copy/pasting the desired information into a new document.

**Static documents:** Current documents are static, no possibilities exist for creating another view on the information described in the document, other than manually creating a new document and copy/pasting the desired information to this new document.

### 1.1.1 Requirements traceability

The problem of not having any consistency checking, as mentioned above, can be rephrased as the lack of requirements traceability. Traceability can be defined as “the degree to which
Introduction

1.2 The proposed solution: Forest

A relationship can be established between two or more products of the development process...” [11]. It is about understanding how requirements transform (e.g. business requirements into functional requirements) and how the design and implementation are based on these requirements. A software requirements specification is traceable if
(i) the origin of each of its requirements is clear, and if
(ii) it facilitates the referencing of each requirement in future development or enhancement documentation [9].

Requirements traceability, if available, allows for several types of analysis, namely coverage analysis, derivation analysis, and impact analysis [8]. These analyses can be useful to a variety of stakeholders, as explained in [11]. Coverage analysis can be used to verify that all requirements are met, i.e. each requirement can be traced to an implementation. Derivation analysis, sometimes called cost/benefit analysis, is done in the opposite direction. It can be used to verify that each implementation artefact is referenced by at least one requirement. If this is not the case for a particular implementation artefact, the costs (for creating the implementation) were made, but it yields no benefits (since no requirement was found that expresses a stakeholders interest). Finally, impact analysis can be used in so-called change management. The goal of an impact analysis is to indicate which part of a system’s architecture, design, or implementation is affected by a change in the requirements. Typically, modifying requirements will show that existing implementation artefacts should be changed or that new implementation artefacts are needed. These types of analysis can support project management (especially when automated) and automated requirements traceability is therefore a desired feature in system documentation.

1.2 The proposed solution: Forest

The Forest project proposes a new method of creating and maintaining documentation. This method requires a well-defined repository as well as a set of tools to store all information in a structured, and traceable manner. The key concepts of Forest and the repository are explained in Chapter 3. More information on the method to use these concepts and the required tools to follow this method can be found in Chapters 4 and 5, respectively.

The name Forest originated from the great number of trees involved in the project. The repository contains information in a structured manner, which can be represented as various trees (e.g. a requirements tree, a tree of design artefacts, etc.) and with incomplete or outdated documentation one cannot see the woods for the trees anymore.

In order to keep the threshold for introducing the Forest method as low as possible, the currently proposed repository concepts are kept very simple. However, care has been taken that the concepts are extensible. Based on problems with current documentation and experience with existing tooling for managing system information artefacts, a number of requirements to the proposed solution were formulated, which are explained next.

1.2.1 Requirements to the solution

The following requirements should make sure that all the individual problems that exist with current documentation, as listed in the previous section, will eventually be solved.
Central repository: All information is to be stored in a repository, that is integrated with the location of the project’s source files. This way, it is possible to verify references to implementation artefacts in the file system. Furthermore, the repository acts as a frame holding all information together. This should improve the coupling and consistency between documents.

Store all types of information during the system’s entire lifecycle: Some existing tools only focus on one phase of the lifecycle, e.g. requirements management or testing. We aim at storing all types of information, to be able to trace requirements in a complete way and to improve consistency.

Automated consistency checking and requirements traceability: A tool should be available that can perform a number of consistency checks automatically. Checks that should at least be available, are:

- Each requirement should be implemented somewhere,
- All implementation artefacts need a least one requirement to explain their presence,
- For external references to the file system, the actual existence of the files should be verified.

Support for reuse: Artefacts that might be reused, like a set of requirements or a (sub)system designed for common problems, should be defined abstractly in reusable systems. Reuse now simply becomes a matter of extending these systems, and it should be verified automatically that all local and all inherited requirements are implemented. Furthermore, a concrete system description should be available for every inherited abstract system definition. Sets of domain-specific requirements, standards, or patterns can now be defined once and reused many times.

Generation of dynamic, personalized documentation: A user should be able to generate documentation specific to his role. For example, a requirements engineer will typically be interested in generating requirements documentation. Furthermore, a user should be able to select the part of the system of which documentation should be generated, as well as the desired level of abstraction. An architect is typically interested in the design of an entire system, at a high level of abstraction; whereas a developer is more interested in a small part of the system, but including all the details. This solves the problem of having static documents providing only a single view on the information.

Document any system: Many systems are limited to documenting software systems only. Since this can be a limitation when creating e.g. embedded systems, the proposed method should allow to describe anything that can be expressed as a “system”, not only software systems. A commonly used example of a “system” to describe is a business hierarchy (i.e. a company with a board of directors, several business led by a manager, employees in each
business unit, etc).

**Easy to use:** The user should be able to fill in as much or as little information as he likes. There should be no blocking error messages or dialog screens that enforce the user to enter more information or invest more time at a certain moment.

**Low threshold to start using Forest:** It should be very little work to define the structure of a system in terms of subsystems. Combined with the previous requirement, the user should be free to explore the possibilities and use as much or as little of the available possibilities as desired.

**Integrated in development environment:** We expect an advantage in efficiency when documentation is well-integrated in the development environment. A developer should be able to find the desired documentation, or the actual implementation of a requirement, faster than in the current situation, by means of additional navigation between implementation and documentation.

**Separate tools and content:** By storing the information artefacts and the relations between them in an open and readable format, e.g. an XML format, a Forest repository is independent of specific tools for viewing or modifications. “Anyone” should be able to build a viewer or editor on top of such an XML-based repository.

**Open and extendable:** Anyone should be able to build tools and extensions for a Forest repository, and the format should still be readable in the future.

**Not intended for code generation:** The Forest project aims at documenting systems, which is in itself already quite a complex task. In order to generate code for a complex system, that system must first be fully and properly specified. Today, this is still a difficulty so this is our primary goal. Perhaps code generation becomes of interest in a later phase of the Forest project.

### 1.3 Scope of this graduation project

The solution as proposed in the previous section is too large for a single Master’s graduation project. The goal of my graduation project is to verify that the proposal can actually be used for documenting systems that are developed in a business environment. My graduation project, defined as such, consists of a number of subprojects.

These subprojects involve an editor to enter and modify information, a repository to enter the information into, a model checker, and a document generator. For the proof-of-concept, I did not investigate and elaborate each of these subprojects to the fullest extent, but rather combined all these subprojects to provide insight in the proposed solution as a whole. See section 7.3 for more information on what still needs to be investigated in the Forest project.
Although it might be seen as an implementation detail, during this graduation project we assumed that the Forest repository would be stored in XML files. See section A.2.2 for the reasons behind this assumption.

1.4 Research Questions

In order to verify that the proposed solution is indeed a solution to the problem identified in section 1.1 (taking into account the scope of my graduation project as explained in the previous section), we defined number of research questions, which are explained next. In order to answer these questions, I have built a prototype implementation (see Appendix A). In the following chapters I will present my answers to these questions and in Chapter 7 I will draw conclusions about about the Forest method in general as well as the underlying concepts.

What adjustments to the previous proposal of the Forest repository concepts are necessary, and why?
The “previous” proposal is a combination of the results of earlier research in this project by Mats Fillerup and Joris Portegies Zwart and the graduation project of a student of Twente University [20]. Unfortunately, the previous proposal has never been implemented, and therefore the concept was not verified. After several discussions, we agreed that a number of details of the specification needed to be changed. Several small additions were made as well. Section 3.2 describes the key concepts of the current Forest repository and section 3.4 explains the changes with respect to previous versions of the repository concepts.

Can the proposed method be used to generate documentation that is comparable to the current, manually created, MS Word documents?
Documentation that is being created using the newly proposed method, should be of a quality that is at least comparable to the current documentation. That is, a stakeholder should be able to express all requirements, architecture, design, and test case artefacts, such that they can be stored and used as contents when generating documentation. Furthermore, the look-and-feel of the generated documentation should resemble the documentation that currently exists. In Chapter 6 several case studies are described, in which documentation is generated.

How to merge system traceability information with automatically generated documentation?
There are at least two views on this question: a software engineering view and a documentation engineering view. The latter is most important, since the information in the document that is being generated, should be understandable. That is, the added traceability information should not make the system description harder to read, yet it should add comprehensive

---

1 XML: the eXtensible Markup Language, is used to add structure (the markup) to the contents of files. For more information, see: http://www.w3.org/XML/
2 A commonly used term to refer to graphical layout, i.e. fonts, colours, headers and footers, etc.
information about the state of the system.

**Does automated requirements traceability have the expected advantages?**

As explained previously in this chapter, the automated requirements traceability feature is expected to have a number of advantages:

- Perform coverage analysis (completeness of implementation)
- Perform derivation analysis (cost/benefits)
- Perform impact analysis (change management)
- Verify that all related artefacts, documents, etc. indeed exist
- Quickly navigate to the documentation of an implementation artefact
- Quickly navigate to the implementation corresponding to a part of the documentation

Chapter 6 presents two case studies in which these questions are answered.

**When applied to software engineering, will this new documentation method become an integral part of software development like unit testing has become?**

Although this is a long-term question, it is expected that a case study can yield a preliminary answer.

### 1.5 Summary

This introductory chapter started with an explanation of the context of my graduation project, after which a proposed alternative for generating system documentation was introduced. Since this proposed alternative involves a project too large for a single Master’s graduation project, I outlined the boundaries of my graduation project in the Scope section. Finally, the research questions that will be answered in the rest of this thesis are posed along with a short explanation.
Chapter 2

Related Work

Creating documentation of large systems is important, not only in computer science. Adding requirements traceability allows for several useful types of analysis. This chapter discusses technologies related to system documentation and requirements traceability. It is divided into two sections: alternative software systems that can be used in the process of creating and maintaining system documentation, and current and past scientific research in this field. The report resulting from the literature study I performed in preparation of my graduation project is included in this thesis as Appendix F.

2.1 Alternative software usable for system documentation

As discussed in my literature study [12], a number of tools and applications already exists but none of the tools I looked at solve all the problems identified in section 1.1. In the previous chapter, these problems were listed.

A summary of the tools examined and their major advantages and disadvantages now follows.

**Enterprise Architect:** A powerful but complex UML-based tool with good documentation output, but primarily aims at Object Oriented systems due to its UML foundation. A licence is required and it does not completely fit into an integrated development environment.

**Requisite Pro:** A requirements management tool that is well-integrated in MS Word. This tool has support for cross-project traceability, but traceability is limited to relations between requirements. Neither architectural nor design artefacts can be entered, nor stakeholders or test descriptions.

**Caliber:** Both Caliber RM and Caliber DefineIT are complete requirements management tools.  

---

1 UML is the Unified Modeling Language, a visual notation for expressing Object Oriented system designs. See http://www.uml.org/ for more information.


tools but do not contain any architectural or design possibilities. Caliber DefineIT requires specifying a complete scenario when adding only a single requirement. The documentation generation features of these tools are limited.

**DOORS:** The Dynamic Object Oriented Requirements System\(^5\) comes with an import/export module for MS Word, which allows for easy adopting of the tool in an existing, MS Word based, documentation environment. Again, design artefact support is not available and the tool features no integration possibilities with an integrated development environment.

**OSRMT:** The Open Source Requirements Management Tool is a compact, intuitive looking tool. It cannot be fully integrated in a development environment and the default documentation output is not useful compared to current, manually created documentation\(^6\).

### 2.2 Scientific research

Research is being done in a number of fields that are closely related to the Forest project. In the Netherlands, the Joint Academic Quality Research and Development (Jacquard) programme hosts several projects in which academic research is done with the goal to increase the Dutch knowledge in the field of software engineering. Several major companies, including Chess, participate in these projects. Some of the Jacquard projects have close relations to the Forest project, so I will briefly describe them here.

To start with, the Quadread\(^7\) project that started recently at Twente University, has quite some overlap with the Forest project. The Quadread project aims at bridging the gap between requirements engineering and architectural design. The Forest project has this goal as well and tries to solve this by means of requirements traceability, annotated with descriptions and motivations. The goals of the Forest project, however, extend to the implementation of a system, meaning that implementation artefacts should be traceable from the requirements and design artefacts. A meeting with researchers in the Quadread project has lead to collaboration with the Forest project: a Master student from Twente University will be doing his graduation project within the Quadread project, using the tools needed for the Forest project as a case study.

Another project of the Jacquard programme is the Griffin\(^8\) project, at the Vrije Universiteit, Amsterdam. The Griffin project aims at documenting the decisions made during software development. The research focuses at the early design stage, to document the rationale behind decisions (the “why”) rather than just the result of decisions. This rationale is captured in a structured manner as explained in [4]. Currently, in the Forest project only

---

\(^5\) [http://www.telelogic.com/Products/doors/](http://www.telelogic.com/Products/doors/)

\(^6\) New versions of OSRMT have been released in the mean time, so its functionality is likely improved. See [http://www.osrmt.com/](http://www.osrmt.com/)

\(^7\) Quality Driven Requirements Engineering and Architectural Design, for more information, look at: [http://quadread.ewi.utwente.nl](http://quadread.ewi.utwente.nl)

\(^8\) Grid for information about architectural knowledge, for more information, look at: [http://griffin.cs.vu.nl](http://griffin.cs.vu.nl)
a description of this rationale can be captured in a motivation artefact, which can be added to design, implementation, and relation artefacts. In this motivation, the rationale can be explained for the respective artefact. As explained in [8], so-called rich traceability can be used to store design decisions as well as alternative solutions that have been tried already, or may be tried if the current solution may eventually not be satisfactory. At this point, collaboration between the Forest project and the Griffin project might be useful to support more rich and structured design rationale.

From another point of view, the Reconstructor project at the Software Engineering department of Delft University of Technology, is also related to the Forest project. The Reconstructor project aims at reconstructing software architectures of existing systems. Problems can occur in case the (architectural) documentation of a system is not synchronized with the system’s implementation, as explained in the problem statement in the previous chapter. An example of these problems is given in [5]. The second experiment described in Chapter 6 shows that the Forest project can aid in redocumenting (the architecture of) an existing system as well.

In the field of requirements traceability, research has been done to automatically reconstruct traceability links from existing project documentation and existing systems. In [13] and [14], latent semantic indexing (LSI) is used to reconstruct a requirements traceability model and to generate requirement coverage views. This technique is applied to several case studies, varying from a lab study to an industrial case study. Related traceability recovery research is presented in [11], where documentation and source code of existing software systems are analysed to recover traceability links. As described in section 7.3.4 it could be very useful to combine the Forest project with the mentioned traceability recovery methods for redocumenting or even reverse engineering existing systems.

2.3 Summary

This chapter started with a discussion of tools, both commercial and open source, that are available for requirements management, storing design, test case management, documentation generation, etc. None of these tools, however, solve all the problems that were identified in the previous chapter. Next, a number of scientific research projects are discussed, and their relation to the Forest project in particular.

9For more information, look at: http://swerl.tudelft.nl/bin/view/Main/ReconstructorProject/
Chapter 3

Forest Concepts

This chapter gives an overview of the key concepts of Forest. Most of these concepts were available but not tested, yet. In the next chapter, I will illustrate a method to use these concepts. When working according to this method a number of tools can be useful, as explained in Chapter 5. I have implemented these tools in a software prototype implementation, see Appendix A.

First, a high-level overview is given of the Forest concepts. Then, the Forest repository concepts are explained. At the end of this chapter, changes with respect to previous versions of the repository concepts are explained.

3.1 Forest concepts at a glance

First, a brief overview of the possibilities and functionalities of the Forest project will be given, after which a number of key concepts are discussed in more detail.

The main goal of the Forest project is to create and maintain documentation of systems. The information that is to appear in this documentation is entered and stored in a structured manner and relations between these information artefacts are stored as well. A variety of checks can now be performed on this structured information, to verify that the information is consistent and complete.

Personalized documentation can now be generated, according to a certain documentation standard, based on the structured information. Personalized documentation here means that that stakeholders in different roles have different views on a system and have different interests. As explained in section 1.2.1 the required level of abstraction also depends on the role of the stakeholder. Typically, an architect is interested in the design of the entire system on a high level of abstraction; whereas to a developer only a small part of the system is relevant, at the lowest level of abstraction.

While generating a document, the results of the checks performed can be used to provide information about the state of the system. For example, requirements that cannot be fully traced to an implementation are visually augmented with a small icon in the margin of the requirements document. Having state information included in the document implies that documents are no longer intended for long-term use. Hence, documents should be genera-
3.2 Forest repository concepts

A Forest repository here means the collection of elements used to store all available information about a system, in a structured manner. This structure is the key concept of the Forest repository, since it defines what metadata\(^1\) is needed to store all information in such a way that parts of the information can be selected for retrieval later. The current repository structure, i.e. the Forest meta model, is based on previous work done by Mats Fillerup and Joris Portegies Zwart as well as a Master’s graduation project by a student from Twente University \[20\].

First, four basic elements are introduced, which can be used to create a simple repository. Then, a number of additional elements are explained giving the repository more functionality.

\(^1\)Meta data is data which describes other data
3.2.1 Requirement

The requirement element represents a requirement that consists of an identifier, title, and a textual description. This element is not only used to express e.g. business or functional requirements to the top-level system artefact but it can also be used to express needs required by design decisions on a lower level subsystem. Hence, an architect or developer responsible for designing a subsystem can introduce new requirements. In fact, a requirement delegated to a subsystem is often implicitly split into many new requirements. A simple reason why this is needed is that the architect of a high-level system artefact often does not know all about the design and implementation details at the lowest levels. It is up to the architect responsible for the new subsystem whether to make this explicit by means of a delegation\(^2\). Tables B.1 and B.2 describe the attributes and subelements of the requirement element, respectively.

3.2.2 System

The system element represents a design artefact. This element is a composite, thus it can contain system elements. Furthermore, most systems will contain a number of requirements as well as a number of atoms\(^3\). Tables B.3 and B.4 describe the attributes and subelements of the system element, respectively.

For reuse, a system containing requirements and/or subsystems can be declared to be abstract. This means that no implementations are expected in this system but the system can be extended (i.e. reusing the abstract declarations). In the extending system, implementations are expected of all requirements including those of the abstract system. Hence, the use of this property is similar to the use of the abstract keyword in Java \([7]\). Furthermore, all subsystems of the abstract system should be present in the extending system. This is visualized intuitively in Figure 3.2. Note that this figure could have been created using the visual UML notation for meta modelling as well. I choose this intuitive drawing method instead since the concepts I want to explain dealing with multiple instances of the same type (e.g. a system) can be seen more clearly from this type of drawings than from an UML diagram. The figure is used as an example rather than as a formal specification. This holds for the following two figures as well.

To avoid having to describe a large system entirely in one repository file, each system element can optionally specify a relative file path at the location attribute. This should be an indication for the parser to look at this new location and obtain the system’s contents from there. Since this is optional, the user is free to decide whether and when to use it. A system’s description can also be moved to a separate file later, when it has become larger than initially expected or to keep synchronized with the implementation after refactoring.

3.2.3 Atom

An atom represents an atomic piece of implementation, which can implement one or more requirements. It is possible to add a reference to a concrete implementation to this atom

---

\(^2\)Explained in subsection 3.2.4

\(^3\)Explained in subsection 3.2.3
Figure 3.2: A system extending another, abstract system.
representation.

Alternatively, an atom can also be used during a design phase to represent a part of the system the architect does not want to elaborate in more detail, but should already take care of the implementation of some requirements. In a later design phase, this atom can be replaced by a system containing more details.

Tables B.5 and B.6 describe the attributes and subelements of the atom element, respectively. Figure 3.3 shows a system that contains requirements and a subsystem containing both requirements and atoms.

![Figure 3.3: A simple repository structure.](image)

### 3.2.4 Delegation

The delegation element can be used to delegate a requirement (the source) to either an atom or a subsystem (the target). When delegating a requirement to a subsystem, one is effectively stating that this subsystem is now responsible for providing an implementation (in the form of an atom) of the delegated requirement.

Alternatively, this element can be used to rewrite a requirement, for example because this requirement is not specific enough. In this case, a requirement is delegated to one (or several) other requirement(s). Tables B.7 and B.8 describe the attributes and subelements of the delegation element, respectively.

In general, these delegation elements represent the relations between artefacts in the repository, which allow for traceability support between these artefacts. As visualized in
3.3 Additional elements

Figure 3.4: A requirement is delegated to an atom which resides in a subsystem. This requirement is traceable to its implementation (as indicated by the set of arrows). The condition for having an atom is satisfied as well since at least one requirement is pointing towards this atom.

Figure 3.4: A more extensive repository structure.

Care should be taken that delegations do not introduce cycles in the model. Since requirements can be delegated to other requirements (e.g. when translating a business requirement to a functional requirement) a simple model checker will not warn when two requirements are translated into each other. For example, requirement $A$ is delegated to requirement $B$ and requirement $B$ is delegated to requirement $A$ again. Obviously, this is not a situation in which the model checker should remain silent. Hence, a cycle-detection check should prevent this in order to ensure consistency.

3.3 Additional elements

Using the following additional elements, the Forest repository can contain more rich information.
3.3.1 Extends

The extends element is used to indicate that the system containing this element and must implement everything that is declared in the extended abstract system. Subsection 3.2.2 and Figure 3.2 explained this notion of extending abstract systems. Table B.9 describes the attributes of the extends element. This element cannot contain any subelements.

3.3.2 Implements

The implements element is a subelement of the atom element and contains a reference to a requirement that is being implemented. Table B.10 describes the attributes of the implements element. This element cannot contain any subelements.

3.3.3 Source

The source element is a subelement of the delegation element and contains a reference to the element that is being delegated, typically a requirement. Table B.11 describes the attributes of the source element. This element cannot contain any subelements.

3.3.4 Target

The target element is a subelement of the delegation element and contains a reference to the element that the respective source is being delegated to, typically a system or an atom. Table B.12 describes the attributes of the target element. This element cannot contain any subelements.

3.3.5 Title

The title element contains a single line of text describing the artefact that contains this title element. The extends element does not contain any attributes and can only contain character data but no subelements.

3.3.6 Description

The description element contains multiple lines of text describing the artefact that contains this description. References to images may be included in this text. The description element does not contain any attributes. The subelements of this element are described in Table B.13.

3.3.7 Motivation

Similar to a description element, the motivation element contains multiple lines of text explaining why the artefact containing the motivation is present. References to images may be included in this text.

The differences between description and motivation are as follows:
3.4 Changes with respect to previous versions

- A motivation is always optional,
- A motivation describes the “why” instead of the “how” and “what”,
- A motivation might contain textual descriptions of alternative solutions, for example because these alternatives do not work or are not explored yet.

The motivation element does not contain any attributes. The subelements of this element are described in Table B.14.

3.3.8 Image

The img element describes a reference to an image file. Images may be used in descriptions or motivations to augment the text. Future versions of the repository shall also contain support for tables, graphs, etc. Table B.15 describes the attributes of the img element. This element cannot contain any subelements.

3.4 Changes with respect to previous versions

As mentioned in the introduction, the current repository concepts are based on previous research. An overview of changes with respect to previous versions now follows.

Subsystem

In a previous version, described in [20], an element called subsystem was used. After discussing the fundamental differences between this and the system element, we decided that we actually meant the same when using either element. Furthermore, having only one (composite) element, indicates that the element can be used at an arbitrary level of abstraction. Hence, we decided to discard the subsystem element.

Delegation and Atom

Since the term mapping of a previous version was considered to generic, we renamed this to delegation. This indicates more intuitively what happens: requirements can be delegated to a system that is from that moment on responsible for their implementation. Furthermore, the term implementation yielded much discussion so we renamed this element to atom. An atom is meant as an atomic (indivisible) artefact, typically used as the representation of a piece of the concrete system implementation.

Extending abstract systems

Several discussions on how to implement the important feature of reuse yielded various possibilities. Since we wanted to keep the meta model as simple as possible, we decided to replace the declaration element by an attribute of the system, called abstract. The advantage of this approach is that concrete systems do not need to be declared first, declaration is done implicitly. The attribute can contain a simple boolean value (the default is false.
if the attribute is not present). If set to true, however, the respective system is declared to be
abstract, which means that no implementations of requirements are necessary.

The `implements` element has been replaced by `extend`, to make this reasoning more
clear. Systems that extend an abstract system are now obligated to implement (or delegate)
the requirements. Furthermore, any subsystem of the abstract system should be available in
the extending system as well.

Requirements attributes

In [8], an extensive list of possible attributes of requirements is presented. Since the reposi-
tory concepts should be kept simple, I selected the four most important attributes and added
these to the Forest meta model. This selection was based on the experience gained by read-
ing existing documentation.

Relations between systems

Some experiments have been done with specifying relations between systems. However,
the many types of relations that can exist between systems did not make the repository easy
to use. First, a simple repository should be available to prove the concepts of this approach.
Hence, this feature was postponed to later research.

Process descriptions

Processes often consist of a sequence of steps that need to be taken. This sequence can
be expressed in several ways and is also related to the problem of relations between sys-
tems. For now, processes can be described by means of creating several requirements and
describing in text in which order the system should fulfil these requirements.

Architectural layers

In a previous version, there was a notion of dynamic architectural layers. When specifying a
system, one could indicate of which layer this system should be a part. In a model checking
phase or a documentation generation phase it could be determined automatically which
layers are available. The user could then make a selection of the desired layer(s) when
generating a document.

3.5 Summary

First, a high-level overview of the Forest concepts was given. The main goal was explained:
creating and maintaining documentation of systems. Second, the various elements that can
be used to form a structured Forest repository were explained.

This explanation started with the introduction of four basic elements: a requirement, a
system, an atom, and a delegation. Using these four elements, the main types of information
and relations can be stored. Next, a number of additional elements are explained that can be
used to enrich the functionality of the repository, such as descriptions, motivations, images, etc. Finally, changes with respect to previous versions are explained.
Chapter 4

Using Forest in practice

This chapter describes how the concepts of the Forest project explained in the previous chapter can be used in practice. I will start with a practical overview of Forest and then zoom in on some details. The concepts and their use are validated by means of two case studies, described in Chapter 6. The proof-of-concept application that was used for validation is described in Appendix A.

4.1 Overview

The Forest project is about documenting systems, throughout the system’s entire life cycle\(^1\). The Forest method rests on two pillars: generation of system documentation on one hand, and adding traceability support to a system and its documentation on the other.

Documentation of a system, e.g. a requirements document or a design document, can be generated by a document generator tool. Such a tool creates a document with chapters according to a certain document standard, and fills these chapters with the desired information from a Forest repository. The resulting document is no longer intended to store on a bookshelf. On the contrary, any time a stakeholder needs information about a system, he can generate the desired document on-the-fly.

Traceability support, i.e. the tracing from requirements to their respective design and implementation artefacts, is possible by means of maintaining relations between artefacts in the repository; and by integration of the Forest repository into the existing repository holding the system’s project files, e.g. the file system.

4.2 Central repository

In order to create system documentation, information about these systems is needed. Information here is defined as the set of requirements, artefacts containing architecture and

\(^1\) Generating all documentation would be the ideal case. For the introduction of the Forest methodology, we do not aim at generating all documentation perfectly and fully traceable, but we rather aim at an improvement of the current documentation situation. Hence, in this thesis, we limited the set of documents-of-interest.
4.3 Entering information into the repository

Using Forest in practice

design descriptions, and test case descriptions. All relevant\(^2\) information about a system is stored in a central location, close to the project files that form the implementation of the system. For example, in case of a software system, the so-called Forest repository is typically located in the source directory in the file system, between the source files of the software project. This has the advantage that the Forest repository (as one or more XML files), is automatically managed by the version control system that is used for the projects source files, e.g. Concurrent Versioning System (CVS)\(^3\) or SubVersion (SVN)\(^4\). More information on the Forest repository can be found in section 3.2.

4.3 Entering information into the repository

Since all information in the repository is stored in XML files, in principle, information could be added to, or modified in the repository by hand. However, when more than only a few requirements or systems need to be stored in the repository, editing the repository manually becomes a tedious operation. The advantage of the XML file format is that “anyone” can build a tool or an editor to view and modify the contents of the Forest repository. This is a great advantage with respect to existing applications that often have their own proprietary file format, which is hardly (or sometimes not at all) accessible by other applications.

Having an editor to modify information in a Forest repository, a practical method of how to start entering information is needed. Systems that are still to be built require a different approach than already existing systems that need to be redocumented.

For new systems, a general approach may be as follows. First, an entity is defined for the system as a whole. Second, high-level requirements to the system are specified and stored in the repository. Next, subsystems are defined to fulfil each of the requirements specified earlier. Each requirement can now be delegated to the respective subsystem. If requirements cannot be fulfilled by one subsystem, the requirement must be split into several others first. Now, one level of abstraction deeper, per subsystem all requirements should be either implemented, or delegated to a subsystem once more. This process should continue until all requirements and architectural information are entered up to the desired level of detail.

For existing systems that need to be redocumented, the approach of entering information into the repository depends on the amount of documentation that is available as well as the accuracy of the existing documentation with respect to the actual implementation. A best-practice for dealing with this situation is to be established.

4.4 Checking for completeness and consistency

A repository is defined to be complete when all information artefacts that are required by other artefacts are available in the repository. For example, requirements to a system are

---

\(^2\)What information is relevant for documentation purposes, i.e. the amount of detail, varies per project.

\(^3\)http://www.nongnu.org/cvs/

\(^4\)http://subversion.tigris.org/
needed in order to know what the system should do. Storing a system artefact in the repository automatically means that one or more requirements need to be available as well. If this is not the case, the system is incomplete.

We define a repository to be consistent when all relations that are defined between the artefacts in the repository are valid. For example, a requirement may be delegated to a system, effectively stating that the system is now responsible for meeting that requirement. For this type of relation to be valid, both the requirement and the system that the relation refers to, should exist in the repository. Furthermore, the now responsible system should provide an implementation for this requirement or another delegation to yet another system.

### 4.4.1 Requirements traceability

When designing a system based on a set of requirements, it is common to rewrite, or translate, (some of) the given requirements. Original requirements might have a description that is too vague, too unclear, or too broad to be implemented directly. Sometimes these requirements are split in several, more specific, and more clear requirements. The opposite might also occur: several requirements that have a similar description can be rewritten to one single requirement. A common case of rewriting requirements is when business requirements need to be implemented, these typically need to be translated into functional requirements, non-functional requirements, or constraints [18]. Furthermore, instead of implementing a requirement directly in a system, a requirement may be delegated to one (or multiple) subsystem(s) first. A subsystem that “inherits” a requirement, is now responsible for either implementing the requirement by itself or delegating it even further.

Requirements traceability is the ability to trace (delegated) requirements through the architecture and design to their corresponding implementation artefacts and possibly even further to the corresponding test cases [8, 9]. In order to trace requirements, links (describing delegations to subsystems, or describing translations of $n$ requirements to $m$ others) need to be known and stored in the Forest repository.

Tracing, which can be done in order to perform a certain type of analysis [12], now effectively means following the links from one artefact to another. The most straight-forward type of analysis, called coverage analysis, checks whether all requirements can be traced down to an implementation artefact or even further to a test case. This type of analysis yields insight in the coverage of the requirements in terms of implementation or test case artefacts.

If a certain requirement cannot be traced to an implementation or a test case, a warning should be generated. During the generation of a document, this warning can be used to provide extra information. How this warning is eventually represented in the document depends on the type of document and the layout template used when generating the document.

### 4.5 Generating documentation

During the entire life of a system, various documents need to be created. For this thesis, we restricted the term *documentation* to two types of documents: a requirements document and
4.6 Redocumenting existing systems

Using Forest in practice

A detailed design document\(^5\). These documents are also part of the documentation standard JSTD-016 [10], which is used in almost every software project that is currently done at Chess. Each of these documents contains several common artefacts as well as a number of document-specific artefacts.

A document generator tool is needed to form the information in the repository into a document. The user creating a document, should select the desired information from the repository, which then appears in the document.

4.5.1 Documentation templates

Instead of selecting the desired information each time a document is created, document templates could be used. These templates contain a description of the chapter layout of the document that should be created according to a certain documentation standard, like JSTD-016. Within these chapter descriptions, queries can be used to select one or more artefacts from the repository. The document generator tool should then place the description of the selected artefact(s) in the respective chapter.

Although more research is required, especially with respect to the query language that is needed to select the desired artefacts from a repository, a preliminary proposal for such a documentation template is given in Listing [D.2] The template contents are, like the Forest repository itself, stored in an XML format.

4.5.2 Documentation augmented with traceability information

The Forest project is not only about generating documentation and not only about traceability (see section 4.4.1), but rather about the combination of these two. That is, generating informative documents about a system containing information about how the system should be built, or how it has been built, as well as information about the current state of the system. This state is in fact the result of various types of analysis, that can be performed by means of the traceability support of the repository.

When creating a document, the user should be able to indicate whether traceability information (results of tracing analyses performed on the repository) are to be included in the documentation. If so, the user can make a selection of the various types of analyses, which are then performed by the model checker and the repository is augmented with the results of these analyses. The document generator should now select the traceability results from the repository as well.

4.6 Redocumenting existing systems

The Forest method does not only apply to creating documentation for systems that still need to be built, but it can be applied to redocumenting existing systems as well. Having

\(^5\) At the start of this graduation project we had two extra types of documents in mind: an architectural overview document and a test case description document. However, since still more research is needed to define and implement a query language for selecting artefacts from the repository, creating these two extra document types with the current prototype implementation did not yield any additional scientific value.
a tool that continuously verifies the consistency and completeness of the system documentation that was entered so far, actually helps finding parts of the system that have not been documented yet. The second case study in Chapter [6] demonstrates this.

4.7 Summary

In this chapter, an overview of the Forest method is given. This method consists of entering information into a central repository and validating this repository for completeness and consistency. The concepts of this repository are explained in the next chapter. Furthermore, this chapter provided an explanation of generating documentation based on the information in the repository and how to augment this generated documentation with the tracing results. Documentation which is augmented with tracing results can provide information about the current state of the described system.
Chapter 5

Forest tools

This chapter describes at a conceptual level what tools are needed and/or useful when working according to the Forest method. During my graduation project, I created a software prototype implementation of these tools (see Appendix A) in order to perform a number of case studies. These case studies are used to validate that the Forest method can be used to generate and maintain system documentation, including the benefits from requirements traceability.

Summarizing from the previous chapter, all information about a system is to be stored in a repository, which needs to be checked for completeness and consistency. A user should be able to add and to modify the information in the repository and generate documentation based on the information in the repository. Generated documents optionally contain traceability information about the system, i.e. an indication of the development status of requirements or generated traceability matrices.

5.1 Forest repository

A location is needed to store all information about a system, which is called a Forest repository. The contents of a Forest repository are stored in a structured manner: an XML-based format, as explained in section A.2.2. The eXtensible Markup Language (XML) has become a standard for storing and exchanging information by computer applications in the last decade, and became W3C Recommendation early 1998\(^1\). The format of a Forest repository, which is open and readable for both computers and human beings, allows for easily creating extra tooling that is to use a Forest repository. This is a great advantage compared to other tools that can be used for requirements traceability, documentation generation, etc., which often store the data in a proprietary format, that is hard to read by other tools.

5.2 Editors

As explained in the previous chapter, an editor is desired to modify the contents of a Forest repository. It is possible to create a “Forest editor”, in which the user can view and mo-

\(^1\) [http://www.w3.org/XML/](http://www.w3.org/XML/)
5.3 Model checker

A tool is needed to verify the completeness and consistency of a Forest repository, as explained before. Similar to a compiler for any modern programming language, e.g. Java, which shows warnings and errors when it encounters syntactical (and nowadays even some semantical) mistakes in the source code, a model checker for a Forest repository should show comprehensive warnings and/or errors if the repository is not complete or not in a consistent state. Furthermore, warnings may be shown in the case where an identifier is assigned to several artefacts in the repository, since in this case the tracing of requirements might not work properly.

A suggested feature for a Forest model checker is to have some configuration possibilities, e.g. to enable or disable any checks that it can perform. The reason for this is that experience shows that every project is different, and hence for some project certain checks might not be relevant. In such a case, the user should then not be bothered with warnings or error messages.

Another suggested feature is to have configurable settings for when to raise a warning, and when to raise an error. For one project, certain checks might have different priorities than for another project. For example, a missing reference to a certain artefact should sometimes be regarded as a warning, while in other, more strict cases, an error should be raised.

5.4 Documentation generator

As explained in the previous chapter, part of the Forest method is generating documentation based on the information stored in the Forest repository. The document generator tool that we need should compose documents, based on a selection of the information artefacts in the Forest repository, while adhering to predefined “styles”. The word style has two meanings here, that are both relevant.
First, there is the visual layout defining the fonts to use, size and positioning of logo’s, page headers and footers, etc. Second, we need a “chapter layout”, stating what kind of information can be found in which chapter. This is typically defined in so-called documentation standards, e.g. JSTD-016 [10]. Ideally, the document generator tool takes the visual layout style and the chapter layout style (in the form of predefined templates) as its input and produces the desired document as its output. A convenient feature of such a tool would be the possibility to select the desired output format\(^3\), e.g. PDF, HTML, RTF, etc.

5.5 IDE integration

Using the previously described tools all separately would not be practical, therefore support by commonly used integrated development environments is desired. Next to the aspect of being practical, we expect that system documentation that is well-integrated with the development environment can improve efficiency during the implementation phase.

For example, in the case of a software project, the source code editor could contain extra navigation options to instantly “jump” to the related requirements or design description. These extra navigation options could be added to the context menu\(^4\) and should be accessible by means of short-keys\(^5\).

5.6 Summary

In this chapter, an overview is given of the tools needed when working according to the previously explained Forest method. The Forest repository is a “tool” which is needed to store all information artefacts and the relations between them. Editors are needed to add and modify this information and a model checker is needed to verify the completeness and consistency. In order to actually obtain documents that contain information from the repository, a document generator is needed. Finally, we expect an efficiency improvement during development when all tools are well-integrated with the development environment that is used to build or modify a system.

---

\(^{3}\) As explained in Appendix A, the current prototype implementation only supports PDF output.  
\(^{4}\) The popup menu that appears when right-clicking on e.g. a class or a method in the source code.  
\(^{5}\) For the Eclipse IDE, a number of user interface guidelines and a list of best practices is available at: http://www.eclipse.org/articles/Article-UI-Guidelines/Index.html
Chapter 6

Case Studies

This chapter describes the case studies that were performed in order to evaluate the Forest concepts. A prototype software implementation was created as a tool to assist in this evaluation. An overview of this prototype, some important design decisions, and a few implementation details can be found in Appendix A.

Since not all concepts of the Forest method can be verified by means of a single case study, two different experiments were performed. The first experiment describes the documentation of a newly built system, namely the Forest prototype implementation. Since quite a number of software engineering projects consist of making changes to existing systems, rather than creating new systems, the second experiment describes applying the Forest concepts to an existing system. For each experiment, several evaluation criteria were defined, as described in the next sections.

6.1 The first experiment: bootstrapping

As a first way of proving the Forest ideas and concepts, we tried to document the Forest prototype by means of bootstrapping\(^1\). That is, using the prototype to document the prototype.

Some initial documentation (requirements and design documents for the first phase of the project: Forest Core and Requirements Editor) was created the old-fashioned way, in MS Word. After a few iterations of improving these MS Word documents, I got the impression that the documents were finished, so we decided to “freeze” these documents. After having implemented the first phase of the prototype, requirements and design documents were created using this prototype according to the new Forest method. This allowed us to see which aspects of the documentation are better when the documents are generated using Forest. In this case, we define better documentation as being more complete and more consistent in terms of references between artefacts. The reason for expecting that the Forest-generated

\(^1\)The term “bootstrapping”, or “booting”, is commonly used to refer to the process of starting a computer system. The term originates from a story of Baron Münchhausen, originally written by Rudolf Erich Raspe, extended and republished by Gottfried Bürger in 1786, after which it became famous. In the story, the Baron escapes from a deep lake by pulling himself out by his own bootstraps.
documents are better, is the support for requirements traceability in Forest which allows for checking on completeness and consistency automatically, as opposed to MS Word.

6.1.1 Problem description

Writing requirements for a new system has a number of drawbacks in the current situation, as explained in Chapter [1]. Documentation manually created in MS Word can not be checked automatically for consistency and completeness. Furthermore, the documentation can not be linked to an implementation such that alarm bells would go off when changing the implementation without updating the documentation.

6.1.2 Evaluation criteria

In this experiment we are interested in the possibilities to capture requirements and design artefacts of a newly built system. Furthermore, we would like to see requirements traceability in action and generate documentation. To be more specific, the following evaluation criteria are defined for this case study:

- Is the meta model of the repository sufficient to capture all requirements and design information as written in the original MS Word documents?
- Does requirements traceability allow us to see warnings (on the screen and in the generated documentation) when requirements have not yet been implemented?
- Is the searchability of the documentation improved?
- Is the maintainability of the documentation improved?
- To what extend are the generated requirements and design documents comparable to the original, manually created, MS Word documents?
- Gain insight in the amount of time it takes to set up and maintain documentation in a Forest repository. As one of the requirements to the proposed solution in Chapter [1] states, we would like to keep the threshold for introduction of the Forest method as low as possible. If it would take too much time to update the repository after making changes to the system’s requirement, design, or implementation, it is not likely that the method will be used.

6.1.3 Case study details

Using the prototype implementation for entering information, tracing requirements and generating documentation during the building of the prototype was not possible due to the incremental development method that was used to build the prototype. However, the (nearly) finished prototype was usable. This served two purposes at the same time: testing and documenting the prototype.

First, a number of requirements and design descriptions were entered, see Figure [6.1]
Figure 6.1: High-level requirements and design of the Forest prototype.
Then, relations between these artefacts were created and the repository contents could be verified with the model checker. Problems with the implementation of requirements were found, as shown in Figure 6.2. In this figure, a requirement has been delegated to a subsystem twice but no implementation can be found. This is correct since no implementation artefacts (i.e. atoms) were entered into the repository, yet.

![Figure 6.2: A problem found by the model checker.](image)

After solving these problems by having atoms implement the various inherited requirements, the Forest model was free of errors and we were able to generate (simple) documentation. An example of a generated design document (generated for the second case study, see section 6.2) can be found in Appendix E.

The next step was to complete the Forest model in terms of design and implementation artefacts and to fill in all descriptions and some motivations of the systems and atoms in the repository. Finally, the atoms (i.e. the representations of parts of the concrete implementation) were linked to the Java files of the implementation. Automatically, a check was performed whether these files existed in the file system.

### 6.1.4 Results

This subsection lists the results of this case study, according to the evaluation criteria defined in subsection 6.1.2.

**Meta model completeness:** All information concerning the contents could be stored in the repository. Additional information, such as a copyright notice or review and distribution lists, is to be added by the document generator. This information, currently provided by the document generator itself, should not be stored in a Forest repository but in a document template instead, see the example in Appendix E.
**Case Studies**

### 6.1 The first experiment: bootstrapping

**Warnings due to traceability support:** This case study showed that a tool can show warnings and errors for requirements that are not satisfied by means of an implementation, pieces of implementation without any requirements explaining their presence, etc. Several possibilities exist to inform the user about these warnings and errors, e.g., using decorators in the IDE or writing error messages in the Problem view. Optionally, these decorators and error messages can be added to generated documents as well.

When the Forest repository describing the prototype was complete, only one warning was left because the atom **Artefact selector** in the **Document generator subsystem** could not be linked to a concrete implementation. This is indeed true, since the mechanism for selecting artefacts to appear in the generated documents has not been implemented, yet. The atom acts as a placeholder and is probably replaced by a system with a more detailed design at the time this mechanism will be designed, as explained in section 3.2.3.

**Searchability:** Since the documentation of a system can not only be accessed by means of digging through a text document but also by means of a tracing tool, searchability of the documentation has improved.

**Maintainability:** As a result of the improved searchability, the documentation has become better maintainable since artefacts are located faster than in the original MS Word documents. Furthermore, the (extensible) model checker can actually follow links from documentation artefacts to a concrete implementation resulting in warnings in the documentation when the implementation changes.

**Document comparison:** During the case study, a requirements document and a design document were generated. While further improving the prototype implementation, these documents became more and more similar to the original MS Word documents. However, the contents remained different between the generated and the original documents, this holds especially for the requirements document.

A fundamental difference between the manually created and the generated documents is that the latter contain by definition more structure. Next to this structure, the first impression is that the generated documents are more complete than the original documents.

Another difference is that the generated documents can optionally contain so-called **decorators** in the margin of each section to add visual status information. If the status information contains one or more errors, an appendix is added to the document containing descriptions of these errors. Optionally, the generated design document can contain generated traceability matrices, per system, although in the current prototype implementation the visual representation of these matrices should be improved.

**Time needed:** The time needed to enter the information artefacts describing the Forest prototype was approximately the same as when using MS Word. Adding new artefacts is slightly more work, since first the structure of the repository needs to be extended and only

---

2 The term decorator was borrowed from the Eclipse framework, where a decorator is a small (overlay) icon providing extra visual information about a resource.
then a description can be typed, as opposed to MS Word. Document layout issues, however, are not present when using the prototype and information could be retrieved much faster than from an MS Word document, since the structured views (Outline and Trace view) assist in navigation.

6.1.5 Conclusions

This subsection reflects on the results of the first case study. All information in the manually created documentation could be stored in the Forest repository. Its traceability support allowed for a number of checks to be performed automatically. While entering or modifying information, this is a great help in ensuring consistency and completeness. Furthermore, the traceability support has increased the searchability of the information contained in the Forest repository.

Increased searchability has improved maintainability of the documentation, which in turn leads to more complete documentation. Another reason for the extra completeness of the generated documents with respect to the original MS Word documents is that the Forest method enforces the user to first specify a system before requirements can be delegated to it. When creating this system artefact, it is very little extra work to immediately enter a title, a short description, and possibly a short motivation for this system.

The differences in the contents of the requirements documents are mainly due to the use of the JSTD016 standard in the MS Word document. When creating the MS Word document, a JSTD016 template was used. Future work is needed for the document generator to support various documentation standards, as mentioned in section 7.3.

With respect to the time needed when documenting a system using Forest, one advantage is that one does not need to be concerned about the layout of the final document while entering information. Compared to using complex MS Word layout features in a document, this can save some time. Furthermore, the time it takes to find the part of the documentation one needs is greatly reduced because of the improved searchability of the information.

6.2 The second experiment: a ticketing system

Although the first experiment already provided us with some answers, this experiment was performed in a “laboratory environment”. To feel confident about the method and the options currently available in the software prototype implementation, we decided to (partially) redocument a project that was actually being done at Chess at the time of writing. The advantage of recreating the documentation is that the generated documents can be compared to the original documents and some conclusions can be drawn with respect to the similarities and the differences.

The project chosen to document using the Forest method involves an already existing ticketing system of a large theatre. Customers of the theatre can subscribe for unlimited theatre access for a fixed monthly fee. The subscription fee (€17.50 per month) should be paid in advance. In processing these subscriptions, a number of changes are requested.

Since explicit change management was outside the scope of this graduation project, the required support was not built in the prototype implementation, yet. Nevertheless, require-
ments (describing the requested changes) and design artefacts (of the existing systems) can be entered into a Forest repository as if these apply to a new system to build.

In this case study, the system overview including the basic system requirements will be documented at a high level of abstraction. The subscription process is elaborated in more detail to show various requirements translations. Finally, the part of the system dealing with monthly payments is elaborated in full detail to demonstrate a trace from the requirements to the actual implementation.

6.2.1 Problem description

The ticketing system that is currently in use by the customer needs to be adapted to better support the theatre’s business process. After having the ticketing system in place, it turned out that quite a number of customers like to pay in advance for several months. Currently, assigning a payment for several months in advance to a customers contract needs to be done manually. Furthermore, sometimes customers pay a non-integer factor of €17.50, which currently cannot be processed.

Another change request is about simplifying contract terms. This involves modifications to the system that generates contracts for new customers as well as a migration of already existing contract data of current customers. Next to changes in payments and contract terms, currently existing reporting possibilities need to be changed. A number of reports are automatically generated on a monthly basis, which should provide more information. Due to time restrictions, the parts of the system relating to the latter two change requests are only documented at a high level of abstraction.

6.2.2 Evaluation criteria

In this experiment we try to verify that the Forest method can be used for documenting requirements and the design of a real system that is to be built (or in this case, adapted) for a real customer. The evaluation criteria for this case study can be divided in two categories: “does the Forest method work” and “advantages”.

In the first category, we define the following criteria:

- Is the current Forest meta model sufficient to capture all different types of requirements found in the original documentation?

- Can we capture all design artefacts in the repository as specified in the original documentation?

- Can we actually trace requirements to other (translated) requirements, to design artefacts, and to an implementation (in this case: a file containing Java source code)?

- Can we generate documents comparable to the existing documents? The term comparable in this criterion is two-fold: the contents of generated documentation should be similar to the contents of existing documents and the graphical layout should be similar as well.
• Gain insight in the number of levels of abstraction that is useful. When specifying a system, adding additional levels of abstraction might cause more overhead than added clarity, compared to having less levels with a longer description.

In the category “advantages”, we look for the following criteria:

• Do we find requirements that are not properly implemented? When some parts of the implementation are created under time pressure, it is possible that the implementation does work as expected and the (high-level) requirements are satisfied but the design documentation was not followed exactly. The problems of making changes to such an implementation are outlined in the problem statement in the introductory chapter.

• Do we find implementation artefacts without requirements? As explained previously, requirements traceability allows for among others a cost/benefit analysis. When using the prototype to document a system, this analysis is performed automatically after every change made.

• Are there any requirements or design patterns to include or can any set of artefacts be identified to extract for later reuse?

6.2.3 Case study details

In this subsection, the details of the case study performed are presented. Based on the existing documentation and discussion with the project’s architect, the high-level description of the system with its top-level requirements is documented as illustrated in Figure 6.3. Note that this high-level system artefact does not yet describe a software system; the term system is merely used as a design artefact. The system as well as each requirement contain rich text descriptions.

In Figure 6.4, the top-level requirements are taken care of: the requirements are delegated to three different subsystems by means of the delegation artefacts shown. First, requirements related to general administrative processes are delegated to the Administrative employee. Then, requirements related to automatic processes are to be implemented by the Automated system. Finally, requirements related to manual processes in a theatre are delegated the Theatre desk employee. The arrows in the figure illustrate the effect of these delegations. The Administrative employee and the Theatre desk employee are modelled as systems, since it was a design decision that these people were needed to perform certain (manual) tasks.

The next step in this case study was to document the subscription process. In Figure 6.3 the top-level requirement Subscribing is already specified in more detail using subrequirements. These subrequirements are now delegated to the various design artefacts. Delegating (or implementing) all subrequirements of a certain requirement already satisfies the model checker, as shown in Figure 6.4. The elaborated design of the subscription process is shown in Figure 6.5. Figure 6.6 shows that requirements can be translated into one or more other requirements, before being implemented or delegated further. Here, the requirement Card creation system needed is translated into four new requirements, once
delegated to the subsystem **UAC creator**. These four new requirements are either delegated further, or implemented directly by means of an atom.

The last step in this case study was to document the monthly payment process in full detail. Requirements at the highest level (typically business requirements) should then be traceable down to their implementation. This was the hardest part of the case study, since the original design document did not completely reflect the system’s implementation (anymore). At this point, I needed to do some reverse engineering in order to “reconnect” the trace from the implementation to the design artefacts already present in the Forest repository. After this, it was possible to completely trace requirements with respect to monthly payments to their corresponding implementation artefacts, as shown in Figures 6.7 and 6.8.

### 6.2.4 Results

This subsection presents the results of the second case study, according to the evaluation criteria defined in subsection 6.2.2.

**Forest meta model**: The requirement and design artefacts as described in the original documentation could indeed be captured and stored in the Forest repository. Some requirements were specified in more detail in terms of sub requirements first, after which these subrequirements were delegated to one or more systems. This is illustrated in Figure 6.4.

**Requirements traceability**: Traceability of requirements to design artefacts is shown in
Figure 6.4: By means of delegations, the top-level requirements of the Theatre ticketing system are delegated to subsystems.
Case Studies

6.2 The second experiment: a ticketing system

Figure 6.5: The subscription process elaborated in terms of requirements, subsystems and atoms.
6.2 The second experiment: a ticketing system

Creating a complete trace of most of the top-level requirements to the implementation was difficult, since a large gap was found between the original design document and the implementation. However, after completing the documentation by means of a little reverse engineering, a complete trace from the top-level requirements to their implementation could be made.

Requirements can also be translated into other requirements, as shown in Figure 6.6. This can also be useful to make a translation of a business requirement to one (or more) functional and/or non-functional requirement(s) explicit.

**Documentation generation:** Using the prototype implementation, a requirements document and a design document were generated from the information in the Forest repository. As found in the previous case study, future research is needed with respect to selecting the
Levels of abstraction to use: As explained in the introduction of this case study, the system overview is documented only at a high level of abstraction. The subscription process is elaborated in more detail, hence more levels of abstraction are used. The monthly payment process is elaborated in full detail, as a trace is shown from the high-level requirements down to the system’s implementation artefacts.

Traceability advantages: Due to the gap between the original design documentation and the system’s implementation, it was not possible to determine whether requirements were implemented in a wrong way. However, the cost/benefit analysis check of the model checker pointed out that several parts of the implementation did not seem to have any requirements to explain their presence.

Reuse The system’s implementation was designed for reuse. Six major parts could be identified, which were implemented as components that could be reused in any other project. These parts were documented as abstract systems in separate Forest repositories. When reusing a software component in another system, the documentation can be reused as well, by extending the abstract systems in the respective Forest repository. The components that were already reused during the implementation of the system were not documented in a Forest repository, yet.
Figure 6.8: A full trace of a split requirement to its implementations.
6.2.5 Conclusions

This subsection reflects on the results of the second case study.

First of all, the requirements and design artefacts written in the original MS Word documents could be entered into the repository. The original design document only contained system and subsystems having a hierarchical relation, which is already supported by the current Forest repository structure. However, we recognize that other relations might be needed, see section 7.3 for more information. The generated documents are comparable to the original documents, although with respect to selecting the desired contents some work still needs to be done.

Although the prototype implementation does not contain all proposed functionalities, yet, (see section 7.3), I have already experienced the advantages of updating documentation within the development environment. Warnings about non-existing references into the file system, atoms without requirements explaining their presence, or requirements without an implementation or delegation were corrected quickly and easily. Artefacts can be located quickly, since double clicking on an artefact in the Outline view immediately displays a full trace in the Trace view in either backward or forward direction. Adding extra navigation options between the actual implementation (e.g. the source code) and the corresponding design artefacts or requirements in the Outline or Trace view, is presumed to further improve efficiency. Hence, the overhead for keeping the repository up to date is reduced with respect to MS Word documents.

Gaining insight in the number of levels of abstraction to use, turned out to be a more difficult task. Since this is expected to vary per project, a general best-practice needs to be established in future research.

Because of the gap between the original design documentation and the implementation, it could not be decided whether requirements were implemented properly. However, a few pieces of implementation without requirements were found! When discussing this with one of the original software architects, I learned that the system was constructed by building a number of components that were intentionally kept as general as possible. This way, the individual components could be reused in other systems. This is an understandable decision but it should be mentioned in the system’s design documentation.

The envisioned reuse of components could be expressed by creating abstract systems in separate Forest repositories. Other systems can now benefit by extending these abstract systems. A number of existing components were used as well to support the system; if Forest repositories would have been available describing these, I could have extended these.

6.3 Summary

This chapter explained the case studies that I performed during my graduation project in order to validate the principles Forest method and gain insight in the usability of prototype. The first case study consisted of documenting the prototype implementation by means of the prototype itself. The second case study consisted of partially redocumenting a real project that was done at Chess at the time of writing; a ticketing system of a large theatre.
6.3 Summary

The main conclusions of these experiments were that the current Forest repository is sufficient to capture all requirements and design artefacts that were found in the original MS Word documents. However, other projects might require additional artefacts or relations between artefacts, see section 7.3.

Documenting systems using the Forest method does not introduce a lot of overhead compared to using MS Word, but it does yield additional benefits because of the requirements traceability support. A best-practice still needs to be established on the number of levels of abstraction to use when adding information to the Forest repository.
Chapter 7

Summary, Conclusions, and Future Work

This chapter first summarizes the project’s contributions. Next, I will reflect on the achievements of this graduation project and draw some conclusions about the Forest method in general as well as the underlying concepts.

Since it was known in advance that the project was too large, given the time for a single Master’s graduation project, quite an extensive list of future work is presented at the end of this chapter.

7.1 Summary

In this thesis, I have explained and evaluated a new way of documenting systems. First, the context was set: documentation is often not up to date or not created at all. Several general and some more specific documentation problems were identified in the current situation of creating and maintaining documentation at Chess. Based on these problems, the requirements to the solution were defined and the proposal of a solution, that was expected to meet these requirements, was introduced. The assignment of evaluating this solution, however, was too large for one Master’s graduation project, hence we limited scope of my graduation project. Within this scope, a number of research questions were posed in order to see whether the proposed solution is indeed a solution to the problems that were identified.

The key concepts of the proposed solution are introduced next. The concepts of the repository, needed for storing information in a structured manner, are explained and an explanation of adjustments made to previous repository concepts is given. With these concepts in mind, a method to use the concepts in practice is illustrated. When working according to this method, a number of tools are useful. These tools are conceptually explained and a discussion of the prototype implementation (which I created during my graduation project for evaluating the Forest method) is presented in Appendix A.

Two case studies are performed to validate the proposed documentation method. The first case was to document the prototype implementation by means of the prototype itself. In the second case study, a real project (done at Chess at the time of writing) was (partially)
7.2 Conclusions

This graduation project contributed to research in the fields of system documentation, automatic generation of documentation, and requirements traceability. A method of creating and managing documentation which only conceptually existed, was investigated, and was found to be useful. For this investigation, I created a prototype software implementation which I used to perform two different case studies. These case studies showed little to no extra overhead in documenting the respective systems, but yielded some important benefits.

However, our conclusions are based on the assumption that all information in the repository is kept synchronized with the actual implementation. In the general case where no implementation-specific extension is available, keeping the Forest repository synchronized with the actual implementation still needs to be done manually. Integration of the Forest editors in the development environment and easy navigating through the existing documentation are presumed to be of assistance and lower the threshold for many developers to actually make the effort of updating the system’s documentation. As with unit testing, a key factor to the adoption of the Forest concepts will be easy-to-use tooling, integrated in the development environment.

The Forest methodology is flexible and so are the tools. This means that the user is not obligated to fill in every detail. Even when not filling in the description and motivation fields of the artefacts, the requirements traceability feature of the structured repository may assist in completing the repository and verifying its consistency. Furthermore, searchability of the documentation has improved because of the traceability support of the Forest repository. However, a best practice needs to be established on the amount of detail used when creating artefacts in the repository. Creating more artefacts yields a more structured repository on one hand, but may take more time to specify on the other hand. When generating documents, more artefacts result in shorter descriptions, which in turn may result in less legible documents since the documents have become too structured. For example, in case of over-specification, a resulting design document may contain several levels of sections and subsections, each containing one or two sentences describing the system at the respective level of abstraction. Whether this is a problem depends on the targeted use of the document: should it be read from cover to cover or will it only be used as a reference guide.

After having discussions with several developers and project leaders, the use of status
information and automatically generated traceability matrices in the documentation was found to be useful in general, although not strictly required for every project. It is clear that more research is needed to improve the visual representation of these matrices.

7.3 Future work

Since it was clear from the beginning of my graduation project that the Forest project at Chess was too large for a single Master’s graduation project, a number of areas within the project still need to be investigated. In this section, I recommend several topics that should be covered in order to improve and enrich the functionality available.

7.3.1 Documentation templates and query language

For most projects done at Chess, there are requirements to the documentation. One of these requirements is often that a documentation standard, e.g. JSTD-016, should be used. This standard describes a number of documents and per document the mandatory and optional contents are listed.

A projection of existing documentation standards onto the information in the Forest object model needs to be devised, to describe the types of documents and describe which information should be placed where in the document. A proposal for this was suggested in Chapter 4 and Appendix D. This proposal contains a number of queries to select information from the current repository model. These queries are not part of an existing query language yet, they simply serve as an intuitive example. The syntax used is based on experience with several high-level programming languages. Typically, the query language that is needed and the method of actually selecting the desired information from the repository need to be investigated.

Together with a documentation template, a layout template should be packaged. This way, “document plug-ins” can be added to the document generator. A dynamic plug-in loading mechanism could be used to instantly generate new types of documents. Finally, a user should be given the possibility to choose which document standard should be used.

7.3.2 Extendibility

In at least two ways we would like to provide extendibility. First, the Forest model (i.e. the syntax of a repository XML file) is currently very simple. Although it was one of the goals to keep the syntax as simple as possible, some artefacts (like a concrete stakeholder), or relations (like relations between systems, other than the hierarchical relation) can currently not be stored in a Forest repository.

As mentioned in Chapter 2, the Griffin project at the Vrije Universiteit in Amsterdam has investigated an extensive structure to store design decisions, whereas the Forest project currently only supports a free text motivation field.

Furthermore, we recognize that currently there is no support for expressing certain artefacts that appear in UML(-like) diagrams such as states, sequences, or classes. This is due to the limited support for relations between system artefacts in the current Forest model.
Future research is needed to identify which other types of relations are useful to add to the Forest model.

Second, we have envisioned a plug-in mechanism for the model checker. Currently the tracing done by the model checker ends at an atom in the repository since we did not want to place any restrictions on the type of system to document, yet. If, however, a software system is designed that should be entirely created in Java, one can think of several extra, Java-specific checks. For example, an atom may refer to a Java class or even to a method inside a class. Now, a Java-specific extension of the model checker could check for presence of the referred class or method. Furthermore, tools can now be extended with navigation support to “jump” from the documentation view of an atom immediately to the Java method that implements what the atom describes. Furthermore, a list can be automatically compiled with classes and/or methods that contain a (part of the) implementation for a particular requirement.

7.3.3 Change management: impact analysis

The current prototype implementation has built-in support for two of the three types of analysis [12] that can be performed by means of the traceability support of the repository; coverage analysis and cost/benefit analysis. Impact analysis, to be used for supporting change management, was left out of the scope of this graduation project.

To implement support for change management in the current prototype, we would prefer the use of so-called delta files. These files contain only artefacts that should be changed in a new version of system, e.g. new requirements or changed atoms. The parser should be able to load the system descriptions of a certain version, e.g. 1.3, from the repository and then load the delta’s from the delta file (containing the changes between version 1.3 and 1.4) while merging these with the current model. The resulting model is now version 1.4 and can be saved separately. Furthermore, support for creating these delta files should become available.

Typically, the model checker shall now complain about incomplete traces from requirements, systems or atoms that have become obsolete, or artefacts that need to be changed. Trying to delegate any new requirements to the appropriate systems and/or atoms will help to estimate the amount of work needed to incorporate all the changes.

7.3.4 Reverse engineering

As explained in this thesis, the concepts of the Forest project could aid in a reverse engineering process. Such a process aims at recovering a system’s architecture from the implementation, possibly including traceability links back to the original requirements. More research is needed to investigate whether techniques for recovering requirements traceability links, like the use of latent semantic indexing [13,14], could be used in conjunction with the Forest project to yield a more complete and accurate system description.

1The term delta, is borrowed from physics, where the Greek letter Δ (capital delta) is used to indicate a change of a certain quantity.
7.3.5 Name spacing

For larger systems, it might be a restriction to use globally unique names. If, for example, a number of subsystems are existing components, it would currently not be allowed to use a name for an artefact in component A, that already exists for an artefact in component B. For the proof-of-concept experiment, relative small systems could be used where unique names can be chosen easily, so this was not a problem yet. However, a mature documentation system should allow the use of names that are only “locally unique”.

A solution to this “name clash” problem, is the use of name spacing. Several implementations of name spacing are available. Explicit declaration of the name space (e.g. when using the Java package keyword [7]), as well as implicit declaration (e.g. when using XPath queries to select certain artefacts from a XML file in general [19]), is possible. When using implicit declaration of the name space, names are resolved by following a path in the data structure. In the Forest object model this would be very well possible, since the model is in fact a structured tree.

7.3.6 Extend requirements traceability: include testing

Testing is an important part of software development. Testing and unit-testing is part of the Extreme Programming (XP) methodology [2], of which some fundamental ideas are applied at Chess. Therefore, it would be useful to extend the tracing of requirements to the (unit) test cases that are written for the implementation of each requirement. The status of the test case, e.g. “implemented” or “test successful”, can then be shown when viewing the requirements, either on-screen or in a generated document. However, next to unit testing (sometimes called component testing), there should also be support for integration testing, system testing, and acceptance testing [8].

Several applications are currently available for managing tests and test cases. Extending Forest to support test cases could be done in at least two ways. The first possibility is to store all test information (test case descriptions, test status, etc.) in the Forest repository and discard all existing tools. This, however, involves quite a change in the entire process of testing. An alternative possibility, which is probably easier to start with, is to interface with test management applications that are currently in use. At first, this test management application is “in control” and contains requirements, test case descriptions based upon those requirements, and manages the process of running the tests to acquire the status of each test. Test case descriptions and test status are “exported” from the test management tool to the Forest repository. Now, a phased transition can be made to make the Forest repository more responsible for the test cases, until finally the running of the test cases is managed by a Forest application.

7.3.7 Tools and User Interfaces

The current prototype provides a number of editing capabilities for the Forest repository. These editing capabilities are made available via a single plug-in for the Eclipse IDE. In the near future, it might be desired to create several more editors, one for each stakeholder. These editors should not depend on the Eclipse framework by definition.
7.3 Future work

The use of a visual editor for creating artefacts and the relations between artefacts needs to be investigated. As described in section A.2.2, I started implementing a visual editor but soon I observed that it would take too much time to complete this editor within my graduation project.

Optimizing the generated traceability matrices is another feature on the wish-list, since the JSTD-016 standard prescribes the use such a matrix in some of the documents. The current prototype is capable of generating these matrices but due to the document generator that is used there are still some visual limitations.

Furthermore, although I tried to follow the Eclipse user interface guidelines as much as possible, it is probably worthwhile to have an expert looking at the user interface of the existing prototype again. While doing the case study, I found that some actions in the user interface are a bit tedious. Furthermore, during editing, the number of artefacts shown in the outline view grows quickly so enhanced filtering of the artefacts shown would improve navigation. For example, a filter option might be to collapse all systems except the one in which the user is currently editing.

7.3.8 Prototype: performance

With respect to the performance of the prototype Eclipse plug-in, there is room for improvement. Currently, the prototype editors update the model on any change and invoke the model checker on any structural change in the model. This introduces some overhead, which might be substantially reduced when using the Eclipse building facilities. This allows for checking only that part of the model that was changed, instead of checking the entire model over and over again.

Furthermore, the Outline view in the prototype Eclipse plug-in should be refreshed (a fast operation) instead of provided with a entirely new model (slow operation, dependent on the size of the model), to improve the user’s experience with the tool.
Bibliography


Glossary

The following list presents explanations of frequently used terms and abbreviations.

**Artefact:** A complex piece of information representing a requirement, the design of a (sub)system, a part of the implementation, or a relation between two or more of these.

**Documentation:** In this thesis, the term documentation is restricted to the minimum set of documents that is created for almost every project at Chess ibusiness. This minimum set consists of requirements documents and detail design documentation. These documents are to be generated, and augmented with traceability information. That is, the documents shall contain information about the state of (various parts of) the system. Being able to generate and augment these two types of documents is representative for generating other types of documents as well.

**Eclipse:** The Eclipse platform is a multi-purpose software framework. Originally intended as an Integrated Development Environment (IDE) for creating new software applications, today Eclipse is an outstanding basis for creating so-called rich clients. Rich clients are stand-alone applications, built for a special purpose, which can use the functionalities offered by the Eclipse framework, such as the ease of creating editors, views, using threads with a progress monitor, etc.

**IDE:** Integrated Development Environment. This is a software application (e.g. the Eclipse IDE) in which source code can be written, which contains an embedded compiler to build the software that is being developed and often various debugging options are available as well.

**MS Word:** Microsoft® Word, a commonly used word processor for office applications. This application is frequently used for writing system documentation as well.

**Repository:** An place to store data, e.g. a computer file system in which files can be stored or a Forest repository used to store documentation artefacts.
**Plug-in:** In general, a plug-in is a software package which can be “plugged into” an existing software application to enrich its functionality. A part of the software prototype I created during my graduation project is meant to extend the Eclipse IDE with functionalities for creating and managing system documentation.

**Subsystem:** A system contained in another system. Typically, a subsystem contains more detailed information than its parent system.

**System:** Anything that is designed based on requirements and consists of smaller individual components (subsystems), which can be designed in more detail, created (implemented), and tested.

**Trace:** The set of consecutive relations between artefacts. Typically, a trace starts at a requirement which is delegated to one or several design artefact(s), and is finally delegated to an implementation artefact. See Figure [6.6](#) for an example.

**Traceability:** The degree to which requirements and/or design artefacts are traceable in forward or backward direction.