

Quantification and Traceability of Requirements

Gyrd Norvoll

Master of Science in Computer Science

Submission date: May 2007

Supervisor: Tor Stålhane, IDI

Problem Description

Software development is a highly dynamic process, primarily caused by its foundation in the dynamic human world. Requirements traceability alleviates the detrimental effects of this dynamism by providing increased control over the artifacts of the software development processes and their interrelationships.

The implementation of requirements traceability in software development projects is a challenging task. The work overhead can increase significantly, and the benefits could soon become second to the amount of work required to harvest them. This situation destroys the purpose of implementing RT, and must be amended by reducing the work overhead associated with RT, which can be accomplished by introducing an RT tool.

This thesis shall research how an RT tool should be implemented to reduce the work overhead associated with implementing RT in a software development project. The research shall conclude with the development of an evolutionary prototype of an RT tool, but preparatory to this, the applicability of the traceability models presented in the preceding in-depth study must be verified by means of empirical work. The internal representation of the artifacts and traces of the traceability models must be determined, in addition to addressing the issue of representation of stakeholders' organisational hierarchies.

Further, the thesis shall present a full requirements specification and an architectural description of an RT tool, suggesting solutions to the issue of increased work overhead when implementing RT. Based on the requirements specification and the architectural description, an evolutionary prototype of the described RT tool shall be developed, to give its users an impression of the functionality of the designed RT tool, and illustrating how full forwards and backwards requirements traceability can be provided.

Assignment given: 19. January 2007
Supervisor: Tor Stålhane, IDI

Abstract

Software development is a highly dynamic process, primarily caused by its foundation in the dynamic human world. Requirements traceability alleviates the detrimental effects of this dynamism by providing increased control over the artifacts of the software development processes and their interrelationships.

This thesis investigates how an RT tool should be designed and implemented in order to assist with the tasks of requirements traceability, and outlines a tool that primarily focuses on reducing the work overhead associated with the tasks of implementing requirements traceability in software development projects.

Preparatory to the development of the RT tool, the applicability of the traceability models presented in the in-depth study has been confirmed through empirical work. A detailed representation of the models has been compiled, elaborating on the internal representation of artifacts and traces. The models were extended to be able to represent organisational hierarchies, enabling trace information analysis to deduce the context of important decisions throughout the software development processes, an important tool in understanding how requirements are determined.

The thesis presents a requirements specification and architecture with a firm foundation in the findings of the in-depth study, outlining an RT tool that addresses important issues concerning the implementation of requirements traceability, in particular focusing on reducing the associated work overhead. Based on the requirements specification and architecture, a evolutionary prototype is developed, giving its users an impression of the functionality of the outlined RT tool. The prototype addresses the issues pointed out by the requirements specification and architectural description, and, throughout development, attention is given the evolvability of the prototype. Consequently, the prototype provides a good foundation for the future development of a complete RT tool.

Preface

This thesis is written as the concluding part of a Master's degree in Computer Science. The thesis is written during the spring semester of 2007, at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU) in Trondheim, Norway.

The thesis includes an in-depth study, researching the field of requirements traceability. The thesis investigates how an RT tool should be designed in order to assist with the tasks of implementing requirements traceability in software development projects.

Thanks is given to the assigned teaching supervisor Tor Stålhane, for his continuous support and feedback.

Gyrd Norvoll

Trondheim, June 2007

Contents

I	Thesis directive	1
1	Introduction	3
1.1	Purpose	3
1.2	Scope	3
1.3	Overview	3
2	Thesis mandate	4
2.1	Background	4
2.2	Motivation	5
2.3	Problem Definition	5
2.4	Context	6
2.5	Audience	6
2.6	Scope	6
2.7	Report Outline	6
3	Thesis processes and methods	8
3.1	Methodology	8
3.2	Preparatory work	8
3.3	Prototype development	9
4	Summary	10
II	Preparatory Work	11
5	Introduction	13
5.1	Purpose	13
5.2	Scope	13
5.3	Overview	13
6	Applicability of results from in-depth study	15
6.1	Industrial experiences	15
6.2	Evaluation of results	16
6.2.1	Using the traceability models as a vehicle of communication	16
6.2.2	An alternative application of the pre-RS traceability model	16
6.2.3	The need for information	17
6.2.4	Challenges in modern IT development	17
6.3	Alterations in TRACY	17
6.3.1	Pre-RS traceability submodel	18
6.3.2	Pre-FRS traceability submodel	18
6.3.3	Post-FRS traceability submodel	19
7	Representation of stakeholders and their organisational roles	21
7.1	Stakeholder hierarchy	21
7.1.1	Status networks	21
7.2	Group dynamics	22
7.2.1	Majority and minority influence	22

CONTENTS

7.2.2	Pressure to conform	23
7.3	Determining stakeholders' roles	23
8	Internal representation of artifacts and traces	25
8.1	Application of Planguage	25
8.2	Traceability granularity	26
8.3	Internal representation of artifacts	26
8.3.1	Pre-RS Traceability Submodel	26
8.3.2	Pre-FRS Traceability Submodel	30
8.3.3	Post-FRS Traceability Submodel	32
8.4	Internal representation of traces	34
8.4.1	Pre-RS traceability submodel	34
8.4.2	Pre-FRS traceability submodel	35
8.4.3	Post-FRS traceability submodel	35
9	Summary	36
 III Requirements Specification		 37
10	Introduction	39
10.1	Document information	39
10.2	Purpose	39
10.3	Scope	39
10.4	Definitions, acronyms, and abbreviations	40
10.5	References	40
10.6	Overview	40
11	Overall description	41
11.1	Product perspective	41
11.2	Product functions	41
11.3	User characteristics	42
11.4	Constraints	42
11.5	Assumptions and dependencies	43
11.6	Requirements subsets	43
12	Specific requirements	44
12.1	External interface requirements	44
12.2	Functional requirements	45
12.2.1	Traceability projects	45
12.2.2	Gathering trace information	47
12.2.3	Maintaining trace information	48
12.2.4	Searching trace information	50
12.2.5	Visualising trace information	51
12.2.6	Help and assistance	52
12.2.7	Administrative functions	53
12.3	Use case modelling	54
12.3.1	The concept of use case modelling	54
12.3.2	Description of actors	54
12.3.3	System use cases	55
12.3.4	Tracing of use cases	55
12.4	Performance requirements	56
12.5	Design constraints	57
12.6	Software system attributes	58
12.7	Other requirements	60
13	Requirements dependencies	61

IV Architectural Description	65
14 Introduction	67
14.1 Document information	67
14.2 Summary	67
14.3 Scope	67
14.4 Context	67
14.5 Glossary	68
14.6 References	68
14.7 Overview	68
15 Identification of stakeholders and concerns	70
15.1 Stakeholders	70
15.2 Concerns	71
16 Architectural strategies	73
17 Selection of architectural viewpoints	78
17.1 Logical viewpoint	78
17.2 Process viewpoint	79
17.3 Development viewpoint	80
17.4 Physical viewpoint	80
18 Architectural views	82
18.1 Logical view	82
18.1.1 The domain model	82
18.1.2 Component-based architecture	84
18.2 Process view	87
18.2.1 Three-tier client-server	87
18.3 Development view	88
18.4 Physical view	91
19 Consistency among architectural views	93
20 Architectural rationale	95
V Prototype development	97
21 Introduction	99
21.1 Purpose	99
21.2 Scope	99
21.3 Overview	99
22 Implementation details	101
22.1 Technical details	101
22.2 Prototype development	102
23 Using the prototype	104
23.1 Accessing the prototype	104
23.2 Managing projects and other administrative work	105
23.3 Project contents	108
23.4 Creating and maintaining trace information	109
23.4.1 Artifacts	110
23.4.2 Traces	114
23.5 Visualising trace information	115
23.6 Search	118
23.7 Help and assistance	118

24 Prototype evaluation	121
24.1 Prototype evolvability	121
24.2 Prototype contributions	121
24.2.1 Fulfilment of functional requirements	122
24.2.2 Fulfilment of non-functional requirements	124
25 Prototype evolution	126
25.1 Model-View-Controller design pattern	126
25.2 Model Persistence	126
25.3 Concurrency	127
25.4 Alpha functionality	127
26 Summary	128
VI Evaluation & Discussion	129
27 Introduction	131
27.1 Purpose	131
27.2 Scope	131
27.3 Overview	131
28 Evaluation	133
28.1 Fulfilment of research agenda	133
28.2 Preparatory work	133
28.2.1 Validity of empirical work	134
28.2.2 Representation of stakeholders and their organisational roles	134
28.2.3 Internal representation of artifacts and traces	135
28.2.4 Evaluation of Planguage	135
28.3 Requirements specification	136
28.4 Architectural description	136
28.4.1 Identified tradeoff points	136
28.4.2 Identified risks and nonrisks	137
28.5 Prototype	137
29 Discussion	139
30 Summary	141
VII Conclusion & Further Work	143
31 Introduction	145
31.1 Purpose	145
31.2 Scope	145
31.3 Overview	145
32 Conclusion	146
33 Further work	147
VIII Appendices	149
A A brief overview of the in-depth study	150
A.1 Study Outline	150
A.2 Important findings	150
B Employed <i>Planguage</i> attributes	152
C Internal representation of artifacts	154

D	Eight golden rules of interface design	156
E	High-level Requirements Specification	158
F	Full Requirements Specification	159
F.1	External interface requirements	159
F.1.1	User interface requirements	159
F.1.2	Hardware interfaces	162
F.1.3	Software interfaces	162
F.1.4	Communications interfaces	163
F.2	Functional requirements	163
F.2.1	Traceability projects	163
F.2.2	Gathering trace information	166
F.2.3	Maintaining trace information	167
F.2.4	Visualising trace information	170
F.2.5	Help and assistance	173
F.2.6	Administrative functions	175
F.3	Performance requirements	176
F.3.1	Throughput	176
F.3.2	Response time	178
F.3.3	Storage capacity	181
F.4	Design constraints	182
F.4.1	Standards compliance	182
F.4.2	Hardware limitations	182
F.5	Software system attributes	182
F.5.1	Functionality	182
F.5.2	Reliability	184
F.5.3	Maintainability	185
F.5.4	Portability	186
F.5.5	Usability	188
G	Use cases	191
H	Prototype development	199
H.1	Prototype design	199
H.2	Prototype installation	205
H.2.1	Downloading the Application Server	205
H.2.2	Installing the Application Server	205
H.2.3	Starting the Application Server	206
H.2.4	Deploying the prototype	206

List of Figures

3.1	Thesis processes	8
3.2	Development process	9
6.1	Pre-RS Traceability Submodel	18
6.2	Pre-FRS Traceability Submodel	19
6.3	Post-FRS Traceability Submodel	19
8.1	Internal representation of the <i>Prose requirement</i> artifact	27
8.2	Internal representation of the <i>Stakeholder</i> artifact	28
8.3	Internal representation of the <i>Functional requirement</i> artifact	31
8.4	Internal representation of the <i>Non-functional requirement</i> artifact	31
8.5	Internal representation of the <i>System component</i> artifact	33
12.1	Summary of external interface requirements	44
12.2	Summary of functional requirements	46
12.3	Tracking matrix - Use cases/Requirements	56
12.4	Summary of performance requirements	57
12.5	Summary of design constraints	58
12.6	Summary of software system attributes	59
13.1	Tracking matrix - Requirements/Requirements	62
13.2	Tracking matrix - High-level requirements/Detailed requirements	63
16.1	Component-based architecture	74
16.2	Model-View-Controller (MVC) design pattern	75
16.3	Virtual Machine	75
16.4	Three-tier Client-Server design pattern	76
17.1	"4+1" view model of software architecture	78
17.2	UML concepts - class diagram	79
17.3	UML concepts - component diagram	79
17.4	UML concepts - sequence diagram	80
17.5	UML concepts - deployment diagram	81
18.1	Domain model	83
18.2	High-level logical structure	84
18.3	Logical structure	85
18.4	Project model	86
18.5	High-level process architecture	88
18.6	The layers of the RT tool	89
18.7	Detailed components	90
18.8	Deployment of the RT tool	91
23.1	TraceMe - Login	105
23.2	TraceMe - Welcome screen	105
23.3	TraceMe - User details	106
23.4	TraceMe - Project details	107

LIST OF FIGURES

23.5	TraceMe - Register new user	107
23.6	TraceMe - Create new project	107
23.7	TraceMe - Edit project	108
23.8	TraceMe - Attach user to project	108
23.9	TraceMe - Project contents	109
23.10	TraceMe - Artifact categories	110
23.11	TraceMe - Artifact instances	110
23.12	TraceMe - Artifact instance details	111
23.13	TraceMe - Artifact instance details - Example 2	111
23.14	TraceMe - Artifact instance details - Example 3	112
23.15	TraceMe - Artifact instance details - Example 4	112
23.16	TraceMe - Create artifact	113
23.17	TraceMe - Import system components	114
23.18	TraceMe - Defined traces	115
23.19	TraceMe - Create trace - step 1	116
23.20	TraceMe - Create trace - step 2	116
23.21	TraceMe - Complex predetermined visualisation	117
23.22	TraceMe - Custom-made visualisation	117
23.23	TraceMe - Custom-made visualisation - Example 2	118
23.24	TraceMe - Visualisations	119
23.25	TraceMe - Search	119
23.26	TraceMe - Help menu	120
A.1	Requirements traceability	151
C.1	Internal representation of artifacts - Pre-RS traceability submodel	154
C.2	Internal representation of artifacts - Pre-FRS traceability submodel	155
C.3	Internal representation of artifacts - Post-FRS traceability submodel	155
H.1	Organisation of JSP files	200
H.2	Prototype design	201
H.3	Detailed specification of the domain object model - pre-RS	202
H.4	Detailed specification of the domain object model - pre-FRS	203
H.5	Detailed specification of the domain object model - post-FRS	204

List of Tables

8.1	Artifacts in the pre-RS traceability submodel	27
8.2	Artifacts in the pre-FRS traceability submodel	30
8.3	Artifacts in the post-FRS traceability submodel	33
12.1	Actor description - regular user	54
12.2	Actor description - administrative user	54
12.3	Actor description - System	55
24.1	Fulfilment of functional requirements	122
24.2	Supported use cases	123
E.1	High-level Requirements Specification	158
F.1	User Interface Requirement - Window-based application	159
F.2	User Interface Requirement - Feedback	159
F.3	User Interface Requirement - Navigation	160
F.4	User Interface Requirement - Location	160
F.5	User Interface Requirement - Escape	160
F.6	User Interface Requirement - Menu	161
F.7	User Interface Requirement - Logout	161
F.8	User Interface Requirement - Eight Golden Rules of HCI Design	161
F.9	Hardware Interface Requirement - Hardware Independency	162
F.10	Software Interface Requirement - Application Server	162
F.11	Software Interface Requirement - Database Server	162
F.12	Communication Interface Requirement - Underlying Communication	163
F.13	Functional Requirement - Create a new project	163
F.14	Functional Requirement - Remove a project	163
F.15	Functional Requirement - Attach users with project	164
F.16	Functional Requirement - Choose project to work with	164
F.17	Functional Requirement - Exit project	165
F.18	Functional Requirement - Determine Traceability Stage	165
F.19	Functional Requirement - Navigating the trace information	165
F.20	Functional Requirement - Creating an artifact instance	166
F.21	Functional Requirement - Choosing a starting artifact instance	166
F.22	Functional Requirement - Choosing an ending artifact instance	166
F.23	Functional Requirement - Registering auxiliary trace information	167
F.24	Functional Requirement - Choose artifact to edit	167
F.25	Functional Requirement - Save changes to edited artifact instance	168
F.26	Functional Requirement - Choose artifact to delete	168
F.27	Functional Requirement - Choose trace to edit	168
F.28	Functional Requirement - Save changes to edited traces	169
F.29	Functional Requirement - Choose trace to delete	169
F.30	Functional Requirement - System-initiated trace deletion	170
F.31	Functional Requirement - Searching trace information	170
F.32	Functional Requirement - Requesting visualisation of single artifacts or traces	170
F.33	Functional Requirement - Requesting complex visualisations	171

LIST OF TABLES

F.34	Functional Requirement - Simple predetermined visualisations	171
F.35	Functional Requirement - Complex predetermined visualisations	172
F.36	Functional Requirement - Adapting simple visualisations	172
F.37	Functional Requirement - Creating custom-made complex visualisations	172
F.38	Functional Requirement - Choosing starting and ending artifact of complex custom-made visualisations	173
F.39	Functional Requirement - Help Messages	173
F.40	Functional Requirement - Help Menu	174
F.41	Functional Requirement - Search Help Menu	174
F.42	Functional Requirement - Adding a user	175
F.43	Functional Requirement - Removing a user	175
F.44	Functional Requirement - Changing user password	175
F.45	Functional Requirement - Changing access level	176
F.46	Functional Requirement - User Authentication	176
F.47	Performance Requirement - Throughput: Number of simultaneously active users	176
F.48	Performance Requirement - Throughput: Number of requests to system	177
F.49	Performance Requirement - Throughput: Number of requests to database	178
F.50	Performance Requirement - Response time: User requests	178
F.51	Performance Requirement - Response time: Database requests	179
F.52	Performance Requirement - Response time: Search	179
F.53	Performance Requirement - Response time: User authentication	180
F.54	Performance Requirement - Response time: Visualisations	180
F.55	Performance Requirement - Storage capacity: Registered users	181
F.56	Performance Requirement - Storage capacity: Trace information	181
F.57	Design Constraint - Architectural description	182
F.58	Design Constraint - Hardware limitations	182
F.59	Software System Attribute - Nonrepudiation	182
F.60	Software System Attribute - Confidentiality	183
F.61	Software System Attribute - Auditing	183
F.62	Software System Attribute - Recoverability	184
F.63	Software System Attribute - Availability	184
F.64	Software System Attribute - Changeability	185
F.65	Software System Attribute - Testability	186
F.66	Software System Attribute - Installability: Installation costs	186
F.67	Software System Attribute - Installability: Time to install	187
F.68	Software System Attribute - Adaptability: Increase number of simultaneous users	187
F.69	Software System Attribute - Adaptability: Alterations to the traceability model .	188
F.70	Software System Attribute - Understandability	188
F.71	Software System Attribute - Learnability	189
F.72	Software System Attribute - Operability: Visualisation flexibility	189
F.73	Software System Attribute - Operability: Change functionality	190
G.1	Use case - User login	191
G.2	Use case - Choose project	191
G.3	Use case - Choose traceability stage	192
G.4	Use case - Working with artifacts	192
G.5	Use case - Create an artifact	193
G.6	Use case - Edit an existing artifact	193
G.7	Use case - Delete an existing artifact	194
G.8	Use case - Locate an existing artifact	194
G.9	Use case - Add a trace between artifacts	195
G.10	Use case - Delete a trace between artifacts	195
G.11	Visualise trace information	196
G.12	Use case - Search through trace information	196
G.13	Use case - Creating a project	197
G.14	Use case - Register a user	197
G.15	Use case - Search through help directory	198

Part I

Thesis directive

Chapter 1

Introduction

In this chapter, a short summary of the purpose and scope of the thesis directive is given, along with an overview of the chapters.

1.1 Purpose

The purpose of this part is to provide a framework for the thesis, to be used as guidance during the remainder of the thesis work.

1.2 Scope

The following chapters describe the foundations of the thesis, i.e. its background, motivation and problem definition. In addition, the scope of the thesis is stated, informing the reader of the focus of this thesis, and defining the boundaries of the scope. The chapters do not describe any of the results, only which results are to be obtained, as well as what methods and processes that should be used in obtaining them.

1.3 Overview

Here we give an overview of the structure and a quick summary of all chapters in this part.

- **Chapter 2 - Thesis Mandate**
This chapter describes the mandate of the thesis, which contains information such as the background and motivation of the thesis, and its research agenda.
- **Chapter 3 - Thesis Processes and Methods**
This chapter describes the processes and methods employed during the work of the thesis.
- **Chapter 4 - Summary**
A summary of the key decisions of the thesis directive.

Chapter 2

Thesis mandate

The thesis mandate presents all necessary information regarding the justification of this thesis, and its purpose. This chapter begins by giving an introduction to the background of the thesis and its field of research, continuing with the motivation of the thesis. The chapter concludes with a research agenda, scope, and context of the thesis.

2.1 Background

In preparation for this Master's Thesis, an in-depth study [Nor06]¹ was conducted, focusing on the theoretical aspects of the field of quantification and traceability of requirements (RT), a field that has arisen as a response to the challenges created when software engineers attempt to keep up with the human dynamism of their customers and the ever-changing nature of software development.

The in-depth study defined requirements traceability as *“the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases).”*

In order to be able to describe and follow the life of a requirement, the in-depth study presented a set of traceability models, which defined artifacts of the software development processes and their interrelationships. Examples of artifacts include stakeholders, requirements, change proposals, system components and verification procedures.

Software engineering is a highly dynamic process, mainly because the process is located in the human world. The traceability models establishes how the artifacts of software development processes are interrelated and influence each other, providing an instrument for controlling not the dynamism of software engineering itself (a seemingly impossible task), but the influence of the dynamism on the results of the software engineering processes, thus alleviating any destructive influences.

The benefits of employing RT in software development projects would incontestably bring added value to a project, particularly in the form of increased control over the influence of dynamism. However, employing RT is not a triviell task, and cannot be accomplished without considerable effort. The added value will soon loose its valour if it can only be achieved through an undesirable work overhead.

¹A brief overview of the in-depth study can be found in Appendix A

2.2 Motivation

In short, requirements traceability requires access to and analysis of information on the artifacts of the software development processes and their interrelationships. This information, referred to as trace information, must be gathered and maintained, tasks that cause undesirable work overhead, reducing the effect of any added value gained by employing RT.

The reason why the tasks of gathering and maintaining trace information incur such work overhead, is found by considering the dynamism of the human nature. Software development processes are located in the human world, and are subject to our inclination to change our minds. Consequently, the artifacts of the system development processes are often altered during their life cycle. Whenever an artifact changes, its trace information must be updated, adding to the work overhead.

By using a tool designed to assist with the tasks of requirements traceability, e.g. with the means of automation, this work overhead can be significantly reduced, allowing focus to be shifted to the analysis of the trace information. In order to harvest the benefits of requirements traceability, the gathered trace information must be analysed, and the conclusions drawn should initiate actions that brings added value to the project, such as quality assurance. If the majority of effort must be spent on gathering the trace information rather than analysing it, it is likely that the analysis and its impact on the software development processes will suffer.

Consequently, the task of implementing requirements traceability in a software development project can be significantly alleviated by providing an RT tool that is able to assist its user with gathering and maintenance of trace information, allowing the majority of effort to be spent on analysis of this information, as well as providing mechanisms for efficient analysis of the gathered trace information.

2.3 Problem Definition

The implementation of requirements traceability in software development projects is a challenging task. The work overhead can increase significantly, and the benefits could soon become second to the amount of work required to harvest them. This situation destroys the purpose of implementing RT, and must be amended by reducing the work overhead associated with RT, which can be accomplished by introducing an RT tool.

Research Agenda

This thesis shall, based on the results of the in-depth study [Nor06], research how an RT tool should be implemented to reduce the work overhead associated with implementing RT in a software development project. The research shall conclude with the development of an evolutionary prototype of an RT tool, but preparatory to this, some issues referred from the in-depth study must be resolved. The applicability of the traceability models presented in the in-depth study must be verified by means of empirical work. The internal representation of the artifacts and traces of the traceability models must be determined, in addition to addressing the issue of representation of stakeholders' organisational hierarchies.

Further, the thesis shall present a full requirements specification and an architectural description of an RT tool, suggesting solutions to the issue of increased work overhead when implementing RT. Based on the requirements specification and the architectural description, an evolutionary prototype of the described RT tool shall be developed, to give its users an impression of the functionality of the designed RT tool, and illustrating how full forwards and backwards requirements traceability can be provided.

2.4 Context

This thesis is a Master's Thesis at the Norwegian University of Technology and Science, and takes place during the spring semester of the fifth year of the Master of Computer Science programme. The thesis includes an in-depth study [Nor06], conducted during the autumn semester of the fifth year of the programme, which researched the field of requirements traceability with the intent of uncovering how RT can assist the processes of software engineering.

2.5 Audience

This study is primarily intended to be of interest to those involved in the requirements engineering processes of software development, in particular those who seek better control of software requirements throughout the project life cycle. The thesis researches how an RT tool can assist with the task of implementing RT in a software development project, and suggests how an RT tool can be implemented. The thesis is directed at those interested in implementing RT in their software development projects, and should be of particular interest to those attempting to develop their own RT tool.

2.6 Scope

In addition to the development of an RT tool, this thesis will limit itself to the preparatory work required to commence the development. This includes the empirical work investigating the applicability of the results of the in-depth study, as well as the representation of organisational hierarchies, and artifacts and traces. All the other required research has been performed by the preceding in-depth study.

The development of the RT tool is primarily intended to provide an illustration of how the functionality of an RT tool can alleviate the task of implementing RT in a software development project. Consequently, the scope of the development is narrowed down to an evolutionary prototyping technique, which allows the development to illustrate the concepts related to requirements traceability without having to invest effort on technical details not of interest to the field of requirements traceability. By creating an evolutionary prototype, a foundation is built from which a complete RT tool can be developed.

In addition to the technical details not of interest to RT, it falls outside the scope of the RT tool to teach its users the elementary processes of software development, such as requirements elicitation. The tool is only intended to assist with the gathering, maintaining and analysis of trace data, thus requiring of its users some knowledge of elementary software development processes. However, knowledge of RT shall not be required to use the system.

2.7 Report Outline

In this section, an outline of the report is given, describing each of the following parts.

Preparatory Work - This chapter describes the results of the preparatory tasks and investigations executed due to mandates prescribed in the in-depth study. These results are required for the development of the RT tool, and thus provide a foundation for the remainder of the thesis.

Requirements Specification - In this part, a precise and detailed specification of all requirements to an RT tool is given. The document targets developers of the tool, but any stakeholder with invested interest will also find the document useful.

Architectural Description - This part documents the architecture of an RT tool, a tool providing traceability of requirements by assisting with the elicitation, quantification, and evolution of the requirements, as well as relating the requirements to other important artifacts of the software development processes, such as system components and verification procedures.

Prototype Development - The purpose of this chapter is to provide an example of how an RT tool could be implemented, based on the requirements specification and architectural description. An evolutionary prototype has been developed, and is described and evaluated in this part.

Evaluation & Discussion - This part evaluates and discusses the results presented in the previous parts. The aim of this part is to validate the results, and point out any weaknesses and strengths of the results.

Conclusion & Further Work - This part offers a conclusion based on both the presented results, as well as the evaluation and discussion of these results. In addition, any further work is described.

Chapter 3

Thesis processes and methods

This chapter presents the processes and methods employed in this thesis, providing a framework for how the work of the thesis is to be conducted.

3.1 Methodology

The activities of this thesis are primarily concerned with the development of the prototype of the RT tool. However, some preparatory work is included, adding an element of research to the software development. The development itself will be done as evolutionary prototyping, an iterative approach to software development. As evolutionary prototyping shifts between the activities of software development, an agile approach is opted for, abandoning the classic waterfall model [Vli00a]. However, the required research must take place before the development iterations can commence, leaving us with a process as shown in Figure 3.1.

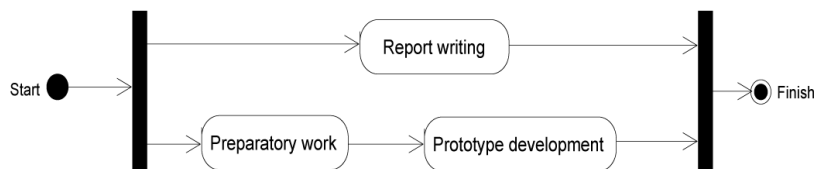


Figure 3.1: Thesis processes

The research and the development processes as a whole are sequential, and performed in parallel with the report writing activity, as shown in Figure 3.1. The development process can be decomposed into four activities, as shown in Figure 3.2. These four activities are performed iteratively, in conformance to the iterative nature of evolutionary prototyping. Testing the prototype will often cause additions or changes to the requirements, causing a new iteration of the development process. When a complete RT tool has been developed, satisfying all tests, the development process is exited.

3.2 Preparatory work

The preparatory work consists of empirical work in the shape of interviews, and detailed investigation of specific findings in the in-depth study [Nor06].

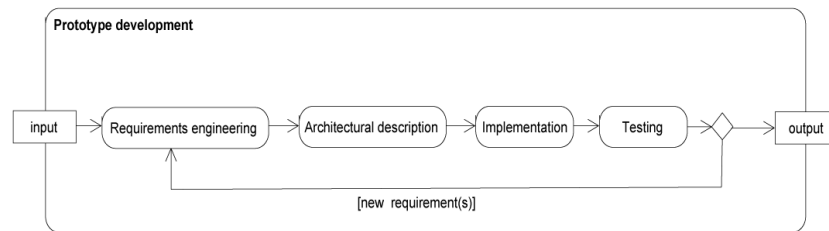


Figure 3.2: Development process

Interviews

The objective of the interviews are to gather expert opinions of the applicability of the results presented in the in-depth study. The interviewees were chosen based on their experience within software engineering and requirements traceability, opting for subjects with extensive experience.

The interviews were prepared by determining which information was needed to gather, and preparing a procedure for gathering this information. This procedure is recounted below, indicating active the participant(s). The interviewees are informed of the purpose of the interview when invited to participate, enabling them to decline the invitation if they do not wish to participate, as well as prepare for the interview session.

- Introduction on the context and purpose of the interview - Interviewer
- Brief presentation of requirements traceability - Interviewer
- Narration on previous experience with requirements traceability - Interviewee
- Brief presentation of traceability models - Interviewer
- Discussion on applicability of traceability models - Interviewer & Interviewee
- Suggestions on changes to traceability models - Interviewee

The interviews are to be conducted in an informal manner, and any divergence from the original procedure is welcomed, as it could lead to important findings otherwise left unnoticed. The interviews are conducted face-to-face, and are estimated to last approximately 1.5 hours. During the interviews, notes are taken. These notes are subsequently analysed, and the results will be presented anonymously, in Chapter 6.

Investigation of previous findings

The investigation of previous findings looks deeper into aspects only slightly touched by the in-depth study. It is primarily a literature study, but it also focuses on how the findings of the study can be applied when implementing an RT tool. The literature study is a narrow study, as its boundaries are determined by the in-depth study. Unlike the in-depth study, an agile and iterative process is not important to ensure the quality of the study. Consequently, the entire preparatory work is executed before the prototype development commences.

3.3 Prototype development

The prototype development is an iterative process, consisting of several sub-activities, which are executed in sequence. At the end of the sequence, the need for another iteration is evaluated by considering the requirements to the prototype. If any requirements are added or altered, a new iteration is required. However, this thesis will not perform more than a single iteration, as this serves to illustrate how an RT tool can solve the problems of implementing requirements traceability.

Chapter 4

Summary

Due to the dynamic nature of requirements engineering, changes are an inevitable part of software engineering. This has introduced the field of requirements traceability, providing an instrument for controlling how this dynamism influences the software engineering processes. Requirements traceability requires access to large amounts of information regarding the artifacts of software engineering processes, which renders the implementation of RT in a software development project difficult, due to a potentially significant increase in the work overhead.

The research agenda of this thesis is to research how an RT tool should be implemented to reduce the work overhead associated with implementing RT in a software development project. A requirements specification and architectural description shall be presented, concluding with the development of an evolutionary prototype, intended to give its users an impression of the functionality of the designed RT tool.

Due to the evolutionary prototyping technique employed in the development of the prototype, the development process of the project will be agile and iterative. In addition to the development process, the thesis has two other main processes, preparatory work, and report writing. The preparatory work is performed prior to the prototype development, and report writing is performed in parallel with the other two processes.

Part II

Preparatory Work

Chapter 5

Introduction

The in-depth study pointed out future work that should be tended to by the Master's Thesis ([Nor06], Chapter 25). The primary task was the development of an RT tool providing full traceability of requirements. However, before this development could commence, preparatory tasks and investigations had to be done. The results of these activities are presented in this chapter.

5.1 Purpose

The purpose of this part is to describe the results of the preparatory tasks and investigations executed due to mandates prescribed in the in-depth study. These results are required for the development of the RT tool, and thus provide a foundation for the remainder of the thesis.

5.2 Scope

The following chapters describe the results of those specific tasks pointed out by the in-depth study, i.e., the empirical work performed to ascertain the applicability of the results presented in the in-depth study, research regarding the representation of stakeholder authority and power interrelations, and a specification of how Planguage is to be employed to represent the artifacts of the traceability models internally.

This part does not look deeper into the primary task pointed out by the in-depth study, i.e., the development of an RT tool. The work related to this task is referred to parts III through V.

5.3 Overview

Here we give an overview of the structure and a quick summary of all chapters in this part.

- **Chapter 6 - Applicability of results from in-depth study**

This chapter discuss the results of the empirical work, in addition to any changes these results have brought about. In addition, the chapter presents a remodelled version of the traceability models in the in-depth study, a remodelling caused by the empirical work.

- **Chapter 7 - Representation of stakeholders and their organisational roles**

This chapter discusses the relationships that can occur in groups, referred to as group dynamics, and looks deeper into how these relationships could be represented in an RT tool.

- **Chapter 8 - Internal representation of artifacts and traces**

In this chapter, the internal representation of each artifact in the traceability models is determined, using Planguage. In addition, the representation of traces are discussed.

- **Chapter 9 - Summary**

A summary of the results presented in this part.

Chapter 6

Applicability of results from in-depth study

As a foundation for the development of an RT tool, an in-depth study was conducted, researching the field of requirements traceability. Preparatory to the work of the thesis, the in-depth study proposed a set of traceability models, modelling the problem domain, and a high-level requirements specification, listing the issues of requirements traceability an RT tool must address. In addition, the in-depth study presented a checklist of key focus areas that should be considered when deciding to implement RT in a software development organisation. This checklist is not further addressed by this thesis.

The in-depth study concluded by presenting the required further work. This work included the need to investigate the applicability of the results from the in-depth study in practice. These results are based upon compilation of literature studies and surveys of existing RT models and tools, and are consequently only connected to reality through the empirical work of other authors, work which might have been conducted several years ago and thus do not necessarily depict the current situation in the industry.

In order to achieve an updated and direct connection to the industrial practices, experiences, and needs, we have interviewed experienced software engineers, mapping the experiences and best practices of the industry, described further in Section 6.1. The gathered experiences and best practices were used to gain a deeper understanding of the current situation in the software engineering industry, and further to evaluate the applicability of the results from the in-depth study, described in Section 6.2.

6.1 Industrial experiences

The interviewees were two software engineers with several years of experience with software projects and hands-on experience from traceability of requirements. They were interviewed separately, making them independent sources of information.

The first interviewee has several years of experience from large consulting houses, working as an IT architect in the majority of these years. After working on projects for one of her employers, she and her colleagues discovered that they needed a tool that could trace requirements, in particular change requests to requirements during the project life cycle. One of the primary drivers for acquiring such a tool was to maintain control over the economic aspects of the software engineering process.

The IT consulting industry revolves around developing and delivering software solutions to customers (henceforth referred to as acquirers), their primary sources of income. In many cases, the acquirer provides a requirements specification, from which the consulting firm estimates its quotation. More often than not, the acquirer will request changes to this requirements

specification throughout the project life cycle. In such cases, requirements traceability, both horizontally and vertically, were found to be a valuable tool, assisting with the task of keeping the change requests under control. The interviewee related that her employer decided that they required a change request traceability system. When looking for existing tools, they discovered no suitable applications nor applications with similar functionality, and consequently they had to develop their own in-house application.

The second interviewee, also with several years of experience, came from a smaller consulting firm, but could still relate his experiences from a large project with approximately 2400 requirements. Due to the sheer amount of requirements, they decided that some sort of traceability would be required, as no human could maintain an inventory of the requirements otherwise, and as a result requirements could easily be lost. In addition, traceability would be useful during the acceptance tests, simplifying the process of proving to the acquirer that all requirements had been fulfilled.

Consequently, they implemented a custom-made web application for registering the requirements and some additional data (i.e., requirement ID, project ID, stakeholders, functionality, type, risk levels, workflow status and a regular status), persisting this data in an underlying database. The requirements were then sorted based on which system module they described, which later proved helpful when linking the requirements to models and tests. Both models and tests were described in other systems, and consequently, the traceability web application was extended to include references to these artifacts. Representations of the requirements were also included in the other systems. However, when requirements changed throughout the project life cycle, as they generally do in practice, problems with determining the prevailing requirement version were discovered. This was caused by the difficulties of propagating changes throughout the collection of systems holding a representation of the requirements.

6.2 Evaluation of results

In addition to relating their experiences with requirements traceability in practice, the interviewees also reviewed the traceability models from the in-depth study, elaborating on how they could be applied in practice. Their remarks and suggestions are related below.

6.2.1 Using the traceability models as a vehicle of communication

When communicating with the customer throughout the project, formal procedures such as minutes of meetings are important, as they ensure agreement on decisions reached, keeping them from passing into oblivion, and ensuring support from all stakeholders. An interviewee suggested to use the traceability models as a supplement to these formal procedures. The models form a framework for what information should be kept, and when maintained, they can be used as a vehicle of communication between the various stakeholders, enabling them at all time to review the current situation of the project. This will give all interested parties the opportunity to express any deviating opinions, which in turn ensures agreement upon reached decisions.

6.2.2 An alternative application of the pre-RS traceability model

In the consulting industry, projects usually take on one of the following two shapes; either the acquirer provides a ready-made requirements specification, upon which a solution is developed, or the acquirer requires help with making the requirements specification. It is suggested by one of the interviewees that the pre-RS traceability model, focusing on the processes occurring when making the requirements specification, is particularly of interest to the acquirer.

The interviewees relate that contact between the acquirer and those developing the solution often consists of just one or a few contact persons, thus shielding the acquirer's organisation from those developing a solution. These contact persons often hold high decision-making authority

within their organisation, enabling them to make impromptu decisions without consulting with other stakeholders in their organisations. As the given requirements specification is often poorly defined, new decisions will often have to be reached throughout the project. This in turn could cause that the solution does not reflect the needs nor opinions of all the stakeholders in the organisation, and can potentially cause dismay when presented to its end users.

It is often the contact person(s) who is held responsible for project success in the acquirer's organisation. Consequently, he or she will attempt to ensure support throughout the acquirer's organisation, and it is suggested that the pre-RS traceability model could assist with this. By employing the model and gathering the required traceability information, the contact person can legitimate the reached requirements, thus gaining support. In addition, employing the model drives the inclusion of stakeholders from the entire acquirer's organisation, thus ensuring that the end users are given the possibility to influence the end result. Further, it is suggested that any organisation could benefit from external assistance with the process of employing the pre-RS traceability model to reach a requirements specification, an assistance that could be delivered by the consulting firm. This will also prove beneficiary for the consulting firm, as it will assure the quality of the provided requirements specification.

6.2.3 The need for information

The interviewees places emphasis on the need for information, in particular focusing on that the gathered traceability information should not exceed the employed traceability information, i.e., that no more information should be gathered than what is useful to the project. The amount of instances of artifacts to trace between will increase exponentially when traversing the model, from fewer artifacts when dealing with pre-RS traceability, to more artifacts when dealing with post-FRS traceability. Serious thought must be given to the granularity of the traces, i.e. how deep should the traces travel. This will vary depending on the project at hand, and a pragmatic attitude must be maintained towards the required amount of traceability information. The generalised artifacts and the possibility to extend or reduce the models help with the process of acquiring precisely the required information. Section 8.2 addresses the issue of trace granularity in detail.

6.2.4 Challenges in modern IT development

The introduction of agile development, and project administration methods such as *Scrum* [RJ00] [BDS⁺00], have brought with them a new need for requirements traceability. This paradigm is highly dynamic, with a high level of influential force throughout the project life-cycle. This means that requirements often will be altered or removed, and new requirements will surface. The processes are informal by nature, making it difficult to find the sources of requirements at a later stage. Employing a requirements traceability scheme will assist with keeping the dynamism under control, but it is important that this is accomplished without inhibiting it. The proposed traceability model is highly adaptable with its generalised artifacts and non-existing restrictions on its environment. The only requirement the model imposes on its environment is the gathering and maintenance of the required traceability information. It says nothing of how this is to be accomplished, and can thus coexist with any project administration method. The task of specifying how the traceability information is to be gathered and maintained falls in the hands of the RT tool, encapsulating the model with constructs suitable for use in the current environment.

6.3 Alterations in TRACY

The information gathered through the empirical work, described in the previous sections, brought about some changes to the traceability models of TRACY. In the following sections, changes to each of the submodel are described, along with a discussion on why the changes

were deemed necessary. In general, the changes were implemented to increase the applicability of the models in real software development projects, i.e. enhancing their reflection of real circumstances.

The old versions of the traceability models can be found in Chapter 15.1 in the in-depth study [Nor06], Figures 15.1 through 15.3.

6.3.1 Pre-RS traceability submodel

The remodelled pre-RS traceability submodel is shown in Figure 6.1.

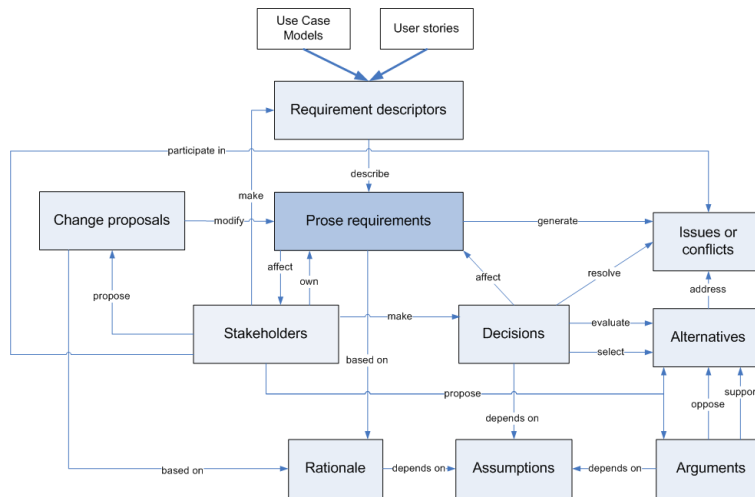


Figure 6.1: Pre-RS Traceability Submodel

The most profound change from the old pre-RS traceability submodel concerns the Stakeholder artifact. Previously, this artifact was only connected to the prose requirements and change proposals. This was found to be inadequate, as it did not illustrate sufficiently the diversity of a stakeholder’s role in the processes resulting in prose requirements. Thus, several new traces were added, connecting a stakeholder to any artifacts her or she are interested in. In particular, a stakeholder is now connected to the Prose requirement artifact via two traces, one registering the “propose” relationship between a stakeholder and a requirement, the other registering a “affect” relationship, signalling that a stakeholder is affected by a specific requirement. The intent behind this trace is the possibility to register how stakeholders can be connected to requirements other than by proposing them (e.g., by designing solutions that fulfil them, implementing these solutions, etc.).

Another important change is the introduction of a generalisation, *requirement descriptors*. This allows for several types of requirement descriptors to be registered, rather than simply just use case scenarios, as in the old version of the model. The Requirement descriptor artifact is also connected to the Stakeholder artifact by a “make” trace.

6.3.2 Pre-FRS traceability submodel

The remodelled pre-FRS traceability submodel is shown in Figure 6.2.

The in-depth study separate between functional and non-functional requirements, and this is repeated by the previous version of the pre-FRS traceability submodel. The remodelled pre-FRS traceability submodel has been adapted to the regulations enforced by IEEE, in Std. 830-1998 [IEEmb]. This standard describes how to specify requirements, and distinguishes five categories of requirements rather than the previous two. The five requirement categories are external interface requirements, functional requirements, performance requirements, design constraints and software system attributes. In addition, Std. 830-1998 employs a generic

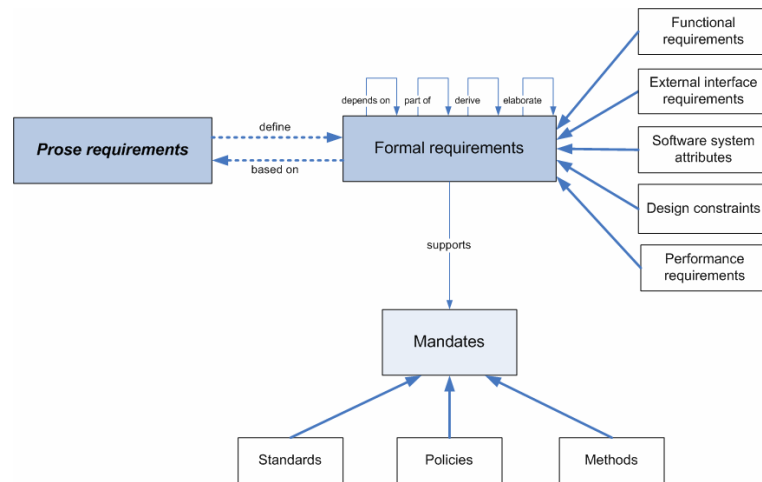


Figure 6.2: Pre-FRS Traceability Submodel

category for requirements that do not fit in any other category. The traceability models classifies these requirements as prose requirements, and employs no generic category among the formal requirements.

The classification of requirements is based on IEEE's regulations, but Planguage will also exert influence on the traceability models. The internal representation of the artifacts will be achieved with the use of Planguage attributes. This is further discussed in Chapter 8.

6.3.3 Post-FRS traceability submodel

The remodelled post-FRS traceability submodel can be found in Figure 6.3.

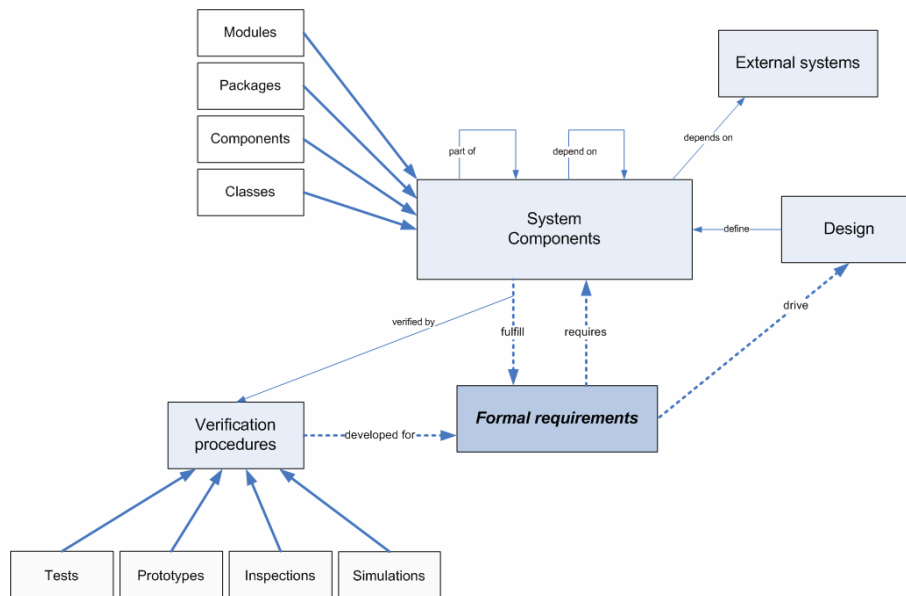


Figure 6.3: Post-FRS Traceability Submodel

Most importantly, the Function artifact has been removed from the post-FRS traceability model. The Planguage attribute *Tag* allows for a hierarchic classification of artifacts by establishing a structure within the tag. This technique is commonly used when determining package names within i.e. the Java programming language. Each artifact is assigned a tag, and this tag potentially consists of elements of other tags. If two requirements describe the same module

within a system, their tag will start with the tag of the module, and append a requirement-specific tag (e.g., *Module1.Requirement1* and *Module1.Requirement2*). Thus, the function of a requirement can be found by analysing its tag. The first parts of the tag describes the function of the artifact, thus enabling the user to quickly determine the context of the artifact.

Secondly, the System components artifact has been decomposed into a number of component types, such as modules, packages, and classes. This enables the user to control the granularity of the traceability, as discussed in Section 6.2.3. This granularity control is further discussed in Section 8.3.3.

Chapter 7

Representation of stakeholders and their organisational roles

Stakeholders are almost always members of an organisation. An organisation more often than not consists of a intricate set of power relations due to the human nature of its members. Specifying requirements can be an unpredictable affair, as discussed in detail in Chapter 7 in the in-depth study [Nor06]. Stakeholders with opinions and requirements that are important to the system to be developed can be overruled by stakeholders with higher authority, regardless of the value of their proposals. The following sections discusses how power relations and their influence upon the stakeholders can be recorded by an RT tool and used to understand how and why the given system is developed.

7.1 Stakeholder hierarchy

There will often exist a hierarchy among the stakeholders, created by the amassed status and authority of the individuals. This hierarchy is based on several factors such as authority and technological expertise. When analysing the inclusion of some requirements and the exclusion of others, in addition to the priority given to specific requirements, it could be useful to have the opportunity to review the influence of the stakeholders, in particular with respect to their organisational position.

A stakeholder hierarchy will to some extent correspond to the organisational hierarchy of the stakeholders' organisation. However, there are some aspects that disturbs this hierarchy. First, it will frequently happen that not all stakeholders originate from the same organisation. Often a customer and a development team will cooperate on developing a solution, and the stakeholders will thus originate from two organisations. This complicates the task of unveiling an organisational hierarchy. Secondly, authority in the requirement specification processes is not only given by a stakeholder's position in the organisation hierarchy. It could also be influenced by the stakeholder's technological expertise. A stakeholder with vigorous technological expertise could potentially overrule any stakeholders with lesser expertise when discussing technical questions. Personal attributes could also influence the amount of authority held by an individual, attributes such as presence, vigour and drive, even attributes such as age, sex, and race could unfortunately influence the authority of an individual. In other words, the field is delicate, with many aspects that must be considered carefully.

7.1.1 Status networks

[For06a] employs the concept of status networks when describing structures in groups. A status network is a stable pattern of difference in authority and power, and could also be described as a chain of command. A status network describe a group structure where some group members

amass a higher level of authority and power (status) than others, thus creating a hierarchy sorting group members based on their status.

No severe difficulties are encountered when trying to *represent* a status network or hierarchy. Hierarchic representations are common in science, and an RT tool could from a technical viewpoint easily arrange stakeholder artifacts in a predetermined hierarchy. The problem lies in unveiling the stakeholders' relative status, and determining the sorting attributes that decide how the hierarchy is arranged. This requires extensive knowledge of the field of group dynamics, and the following sections look at some of the research in this field, particularly looking into how group members influence each other.

7.2 Group dynamics

A group is defined by [For06b] as *two or more individuals who are connected to one another by social relationships*. Requirements engineering is a highly social process [Gog93], as discussed in Chapter 7.3 of the in-depth study [Nor06]. Consequently, the participants of these processes are members of a group, connected to each other by social relationships. When determining a status network within this group, it is these social relationships that must be unveiled.

The social relationships seldom remain unchanged throughout the lifecycle of the group. [Tuc65] describes how a group pass through five stages during its development; the *forming* stage (formation of the group), the *storming* stage (conflicts arise on the group), the *norming* stage (norms are developed), the *performing* stage (the group becomes productive), and the *adjourning* stage (the group is dissolved). Throughout these stages the individuals constituting the group, in this context henceforth referred to as stakeholders, establish and maintain a status network, which evolves just as the group itself evolves.

From a requirements traceability perspective, a stakeholder's role in a group or organisation is important for determining why requirements are included or excluded. Stakeholders are profoundly influenced by the group they belong to [For06b]. If one wishes to understand the behaviour of individual stakeholders, it is necessary to understand the group they belong to, and how the group's members influence each other. As the social relationships in according to [Tuc65] changes throughout the lifecycle of the group, requirements traceability is required to update any information relating to a stakeholder's position within a group.

7.2.1 Majority and minority influence

[For06c] introduces the concepts of majority and minority influence. Majority influence occurs when a majority of group members fronts a specific solution, a solution that not necessarily represent the best option. The remaining minority stands little chance of resisting the opposition, and experiments have shown that the minority is most likely to conform to the majority's choice.

Usually a minority resisting the choice of the majority will not waver the opinion of the majority. The minority will in such cases be ignored or even ridiculed. However, occasionally the majority will listen to the minority. This is most likely to occur when the minority argues its case with consistency, when the minority consists of group members with credibility, and when the minority is growing. Persuasion, loosely defined by [O'K02] as "a successful intentional effort at influencing another's mental state through communication in a circumstance in which the persuadee has some measure of freedom", is an important success factor for minority influence. A group member that *converts* from supporting the majority to supporting the minority demonstrates a stronger will than a group member *complying* with the majority's choice, as conversion requires an inner acceptance caused by persuasion rather than simply acquiescence.

Understanding majority and minority influence is important for analysing how requirements were chosen. An observer could be able to spot if the requirement resulted from major influence or minor influence, and an RT tool could in turn allow the observer to tag the requirements, thus storing this information for future use. An RT tool can not directly influence the processes

of decision making, and the observer's task of detecting the true influence behind a decision is difficult, as it delves into the field of human psychology. But if the required observations are present, the RT tool can provide a mechanism for attaching the observations to the artifacts for future reference. A requirement accepted due to majority influence could prove dangerous in the long run, as important opinions of the minority drown in their compliance. Thus, such requirements potentially represent a greater risk than requirements accepted due to minority influence, and flagging a requirement as "majority-accepted" could be helpful as an aid in risk management. Minority-accepted requirements often represent a source of innovation, spotting new and previously unknown issues regarding the system to be developed, and flagging requirements as "minority-accepted" could influence e.g. their assigned priority.

7.2.2 Pressure to conform

Groups traditionally meet face to face, and are thus faced with immediate scrutiny from others and public evaluation. This is commonly understood to amplify conformity among group members [For06c]. Requirements engineering is an inherently social process, thus it is difficult to imagine the activities of this process to take place in a situation where participants are separated by time and place. However, this might occasionally be the case, in particular with the increasing focus on outsourcing of software development [out03] [HQ95]. Computer-Supported Cooperative Work [CS99] addresses the issues concerning group interaction through computers, and can be of assistance in situations where requirements engineering takes place in locations separated in both time and geography. An RT tool supporting cooperative work can prove valuable, helping the participants in keeping control over contributions. This is especially true if the tool leaves an auditing trail of contributions.

It has been suggested that CSCW reduces the pressure to conform to the majority's opinion. However, research has shown that this alleged reduction in conformity pressure does not exist. [SPLW02] proposes that people often conform more when interacting through a computer-supported medium. This is suggestively caused by depersonalisation of group members, thus increasing focus on the group identity, which in turn increases the conformity of the group. Consequently, an RT tool designed for CSCW-specific purposes will not necessarily reduce or remove the pressure to conform within the group applying the tool.

Thus, the desire to conform is difficult to avoid, and detecting when requirements are included or excluded due to conformity remains an important task. It is a task that requires human effort, a task that cannot be automatically solved by the RT tool. The concluding section of this chapter looks deeper into how the problem of conformity should be recorded by the RT tool, in addition to discussing how status networks should be determined within a group or organisation.

7.3 Determining stakeholders' roles

The objective of determining stakeholders' roles in the requirement engineering processes is to be able to analyse why and how the given stakeholders became those that directed the proposed solution. Detecting this "why" and "how" is important for providing qualitative pre-RS traceability, and could, as pointed out by an industrial representative (see Section 6.2.2), become important to those customer representatives seeking to justify the end result within their own ranks.

The RT tool should thus store information on the status (position in the status network) of the stakeholders, in addition to tagging requirements as results of either majority or minority influence. Both these pieces of information could prove difficult to ascertain, in particular the status of the stakeholder, as this status could be gained by different means, not only a organisational position. However, it is not the task of the RT tool to gather this information, it is even beyond the tool's abilities to do so, as such a task is complicated and requires extensive human effort, such as observations and interviews. The RT tool is left the responsibility to register and preserve the information for future analysis.

Tagging requirements as minority or majority-decided could be considered a question of granularity. The tool could (applying a coarse granularity) for instance register that *requirement A was determined by a majority among the proposing stakeholders*, trace information with little detail and consequently little value. If more detail is required, the tool could (with a fine granularity) register that *requirement A was determined by a majority consisting Stakeholder 1, 2, 3, and 4, where Stakeholder 3 and 4 only complied with the majority decision*. This latter version provides more detail and thus a better traceability, enabling the user to determine the roles of the various stakeholders. However, the RT tool must remain adaptable in order to ensure the user's freedom when employing the tool, and should thus leave it to the user to determine the level of detail in the registered information.

The tagging of requirements as minority or majority-decided is accomplished using the Decision artifact of the pre-RS traceability submodel. As explained in Section 8.3.1, the Decision allows the user to register how a decision was reached and which stakeholder made it. This information could then be analysed, providing the analyst with important insight in the power relations within a software development project.

When determining a stakeholder's role in the group or organisation, it is natural to attempt to ascertain the stakeholder's authority and power within the group or organisation. As mentioned earlier, this authority and power can be caused by different factors, not only the stakeholder's official position. Still, it remains outside the scope of the RT tool to provide a means for determining the status of stakeholders. Only a means for registering and preserving this information is required, leaving the comprehensive task of investigating group status networks to humans. Registering stakeholder status is accomplished by adding attributes to the internal representation of the Stakeholder artifact that addresses the position and authority of the stakeholder, explained further in Section 8.3.1.

The RT tool should thus be able to describe a stakeholder's authority and power, and use this attribute to place the stakeholders in a reciprocal hierarchy, thus illustrating which stakeholders have more authority and power than others. This could at a later point be employed to discovering why certain stakeholders' requirements were chosen above others.

Chapter 8

Internal representation of artifacts and traces

Chapter 6 presents three traceability submodels, which when combined provides full traceability. These models consists of artifacts and traces between these artifacts. However, discussions concerning the internal representation of artifacts were deferred to this chapter, and will consequently be discussed in the following sections.

8.1 Application of Planguage

The in-depth study determines that Planguage, a planning language proposed by Tom Gilb, and its subset the Requirements Specification Language, shall be employed for specifying the requirements in a formal manner. As discussed in the in-depth study, requirements are often proposed in a prosaic manner at first by the non-technical stakeholders (i.e., the customer). This is captured in the traceability models, separating between *prose requirements* and *formal requirements*. Planguage will first and foremost be applied to transform the prose requirements into formal requirements, applying quantifying attributes that assist in making the requirements measurable and testable, as discussed in the in-depth study. This is accomplished by employing specific attributes that establish the foundations for judging whether a requirement is fulfilled by the proposed solution. This is discussed further in Section 8.3.2. These attributes are referred to as the *core representation* of the requirements.

Secondly, Planguage contains attributes that are originally intended to be used to specify the requirements further, such as the *Stakeholder*, *Rationale*, and *Priority*. These attributes are applied to specify, amongst others things, the origin of the requirements, invested interest, and decision-making authority, and are collectively referred to as the *added requirement representation*. However, the traceability models have created additional artifact categories (henceforth simply referred to as artifacts) that capture these specifications, such as *Stakeholder*, *Decisions*, and *Rationale*, connecting them to each other and the requirement artifacts (i.e., the core representation) with the use of traces. Thus, the attributes that constitute the *added requirement specification* are either contained within these additional artifacts, or represented by the use of traces between artifacts. The following sections elaborate this topic further by presenting the artifacts of the traceability models and how they should be internally represented. However, the issue of traceability granularity is addressed first, discussing how the model must allow the user to decide the desired granularity.

8.2 Traceability granularity

Section 6.2.3 pointed out the issue of traceability information granularity. When gathering traceability information, it is important to do so with a pragmatic approach. Spending time and effort gathering traceability information that will never be used is pointless, and the internal representation of both artifacts and traces must leave room for the user to decide the desired granularity of traceability.

Each artifact is represented with the means of Planguage attributes, and by leaving non-identifying attributes optional, the user is allowed to determine the level of detail included in the model. However, this only addresses the internal granularity of each artifact. Granularity can also be found in the hierarchies of artifacts, and in the traces connecting artifacts.

By creating a hierarchy of artifacts, the model allows the user to determine how far down the hierarchy he or she wishes to trace. If fine granularity is deemed necessary by the user, additional hierarchy levels can be added. The System component artifact of the post-FRS traceability submodel exemplifies an artifact hierarchy, where additional hierarchy levels can be added to allow the user a finer traceability granularity. The System component artifact is addressed further in Section 8.3.3.

The granularity of traces is determined not only by the artifacts they connect (i.e., where in the hierarchy the artifacts reside), but also by the auxiliary trace information included. The concept of auxiliary trace information is further explained in Section 8.4, and by allowing the user to determine which information is required, a finer or coarser granularity can be achieved when desired.

Overall, varying traceability granularity is primarily achieved by allowing the user to determine the required level of gathered trace information, i.e. by not imposing any restrictions on the user other than the gathering of identifying trace information, information that assists with uniquely identifying artifacts and traces. Such information (i.e. identification numbers, tags, etc.) cannot be left optional, as this would render the model useless for requirements traceability purposes.

8.3 Internal representation of artifacts

Appendix B contains an alphabetical list of the attributes discussed in the following sections, providing a short description of their purpose and contents. The reader is referred to this appendix when looking for further detail on the attributes and how they are used.

Each artifact contains an ID attribute, providing a unique identification of the artifact. The ID attribute coexists with the tag attribute, which in turn provides useful meta-information in addition to serving as an identification means.

8.3.1 Pre-RS Traceability Submodel

The pre-RS traceability submodel presents a large number of artifacts, the submodel with the most artifacts. Pre-RS traceability focuses on capturing the actions and processes that lead to a requirement, and the artifacts reflect this. A short summary of the artifacts is given in Table 8.1. In the following sections, a discussion of the internal representation of each of these artifacts will be presented, concluding how to represent the artifact in question.

Prose Requirements

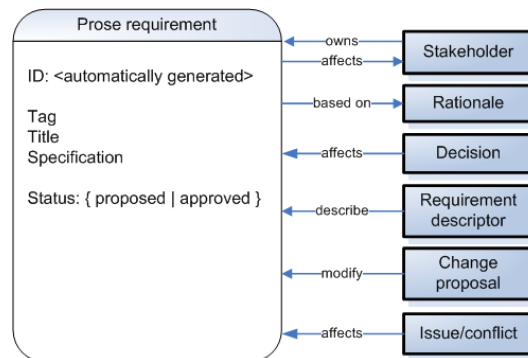
A prose requirement is a small piece of text that describes a requirement, often lacking in detail and precision. This is due to the human aspect of software development (discussed in detail in

Table 8.1: Artifacts in the pre-RS traceability submodel

Artifact	Description
Prose Requirements	The prose specification of the requirement.
Stakeholders	Information on stakeholders of the system under development.
Change proposals	Proposals for changes made by stakeholders.
Rationale	The underlying rationale behind prose requirements.
Requirement descriptors	Documents, models, etc., describing the prose requirements.
Decisions	Decisions made during the process of specifying requirements.
Assumptions	Assumptions that decisions are dependent upon.
Alternatives	The alternatives between which a decision-maker chooses when making a decision.
Arguments	The arguments supporting and opposing alternatives.
Issues or conflicts	Problems arising when specifying requirements, problems that are solved by making decisions.

the in-depth study), causing modern software development methodologies to assume an agile and iterative nature.

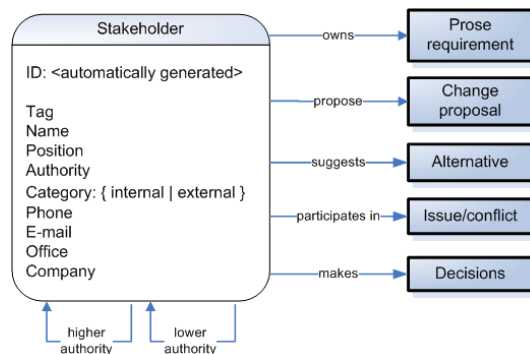
The prose requirements will be transformed into formal requirements at a later stage, and consequently the internal representation of this artifact is minimal compared to a formal requirement, only providing an id, title, status, and prose specification of the requirement. The Prose requirement artifact is one of the most important artifacts of the pre-RS traceability submodel, and the id, title, status, and prose specification alone is not sufficient in providing full traceability. However, the other artifacts of the submodel, and the important traces between these artifacts capture the necessary information for providing full pre-RS traceability. The internal representation of these artifacts will be discussed in the following sections. Figure 8.1 presents the internal representation of the Prose requirement artifact, also illustrating how this artifact connects to other artifacts in order to provide full pre-RS traceability.

Figure 8.1: Internal representation of the *Prose requirement* artifact

Stakeholders

Stakeholders represent those people with an invested interest in the system, both as proposers of requirements, and as people being affected by the requirements. A stakeholder is connected to several of other artifacts in the pre-RS traceability submodel, depending on the stakeholder's degree of participation in the processes leading to a requirements specification. This is illustrated in Figure 8.2, along with the internal representation of a Stakeholder artifact.

The most important part of the internal representation of a stakeholder is its contact information. The primary objective of including a stakeholder as an artifact in the traceability model

Figure 8.2: Internal representation of the *Stakeholder* artifact

is to be able to contact this stakeholder for clarification if a requirement appears ambiguous.

Sorting stakeholders into categories depending on their connection to the project assists in the task of determining where the requirements originated, e.g. from the customer or the development staff. This is particularly helpful in dynamic development processes such as Scrum, where new requirements surface in a setting where it could prove difficult in retrospect to determine who proposed the requirement. Thus, by being aware that a proposer must be registered, such issues can be avoided.

Section 7.3 describes how the RT tool will be able to place stakeholders into a hierarchy (a status network) according to their amassed authority and power. The internal traces of the Stakeholder artifact (*higher authority* and *lower authority*), shown to the left in Figure 8.2, creates a status network by allowing a stakeholder to sort other stakeholders either below or above him or her in the hierarchy. In addition, the authority attribute allows the user to classify the authority of a stakeholder as either *low*, *medium*, or *high*, relating the decision-making abilities of the stakeholder will often prove valuable, as decisions made by certain stakeholders will have stronger impact.

Change proposals

Change proposals are suggested by stakeholders when they feel that an existing requirement is inadequate, and describe an alteration in this specific requirement. Figure C.1a of Appendix C describes the internal representation of a change proposal, in addition to the connecting artifacts.

A tag is as always required, in order to be able to identify the change proposal at a later stage. In addition, a description of the change proposal as well as a status indicator is required to model a complete change proposal. The status indicator relates whether the change proposal has been accepted by all stakeholders and consequently been included in the requirement. It is important to be able to tell which requirement the change proposal affects, and this is recorded by a trace between the change proposal and the affected Prose requirement artifact. The rationale behind the change proposal is recorded by a trace to a Rationale artifact, upon which the change proposal is based.

Rationale

A requirement is always based upon a rationale, and the Rationale artifact relates the underlying reasons for specifying the requirement in question. Figure C.1b of Appendix C presents the internal representation of the Rationale artifact, including the traces to other artifacts.

The Rationale artifact is quite simple, with only a tag and a specification of the rationale itself. However, the rationale behind a requirement can be based upon several assumptions, ranging from simple project-specific assumptions to external and possibly legislated constraints that

serve as assumptions and could consequently affect the choice of requirements. This is captured in the model by a trace between the Rationale artifact and the Assumption artifact. In addition, both the established prose requirements and any change proposals are based upon a rationale, captured by traces between these artifacts.

Requirement descriptors

Requirement descriptors are any documents, models and alike that describe the requirement in an alternative fashion, such as user stories or use case models. Figure C.1e of Appendix C depicts the internal representation of the Requirement descriptor artifact, and the traces that assist with providing full traceability.

In addition to the required tag, the other attributes of the Requirement descriptor artifact describes what kind of requirement descriptor the artifact represents, and where the actual artifact can be found. As requirement descriptors often are models or documents existing in specific formats, they cannot be fully represented as pure artifacts and should preferably be handled as objects. Consequently, a link to the descriptor location is provided in addition to the other artifact attributes, enabling the user to look up the descriptor. In addition to this location attribute, the type is captured in the artifact.

The stakeholders participating in the development of the requirement descriptors are captured by a trace to the Stakeholder artifact. Several stakeholders can be included in this process, as authors and contributing resources.

Decisions

Decisions are made throughout the process of specifying requirements, and they affect the resulting set of requirements, often pointing to specific requirements. Decisions choose between alternatives that address issues or conflicts between the stakeholders, and the stakeholders that make the decisions do this on the basis of specific assumptions. This is all captured by the traceability submodel, and the internal representation of the Decision artifact and its traces to other artifacts are depicted in Figure C.1c of Appendix C.

A context attribute is employed to describe the context in which the decision was made. The context can include any information regarded as useful, for instance whether a decision was made under minority or majority influence. In addition to the context, the Decision artifact registers a decision maker. This information in combination with the context can e.g. provide the user with knowledge of the power hierarchy of stakeholders. The decision is further elaborated by the trace to an Assumption artifact. In addition, a description of the actual decision is required. A decision affects a prose requirement, recorded by a trace to a Prose requirement artifact. A timestamp is also included, indicating the time at which the decision was made.

Assumptions

The Assumption artifact describes any conditions, constraints, or similar, that are assumed by the stakeholders. Reasoning behind requirements, such as rationale and decisions, could depend upon assumptions, and this is captured in the traceability submodel by adding traces between the Assumption artifact and the Rationale, Decision, and Argument artifact. The internal representation of the Assumption artifact is shown in Figure C.1h of Appendix C.

Alternatives

When a decision is made, an evaluation of alternatives is performed, and a selection of the most appropriate alternative is made. Figure C.1d of Appendix C depicts the internal representation of the Alternative artifact, along with its connecting traces.

The internal representation of the Alternative artifact is simple, with a tag and a description attribute. However, the traces connected to this artifact are many. Decisions evaluate and select alternatives, whilst arguments oppose or support them. Alternatives are proposed by stakeholders, and stakeholders with opposing alternatives often participate in a conflict or issue on the subject.

Arguments

The same stakeholders that propose alternatives will often simultaneously propose arguments that support their alternatives. These arguments can depend upon assumptions made by the stakeholder proposing them. Figure C.1g of Appendix C shows the internal representation of the artifact, along with traces to other artifacts.

Just like the Alternative artifact, the Argument artifact is simple in its internal representation, with just a tag and a description attribute. The description attribute provides a specification of the argument, and how it supports or opposes a specific alternative.

Issues or conflicts

The Issue/conflict artifact describes any situations that arise due to opposing forces within the group of stakeholders. Again, this is a simple artifact, it only provides a tag and a description of the issue or conflict. Additional important traceability information is captured by the traces between this and other artifacts, such as participating stakeholders, resolving decisions, and last but not least the prose requirement that spurred the conflict or issue. Figure C.1f of Appendix C depicts the internal representation of the Issue/conflict artifact, as well as traces to and from the artifact.

8.3.2 Pre-FRS Traceability Submodel

The pre-FRS traceability submodel's primary focus is on converting the prose requirements into formal requirements while ensuring traceability between these two categories of artifacts. Table 8.2 summarises the artifacts of the pre-FRS traceability submodel.

Table 8.2: Artifacts in the pre-FRS traceability submodel

Artifact	Description
Prose Requirements	A connecting artifact, repeated from the pre-RS traceability submodel.
Formal requirements	A formalised and quantified version of the prose requirements.
Mandates	Mandates placing constraints on how the formal requirements are shaped.

Formal requirements

The formal requirements are split into five major categories, as shown in Figure 6.2. The establishment of these categories were discussed in Section 6.3. The task of refining prose requirements into formal requirements is a task not only of sorting requirements into these five categories, but also managing the prose specification of the requirements. The prose specification must be made quantifiable, and for this we will use some of the attributes of Planguage. A basic set of attributes are applied in all categories of requirement artifacts, providing basic information on the requirement, such as its tag, type, version, status, priority and description.

In addition, traces connect the requirement artifacts to other artifacts, thus representing other attributes of Planguage, such as owner (a stakeholder owns a requirement), and rationale (a requirement is based on a rationale). This is accomplished in the traceability model by connecting the pre-RS and pre-FRS traceability model. Thus, to be able to find the stakeholders of a formal requirement, a backwards trace must be performed via the connected prose requirement(s) to the pre-RS traceability model.

A hierarchical relationship between requirements are represented by the concept of *internal traces*. An internal trace connects two instances of the same artifact, and the internal traces of the Formal Requirement artifact are illustrated in the pre-FRS traceability model, in Figure 6.2. As these internal traces are the same for all types of formal requirements, they are not repeated in the individual figures.

The two main categories of requirements are functional requirements (relating what the system should do) and non-functional requirements (relating how the system should perform its functions). The functional requirements can be divided into to subcategories: external interface requirements, functional requirements, and design constraints. Non-functional requirements is divided into the subcategories performance requirements and software system attributes. Figure 8.3 describes the internal representation of functional requirements, whilst Figure 8.4 describes the internal representation of non-functional requirements.

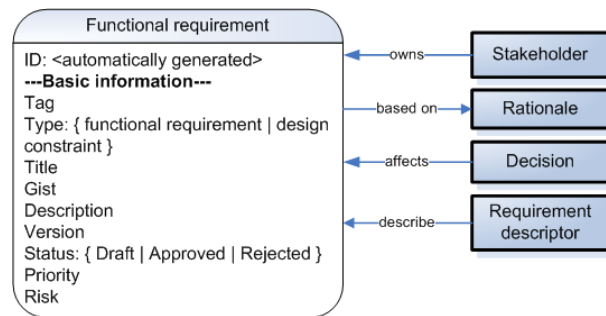


Figure 8.3: Internal representation of the *Functional requirement* artifact

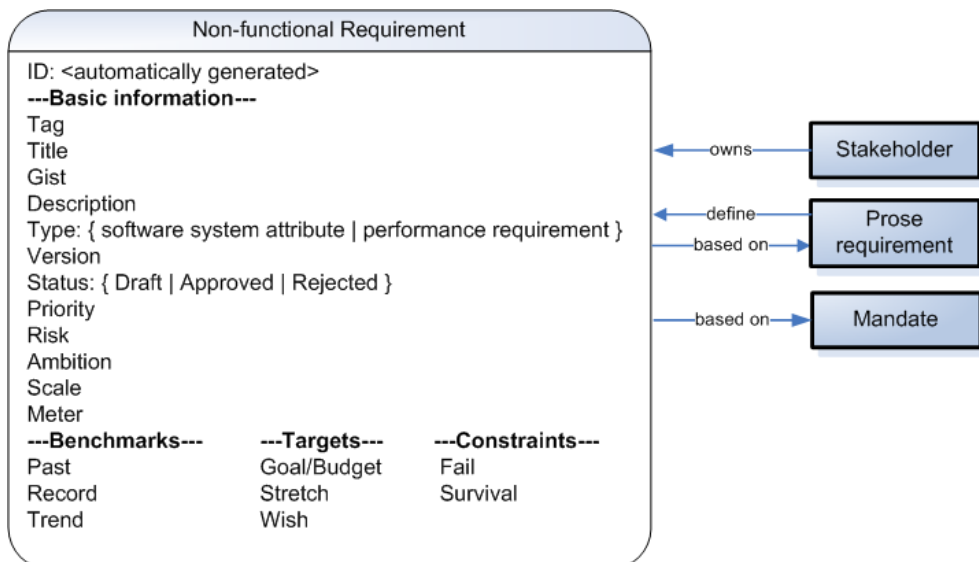


Figure 8.4: Internal representation of the *Non-functional requirement* artifact

The non-functional requirements could also be referred to as *scalar* requirements, as their presence in a system is measurable. In contradiction, functional requirements are either present or not; a *binary* circumstance. Measuring the fulfilment of a functional requirement is in comparison to non-functional requirements a relatively simple task, involving a check to see

whether the required functionality is present or not. Measuring the fulfilment of a non-functional requirement is more demanding, as such requirements can be partially fulfilled, bringing with them the need to ascertain whether the degree of fulfilment is satisfactory to the stakeholders.

Thus, the formalisation of non-functional requirements requires the specification of several levels and goals, which assist with measuring the fulfilment of the requirements. As shown in Figure 8.4, the attributes employed to enable measurement of a non-functional requirement are grouped into three main categories, *Benchmarks*, describing past numeric values of existing or similar systems, *Targets*, describing future numeric values of the system to be developed, and *Constraints*, describing any restrictions to the system to be developed. The numerical values of benchmarks, targets and constraints define attribute levels, creating intervals of acceptance and rejection, referred to as e.g. Goal, Stretch, Fail, and Survival. Thus, a scalar representation of the requirement and consequently a formalisation of a non-functional requirement is provided.

The five categories of requirements does not cover all existing requirements. As discussed in Section 6.3, IEEE Std. 830-1998 employs an additional generic category allowing the inclusion of requirements that do not fit into any of the other categories, such as resource requirements. These requirements (relating how much resources are available to the system and its development) could be considered a non-functional requirement type, but as they do not place any direct requirements to the system itself, only to the development of the system, they require a separate category. As concluded in Section 6.3, the Prose requirement artifact replaces the generic category of the IEEE standard, including resource requirements. An important aspect of the relationship between the non-functional and resource requirements is their contradicting goals. Non-functional requirements seek high performance and quality, which often come at a high price, whilst resource requirements attempt to keep costs at a level acceptable to the stakeholders. This is often referred to as a trade-off in system engineering, and the science of interlacing multiple resource and non-functional requirements in a balanced fashion falls somewhat outside the scope of this Master's Thesis. However, it is not entirely without purpose that the concept of requirement trade-offs have been brought up. By quantifying non-functional requirements, it is easier to calculate the costs of achieving those requirements. Thus, the process of balancing the requirements and achieving the best possible trade-offs is simplified by demanding quantification of non-functional requirements by employing the Planguage attributes.

Thus, only those requirements that can be partially implemented, i.e. non-functional requirements (scalar requirements), demands an extended use of Planguage attributes, in particular the sets of benchmarks, targets, and constraints. Binary requirements, those who are either fulfilled by a solution or not, require a smaller set of attributes, avoiding the specification of benchmarks, targets, and constraints. These requirements include functional requirements, design constraints and condition constraints.

Mandates

The Mandate artifact describes any methodologies, standards or policies that the formal requirements support. These superimposed guidelines can be put down by authorities within both the customer's organisation and the development organisation, or even by governmental authorities. Mandates often exist in a separate document, consequently only a referral to the actual document, its type, and a short description is necessary, in addition to a tag. Figure C.2 of Appendix C depicts the internal representation of the Mandate artifact.

8.3.3 Post-FRS Traceability Submodel

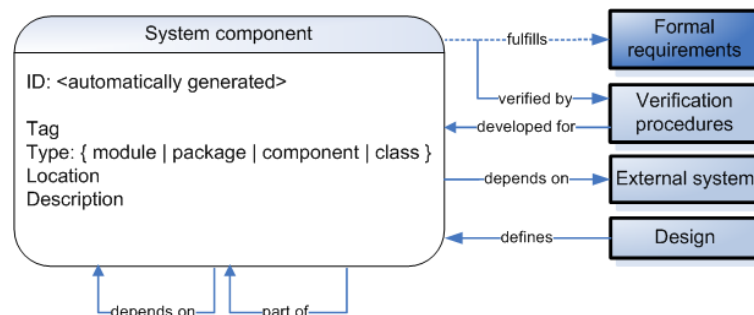
The post-FRS traceability submodel connects the formal requirement artifacts to any artifacts created during the process of developing a solution fulfilling the requirements. The artifacts are characterized by this, representing mainly tangible products of the software development processes, such as design artifacts, system components, and verification procedures. Table 8.3 summarises the artifacts of the post-FRS traceability submodel.

Table 8.3: Artifacts in the post-FRS traceability submodel

Artifact	Description
Formal requirements	A connection artifact, repeated from the pre-FRS traceability submodel.
System components	The key components of the system to be developed, such as packages, classes, components and modules.
Design	Artifacts within the design process, such as domain models and architecture.
External systems	Any external systems that the system under development will be dependent upon.
Verification procedures	Artifacts that verify that the requirements have been fulfilled by the system.

System components

The System component artifact relates to how the formal requirements are fulfilled by the system. The artifacts point to components in the system, such as packages, modules, and classes, thus linking requirements to the specific system component that fulfils it. The System component artifact holds a description of the component, and a reference to its location, as the actual component is located outside the traceability model, e.g. in an integrated development environment (IDE) [Env]. Figure 8.5 shows the internal representation of the System component artifact.

Figure 8.5: Internal representation of the *System component* artifact

In addition, a tag attribute identifying the component is employed, as well as a type attribute specifying the type of system component. System components exist in a hierarchy, where modules contain packages, and packages contain classes. The tag attribute supports the establishment of such a hierarchy. However, the granularity of the hierarchy must be carefully considered. When tracing from requirements to system components, one must consider how deep the trace is required to be. The deeper the trace (i.e., finer granularity), the higher load on all requirements traceability efforts, both gathering of trace information and storing and maintaining it. Carefully balancing the benefits and the required efforts, thus creating an acceptable trade-off, is an important aspect of requirements traceability. By employing a type attribute that can be set to different levels of the hierarchy, the model leaves to the user to determine the desired granularity of the traceability.

Design

The Design artifact introduces to the traceability model those artifacts that are a product of the design processes of the software development processes, such as architecture specifications, UML diagrams, and graphical user interface (GUI) designs. As with system components, the artifact

is stored outside the traceability model, with the Design artifact serving as a representation of the actual artifact. Thus, the attributes include a tag, the type of design, a short description of the design, and a reference to its location. The internal representation of the Design artifact is given in Figure C.3a of Appendix C.

External systems

More and more often, systems communicate with other systems in order to perform their tasks. The External system artifact represents those systems that the system to be developed will communicate with. The internal representation of this artifact includes a tag, a type describing what sort of system is represented, a owner attribute specifying the origin of the system, and a short description of the system and its purpose. Figure C.3b of Appendix C depicts the internal representation of the External system artifact.

Verification procedures

Verification procedures are employed to ensure the stakeholders that their requirements are fulfilled by the proposed system. Several verification procedures exist, and which of these that are employed will change from project to project. Consequently, the Verification procedure artifact contains a type attribute specifying some of the most common verification procedures, allowing the user to specify other procedure types. The verification procedure itself will be stored outside the traceability model, leaving a location attribute as a reference. In addition, a short description of the verification procedure is given. The internal representation of the Verification procedure artifact is given in Figure C.3c of Appendix C.

8.4 Internal representation of traces

The artifacts of the traceability models are connected by traces. A trace illustrates a relationship between two artifacts, either of the same type, or of two different types. Several kinds of traces exist, conveying the diversity of relationships between artifacts. Traces can occur in both directions between two artifacts, illustrating different relationships. In addition, several traces can cooperate and connect several artifacts of the same artifact category to a single other artifact.

A trace should in general be able to identify its two connecting artifacts, but occasionally it is also required to relate additional information describing the relationship further, referred to as auxiliary trace information. The following sections discuss the traces of each of the submodels, concluding how they should be internally represented.

System components exist in a hierarchy, where modules contain packages, and packages contain classes. The tag attribute supports the establishment of such a hierarchy. However, the granularity of the hierarchy must be carefully considered. When tracing from requirements to system components, one must consider how deep the trace is required to be. The deeper the trace (i.e., finer granularity), the higher load on all requirements traceability efforts, both gathering of trace information and storing and maintaining it. Carefully balancing the benefits and the required efforts, thus creating an acceptable trade-off, is an important aspect of requirements traceability. The required granularity will vary, depending on the nature of the software development project being modelled. By allowing the user to add and remove artifacts to the hierarchy of system components, and specifying their position in the hierarchy with the tag attribute, the model leaves to the user to determine the desired granularity of the traceability.

8.4.1 Pre-RS traceability submodel

In the pre-RS traceability submodel, only the traces between the Stakeholder artifact and the Prose requirement artifact requires additional trace information. All other traces is only

required to register the ID and tag attribute of their two connecting artifacts.

The *affect* trace must contain information, relating to how the stakeholder is affected by the prose requirement. This is because the *affect* trace is a generalisation of the many relationships that can exist between a stakeholder and a requirement (e.g., “implements”, “designs”, “tests”, etc.). Thus, the *affect* trace must contain the ID and tag of the two connecting artifacts, in addition to an attribute specifying the type of relationship.

The *propose* trace is a complex trace, as it is used to reflect how the specification of the proposed requirement was reached. The presence of minority or majority influence, described in Section 7.2.1, must be captured by this trace. Section 7.3 explains how the level of detail stored by the RT tool must be left to the user. Thus, this trace allows the user to connect one or more stakeholders to a prose requirement, still appointing one of the stakeholders as an owner of the requirement. In addition, for each stakeholder their position regarding the requirement can be recorded as either “compliance” or “conversion”, implying that they either quietly accept the requirement although they hold a different private opinion, or that they privately accept the requirement. If the user do not wish to provide this elaborate information, only an proposing owner of the requirement need to be provided.

8.4.2 Pre-FRS traceability submodel

The pre-RS and pre-FRS traceability submodel are connected by two traces, ensuring bidirectional traceability. Only the ID and tag of the connecting artifacts need to be registered.

The same applies to the remainder of the traces of the pre-FRS traceability submodel, only the ID and tag of the two connecting artifacts need to be registered.

8.4.3 Post-FRS traceability submodel

The pre-FRS and post-FRS traceability submodels are connected by four traces, and still the same applies to these and almost all other traces of this submodel; only the ID and tag of the two connecting artifacts need to be registered.

However, this submodel contains an extraordinary trace, the *verified by* trace, connecting the “fulfil” trace to the Verification procedure artifact. Instead of connecting two artifacts, the trace connects another trace to an artifact. Thus, the trace must register the ID of the other trace, as well as the ID and tag of the connected artifact.

Chapter 9

Summary

Before the task of developing a tool providing requirements traceability could commence, the in-depth study [Nor06] specified preparatory tasks and investigations that had to be done in order to verify the results presented in the study.

First, interviews were conducted with experienced software engineers, mapping the experiences and best practices of the industry concerning requirements traceability. The interviewees related their experiences with requirements traceability in practice, in particular focusing on situations where RT tools had been developed and used to keep in control of the project. In addition, they reviewed the traceability models from the in-depth study, elaborating on how they could be used in practice. The interviews resulted in a few changes to the traceability models, making them reflect the conditions and relationships in real software development projects.

The stakeholders' role in the organisational hierarchy and how this could be represented in the traceability models was investigated, concluding that the models should capture the position and authority of a stakeholder, in addition to tagging how requirements are determined by storing the circumstances of decisions affecting the requirement.

Finally, the preparatory work reviewed the internal representation of the artifacts and traces of the submodels. The planning language *Planguage* provides a set of attributes that are used to specify and quantify requirements, and these attributes in addition to some custom-made attributes are used to specify the artifacts of the traceability models.

Part III

Requirements Specification

Chapter 10

Introduction

This chapter provides a full description of the requirements to a requirements traceability (RT) tool, in addition to other important attributes of the finished product, such as its context of use, user characteristics, and all constraints and dependencies. The structure of the chapter is based on the IEEE std. 830-1998: Software Requirements Specification [IEEmlb].

10.1 Document information

This section lists information regarding the identification of the requirements specification, its status, version number, and origin.

Date of Issue: June 1st, 2007.

Status: Ready for implementation.

Issuing organisation: Gyrð Norvoll, IDI/NTNU.

Change history: No current change history.

10.2 Purpose

The purpose of this part is to provide a precise and detailed specification of all requirements to an RT tool. The document targets developers of the tool, but any stakeholder with invested interest will also find the document interesting.

10.3 Scope

The primary purpose of the RT tool described by the requirements is to provide the users with full traceability of requirements. This is accomplished by assisting the users with gathering, storing, and visualising traceability information. The RT tool will not dictate how the requirements engineering processes or any other software development processes are executed, other than specifying what information is required to provide full traceability. Even though not included in the IEEE standard, a use case model is included to the requirements specification, providing added value to the specification of functional requirements.

10.4 Definitions, acronyms, and abbreviations

For clarification of terms and concepts from the domain of application, i.e., quantification and traceability of requirements, the reader is referred to the in-depth study [Nor06], Part II.

ORM (Object-relational mapping) - Creating a mapping from the objects of an object-oriented programming language to the structure of a relational database. *Hibernate* is an example of ORM.

Java EE (Java Enterprise Edition) - A programming environment for developing and running distributed Java applications based on a tiered architecture.

10.5 References

The IEEE standard upon which this architectural description is based, was approved in 1998 by the IEEE Standards Board, and is named *IEEE Recommended Practice for Software Requirements Specifications* [IEEmlb].

The remainder of this requirements specification refers to no specific documents, founding its specifications on the findings throughout both the in-depth study and part II of this thesis.

10.6 Overview

Here we give an overview of the structure and a quick summary of all chapters in this part. As this part is based on the structure of the IEEE std. 830-1998, a summary of the part is not included at the end.

- **Chapter 11 - Overall description**

This chapter describes any aspects that concern the overall product and its requirements, and provides a foundation for understanding the requirements specified later in the requirements specification.

- **Chapter 12 - Specific requirements**

In this chapter all requirements are stated in detail, with all information necessary to commence the design of the system. Both functional and non-functional requirements are given, in addition to use cases describing the functional requirements in detail.

- **Chapter 13 - Requirements dependencies**

This chapter provides a tracking matrix, visualising the dependencies between requirements, assisting with determining the priorities of requirements.

Chapter 11

Overall description

This requirements specification dictates how an RT tool should function in order to provide full traceability of requirements by specifying both functional and non-functional requirements. The following sections provide the reader with a perspective for understanding the requirements of Chapter 12.

The in-depth study [Nor06] presented a high-level requirements specification (given in Appendix E) that shapes the foundation of the RT tool that is to be developed as a result of this Master's Thesis. The in-depth study underlines that this requirements specification is at too high a level, and should thus be further developed in the Master's Thesis. Functional and non-functional requirements will be further specified in Chapter 12, and use case modelling will be employed for specifying the functional requirements further, the concept of which is briefly described in Section 12.3.1.

11.1 Product perspective

The RT tool (in this chapter henceforth referred to as the *product*) is not part of a larger product, and should thus be considered an independent product. The product will not require other hardware than a client computer and a server.

11.2 Product functions

An overview of the main functions of the product is listed below. These functions are described further in Chapter 12.

- Traceability projects
 - Managing projects
 - Working with projects
- Gathering trace information
 - Creating artifacts
 - Creating traces
- Maintaining trace information
 - Search for artifacts
 - Editing artifacts
 - Deleting artifacts
 - Search for traces
 - Editing traces

- Deleting traces
- Searching trace information
- Visualising trace information
 - Viewing predetermined visualisations
 - Creating and viewing custom-made visualisations
- Help and assistance
 - Help Messages
 - Help Menu
- Administrative tasks
 - Maintaining a user list
 - User Authentication

11.3 User characteristics

The users of the product can be divided into two main categories: non-technological and technological users.

The *non-technical users*, often represented by those stakeholders partaking in the software development processes as customer representatives, have no or little experience with software development. These users take part in the activities of requirements elicitation, and will thus primarily be exposed to the pre-RS traceability stage of RT.

The *technical users* primarily constitute engineers and developers, but in general, this group of users consist of any user with knowledge of the processes and activities of software development. These users will generally partake in all activities occurring during the software development lifecycle, but due to resource management, such users are often assigned specific roles within a software development project, such as requirements analysts, architects, designers, programmers, and testers. Thus, they will primarily be exposed to those stages of traceability that relate to their field of expertise.

The *administrative user* is a user group with a higher access level than the *regular user*. The administrative user can perform all tasks that the regular user can perform, but have been given the authority to perform some additional tasks, such as maintaining a user list for authentication purposes. Administrative and regular users can be either technical or non-technical users, although administrative users will often be technical users.

11.4 Constraints

A product of this kind will contain information that is often regarded as sensitive for its users, and if accessed by intruders with bad intentions, the information could be abused, e.g. for reengineering purposes. The product is particularly vulnerable to malicious attacks as it is based on a client-server architecture, thus dependent on network communication. Security regulations are thus required, and will be specified further in Section 12.6.

The product represents a source of information that can be employed amongst others in problem analyses, and business and strategy decisions. Making important decisions based on faulty information makes up a significant risk in any project, in the worst case the decision could obliterate any chance of project success. It is important that the information is accurate and precise, and reliability is a significant constraint of the product.

In addition, the product's ability to gather the required amount of trace information without neither creating an imposing work overhead for the user nor gathering useless traceability information, and its ability to adapt this to the current environment, is an important constraint of the product. The granularity of the trace information must be adaptable to the environment and user at hand, as well as the envisioned usage of the trace information.

11.5 Assumptions and dependencies

The system described by this requirements specification is of such a nature that hardware dependencies are few, only relying on hardware technology common today, as this delivers the required interfaces, performance, and possibilities. Still, the requirements specification leaves room for choosing between a range of technical possibilities, creating no particular dependencies.

However, the performance requirements make some architectural assumptions, i.e. assuming that the RT tool will employ a three tier Client-Server design pattern. The architecture of the RT tool is further discussed in Part IV, and other than the assumption of the Client-Server design pattern, no assumptions on the architecture will be made.

11.6 Requirements subsets

The functional requirements of *custom-made visualisations of trace information* could be referred to future works, as this is not vital for the first version of the RT tool. Predetermined visualisations, a simpler alternative to the custom-made visualisations, provide enough traceability functionality to offer the user a satisfactory preliminary solution. Likewise, search functionality can be referred to future works, as the overall functionality of the tool is not dependent on search functionality.

Chapter 12

Specific requirements

This chapter presents the requirements specifying how an RT tool should function. The information stated in this chapter will be of value to any work related to the architectural description of the RT tool. All stated requirements are quantifiable, thus ensuring objectivity when determining whether a requirement has been fulfilled or not. The requirements are stated with the means of Planguage (see Chapter 8), based on the determined internal representation of formal requirements given in Section 8.3.2. Explanation of Planguage attributes can be found in Appendix B.

In order to demonstrate the hierarchy of requirements in the requirements specification, special attributes are employed. A requirement residing at the top-most level of the hierarchy is referred to as a *complex requirement*. Such a requirement is identified by the existence of *sub-functions*, which are requirements extracted from the complex requirement. This decomposition can be spotted in both the *tag* attribute of the decomposed requirements (the sub-functions), and in the *sub-functions* attribute of the complex requirement. Likewise, decomposed requirements list their supra-functions in a *supra-functions* attribute.

Full requirement specifications are listed in tables in Appendix F, and this chapter only summarises the main categories of requirements, describing their overall purposes and intentions.

12.1 External interface requirements

This section states the external interface requirements of the product, and describes in detail the inputs and outputs of the product. Four different types of interfaces are established, ranging from interfaces with humans, with other systems, and hardware, as well as interfaces for intersystem and intrasystem communication. A summary of the external interface requirements are given in Figure 12.1, presenting the requirements according to their hierarchical location.

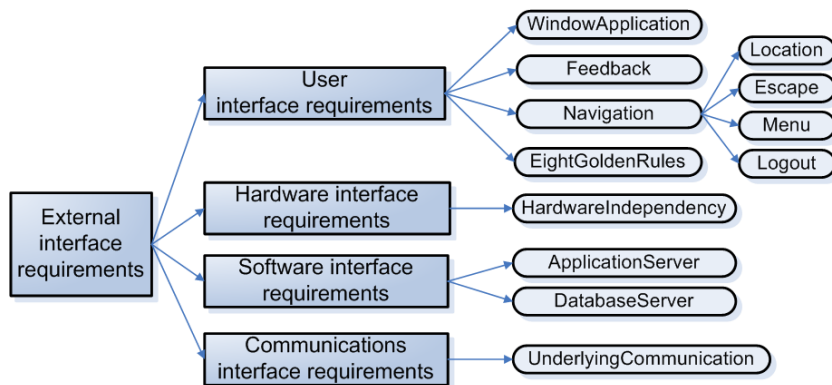


Figure 12.1: Summary of external interface requirements

User interfaces

The product must be usable through a window-based application, as most users currently employ such user interfaces and are accustomed to how they work. This requirement is stated below, in Table F.1. The other user interface requirements are given in Tables F.2 through F.8.

Hardware interfaces

The hardware interface requirement is stated in Table F.9.

Software interfaces

The software interface requirements are stated in Tables F.10 through F.11.

Communications interfaces

The communication interface requirement is stated in Table F.12.

12.2 Functional requirements

The functional requirements are classified according to the main functions listed in Section 11.2. Each category contains several features, which in turn is described by one or more functional requirements. For each requirement, key attributes like priority, required inputs and outputs, and assumed frequency of use are listed.

The features described by the functional requirements are also described by use cases, in Section 12.3. Use cases provide an extended description of the context of the functional requirements. A summary of the functional requirements are given in Figure 12.2, presenting the requirements according to their hierarchical location.

12.2.1 Traceability projects

The following section describes the features associated with creating and managing traceability projects. Traceability projects constitute the topmost categorisation of trace information, associating the gathered trace information with a specific development project.

Managing projects

Tag: `Functional.ManagingProjects`

Description The gathered trace information must be categorised according to which project it belongs to. All artifacts and traces, i.e., an instance of TRACY, must be associated with a specific project. An administrative user must be able to create and remove projects, specifying which users are allowed to access the projects.

Requires An administrative user that has logged on to the system.

Ensures Project-wise association of artifacts and traces.

Inputs An administrative user requests the system to create a project with a given name, and states the users that are allowed to work with the project. The administrative user can notify the system to remove users from a project, in addition to removing entire projects.

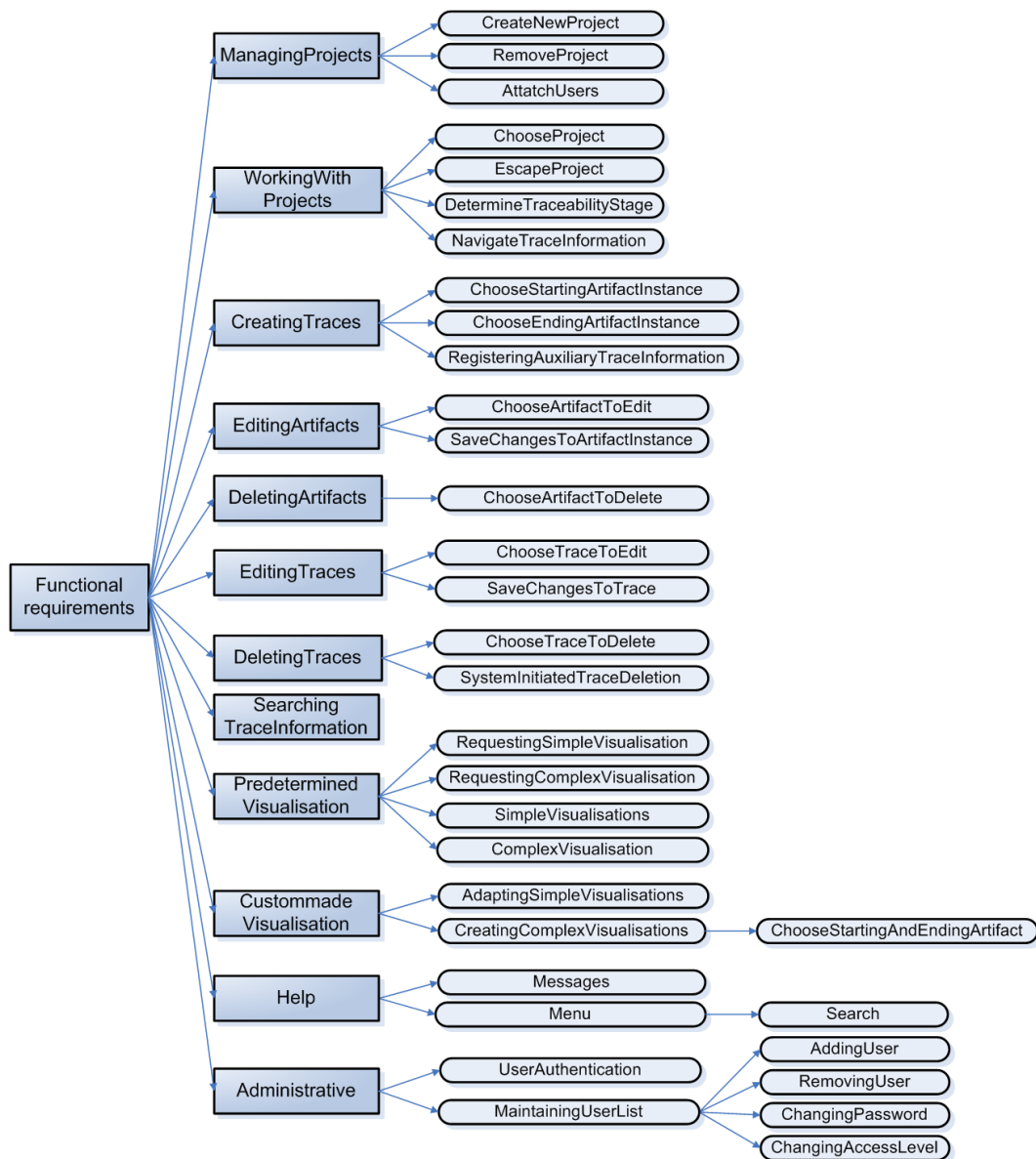


Figure 12.2: Summary of functional requirements

Outputs The system creates a project with the given name, and attaches the specified users to the project, allowing them to access the traceability stages of the project. The system will also, if asked, remove users from projects or remove entire projects. The system always returns a feedback message to the administrative user, informing him of the result of the request.

Priority High.

Frequency of use High.

Functional requirements

The functional requirements are listed in Tables F.13 through F.15.

Working with projects

Tag: Functional.WorkingWithProjects

Description When gathering trace information and registering it in the system, the user must

associate the information with a specific project. Rather than specifying an owner relationship each time a new piece of information is entered, the user chooses a project to work with, and must exit this project if he or she is to work with another project.

Requires An user that has logged on to the system.

Ensures Project-wise association of artifacts and traces.

Inputs A user informs the system which project he or she wishes to work with. If necessary, the user asks the system to quit the project, and can then tell the system to work with another project.

Outputs When requested by the user to work with a specific project, the system presents the traceability stages of the project, giving the user full access to all functions related to gathering, maintaining, searching and visualising trace information.

Priority High.

Frequency of use Medium.

Functional requirements

The functional requirements are listed in Tables F.16 through F.19.

12.2.2 Gathering trace information

The features of this section describe functions that regard the collection of trace information.

Creating artifacts

Tag: `Functional.CreatingArtifacts`

Description Creating artifacts requires the ability to create instances of the artifacts of the traceability models of TRACY (see Section 6.3), registering the trace information TRACY has determined necessary for the artifact in question, and finally storing this information in a persistent manner.

Requires A user that has logged on to the system and are able to provide the required trace information.

Ensures Creation of a new artifact of the desired type, and persistent storage of this artifact and any required trace information.

Inputs The user informs the system of which artifact he or she wishes to create an instance of. When requested by the system, the user states the required trace information required by the desired artifact and requests that the system stores the information.

Outputs The system, after requested to create an artifact instance, presents the user with a form requesting the required trace information. After storing the information, the system outputs feedback, stating either success or failure.

Priority High.

Frequency of use High.

Functional requirements

The functional requirement is listed in Table F.20.

Creating traces

Tag: `Functional.CreatingTraces`

Description Traces connect the artifacts of the traceability model, representing relationships between the artifacts. Traces are the glue that hold the traceability model together, and without them, no traceability of requirements can be offered. Representation of traces usually requires a user specification of which two artifacts are to be connected, as well as any other information determined necessary in Section 8.4.

Requires A user that has logged on to the system and is able to provide the required trace information. In addition, the artifacts that are to be connected, must have existing instances.

Ensures The registration of a relationship between two artifact instances in accordance with the relationships presented by the traceability models, thus creating a trace between these artifacts.

Inputs The user specifies which existing artifact instance he or she wishes to trace *from*. When the system provides a list of potential artifacts to trace *to*, the user selects the appropriate artifact, provides any requested auxiliary trace information and requests that the system stores the information.

Outputs When informed of the starting artifact of the trace, the system provides a list of all possible ending artifacts, in accordance with the traceability models. When the starting and ending artifact of the trace have been determined by the user, the system checks the internal representation of the trace, and asks the user for any auxiliary trace information. After storing the information when requested, the system outputs feedback, stating either success or failure.

Priority High.

Frequency of use High.

Functional requirements

The functional requirements are listed in Tables F.21 through F.23.

12.2.3 Maintaining trace information

The features of this section describe functions that regard the maintenance of trace information, an important part of requirements traceability due to its lifecycle perspective.

Editing artifacts

Tag: `Functional.EditingArtifacts`

Description Artifacts change throughout the project lifecycle, and it is important to be able to edit an artifact instance when its trace information changes. The alternative is to create a new instance, and then delete the old instance, which is not a particularly effort-saving approach. Editing artifacts involve locating the existing instance, editing its trace information, and store it for future use.

Requires A user that has logged on to the system and is able to provide the required traceability information. The artifact instance that is to be edited, must already exist within the system.

Ensures Updatable, maintainable, and current representations of the artifacts of the project.

Inputs User informs system of which artifact he or she wishes to edit, states the new trace information, and requests that system stores the information.

Outputs When being informed by the user which artifact is to be edited, the system presents the existing trace information recorded in that artifact, and allows this information to be edited

by the user. After storing the information when requested, the system outputs feedback, stating either success or failure.

Priority High.

Frequency of use Medium.

Functional requirements

The functional requirements are listed in Tables F.24 through F.25.

Deleting artifacts

Tag: Functional.DeletingArtifacts

Description In order to depict the current situation in a project, it will at times be necessary to delete artifacts. When artifacts are deleted, any of their belonging traces must also be deleted.

Requires A user that has logged on to the system and is able to provide the required traceability information. The artifact instance that is to be deleted, must already exist within the system.

Ensures Removable and current representations of the artifacts of the project.

Inputs User informs the system of which artifact instance that is to be deleted, and when system requests confirmation, the user provides a suitable reply, either acknowledging or renouncing it.

Outputs When requested to delete an artifact instance, the system requests the user for a confirmation of the request. Depending on the user's answer, either acknowledging or renouncing the request, the system deletes or keeps the artifact instance, presenting a feedback message to the user with the result of the request.

Priority High.

Frequency of use Low.

Functional requirements

The functional requirement is listed in Table F.26.

Editing traces

Tag: Functional.EditingTraces

Description Just as with artifacts, traces will occasionally require an update in order to be able to represent the current situation within a project. Editing traces involve locating the existing instance, changing any of the two connecting artifact instances, edit its auxiliary trace information, and store any changes for future use.

Requires A user that has logged on to the system and is able to provide the required trace information. The trace that is to be edited, must already exist within the system.

Ensures Updatable, maintainable, and current representations of the relationships between artifacts of the project.

Inputs User informs the system of which trace he or she wishes to edit, alters the connected artifacts to any other existing instances of the same artifact category, states any new auxiliary trace information, and requests that system stores the information.

Outputs When being informed by the user which trace is to be edited, the system presents the existing trace information recorded on that trace, i.e. the connecting artifacts and any auxiliary trace information, and allows this information to be edited by the user. After storing the information when requested, the system outputs feedback, stating either success or failure.

Priority High.

Frequency of use Medium.

Functional requirements

The functional requirements are listed in Tables F.27 through F.28.

Deleting traces

Tag: `Functional.DeletingTraces`

Description In order to depict the current situation in a project, it will at times also be necessary to delete traces. This can both be due to relationships that cease to exist between two still existing artifacts, or that an artifact is deleted and consequently all its relationships are terminated. In the case of a deleted artifact, the request for deletion of a trace originates from the system itself.

Requires A user that has logged on to the system and is able to provide the required traceability information. The trace that is to be deleted, must already exist within the system.

Ensures Removable and current representations of the relationships of the project.

Inputs A user or the system itself informs the system of which trace that is to be deleted, and when the system requests confirmation in the case of a user-initiated request, the user provides a suitable reply, either acknowledging or renouncing it.

Outputs When requested by a user to delete a trace, the system requests the user for a confirmation of the request. Depending on the user's answer, either acknowledging or renouncing the request, the system deletes or keeps the artifact instance, presenting a feedback message to the user with the result of the request. When requested by the system itself to delete a trace, the system presents a feedback message to the originating system process.

Priority High.

Frequency of use Low.

Functional requirements

The functional requirements are listed in Tables F.29 through F.30.

12.2.4 Searching trace information

Tag: `Functional.SearchingTraceInformation`

Description The system gathers potentially large amounts of trace information, and it is important that the user is able to locate this information quickly. Thus, a search function is important, as it provides a quick and easy manner to find the information needed. A search function uses search terms specified by the user, and lists any information including those terms.

Requires A user that has logged on the system.

Ensures Quick access to the needed information.

Inputs A user specifies a set of search terms, and requests the system to list all information containing these search terms.

Outputs When being requested by the user to search for information containing a set of search terms, the system locates all data containing the search terms, and presents this to the user.

Priority Medium.

Frequency of use Low.

Functional requirements

The functional requirement is listed in Table F.31.

12.2.5 Visualising trace information

The features of this section describe functions regarding the visualisation of the available trace information, ranging from simple visualisation of singular artifacts and traces, to the more complex visualisations of full traceability models.

Viewing predetermined visualisations

Tag: `Functional.PredeterminedVisualisations`

Description Predetermined visualisations are predetermined by the system, and the user cannot change the visualisations to suite his current needs. A predetermined visualisation can either be a simple visualisation of the internal representation of an artifact instance or a trace, or it can be a complex visualisation showing the traces between several artifacts, even crossing the boundaries of the traceability stages (pre-RS, pre-FRS, and post-FRS traceability). Common for them both is that the user cannot change what information is displayed, or which artifacts are included in the visualisations. Simple visualisations only display a single artifact or trace at the time, chosen by the user. For complex visualisations, the included artifacts are predetermined by the system, illustrating e.g. the pre-RS traceability artifacts and how they interrelate.

Requires A user that has logged on to the system and is able to provide the required traceability information. For simple visualisations, a generic view must exist for representing the trace information, whilst for complex visualisations, a set of views for predetermined visualisations must exist. These generic views leaves only one task for the system; inserting the gathered trace information in the views and presenting them to the user.

Ensures Visual representations of the gathered trace information, allowing analysis of the data.

Inputs A simple visualisation requires the user to select an artifact instance or a trace, whilst a complex visualisation requires the user to choose among a set of predetermined visualisations.

Outputs When requested by the user to display a specific visualisation, the system gathers the required trace information, inserts this information into the prespecified view, and presents the view to the user.

Priority High.

Frequency of use High.

Functional requirements

The functional requirements are listed in Tables F.32 through F.35.

Creating and viewing custom-made visualisations

Tag: `Functional.CustommadeVisualisations`

Description Custom-made visualisations consists of either simple custom-made visualisations or complex custom-made visualisations. Simple custom-made visualisations are simple predetermined visualisations regulated by filters activated by the user, thus enabling the user to adapt the amount of presented information and how it is presented. Complex custom-made visualisations could be viewed as an extension of the complex predetermined visualisation, allowing the user to a certain degree to decide which artifacts are included in the visualisation. However, the traceability models imposes certain restrictions on the visualisations, reducing the range of user options to the selection of a starting and ending artifact instance. The remainder of the included artifact instances is chosen from the traceability models; any artifact instances that are required to trace from the chosen starting and ending artifact instance.

Requires A user that has logged on to the system and is able to provide the required traceability information. For simple visualisations, a generic view alterable by predefined filters must exist for representing the trace information.

Ensures Dynamic visual representations of the gathered trace information, allowing analysis of the data.

Inputs A simple visualisation requires the user to select an artifact instance or a trace and activate the desired filters, chosen from a list. A complex visualisation requires the user to choose a starting artifact among the existing artifact instances, and when presented with a set of compatible ending artifacts, choose the desired ending artifact.

Outputs When requested by the user to display a specific simple visualisation, the system gathers the required trace information, inserts this information into the prespecified view, and presents the view to the user. Depending on the filters activated by the user, the system must alter the view to represent the trace information differently. If a user requests a complex visualisation, the system must present the user with a list of possible starting artifacts (any artifact within the chosen traceability stage). When the user have selected the desirable starting artifact instance, the system must, with the means of the traceability models, determine any connected artifact categories, and display their instances as possible ending artifacts. When the user has chosen a ending artifact instance, the system must present the user with a visualisation of these two artifacts, along with any artifacts required to trace between them.

Priority Medium.

Frequency of use Low.

Functional requirements

The functional requirements are listed in Tables F.36 through F.38.

12.2.6 Help and assistance

Tag: Functional.Help

Description Providing electronic help functionality enables the user to solve problems related to executing tasks of the system, without having to call on external resources. Such functionality is an important part of fulfilling the usability requirements to the system, specified in Section 12.6. When implementing help functionality, several factors must be considered. The user must be given the opportunity to review and search through a directory of documents describing common problems and course of actions, referred to as *passive* help functionality. In addition, the user must receive feedback when performing tasks incorrectly, referred to as *active* help functionality.

Requires A user that has logged on the system.

Ensures A user that is able to use the system independently and solve his or her problems without calling on external resources, such as expert users.

Inputs The user executes a task within the system incorrectly, causing the system to respond with feedback on what went wrong, or the user visits the help functionality independently of executing tasks, looking through and searching for interesting topics.

Outputs When the user has executed a task incorrectly, the system responds by displaying a feedback message stating what went wrong, and how the user should correct the problem. If the user visits the help functionality independently of task execution, the system lists the help topics, allowing the user to search through and read interesting topics.

Priority Medium.

Frequency of use Medium.

Functional requirements

The functional requirements are listed in Tables F.39 through F.41.

12.2.7 Administrative functions

The administrative functions include every task in the system that does not influence the gathered trace information, but rather focus on tasks such as maintaining a list of users of the system, and appointing access levels to users.

Maintaining a user list

Tag: `Functional.Administrative.MaintainingUserList`

Description A list of users specify the users of the system, their access level, and authentication passwords, and are used by the system as a security measure against intruders. Only registered users with validated passwords can access the system. Chosen users can be given administrative rights, enabling them to add other users.

Requires An administrative user that has logged on to the system.

Ensures The ability to add and remove users of the system and assign user-specific access rights.

Inputs An administrative user requests that a new user is to be added to the system. The administrative user specifies the access level, authentication password and username of the user.

Outputs When being requested by the user to add a new user, the system requests the user to specify the necessary information (access level, username and authentication password). When the necessary information is stated, the system stores a new user instance persistently in the user list, allowing the specified username and password access the system with the given access level.

Priority High.

Frequency of use Medium.

Functional requirements

The functional requirements are listed in Tables F.42 through F.45.

User Authentication

Tag: `Functional.Administrative.UserAuthentication`

Description In order to secure the system, user authentication is required. This makes it more difficult for intruders with malicious purposes to access the system. A user is required to log on to the system before gaining access to system resources, by providing a username and password that the system will validate before admitting the user.

Requires A user that is registered as a user of the system, and has been given a username and password.

Ensures A secured application with user authentication.

Inputs A user provides a username and a password for validation by the system.

Outputs The system validates the given username and password, and if the user is an allowed and authenticated user of the system, the system presents the list of projects the user is attached to, granting the user his given access level. If authentication failed, the system informs the user of this, and the user is asked to re-enter his username and password. If authentication failed in three consecutive attempts, the user is requested to contact the system administrator.

Priority High.

Frequency of use Medium.

Functional requirements

The functional requirement is listed in Table F.46.

12.3 Use case modelling

Section F.2 specify the functional requirements of the system to be developed. This section extends a selection of these requirements by specifying a set of use cases, illustrating the use of the main features described by the requirements. Thus, a single use case can include the functionality specified in several requirements. The remaining non-functional requirements will be further elaborated in Sections 12.4 through 12.7.

12.3.1 The concept of use case modelling

Use case modelling is the process of building a use case model, a model that describes the functional requirements of the system to be developed. First, descriptions of participating actors are given, continuing with the use cases themselves. The use cases capture what could be thought of as a *contract* between the stakeholders of a system about its behaviour [Coc05]. Use cases describe scenarios, exemplifying the use of the system, and consequently illustrating the functional behaviour of the system in a different manner than the requirements specification, which could enhance the understanding among the stakeholders.

Use cases can be presented in two different forms, a fundamental text form and a graphical form such as flow charts, UML diagrams, and so forth. The use cases presented in this chapter will be textual use cases, allowing a greater level of detail than graphical use cases and consequently richer and more informative use cases, better suited as tools for communication [SK06] [Coc05].

12.3.2 Description of actors

In this section, a textual description of each actor that will appear in the use cases is given. For each actor, important attributes are listed, i.e., name, description, and examples. In Table 12.1, you will find a description of the regular users of the application, i.e. the ones without administrative rights. In Table 12.2, a description of administrative users of the application is given, and in Table 12.3, a description of the system itself is given.

Table 12.1: Actor description - regular user

	Comment
Actor	Regular user
Description	The regular user of the application will primarily use it for registering and viewing traceability information. This actor will be the most frequent user of the application, and any technical knowledge is most likely highly fluctuating, ranging from super users to inexperienced users.
Examples	Customer representatives, business intelligence analysts, requirements engineers, developers, testers

Table 12.2: Actor description - administrative user

	Comment
Actor	Administrative user
Continued on next page	

Table 12.2 – continued from previous page

	Comment
Description	The administrative user of the application have extended user rights, enabling this actor perform administrative tasks within the system, such as registering new users. This indicates that this user has a higher degree of authority within the project than regular users. Technical knowledge is assumed to be substantial.
Examples	Architects, team leaders, project managers, main customer representative

Table 12.3: Actor description - System

	Comment
Actor	System
Description	The system collects, stores, and visualises the trace information in the project, in addition to exert control over the users of the system.

12.3.3 System use cases

System use cases describe how actors and the system interacts, and includes information concerning actors, triggers, pre-conditions, scenarios and exceptions to scenarios. Each use case is given a descriptive title, referring to its contents, and are presented in Tables G.1 through G.15, listed in Appendix G.

12.3.4 Tracing of use cases

This section presents a tracking matrix visualising how the use cases describe specific requirements. In their specification, each use case lists the requirements it extends, thus providing a link between the requirements specification and the use cases.

The matrix is given in Figure 12.3. The use cases are listed horizontally and requirements vertically. When a use case addresses a requirement, this relationship is indicated by an occurrence in the matrix. For example, if use case X addresses requirement Y and Z, this is indicated in the tracking matrix by an occurrence at the Yth and Zth element of the Xth row.

Use cases describe the most common uses of the system, and explain in simple terms how the user interact with the system. Thus, the requirements they extend are important requirements to the system, and visualising which requirements are extended helps to determine the priority of functionality corresponding to the use cases and requirements when implementing a solution. According to the occurrences of the tracking matrix, concentrating on `Functional.WorkingWithProjects` and `Functional.Administrative.UserAuthentication`, functionality providing a user with the opportunity to log in, and work with projects stands out as vital functionality. Priority is further discussed in Chapter 13, where dependencies between requirements assist with defining the priority of requirements further.

Functional.Administrative.UserAuthentication	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Functional.Administrative.ChangingAccessLevel															X
Functional.Administrative.ChangingPassword															X
Functional.Administrative.RemovingUser															
Functional.Administrative.AddingUser															X
Functional.Help.Menu.Search															X
Functional.Help.Menu															X
Functional.Help.Messages															X
Functional.CustommadeVis[...].ComplexVis[...].ChooseStartingAndEndingArtifact												X			
Functional.CustommadeVisualisations.CreatingComplexVisualisations												X			
Functional.CustommadeVisualisations.AdaptingSimpleVisualisations												X			
Functional.PredeterminedVisualisations.ComplexVisualisations												X			
Functional.PredeterminedVisualisations.SimpleVisualisations												X			
Functional.PredeterminedVisualisations.RequestingComplexVisualisation												X			
Functional.PredeterminedVisualisations.RequestingSimpleVisualisation												X			
Functional.SearchingTraceInformation												X			
Functional.DeletingTraces.SystemInitiatedTraceDeletion									X						
Functional.DeletingTraces.ChooseTraceToDelete												X			
Functional.EditingTraces.SaveChangesToTrace															
Functional.EditingTraces.ChooseTraceToEdit															
Functional.DeletingArtifacts.ChooseArtifactToDelete												X			
Functional.EditingArtifacts.SaveChangesToArtifactInstance												X			
Functional.EditingArtifacts.ChooseArtifactToEdit												X			
Functional.CreatingTraces.RegisteringAuxiliaryTraceInformation												X			
Functional.CreatingTraces.ChoosingEndingArtifactInstance												X			
Functional.CreatingTraces.ChoosingStartingArtifactInstance												X			
Functional.CreatingArtifacts.CreatingAnArtifactInstance										X					
Functional.WorkingWithProjects.NavigateTraceInformation										X				X	
Functional.WorkingWithProjects.DetermineTraceabilityStage			X	X	X	X	X	X	X	X	X				
Functional.WorkingWithProjects.EscapeProject		X													
Functional.WorkingWithProjects.ChooseProject		X	X	X	X	X	X	X	X	X	X				
Functional.ManagingProjects.AttachUsers															X
Functional.ManagingProjects.RemoveProject															
Functional.ManagingProjects.CreateNewProject															X
Functional Requirements															
	Use Case														
	UseCase.UserLogin														
	UseCase.ChooseProject														
	UseCase.ChooseTraceabilityStage														
	UseCase.WorkingWithArtifacts														
	UseCase.CreateArtifact														
	UseCase.EditArtifact														
	UseCase.DeleteExistingArtifact														
	UseCase LocateExistingArtifact														
	UseCase.AddTrace														
	UseCase.Delete Trace														
	UseCase.VisualiseTraceInformation														
	UseCase.SearchTraceInformation														
	UseCase.CreateProject														X
	UseCase.RegisterUser														
	UseCase.SearchHelpDirectory														

Figure 12.3: Tracking matrix - Use cases/Requirements

12.4 Performance requirements

This section describes the performance requirements of the system: non-functional requirements. Performance requirements describe how the system must perform by stating “how much” the system must perform. These requirements describes the workload capacity of the system, specifying the amount of work the system must be able to handle.

Specific requirements can also be stated describing how much resources (e.g. time and effort) should be saved in comparison with a benchmark system, often a previous version of the system to be developed. However, such requirements will not be given in this requirements specification, as the system to be developed have no significant benchmark system it can be compared with. Other requirements traceability tools exist, but they are hard to use for comparison, as they all approach the task somewhat different. The reader is referred to the in-depth study for more information on other RT tools.

The performance requirements are given below, in Tables F.47 through F.56. The system’s response time must be considered the primary objective for the performance requirements. The

response time will primarily depend on the performance of i.e. web and database servers, as well as communication lines. The performance requirements regarding response time are based on response-time guidelines in [Shn98a]. A summary of the performance requirements are given in Figure 12.4, presenting the requirements according to their hierarchical location.

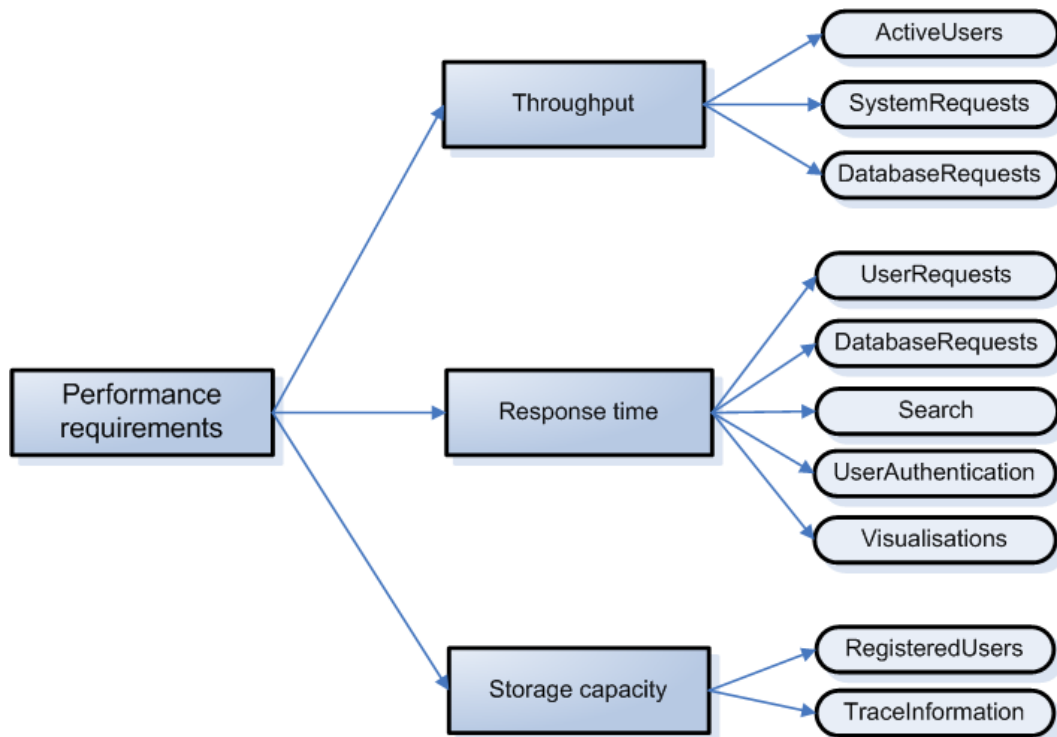


Figure 12.4: Summary of performance requirements

Throughput

Throughput measures the system's ability to process work, and is measured along a scale consisting of the amount of work done per unit of time. Tables F.47 through F.49 lists the quality requirements concerning throughput.

Response time

The response time illustrates the delay between initiating or requesting some action, and the arrival of the result. The response time requirements are stated in Tables F.50 through F.54.

Storage capacity

Storage capacity is a system ability to store units of some defined kind, such as users or units of information, in this case trace information. The storage capacity requirements are stated in Tables F.55 through F.56.

12.5 Design constraints

Design constraints are, like functional requirements, nonnegotiable. They are either fulfilled or not. As explained earlier, this is a binary circumstance. Design constraints consists of standards compliance, introducing constraints put down by either external or internal authorities, and

hardware limitations, constraints caused by the use of certain hardware. A summary of the design constraints are given in Figure 12.5, presenting the requirements according to their hierarchical location.

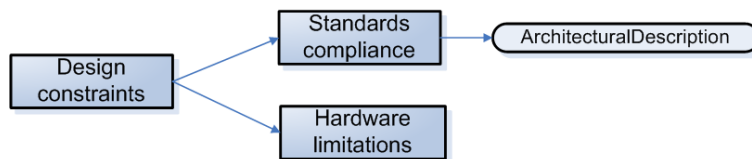


Figure 12.5: Summary of design constraints

Standards compliance

Several standards exist in the world of software development, both universally accepted standards proposed by boards or organisations such as ISO or IEEE, and internal standards that are valid only within the proposing organisation. A standard put down a set of rules that must be followed if standards compliance is the objective. Only a single standards compliance requirement apply to the development of the system, given in Table F.57.

Hardware limitations

Hardware limitations are requirements that influence the software requirements, placing constraints on how the software should be developed in order for it to be compatible with the given hardware. Only a single hardware limitation is put down for the system, listed in Table F.58.

12.6 Software system attributes

Software system attributes describe quality requirements to the system, focusing on selected attributes. It is important that these requirements are measurable and verifiable. ISO 9126 is an international standard for the evaluation of software, presenting a classification of software system attributes. This classification form a foundation for the chosen attributes in this requirements specification. The requirements are sorted after the attributes they concern, and are listed in Tables F.59 through F.73. A summary of the software system attributes is given in Figure 12.6, presenting the requirements according to their hierarchical location.

Functionality

The software system attribute of functionality addresses the quality of the software functions, and focuses on security. The security attribute measures the system's ability to resist unauthorized access and usage while still being able to provide its legitimate users with its services. A secured system can be established in a number of ways, e.g. by providing nonrepudiation, confidentiality, integrity, assurance, availability, and auditing. The requirements related to the functionality attribute are listed in Tables F.59 through F.61.

Reliability

The reliability attribute addresses the software's capability to maintain its performance level under stated conditions such as load, system mode and period of time. The two reliability attributes focused on in this requirements specification are recoverability and availability. Recoverability addresses how well the system can recover after failure, whilst availability is defined

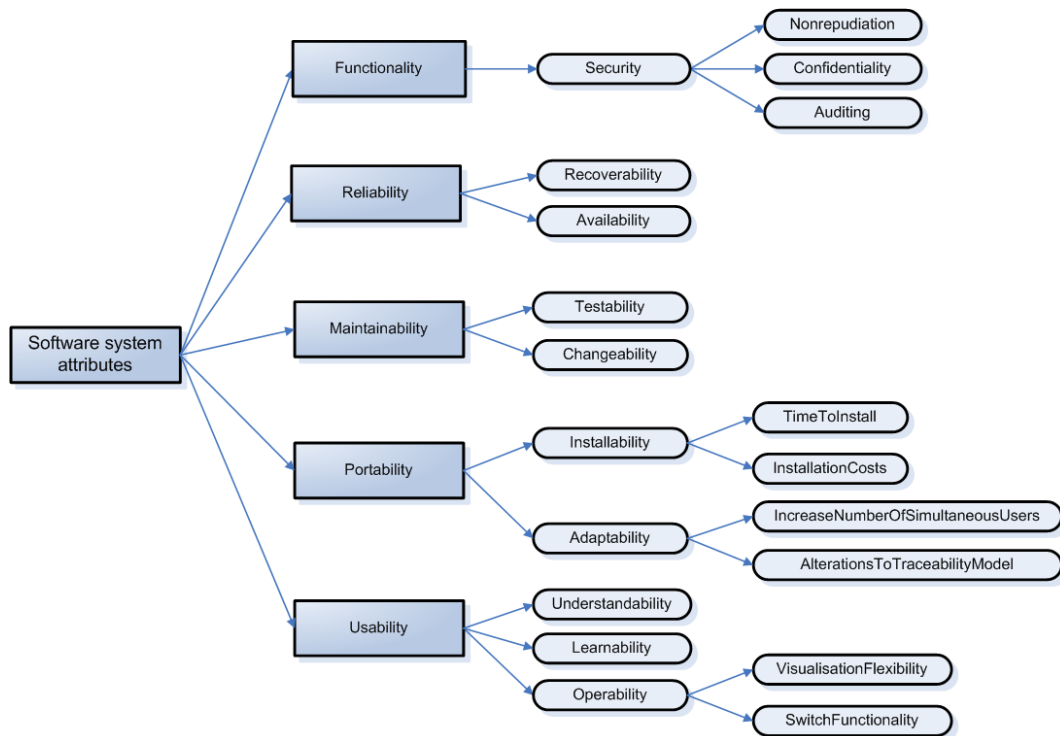


Figure 12.6: Summary of software system attributes

in accordance with Equation 12.1. Tables F.62 through F.63 lists the quality requirements concerning reliability.

$$Availability = \frac{Mean\ time\ to\ failure}{Mean\ time\ to\ failure + Mean\ time\ to\ recover} \quad (12.1)$$

Maintainability

Maintainability concerns the efficiency and cost with which a system can be changed. The issues of maintainability addressed in this requirements specification are testability and changeability. The requirements to maintainability are listed in Tables F.64 through F.65.

Portability

A system's ability to be transferred from one environment to another is referred to as its portability. The portability attribute addresses the issues installability and adaptability. The quality requirements concerning portability are listed in Tables F.66 through F.69.

Usability

A system's ease of use is often referred to as its usability. This attribute is concerned with how easy it is for the user to execute a desired task, and is one of the most common quality attributes considered when developing a system intended for human use. Tables F.70 through F.73 lists the requirements concerning usability.

12.7 Other requirements

There are no other requirements at this stage.

Chapter 13

Requirements dependencies

Figure 13.1 illustrates the dependencies between the requirements by displaying the requirements in a matrix. An occurrence in the intersection between a row and a column indicates a dependency between the requirements. These dependencies are also listed in the description of each requirement in the requirements specification, also illustrating the direction of the dependency; a requirement X listing requirement Y in its dependencies needs requirement Y to be fulfilled in order for itself to be fulfilled. Thus, dependencies help define the priority of requirements.

Hierarchical relationships (sub- and supra-functions) also represent a dependency, as a supra-function requires the fulfilment of its sub-functions in order for itself to be fulfilled. Hierarchical relationships have been illustrated in the figures summarising the requirements of this requirements specification, and are found in Figures 12.1, 12.2, 12.4, 12.5, and 12.6.

As can be seen in the matrix, many requirements are dependent upon the requirements of creating artifacts and traces, *Functional.CreatingArtifacts* and *Functional.CreatingTraces*. These requirements are in turn dependent upon the requirements belonging to *Functional.ManagingProjects*, *Functional.WorkingWith Projects* and last but not least *Functional.Administrative.UserAuthentication*. Every requirement is dependent upon user authentication, so this requirement is the most important requirement of the entire system. Thus, the matrix helps visualise the priority of the requirements.

The use case tracking matrix of Section 12.3.4 also points out that functionality regarding user authentication and working with projects are important, which supports the priority reached when studying the requirements dependencies.

Due to the spatial limitations, only the requirements with dependencies are listed in the tracking matrix.

The requirements of this requirements specification is based on the high-level requirements defined in the in-depth study, summarised in Appendix E. Figure 13.2 shows a tracking matrix linking the high-level requirements to the requirements detailing them in the requirements specification. A single high-level requirement can be linked to several of the specific requirements in the full requirements specification.

Part IV

Architectural Description

Chapter 14

Introduction

In this chapter, a short summary of the purpose and scope of this part is given, along with an overview of its chapters.

14.1 Document information

This section lists information regarding the identification of the architectural description, its status, version number, and origin.

Date of Issue: June 1st, 2007.

Status: Ready for implementation.

Issuing organisation: Gyrð Norvoll, IDI/NTNU.

Change history: No current change history.

14.2 Summary

This architectural description documents the architecture of an RT tool, a tool providing traceability of requirements by assisting with the elicitation, quantification, and evolution of the requirements, as well as relating the requirements to other important artifacts of the software development processes, such as system components and verification procedures.

14.3 Scope

The scope of this architectural description is limited to the elaboration of the architecture of the system outlined by the requirements listed in the requirements specification of Part III. The architectural description will not handle any aspects beyond this requirements specification.

14.4 Context

This architectural description is part of a Master's Thesis concerning the topic of quantification and traceability of requirements, and is written according to the IEEE std. 1471-2000. Consequently, this document can be read independently of the remainder of the Master's Thesis. However, it is recommended that the user reviews the effort put down in order to reach the current architecture, including both an in-depth study [Nor06], additional preparatory work (see Part II) and a requirements specification (see Part III).

14.5 Glossary

Problem domain

A problem domain is a domain where the parameters defining the boundaries of the domain, the entities of the domain, and the relationships between these entities are not well enough understood to provide a systematic description of the domain [dom]. In simpler terms, the problem domain represents an unknown situation or environment where few or no conditions are known.

Problem domain analysis

By performing a problem domain analysis, the analyst acquaints himself with the problem domain and creates a domain model describing the problem to be solved. This model is completely independent of the constructs of the solution, and once the model is built it can be applied to a wide range of technologies. This creates a good approach to software evolution, as older technologies can be replaced by new and emerging technologies in the future [da].

14.6 References

The IEEE standard upon which this architectural description is based, was approved in 2000 by the IEEE Standards Board, and is named *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems* [IEEmla].

14.7 Overview

Here we give an overview of the structure and a quick summary of all chapters in this part. As this part is based on the structure of the IEEE std. 1471-2000, a summary of the part is not included at the end.

- **Chapter 15 - Identification of stakeholders and concerns**
This chapter introduces the stakeholders of the architecture and consequently the system itself. In addition to a description of the stakeholders, the chapter identifies their main concerns which are architecturally relevant.
- **Chapter 16 - Architectural strategies**
The system is constrained by a number of quality requirements, and this chapter investigates which architectural strategies should be applied to help with the fulfilment of quality requirements.
- **Chapter 17 - Selection of architectural viewpoints**
In this brief chapter, a description of the selected viewpoints is given, along with the rationale behind the decisions.
- **Chapter 18 - Architectural views**
The views presented in this chapter consists of architectural models that describe the behaviour of the system, and each view corresponds to exactly one viewpoint of the previous chapter.
- **Chapter 19 - Consistency among architectural views**
Any known inconsistencies among the views described previously, are described in this chapter, in addition to an analysis of the consistency across all of the architectural views.

- **Chapter 20 - Architectural rationale**

This chapter presents the architectural rationale, which explains the choice of architectural concepts throughout the architectural description.

Chapter 15

Identification of stakeholders and concerns

Architectural descriptions have significant influence on the final result of the system development processes, and are thus of great importance to all the stakeholders in the project. This chapter identifies these stakeholders, in addition to the main concerns of the architecture.

15.1 Stakeholders

The RT tool described by this architecture, is more of an *off-the-shelf* application than a custom-made application, with no specified acquirers or customers. The tool is developed with a specific use in mind; assisting software developers with the quantification and tracing of requirements. Thus, the stakeholders of this architecture are identified within the group of people associated with a general software development project, such as users, acquirers, and developers.

Users

The users of the system are divided into two major groups; technical and non-technical users. Technical users often have an IT-related education, work with IT, and are experienced with software development processes in general, and often requirements engineering in particular. The non-technical users do not work with IT-related tasks, and are inexperienced with software development and its processes.

The non-technical users often have a rather limited perspective on the potential benefits of requirements traceability, seeing only the immediate effort required. The technical users have an extended perspective, seeing more of the long-term benefits. However, these users can also be overwhelmed by the immediate effort required. Thus, the primary concern of the users, whether they are technical users or non-technical users, is the usability of the system. This is a key quality attribute of the system, and the users pay particular attention to the ease with which the repetitive tasks of requirements traceability can be executed. It is important to avoid situations creating additional workload among the users, rendering the activities of requirements traceability an undesirable hassle. As the primary objective of the system is to alleviate the activities of requirements traceability, it is important to avoid such situations by carefully considering the usability of the system.

Acquirers

The acquirers of the system are assumed to be those participants of software development projects with the authority of introducing the use of an RT tool into the existing procedures

and processes of the project. They have extensive experience with both technical and non-technical aspects of software development, and are distinguished by a long-term prospective on the benefits of requirements traceability.

Acquirers are primarily concerned with the effort needed for introducing the system, and the benefits gained by its introduction. The effort required for introducing the system addresses several quality attributes, among them installability, portability, and learnability.

Requirement Analysts

The requirement analysts communicate with the acquirers, attempting to uncover their requirements to the product. The analyst have considerable skills in uncovering both latent information and objectively extracting obvious information from the acquirers, and are primarily concerned with creating a requirements specification containing all requirements of the acquirer.

Architects

The architects are presented with a full requirements specification, and given the task of creating an architecture that satisfies the requirements. They have an all-encompassing view of the problem domain, and can consider both sides of conflicting requirements through trade-off analyses. The requirement analyst and the architect roles could be held by the same person, as each role require extensive knowledge of the tasks and results of the other.

Developers

The developers are responsible for the implementation of the architecture, and are thus keenly interested in both the functional and non-functional requirements of the system, and how the architecture addresses these requirements. In addition, the developers are concerned with how the development of the system will be organised, e.g. in software increments, modules, etc.

Testers

The task of the testers is to ensure that the system fulfils its requirements as well as its architecture, and their primary concern is the testability of the system. Testers are generally concerned with every aspect of the system, depending on which level tests are executed.

Maintainers

Maintainers handle the system throughout its lifecycle, solving problems that arise as time goes by. They are naturally interested in the software system attributes regarding maintainability, as the effectiveness of their work is dependent on these attributes.

15.2 Concerns

The primary purpose of the system is to assist software development projects with the task of requirements quantification and traceability, without impeding the other activities of software development. Requirements traceability involve large amounts of information, connecting all artifacts of the software development processes, such as documents, requirements, use cases, designs, and system components. Maintaining requirements traceability by hand is a task too demanding and cost-inefficient to tolerate, consequently introducing the need for an electronic

system that at a low cost can trace from e.g. a requirement to a system component, including all influencing artifacts along the way. The major concern of the system described in this architectural description is to provide a dynamic and efficient approach to requirements traceability, overcoming the significant workload represented by the task of gathering, maintaining, and visualising trace information.

A significant risk when developing and employing the system, is the integration of the system within the employing organisation, i.e. how well the system can be adapted to the projects it is used within, and its unimposingness on the users of the system. This aspect must be considered throughout both development, deployment, and operation of the system. Otherwise, the system would risk being regarded as a hindrance rather than an aiding tool, which in turn would reduce the tool's support among its users. This would significantly reduce the benefits gained from introducing an RT tool.

Privacy and security are also important concerns when defining an architecture. An RT tool contains detailed information on both the development processes and the end product itself, and security breaches rendering the information visible to malicious users could cause serious damage to the organisation, and its competitiveness.

Chapter 16

Architectural strategies

This chapter takes a closer look at the non-functional requirements of the high-level requirements specification¹, attempting to uncover architectural strategies [LB03] fulfilling these requirements. An architectural strategy is a collection of chosen tactics, which in turn are design decisions that influences the fulfilment of non-functional requirements. Several tactics combined are referred to as a design pattern, representing a best practice solution to a common problem. In addition to presenting suitable architectural strategies and discussing their functionality, this chapter goes beyond architectural decisions and discusses some implementation-specific details such as programming language. Choices made regarding implementation do not belong within the architectural description, and will be properly addressed in Part V. However, as these choices influence the fulfilment of the non-functional requirements, they have been included in this chapter. The functional requirements are not considered, as they are not influenced by the choice of strategy to the same degree as the non-functional requirements.

The non-functional requirements place constraints on the choice of strategy used in the development of the outlined system by addressing e.g. the reliability, adaptability, platform independency, and scalability of the system. This chapter investigates these constraints, with the aim of uncovering the best assembly of strategies and technology. As a natural part of the investigation, the rationale behind the chosen architectural strategies is to some extent included in this chapter. However, the architectural description has devoted a separate chapter to the presentation and discussion of the architectural rationale, and the rationale will here be discussed in detail. The remainder of this architectural description outlines in full how the strategies outlined in this chapter will be applied to provide a satisfying architecture of a solution, fulfilling all the requirements of the full requirements specification.

Each of the following sections discusses how architectural strategies can be applied to help satisfy the high-level requirements, i.e., how tactics, design patterns, and technology can be a *means* to an *end*. The full high-level requirements are given in Appendix E, and only an ID and key title are given here. Chapter 13 provides a tracking matrix (Figure 13.2) linking the high-level requirements to the complete set of requirements specified in the full requirements specification.

NFR1 - Persistent storage

This requirement states that all information gathered by the system, must be stored persistently, which in turn improves the reliability of the system, as it reduces the probability of data loss. This is usually accomplished by employing a *database*, most commonly a relational database such as MySQL or Oracle. Thus, by building the system atop a database management system (DBMS), persistence is assured. A DBMS handles all maintenance tasks such as backup, and

¹The reader is referred to Chapter 13 for a tracking matrix linking the high-level requirements to the requirements of the full requirements specification.

provides an interface through which the system can retrieve and pass on data, functioning as a database server.

A relational database's viewpoint on data differs widely from the object-oriented programming paradigm's viewpoint [Kin04]. The object-oriented programming paradigm is currently one of the most popular programming paradigms, and consequently problems arise when attempting to map objects to a relational database. Many programming languages support the object-oriented programming paradigm, among them Java. *Hibernate* is an object-to-relational mapping (ORM) persistence framework for Java, which allows the programmer to ignore the problems arising when mapping objects to a relational database [Hem06].

NFR2 - Adaptability

The RT tool will be employed in widely different settings, in projects of different sizes and complexities. Adaptability is important to consider when designing how the system gather traceability information, making sure that the user can mould the system into a tool that is appropriate and useful in the user's context. In other words, it is important that the front-end functionality of the system, i.e., the functionality experienced by the user, can provide the user with a sense of dynamism and adaptability. This is a challenging task, and will be addressed by both the functional and non-functional requirements of the system, as it relies upon a responsive system design, both in functionality and quality.

In addition, adaptability should be considered when designing the system itself. By creating a component-based architecture, the system will be able to handle future needs and changes better than if all functionality in the system was interwoven. Component-based architecture divides the functionality into blocks with determined interfaces (referred to as components), thus enabling quick replacement of functionality. The component-based architecture can be combined with a layered approach, as depicted in Figure 16.1. This improves the compartmentalisation of the system, by letting each layer contain several components. The only prerequisite for replacing components is that interfaces must remain unchanged. Components can be added and removed when desirable, thus creating an adaptable system.

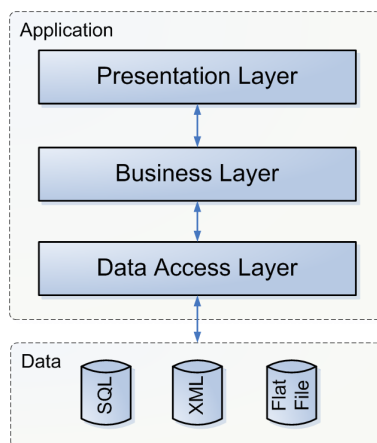


Figure 16.1: Component-based architecture

The component-based architecture could be further extended with a Model-View-Controller design pattern, an architectural choice that allows the application to separate between application components that handles the application logic (the controller), the application data (the model), and the presentation of the application data (the view). This design pattern is illustrated in Figure 16.2. Employing this pattern simplifies the process of replacing specific parts of the application, e.g. all parts that are related to a specific view, primarily because of the separation of functionality.

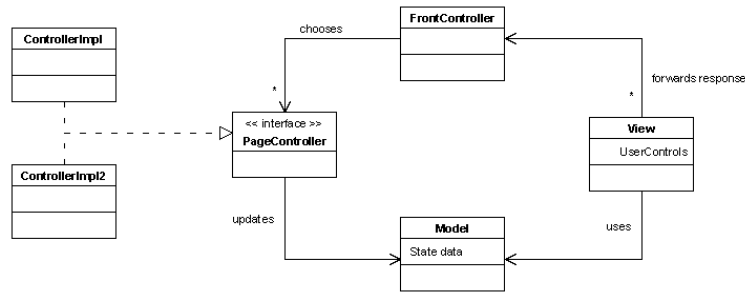


Figure 16.2: Model-View-Controller (MVC) design pattern

NFR3 - Platform independency

Platform independency ensures that the system is independent of its environment. It is important that the RT tool can operate in any environment, as this will lower the threshold for using the tool. The degree of platform independency will vary depending on the choice of technology.

By employing a *virtual machine*, such as in Sun's Java technology [Tec], an application will run in the runtime environment provided by the virtual machine rather than in the enveloping operating system, thus removing itself from its external environment, resulting in platform independency. This is depicted in Figure 16.3.

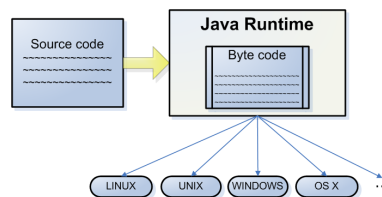


Figure 16.3: Virtual Machine

However, virtual machines provide a limited platform independency, as they require that the user have physical access to the hardware on which it executes. Designing an application in conformance to the client-server architectural style [BCK05a] allows the user to be physically removed from the application, contacting it via some sort of communication technology, most commonly a network. Communication technology falls outside the scope of this discussion, and it is merely assumed to exist, providing the necessary services.

The client-server architectural style requires that the application resides either completely or partially within a server, offering a set of defined services to clients (i.e., the users of its services). The application might in turn require other services, such as a DBMS. The DBMS often resides in a separate server (a database server), and the application server connects to this server, and is offered the required services. Other potential resources required by the application, such as web services [W3C04], offer their services, and the application is only required to connect and request their services, possibly required to authenticate itself, depending on the nature of the requested service.

The client-server architectural style (shown in Figure 16.4) have been further divided into thin and thick clients. Thin clients have little functionality, relying heavily on the server, whilst thick clients implement more functionality, removing themselves from the server. When discussing platform independency, a thin client is often preferable to a thick client, as a thin client will be less dependent upon its environment, more or less ignoring attributes such as client computing power. Web applications are examples of thin clients, where the server performs most tasks, and the client (often a web browser) is only required to forward user input to the server, and display the result handed over from the server. In addition, a web application can be configured to handle requests from a wide variety of clients, such as web browsers, mobile phones, and

PDA's, thus providing the user with an application that can function independently of the user's context, a high level of platform independency.

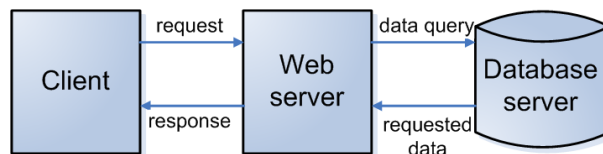


Figure 16.4: Three-tier Client-Server design pattern

Many technologies exist that support the client-server design pattern. However, Sun's Java EE [EE] technology combines the virtual machine with the client-server architectural style, thus providing a large degree of platform independency. By using this technology, applications can be created without particular requirements to local computing resources, thus removing most client-side dependencies. In addition, Hibernate supports Java EE, which simplifies the object-relational mapping between the domain model of the application and the relational model of the database.

NFR4 - Scalability

When creating a client-server application, it is important to ensure scalability. The server must handle fluctuations in the number of clients requesting its services, and it must also handle fluctuations in the workload requested by each client. Scalability is a quality requirement that is easily quantifiable, and by putting down measurable requirements to the scalability of the system, the system can be tested to ensure that it handles the required down- or up-scales in load. Scalability is tightly connected with performance, and the system must be able to down- or up-scale without degrading the performance.

Key attributes are the number of client connections and sessions a server shall handle, how many database connections are allowed, and the maximum allowed response time of the server (which in turn places constraints on the number of simultaneous connections, etc). In addition, scalability must be considered when designing the system; a web application that produces more server requests than necessary will up-scale poorly when compared to an application that produces a minimal amount of requests.

NFR5 - Unintrusiveness

Unintrusiveness is a quality that is removed from the choice of technology. Whether choosing to develop the application with one programming language instead of another, or applying a client-server design pattern, the quality of unintrusiveness will remain largely unaffected. Unintrusiveness is achieved by the system's ability to seamlessly integrate with the user's daily tasks and work environment, and is thus achieved by system functionality and operation, and hardware interfaces rather than the technology implementing the system. Thus, the fulfillment of this non-functional requirement is not dependent upon the choice of implementation technology used in developing the system, but rather depends on the design choices made when designing for usability, flexibility and other software system attributes.

NFR6 - Usability

Usability refers to the ease with which the system and user communicate [Sch98], and is tightly interconnected with the unintrusiveness of the system. When designing a system with usability in mind, the system can be viewed from two different angles, the design of the graphical user

interface (GUI), and the design of the processes the user performs when using the system. However, these prospects are tightly interconnected, as a high-quality GUI can improve the usability of the user-system communication processes amongst others by improving the learnability of the system.

When designing a system with usability in mind, it is important not only to make the graphical user interface intuitive and easy to grasp, it is also important to ensure that tasks are not complicated unnecessary. Unnecessarily complicated tasks implies that the user uses more keystrokes than necessary performing the task, the work becomes tedious, and thus the usability is rendered poor.

When implementing a system of high usability, certain guidelines exist (see Appendix D), based upon best practices and experiences of the industry. Simplicity and consistency are two important features that should be considered during the design process. Other guidelines regards features such as learnability and memorability. Following these guidelines are important to achieve a high usability.

Chapter 17

Selection of architectural viewpoints

Architecture is described by different views, where each view conforms to a viewpoint. A viewpoint determines how a view is created, illustrated, and analysed by establishing conventions concerning the languages employed for conveying a view. The viewpoints employed in this architectural description reflects the stakeholders identified in Chapter 15, and adheres to the “4+1” view model of software architecture created by P. Kruchten [Kru95]. This model is depicted in Figure 17.1. This model starts with a *logical viewpoint*, representing the functionality of the system. These functional components are mapped onto run-time processes in the *process viewpoint*, and onto development modules in the *development viewpoint*. The processes and modules of these views are in turn mapped onto physical hardware in the *physical viewpoint*. The scenarios are instances of use cases, illustrating how the elements of the four views work together. The scenarios provide no new information to the architectural description, they only function as a validation and illustration tool after the completion of the architecture design. However, this architectural description will limit itself to the use of the four other views, and does not encompass redundant scenarios. The combination of the use cases defined in Part III, and the prototype development of Part V replaces the scenarios as descriptors of system functionality.

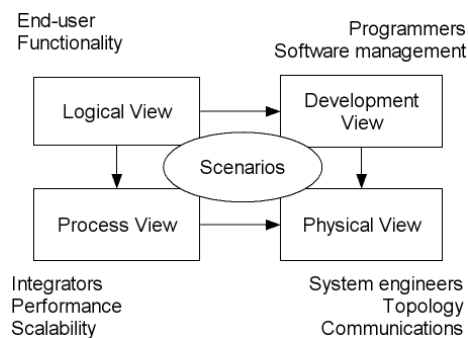


Figure 17.1: “4+1” view model of software architecture

17.1 Logical viewpoint

The logical viewpoint describes the functional requirements of the system, and applies an object-oriented decomposition of the system, describing the functionality of the system.

The Unified Modelling Language (UML) provides a notation for views conforming to the logical viewpoint. For more information on this and other notations of UML, the reader is referred

to [Fow03]. The *class diagrams* of UML describe the objects of the system, thus providing the object-oriented decomposition, and the static relationships between these objects. Figure 17.2 summarises the most important concepts of class diagrams.

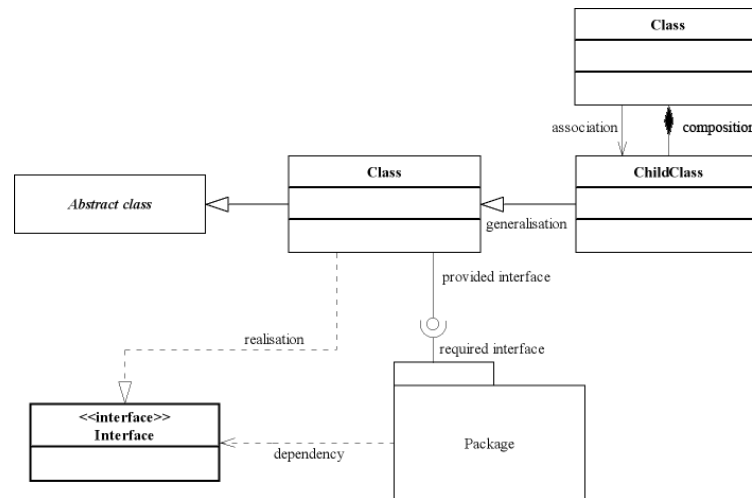


Figure 17.2: UML concepts - class diagram

The *component diagrams* of UML describe the modules of the system, their interrelationships through interfaces, and their decomposition. Figure 17.3 summarises the concepts of component diagrams.

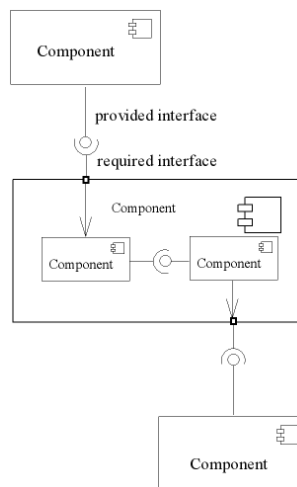


Figure 17.3: UML concepts - component diagram

The logical viewpoint is directed towards the users, developers, maintainers, and testers of the system, as it presents the system in a prospective that is useful for these stakeholders. Further, the viewpoint addresses the concerns regarding the system's ability to ensure requirements traceability by addressing the functional requirements of the system, which is the primary rationale for applying the logical viewpoint.

17.2 Process viewpoint

The process viewpoint addresses the behavioural aspects of the system, mapping the functional components of the logical viewpoint onto run-time processes. When describing the run-time processes of the system, the process viewpoint also takes into account some of the non-functional requirements of the system, such as performance and availability.

The UML concept of Sequence diagrams is used for modelling the process viewpoint. *Sequence diagrams* models the main processes of the system, where the actions of the system (small independent operations containing key functionality) are included. The actions are described by means of use cases, explained in Section 12.3.1. Figure 17.4 summarises the concepts of sequence diagrams.

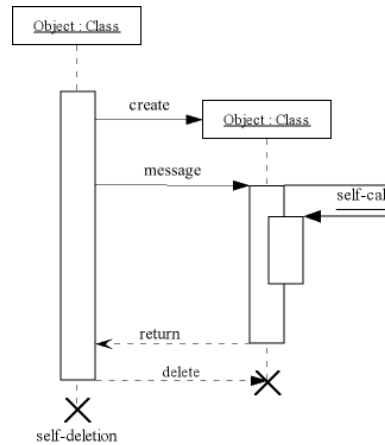


Figure 17.4: UML concepts - sequence diagram

The process viewpoint is found most interesting by the developers, testers, and maintainers, and addresses the concerns of organisation integration, e.g. by allowing the stakeholders to review how much interaction the system requires, which in turn affects the usability of the system. In addition, the performance and availability of the system can be understood through analysing the models of the process viewpoints, thus rendering the viewpoint interesting to maintainers. Thus, the process viewpoint is included in the architectural description amongst others due to its abilities to focus on the non-functional requirements of the system.

17.3 Development viewpoint

The development viewpoint maps functional components onto development modules, thus describing the actual organisation of the software modules within the system. The development viewpoint is partially regarded as an external viewpoint, assisting activities that are external to system functionality and processes, such as work assignment, team organisation, project progress, and cost evaluation.

The development viewpoint uses a simple layered notation for presenting the layers of the system. System components represented with the UML notation for component diagrams (see Figure 17.3) are divided into layers with the use of boxes, and dependencies between layers indicated by arrows.

The other notation of the development viewpoint is taken from UML, where the *class diagrams* describe the system components established in the logical viewpoint in a more detailed manner, decomposing them into packages and classes. This notation is also employed by the logical viewpoint, and Figure 17.2 summarises the concepts of class diagrams.

The development viewpoint is primarily aimed at the needs of the acquirers, developers, testers, and maintainers of software, as it focuses on how the outlined system is to be developed.

17.4 Physical viewpoint

The processes and modules of the process and development viewpoints must be mapped onto physical hardware. This mapping is described in the physical viewpoint with the means of *deployment diagrams*, describing which modules are deployed where. Thus, the physical viewpoint

pays attention to the performance requirements of the system, enabling the representation of deployment of multiple resources. Figure 17.5 summarises the concepts of deployment diagrams.

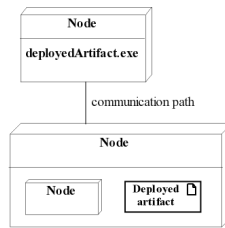


Figure 17.5: UML concepts - deployment diagram

The physical viewpoint is primarily of interest to the testers and maintainers of the system, as these are the stakeholders in charge of testing that the system fulfils its performance requirements, and ensuring that the system continues to fulfil the performance requirements throughout its lifecycle.

Chapter 18

Architectural views

This chapter presents the views of this architectural description, describing the system from several angles. The architectural views conform to the four viewpoints defined in Chapter 17. For each viewpoint, a set of views can be defined, but each view can only conform to a single viewpoint. The models presented in the views must conform to the defined notations of the belonging viewpoint, following the UML standard [Fow03].

The architectural decisions made in this chapter originate from the discussions in Chapter 16, and are explained further in Chapter 20, stating the architectural rationale. A separation of the presentation of the architectural choices and the reasoning behind these choices, i.e. the rationale, is in accordance with the IEEE Std. 1471-2000, and allows the reader fast access to any required information.

18.1 Logical view

The logical view presents the services the system must be able to provide to the user, and is strongly interrelated with the functional requirements of the system. Chapter 16 discusses a range of architectural strategies and styles that assist in fulfilling the non-functional requirements of an RT tool. It was concluded that a component-based architecture combined with a client-server design pattern will provide a maintainable, platform-independent, and adaptable architecture. This section presents the logical structures of the system, conforming to the chosen architectural styles. Before these structures can be presented, an important concept referred to as the domain model must be explored further, as this concept serves as the foundation for all other logical structures of the RT tool. Section 18.1.1 presents this concept, before focus is returned to the high-level logical structures of the tool in the sections that follows.

18.1.1 The domain model

The functional requirements of the requirements specification in Part III describe what users require of the system, outlining the *problem domain* in which the system will exist. A *problem domain analysis* examines the problem domain in detail, and creates a domain model, representing the problem to be solved without regard to any potential solution. This domain model will then serve as a problem description on which a number of different solutions can be founded. The traceability model referred to as TRACY (first introduced in the in-depth study, then remodelled in Part II) serves as a first approach to a domain model, and is shown in Figures 6.1 through 6.3. However, the traceability submodels of TRACY are inadequate for architectural purposes, as they do not comply with the object-oriented paradigm. Figure 18.1 presents the object model of the problem domain, where each artifact of the traceability submodels is represented as an object. Each object belongs to a specific traceability stage, as determined by the traceability submodels, and the boundaries of the submodels are indicated by packages.

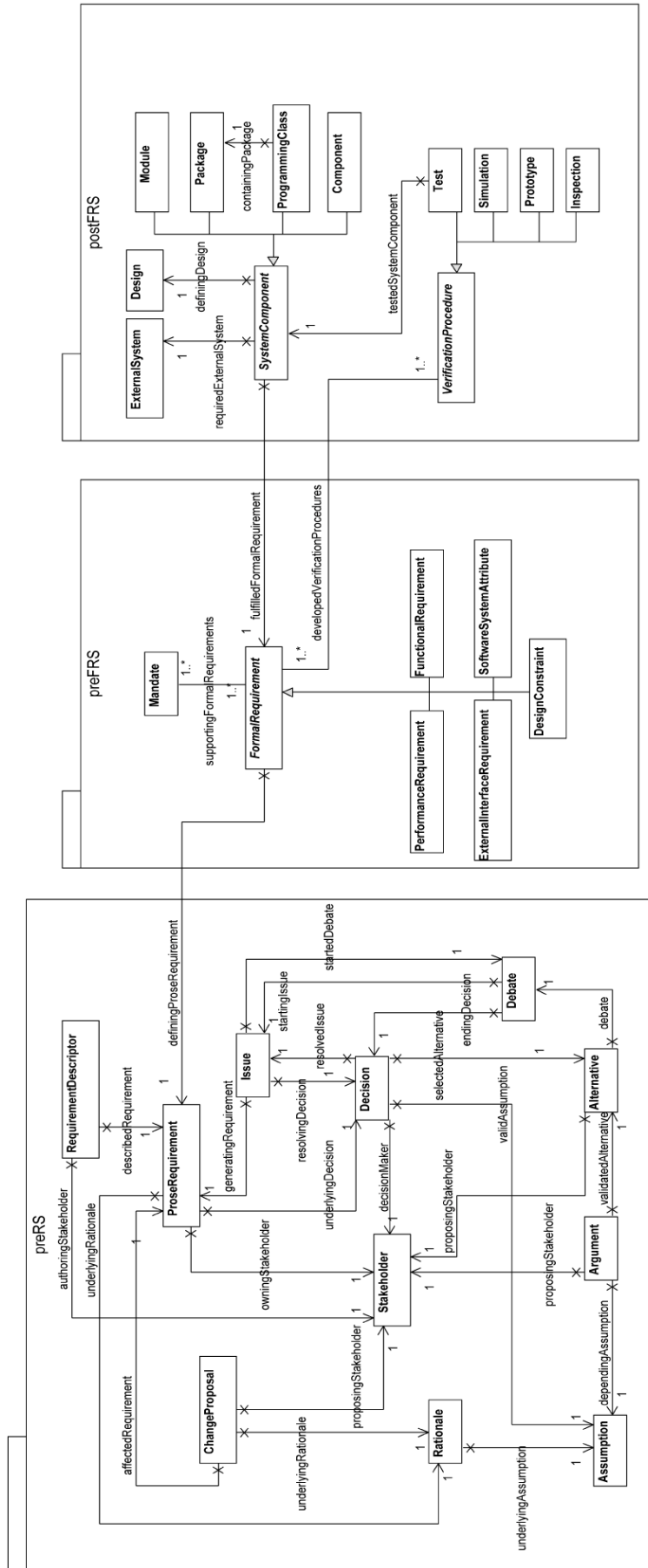


Figure 18.1: Domain model

Critical to the ability to provide full traceability (i.e. the ability to trace between artifacts of all traceability stages) are the connections between the traceability stages, which does not exist between the traceability stages themselves, but rather between key artifacts of the stages. These key artifacts were pointed out in the traceability submodels, referred to as *key connecting artifacts*. In TRACY there are two key connecting artifacts, i.e. *ProseRequirement* and *FormalRequirement*. These remain key connecting artifacts, but in addition we need two new artifacts, *SystemComponent* and *VerificationProcedure*, as the term key connecting artifact is broadened to include both parties of the relationship tying together two traceability stages.

Each of the objects in the domain model of Figure 18.1 are shown only with their name and their associations to other objects, as this provide a simplified, yet substantial overview of the logical architecture of the model. However, more details could provide useful when we need an understanding of the high-level architecture. This issue is further addressed in the development view of Section 18.3.

18.1.2 Component-based architecture

The domain model presented in the previous section represents a description of the problem. A solution must use this model, providing constructs that encapsulate it and assist with solving its problem. A component-based architecture combined with a layered approach, as described in Chapter 16, allows the domain model to be encapsulated at a separate layer, referred to as the business layer. In addition, applying the MVC design pattern (also described in Chapter 16) enhances the compartmentalisation of the system. The rationale behind a component-based architecture encapsulating the model is discussed further in Chapter 20. Figure 18.2 shows a high-level representation of the logical structure of the system as a class diagram, conforming to the MVC design pattern.

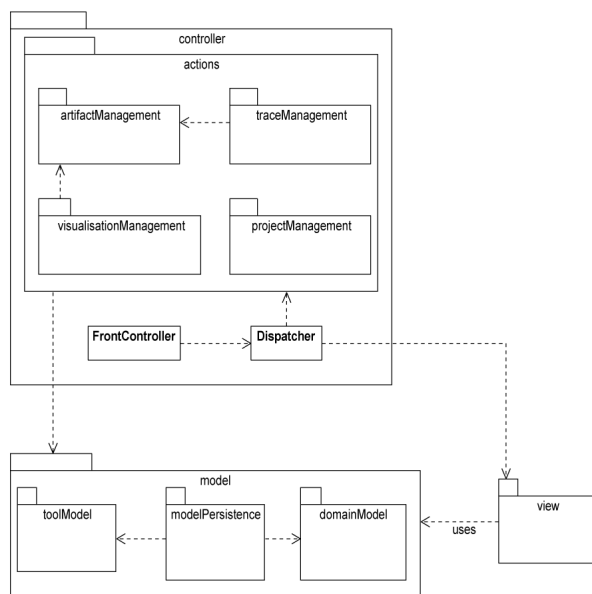


Figure 18.2: High-level logical structure

As a result of using the MVC design pattern, the system is divided into three main components: the controller, the model, and the view, working together on performing the system tasks. The controller represents the authority within the system, and the main functionality of the system is implemented with the means of *actions*, existing within the controller.

An action represents a single task within the system, causing an update of the domain model, and will often have an equivalent among the functional requirements or the use cases, as it represents a basic functionality of the system. The actions are supported by a framework

consisting of a FrontController and Dispatcher, which control which actions are performed and selects the views presenting the results of the chosen action.

As illustrated in Figure 18.2, the controller depends on both of the two other main components of the system, whereas the view depends on only the model, and the model depends on neither of the other two components. This ensures encapsulation of the model, an important aspect of reuse and evolution of code.

Figure 18.3 shows a detailed representation of the logical structure of the system as a component diagram, where the three main components of the system are decomposed.

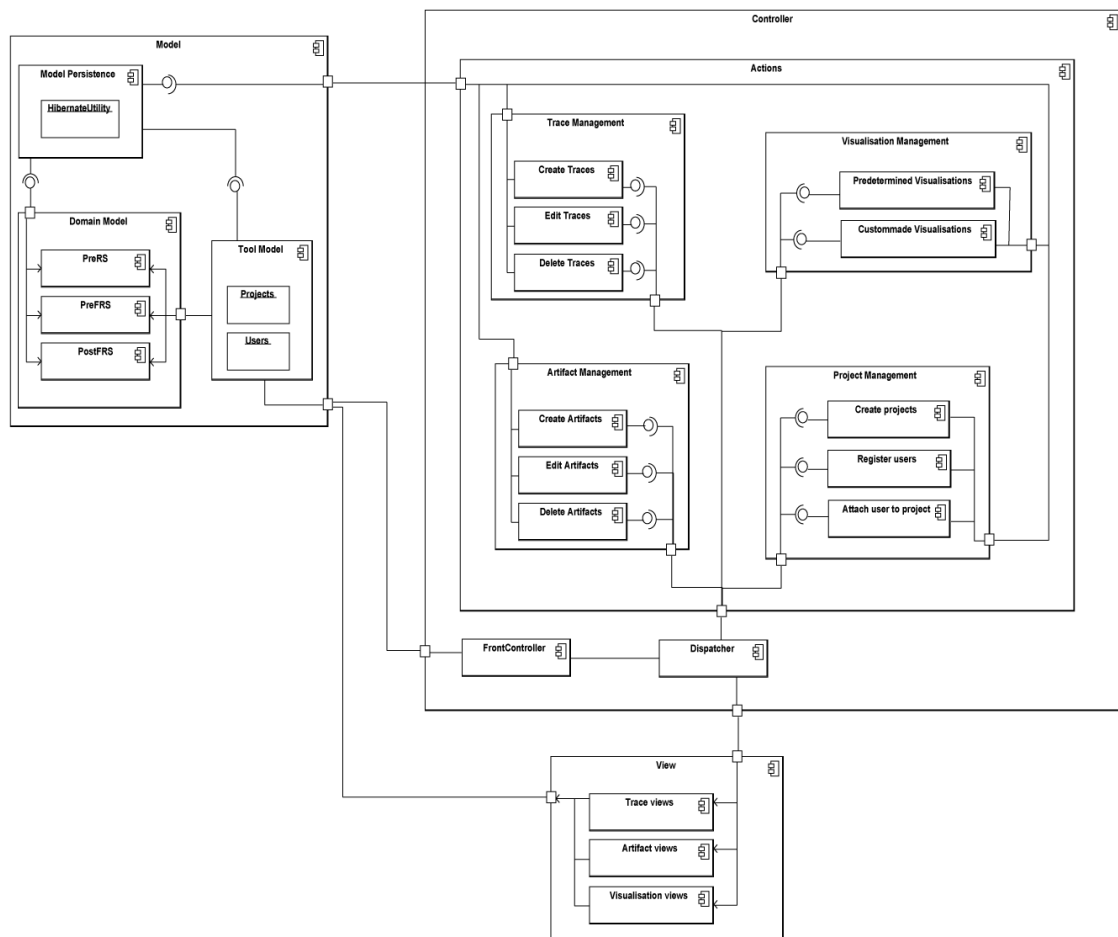


Figure 18.3: Logical structure

Figure 18.3 shows how the MVC pattern is implemented, encapsulating system behaviour in the Controller component, the domain model in the Model component and the presentation of data (the views) in the View component. Each of these main components are decomposed into several smaller components, and are discussed below.

The Model Component

The responsibility of the Model Component is to create a representation of the domain model, shielding it from the remainder of the system. The model must provide persistence mechanisms, and interfaces towards the other components of the system.

The Model component are decomposed into three smaller components. Their importance are equivalent, but their purpose vary. The Domain Model subcomponent represent the domain model, sorting the objects into subcomponents according to their traceability stage. The Domain Model subcomponent exists independently of the other components in the Model com-

ponent, and are not aware of any of them. Thus, the Domain Model subcomponent does not rely on any other components to provide its functionality, and the other subcomponents of the Model component make use of its interfaces.

The Model Persistence subcomponent's sole responsibility is to assure persistent storage of the Domain Model subcomponent's data. With the means of HibernateUtility, the Model Persistence subcomponent provides an interface used by the Controller component for persisting the domain model to the database, and removes all references to a database from the domain object model, providing encapsulation of the model. This is accomplished with the means of Data Access Objects (DAOs), a common design pattern [CK03]. A DAO contains all necessary data access code for a specific object of the domain model, and defines an interface to persistence operations relating to a specific object of the domain model.

The Tool Model subcomponent models the RT tool itself, rather than the problem domain, and the full tool model can be seen in Figure 18.4.

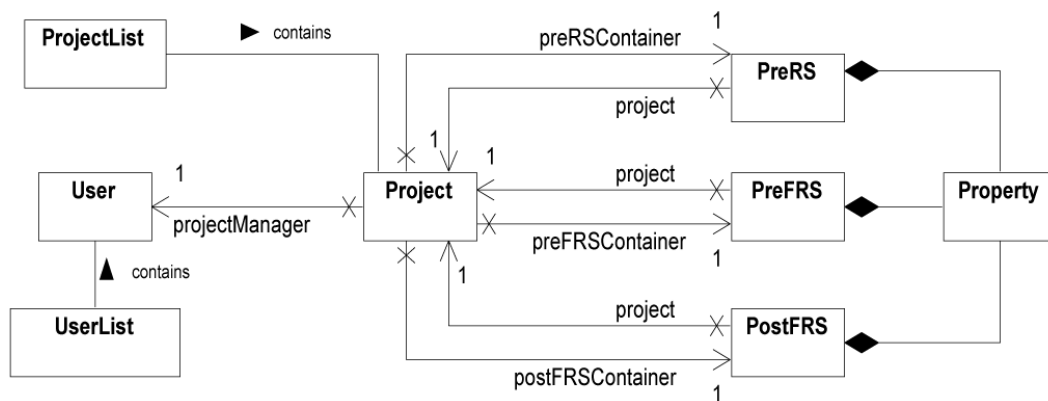


Figure 18.4: Project model

A Project object represents a software development project, and the user can attach any object (i.e., artifact) of the domain model to a project. Thus, each project contains a collection of objects from the domain model. In order to control these objects, a project contains three traceability stage containers, which represent each of the three traceability stages. A project's artifacts are sorted into the appropriate traceability stage container. The Property object is a helper class that helps the traceability stage containers describe the properties of their contained artifacts. The RT tool can contain several projects, each of which encompasses an independent set of objects from the domain model, separated into traceability stage containers. Several users can be attached to a single project, and the RT tool keeps track of its users and projects by maintaining a user list and project list.

The View Component

The main responsibility of the View component is to present the data chosen by the Controller component and held by the Model component. The View component is decomposed into three subcomponents: the Trace Views, the Artifact Views, and the Visualisation Views. This decomposition supports the functional decomposition of the Controller component. Each of these View subcomponents contains two types of views: views presenting functionality the system provides (represented by the actions of the Controller component), and views presenting the data of each artifact and trace. Thus, the Controller component determines which views must exist in order to provide the functionality implemented by the actions, whilst the Model component determines which views must exist in order to present the data of the objects of the domain object model.

The Controller Component

The Controller component's main responsibility is to control the behaviour of the system, with the aid of the Model and View components. The Controller component uses the other two main components of the system to provide the functionality demanded by the functional requirements, and represents the authority within the system.

The Controller component is decomposed into six subcomponents. Two of these, the Main Controller and the Dispatcher component, focus on controlling the behaviour of the system, whilst the remaining four, the Trace Management, Artifact Management, Visualisation Management, and Project Management components, implement actions that provide the functionality prescribed by the requirements specification.

The Main Controller intercepts all requests directed to the system, performs any common tasks that apply to all requests, and then forwards the request to the Dispatcher. The Dispatcher then examines the request and selects the appropriate action, contained within one of the three components implementing actions (Trace Management, Artifact Management and Visualisation Management). The action performs the required manipulations and updates of the domain model, after which the Dispatcher selects the appropriate view for displaying the data. Thus, the actions are not responsible for view selection, as all navigation is encapsulated in the Dispatcher. Functionality that does not require updates of the domain model, only access to existing data, do not require an action. Instead, a view will suffice, presenting the required data. This process is described further in Section 18.2. The Main Controller uses the Tool Model subcomponent (in the Model component) as a source of information assisting with maintaining control over the administrative aspect of the RT tool, such as existing projects and users, which users are attached to which projects, and the collection of artifacts within each project.

The actions implemented within the Controller component uses the Model Persistence subcomponent of the Model component. This gives them access to the persisted data held by the Domain Model component, an important part of the functionality provided by the system. Actions perform the necessary operations and updates on the data, depending on the request initiating the action. The updated data is then persisted to a database, and displayed to the user with the means of the View component. This process is further described in Section 18.2.

18.2 Process view

The process view describes the behavioural aspects of the system. The main abstractions of the logical view are mapped onto process elements, describing how each component's operation is executed. The presentation of the process view begins with a discussion of the chosen architectural style, three-tier client-server, before focusing on the models describing the view.

18.2.1 Three-tier client-server

Employing the architectural style referred to as *three-tier client-server* implies designing the RT tool as a web application with a separation between a client tier, a presentation tier, and a business tier. This assists with ensuring platform independency on the client side, as several types of clients can be supported. The three-tier Client-Server design pattern is highly compatible with the component-based architectural approach, as it exhibits a natural division into layers (i.e., tiers). As the mapping of system components to processing nodes (e.g. application and database servers) is further discussed in the physical view of Section 18.4, this section will limit itself to elaborating the flow of control within the system, i.e. the internal processes.

Figure 18.5 depicts the high-level process blueprint of the system when receiving a request from a user, and is an implementation of the Service to Worker design pattern. The process conforms to the use of the Model-View-Controller design pattern as prescribed by the logical view. The process resides in an execution environment (i.e., an application server), providing a means of sharing variables between e.g. requests or sessions, referred to as scope variables.

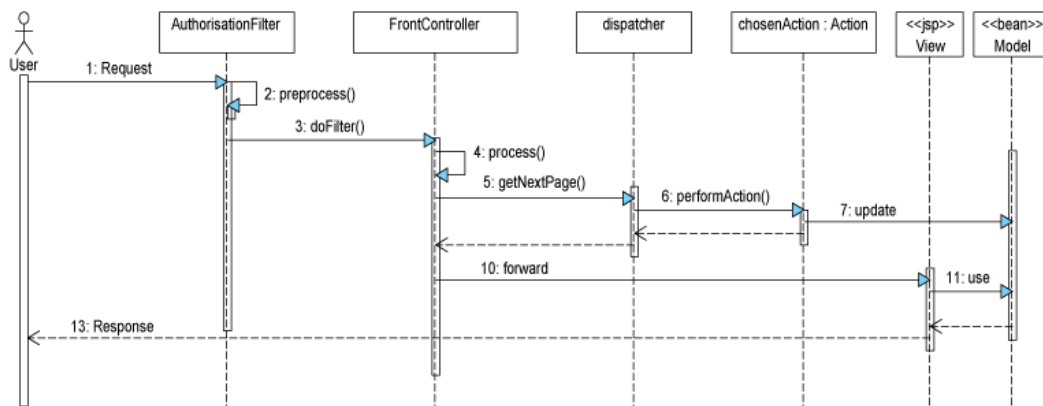


Figure 18.5: High-level process architecture

The execution environment receives all client requests, and passes them through an authorisation filter. If the user is not authorised, the filter initiates an authorisation, and when this has been completed, the FrontController checks to see whether a Dispatcher instance exists as a scope variable. If so, the FrontController uses the existing Dispatcher instance, which may have a different internal state than a newly created instance. If no Dispatcher instance exists as a scope variable, the FrontController creates a new instance of the appropriate dispatcher, based on the nature of the request. When a dispatcher has been found, the FrontController allows it to continue the processing of the request.

The Dispatcher, which is a state machine, consults its inner state, performs the action that belongs to its current state, and informs the FrontController of which view should be presented. The FrontController then forwards control to the selected view, which presents the requested data to the user.

The actions are an important part of this blueprint, as this is where the main functionality of the system resides. Depending on the request, different actions will be invoked by the Dispatcher. Use cases G.1 through G.15, found in Appendix G, describes the actions of the system.

18.3 Development view

The development architecture focuses on the organisation of the software components. Starting with the components specified in Section 18.1, the development view reveals further detail and establishes smaller units that can be developed by one developer or a small team. However, before reviewing these units, we present the organisation of the software components of the logical view into layers. Figure 18.6 presents the software components of the system in a layered style. A component in a certain layer can only depend on components in the same or lower layers.

As shown in Figure 18.6, the models of the bottom layer are independent of all other components, due to their problem-describing nature, as explained in Section 18.1.1. The models only describe the problem, they do not offer any solution. The solution construct is provided by the other layers of the system, which encapsulates the models and provide the mechanisms that constitute an RT tool. Thus, all other layers are directly or indirectly dependent on the Model layer.

The View component accesses the Model Persistence component to fetch trace information to display to the user. If executed by the Dispatcher, an Action will also connect to the Model Persistence component, performing updates and/or manipulations of the data. At the topmost layer, the Behavioural layer, the FrontController and the Dispatcher, which determines the behaviour of the system, resides, and are dependent on all the other layers in order to perform their tasks.

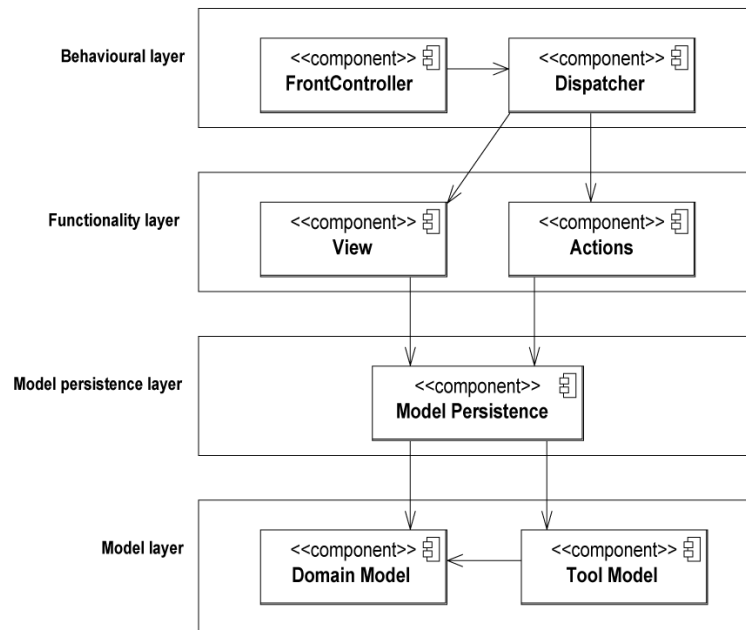


Figure 18.6: The layers of the RT tool

Figure 18.6 suggests the order in which the components can be developed, as it is advisable that development start with the component at the bottom layer, i.e. the components with no outwards dependencies. The components, however, only provide a high-level overview of the system, identifying only the major elements of the software. Figure 18.7 provide some additional detail to these major elements, explaining how the components function and interact.

The chosen main architectural style applied to the system models a three-tier client-server application. Chapter 16 discusses how Java EE, a programming platform specialising in developing and running distributed Java applications, should be employed to create a platform-independent application. In this view, we briefly describe some Java EE-specific details concerning the development of a web application, but as the application of a programming language falls outside the scope of the architectural description, most details are referred to Part V, describing the development of a RT tool prototype. However, the details regarding deployment of the web application are discussed in the physical view of Section 18.4.

The Java EE technology employs regular java classes for implementing system functionality, but in addition introduces the concept of servlets and Java Server Pages (JSP files). Servlets are created by extending the `HttpServlet` class provided by the Java EE programming platform. As shown in Figure 18.7, the `FrontController` is a servlet, as it extends the `HttpServlet` class. This allows the `FrontController` to process requests from clients, determining the proper action required to return a response. As explained in the process view of Section 18.2, the `FrontController` lets the `Dispatcher` decide the required operation, actions are performed if necessary, and a view is chosen, providing a response that is returned to the client.

The model package of Figure 18.7 (the development view equivalent of the Model component of Figure 18.3) is based on the Java standard edition programming language, and is encapsulated by the other Java EE-compliant components. Thus, the model package has no knowledge of the Java EE programming platform, allowing it to be reused by other applications. The only dependency of the model package is created by the model persistence package, which implements the Hibernate framework for persistent storage of the objects of the domain model. Hibernate acts as an interface to a number of database management systems, abstracting their operations and consequently allowing the application to disregard any vendor-specific details concerning the use of a database. The views are implemented as JSP files, allowing both dynamic and static content. The dynamic content is fetched from the domain model, represented in Figure 18.7 by the `domainModel` package.

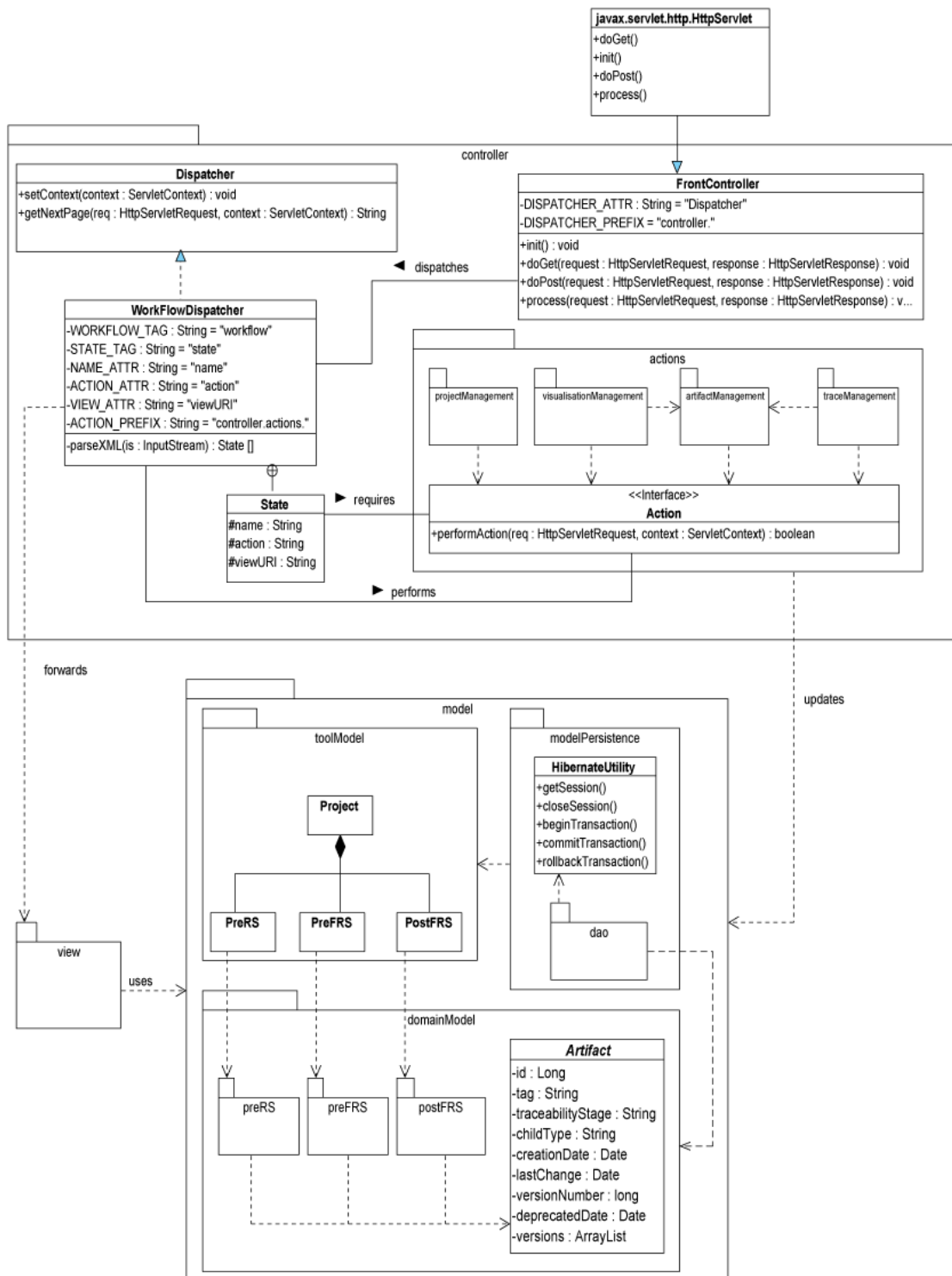


Figure 18.7: Detailed components

The package structure of Figure 18.7 correspond to the components determined in the logical view of Section 18.1. When a package has been considered too complex to develop without additional information, it has been decomposed, revealing its internal structure. The dependencies and associations between the objects of the domain model are only depicted in Figure 18.1, allowing the presentations of the detailed specification of the domain model to focus on the internal structures of each object. Consequently, the implementation of the domain model will not be further addressed here, but referred to Part V.

18.4 Physical view

The physical view presents how the software is mapped onto the hardware providing its execution environment, primarily focusing on the non-functional requirements of the system. Figure 18.8 shows how the web application is deployed on a Java EE-compliant application server, where a web container provides an execution environment.

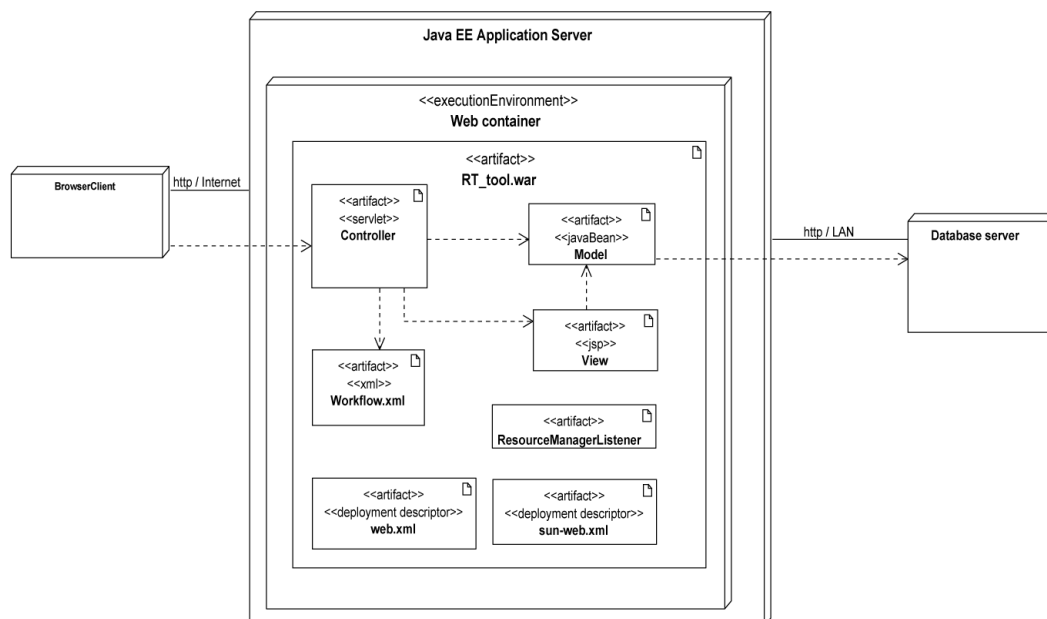


Figure 18.8: Deployment of the RT tool

The web application is packaged into a JAR (a Java library) file with a .war (Web ARchive) extension, depicted in the figure as `RT_tool.war`, and is referred to as a web module. A web module contains compiled servlets, JSP files, supporting class files, and any required static resources, such as GIF images or HTML files. Figure 18.8 shows the most important parts of the contents of the web module package, consisting of the three main components of the web application, the model, the view, and the controller. In addition, four other artifacts are displayed, two deployment descriptors, XML files defining the workflow of the application, and a `ResourceManagerListener` responsible for initialising the web application when first deployed.

The deployment descriptors are XML files that describe the deployment settings of the web module, and two types exist. The Java EE deployment descriptor (shown in Figure 18.8 as `web.xml`) is used to configure deployment settings on any Java EE-compliant implementation. The runtime deployment descriptor (shown in Figure 18.8 as `sun-web.xml`) is used to configure Java EE implementation-specific parameters. Together, the two deployment descriptors ensures that the web application is configured and functions properly.

The XML files defining the workflow of the system is read by the Dispatcher within the Controller component at runtime, allowing the system's behaviour to be altered without changing any source code or recompiling it, ascertaining application adaptability.

The `ResourceManagerListener` is only called by the application when first deployed, and is responsible for initialising the resources required by the application, primarily the `HibernateUtility` of the Model component, ensuring model persistence.

The web application connects to a database server, employing its services when persisting the model, and the `HibernateUtility` of the Model Component provides the required interface for communicating with the database server.

Chapter 19

Consistency among architectural views

Chapter 18 presented four views, in accordance with Kruchten's 4+1 view model of software architecture [Kru95]. This chapter focuses on describing the consistency among the views. As shown by the arrows in Figure 17.1, the process view and the development view are based on the logical view, whilst the physical view is in turn based on the process view and the development view. A similar approach is taken in this chapter, discussing the consistencies among the views in the same order as they were defined. Thus, the chapter commences with a description of the consistencies among the logical view and the process view, continuing with the consistencies among the logical view and development view, concluding with the consistencies among the process and development view and the physical view.

Logical view to process view

The logical view presents the main functional components of the system, amounting to three components, the model, the view, and the controller. These main components were decomposed, revealing their functionality. The process view discusses how the system behaves to provide this functionality, using the same (decomposed) components as the logical view. This ensures consistency among the views. In addition, the process view extends the presentation of the actions in the system, as each action is presented as a separate workflow.

Logical view to development view

The development view layers the components presented in the logical view, pointing out the dependencies between the components. The same components are employed, only viewed from a different viewpoint emphasising the dependencies between the components rather than their functionality. Further, the development view presents a detailed decomposition of the logical structure presented in the logical view, establishing units of software ready for development.

Process and development view to physical view

The physical view explains how the web application is deployed to an application server, and specifies some deployment-specific artifacts not previously described by any of the other views, i.e., the workflow XML files, the ResourceManagerListener, and the deployment descriptors. These artifacts, with the exception of the workflow XML files, are only useful to consider in the physical view, as they only influence the deployment of the application. The workflow

XML files, however, describe the actions of the system, and are based on the action workflows presented in the process view. In addition, the view shows how the three main components of the application, the model, the view, and the controller component, are mapped onto a web container providing an execution environment, allowing them to provide their services to the client, accessing them through a browser.

Overall consistency

The consistency among the four view is overall good, and the views generally discuss the same components, but from a different viewpoint. Decomposition of components occur, to provide a more detailed view, but these decompositions do not affect the consistency among the views. The only exception is the specification of deployment-specific artifacts in the physical view, which are not discussed in any other view. The process view could have included the artifacts, but as they focus on factors outside the internal processes of the application (the focus area of the process view), such as mapping of requests, they were not.

Chapter 20

Architectural rationale

Chapter 16 discusses the choice of architectural strategies for the RT tool, strategies that assist in fulfilling the non-functional requirements of the system. The rationale behind the chosen strategies is partly discussed absently, and this chapter completes the discussion by presenting the full architectural rationale.

The strategies suggested in Chapter 16 can be divided into two categories. One category contains all high-level strategies, i.e., strategies that places constraints on high-level design decisions. The discussions emphasised that building a component-based architecture ensures adaptability, and by extending it with the use of the Model-View-Controller (MVC) design pattern, a robust and extensible architecture is created. Further, using a client-server design pattern and building the system as a web application rather than a local desktop application improves the platform independency. A database management system ensures persistent storage of data. The low-level strategies are of a more technical nature than the high-level strategies, focusing on distinct technologies, such as programming languages and frameworks, thus removing itself from the unbiased approach of an architectural description. Other than the short introductions given in this part, these low-level solutions will not be discussed further until Part V, where they will be revisited.

The discussions emphasised that employing the Java programming language improves the platform independency of the system, as this is a programming language that relies on a virtual machine, enabling cross-platform deployment. In addition, the Java alternative is attractive due to previous development experience. Using Java as the programming language of choice enables the use of the Hibernate framework, simplifying the task of storing object-oriented information in a relational database, e.g. by allowing automatic database schema generation from a domain model, thus eliminating the need to design both a domain model and a corresponding database schema. In addition, the Hibernate framework provides an abstraction of the database, allowing the system to ignore the nature of the database operations. Further, as the Java EE technology is especially designed to develop web applications, its use coincides with the client-server design pattern, a chosen high-level solution.

The MVC design pattern, which creates a clean cut between system data, presentation of data, and system behaviour, has been extended with the Service to Worker pattern, a pattern that specifies in detail how the system should handle a large and complex implementation of the MVC pattern, ensuring a clean separation of concerns among the components of the system. A division of labour is particularly useful in a web application, where the FrontController can screen all user request and perform common functionality such as authorisation and navigation, whilst the distinguished Dispatchers focus on manipulating the required model and views. This will in turn increase the adaptability of the system, as replacing components becomes a less complex task. In addition, taking care when designing the actions of the Controller component can enhance the scalability of the system by reducing the amount of generated requests, and decrease the intrusiveness of the system by requiring the minimum amount of user interaction, which in turn enhances the usability of the system.

Part V

Prototype development

Chapter 21

Introduction

SO far, a requirements specification (see Part III) and an architectural description (see Part IV) has been presented, describing an RT tool. This chapter presents a prototype of the RT tool, referred to as TraceMe.

21.1 Purpose

The purpose of this chapter is to describe how an RT tool could be implemented, based on the given requirements specification and architectural description. An evolutionary prototype has been developed, and is described and evaluated in this part.

21.2 Scope

The developed prototype focuses on exemplifying how an RT tool could provide full requirements traceability. Consequently, the prototype concentrates on how trace information is gathered from and conveyed to the user. The following chapters reflect this, and describes how the system presents its information rather than underlying technical solutions. However, a short summary of the technical aspects of the prototype is also given. In addition, it falls within the scope of this part to describe the work remaining to turn the prototype into a usable application.

21.3 Overview

An overview of the structure and a quick summary of all chapters in this part is given below.

- **Chapter 22 - Implementation**
This chapter provides a short description of the underlying technical solutions of the prototype, clarifying the design of the prototype.
- **Chapter 23 - Using the prototype**
In this chapter, the functionality of the prototype is described. Particular attention is paid to how the system is used, i.e., system workflow and the interaction between the system and its users.
- **Chapter 24 - Prototype evaluation**
This chapter evaluates the prototype, paying particular attention to the fulfilment of the functional requirements.

- **Chapter 25 - Prototype evolution**

The prototype does not represent a usable application, and this chapter focuses on the remaining work that must be done before the prototype is rendered usable in real life, thus developing it from a prototype into a full-scale RT tool.

- **Chapter 26 - Summary**

A summary of the results presented in this part.

Chapter 22

Implementation details

This chapter discusses the implementation of an RT tool based on the decisions made in Parts III and IV, and starts with a discussion on the technology used for implementing the RT tool. Further, the chapter discusses the implementation of the prototype, pointing out how it differs from a complete RT tool implementation.

22.1 Technical details

The implementation of the RT tool outlined in Parts III and IV requires the use of technical solutions previously discussed in Chapter 16. In other words, the RT tool is dependent on its employed technology. This is primarily a software dependency, and not a hardware dependency, as the application places no extraordinary requirements on the choice of hardware technology, the requirements specification even emphasises that the system is to be implemented in a hardware-independent fashion (see Section 12.1).

The discussion on architectural strategies concluded with the use of Java EE [EE] as its primary technology, with assistance of the Hibernate framework [Kin04] for object-relational mapping (ORM) purposes. In order to develop and deploy web applications, Java EE requires a container, also referred to as an application server. Java EE is freely distributed by Sun Microsystems, and this distribution includes the application server *Sun Java System Application Server*. Open source solutions providing analogous container capabilities exist, but due to simpler installation, the bundled server is employed in the implementation of the RT tool. Thus, the RT tool is dependent on the continued support and updates from Sun Microsystems for the Application Server. If problems occur, the RT tool could migrate to any of the open source solutions supporting Java EE, such as Apache Tomcat [Tom].

The RT tool also assumes the availability of a relational database, for persistent storage of traceability data. In addition to the Application Server, a relational database named Derby is distributed with the installation of Java EE, and is used by the RT tool due to the ease with which it is combined with the Application Server. Other relational databases exist, one popular option being MySQL [MyS], and thus the RT tool could migrate to this or another solution if support for Derby discontinues.

Java EE and any related technologies, e.g. application servers and Hibernate, are popular technologies at the present. However, in the future new technologies will probably be introduced, maybe leaving present technologies obsolete. Still, this is not a factor that can be reckoned with, as it is far too difficult to predict, and even so, hard to counteract.

22.2 Prototype development

The technique employed for developing the prototype is referred to as evolutionary prototyping [Vli00b]. Starting with the requirements, a basic prototype is created, which acts as a first version of the RT tool. The main purpose of evolutionary prototyping is to create a prototype that can evolve to the final product. Rather than spending time and effort focusing on the technical details of the implementation of the tool, attention should be directed at the tasks of gathering and presentation of trace information. Consequently, the first version of the RT tool focuses on the information it contains rather than technical details concerning e.g. persistence, concurrency, or performance. The user starts working with the first version, which in most cases spurs changes to the requirements. The next version can then be developed, and after a number of such iterations, the final RT tool can be delivered. The prototype developed in this thesis represents a first version of the RT tool, as it focuses on illustrating how the functional requirements can be fulfilled, rather than making large investments of effort realizing the system. Technical details of a web application are not of interest, and a prototype helps focusing on those aspects that are important: delivering functionality that provides full requirements traceability in a user-friendly and cost-effective manner providing its users added value. As a result of the focus area of the prototype, it contains what could be referred to as alpha functionality, i.e., functionality that has not yet been implemented, but provides the user with an understanding of how it is intended to work. This underlines the evolutionary nature of the prototype.

Even though the prototype primarily focuses on illustrating the functionality of the system, it is important when developing an evolutionary prototype to give attention to software system attributes regarding adaptability and changeability. As an evolutionary prototype, the prototype is intended to be altered, perhaps frequently, and the ease with which it can be altered is vital for the success of the prototyping process. Consequently, attention is given those architectural strategies of Part IV that focus on the adaptability of the implementation of the tool. The prototype is developed with a component-based architecture, ensuring simple expansion of functionality. The Model-View-Controller design pattern is employed in a simplified manner, making sure that the simplifications does not prohibit transition to the complete design pattern. In addition, the prototype is shaped in conformance to the client-server architectural style. As extensive knowledge of the implementation details are not required to discuss and evaluate the prototype, the implementation details of the prototype are referred to Appendix H.1.

The prototype has been developed as a web application (using Java EE) accessible through a web browser, and is deployed to a Sun Java System Application Server for testing purposes. The configured application server is generally not accessible, and if requiring explicit access to the prototype, e.g. for testing, the user is referred to Appendix H.2 for information on how to install an application server and deploy the prototype.

The development gives particular attention to the workflows of the system, attempting to achieve the best possible interaction between system and user. The eight golden rules of human interface design, found in Appendix D, have been used as guidelines throughout the development process. Chapter 23 presents screenshots of the prototype, showing the human-computer interfaces, and discusses how the golden rules of interface design have been applied.

As discussed earlier, spending effort on designing system workflows and human-computer interfaces implies spending less effort on the underlying technical aspects of the implementation. In keeping with this, the prototype has been stripped of functionality regarding persistence of data and concurrency management. The domain model presented in Figure 18.1 of Section 18.1.1 form the foundation of the RT tool, and has been implemented as a separate component in the Prototype, in conformance with the Model-View-Controller design pattern (acting as the Model component of the pattern). In order to reduce the complexity of the prototype, the View component and the Controller component of the MVC design pattern have been merged. Thus, in the prototype JSP files serve both as the view and the controller, determining the behavioural and presentational aspect of the application.

The prototype does not persist the model with the use of Hibernate. Instead, it relies on the combination of a predefined set of domain objects initialised at deployment, and additional objects defined by the user when using the prototype. These additional objects will be lost if

the application fails, but as it is a prototype not employed for gathering real traceability data, this is not a problem. How to implement model persistence is further addressed in Chapter 25.

Issues concerning concurrency has not been addressed by the prototype, as the prototype is intended to illustrate the concepts of requirements traceability to a single user at the time. As the prototype is developed as a web application, it is possible for several users to access the application simultaneously, although with a limited degree of success. They might experience apparently inexplicable loss of data, or other errors, as two users requests changes or access to the same domain object simultaneously. Chapter 25 discusses how concurrency issues should be handled by the final RT tool.

Chapter 23

Using the prototype

This chapter presents the functionality of the prototype (which has been given the name *TraceMe*), discussing the human-computer interfaces and how the user interacts with the system. The prototype's main goal is to give the user an impression of how traceability of requirements can be achieved, and the discussions of the prototype functionality concentrate on highlighting how the given functionality assists with providing full requirements traceability. Throughout the discussions, references will be made to the eight golden rules of human-computer interface design, pointing out how they were applied to the prototype's interfaces. The rules provide a guideline for developing human-computer interfaces of high quality, and Appendix D summarises the rules.

The functionality of the prototype is separated into subfunctionalities, each of which is described in a separate section below. The main subfunctionalities correspond roughly to the major groups of functional requirements as described in Section F.2. In addition, screenshots from the prototype will be presented, illustrating the available functionality.

Chapter 24 evaluates the prototype, and summarises which requirements are addressed by the prototype.

23.1 Accessing the prototype

When the application server to which the prototype has been deployed is online (as described in Appendix H.2), the prototype is accessed by opening a Firefox web browser and entering the following address.

```
http://localhost:8080/traceme/
```

Due to its status as a prototype, the development has given little attention to the detailed non-functional requirements of the system. However, the software system attribute *security* has been considered to a certain degree, as it is important to the concept of requirements traceability. In addition to considering this attribute when developing the functionality of the system (which will be discussed in the following sections), the system presents a login screen when it is first entered, shown in Figure 23.1.

Login functionality is alpha functionality, but it provides the user with an impression of the security of the system. Figure 23.1 shows how the system is accessed through a web browser. In future screenshots, only the screen of the system and not the surrounding web browser will be displayed.

When the user has logged on the system (with a random username and password), he or she will be greeted by the welcome screen of the prototype, given in Figure 23.2. The welcome screen greets the user by name, and displays the upcoming events within the system through a message board, such as maintenance notifications. In addition, a shortcut to the ten most

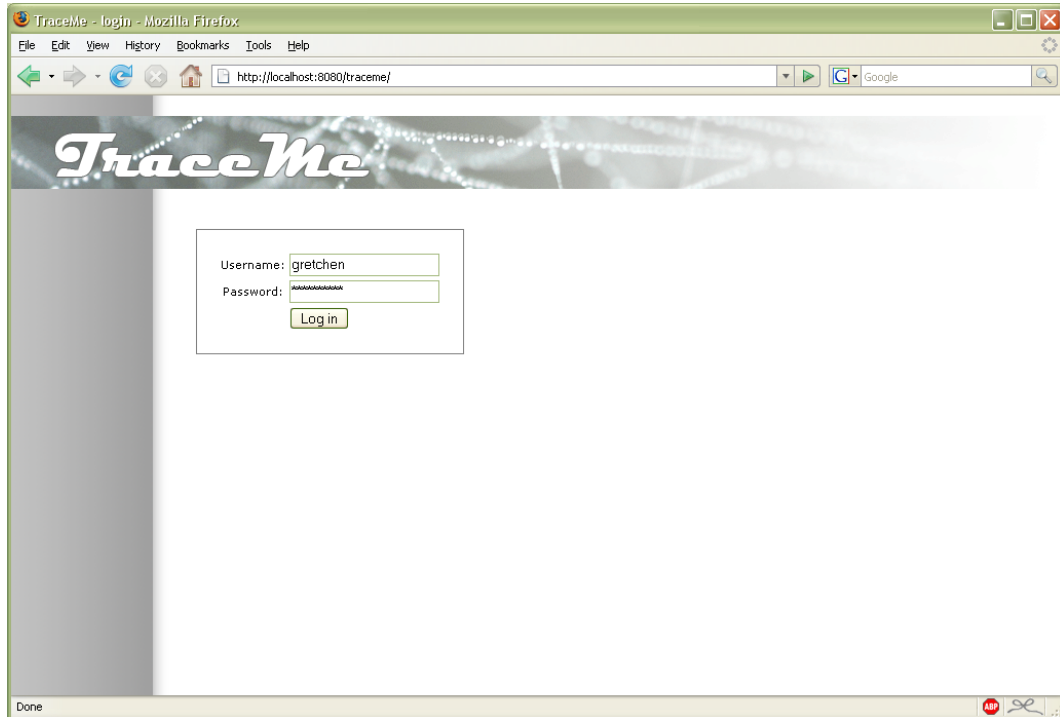


Figure 23.1: TraceMe - Login

recent projects existing within the system is given. The human-computer interface design rules recommends the use of shortcuts, as it allows frequent users to increase the pace of interaction.

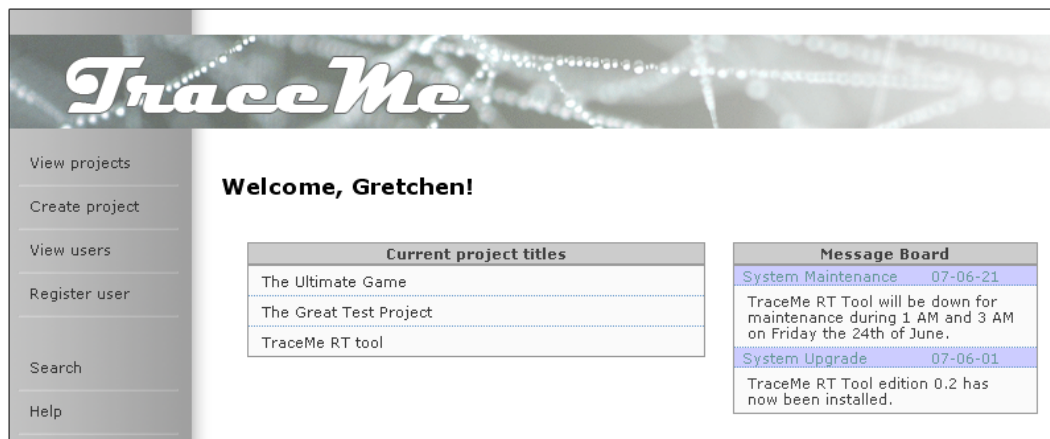


Figure 23.2: TraceMe - Welcome screen

By employing Cascading Style Sheets, the prototype's graphical design is rendered consistently throughout the system. The look of the banners, menus, tables, and text presented in the welcome screen remains the same throughout the system, thus applying the first rule of the interface design rules.

23.2 Managing projects and other administrative work

As discussed in Appendix H.1, the prototype consists of two abstract components, where one (the administrative component) encapsulates the other (the project component). The current section describes the functionality of the administrative component, which includes the regis-

tration of users, creation of projects, and attaching users to projects, granting them access to the contents of each project. In addition, the administrative component provide functionality for manipulating the attributes of users and projects, such as user access rights and project descriptions.

The vertical left-hand menu presented in the screens of the project component allows the user to choose between the most important functionality of the component. As shown in Figure 23.2, the administrative component allows the user to view existing projects and users and their attributes (Figures 23.3 and 23.4), register new users (Figure 23.5), and create new projects (Figure 23.6). In addition, menu items providing search and help functionality is included at the bottom, allowing the user to search for the desired project or user, or seek assistance in the help menu. Section 23.6 describes the search functionality further, whilst Section 23.7 discusses the help menu. The menu items in the left-hand menu have been given a hover effect, in compliance with the third rule of the interface design rules. When the user hovers the mouse pointer above the menu item, the colour of the text will change, signalling to the user that an action can be performed.

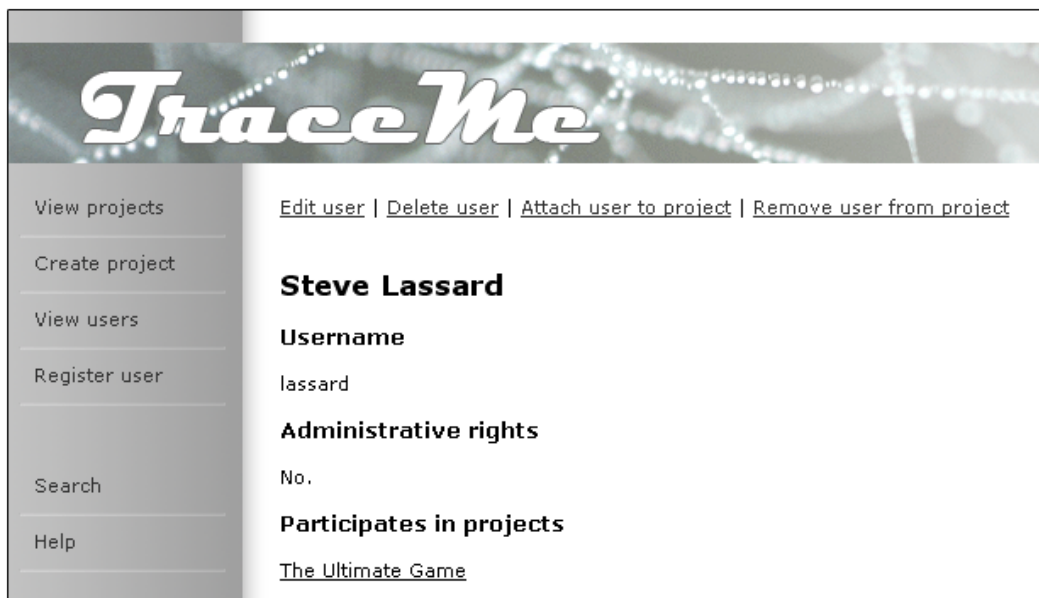


Figure 23.3: TraceMe - User details

In addition to the left-hand menu choices, some screens provide a horizontal detailed menu, as can be seen in Figure 23.4, which shows the screen presenting project details, amongst others references to users attached to the project. All these references are linked to the screen presenting user information (Figure 23.3), allowing the user quick access to the details on the attached users.

The detailed menu allows the user to edit (Figure 23.7) or delete the project, and maintain the list of users added to the project (Figure 23.8). When editing or deleting projects or users, the system offers feedback informing the user of the result of the operation, as can be seen in Figure 23.7b. This complies with the third and fourth rule of the interface design rules, signalling closure of the action, and offering informative feedback of the result of the action.

Most importantly, the detailed menu allows the user to open the project and view its contents. This causes the user to enter the project component, thus losing access to the functionality of the administrative component until the project component is exited.



Figure 23.4: TraceMe - Project details



Figure 23.5: TraceMe - Register new user

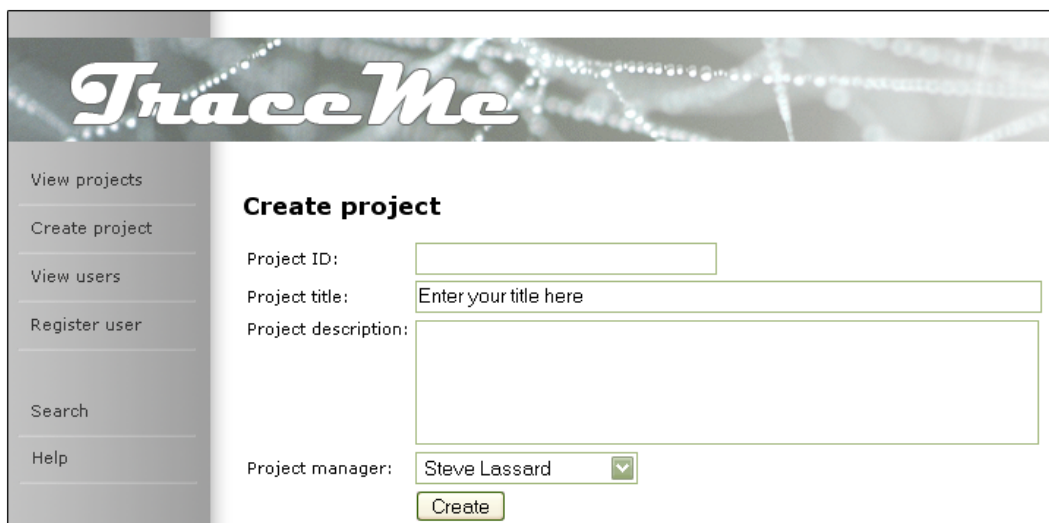


Figure 23.6: TraceMe - Create new project



Figure 23.7: TraceMe - Edit project

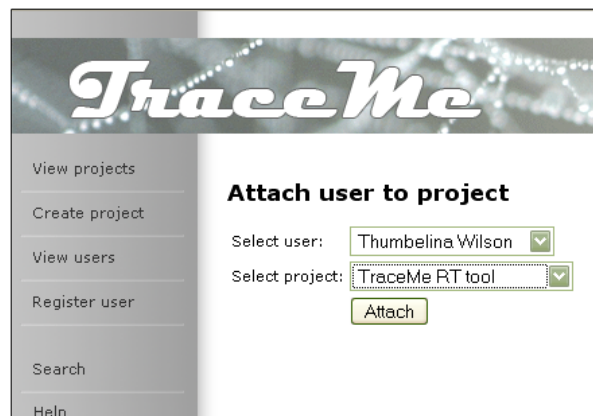


Figure 23.8: TraceMe - Attach user to project

23.3 Project contents

Each project has its own instantiation of the domain model, and maintains a set of artifacts instantiated as domain objects and sorted according to the traceability stage they belong to. When opening the contents of a specific project¹, the user is presented with the screen displayed in Figure 23.9.

This is a summary screen, allowing the user to quickly get an impression of the current situation within the projects. Note that the graphical design have changed slightly from the administrative module, but that major features still resembles each other. This is done in order to ensure consistency, whilst still providing the user with recognisable features, enabling him or her to distinguish the administrative component from the project component. In addition, the left-hand menu has new options, reflecting the functionality of the project component. By choosing the “Exit project” option, the user will exit the project component and re-enter the administrative component, allowing him or her to e.g. open another project.

The title of the chosen project is displayed near the top of the screen, informing the user of which project he or she is currently working on. Throughout the screens of the project component, the project title is always displayed in the same area of the screen, ensuring conformance to the first rule of the interface design rules. In addition, it reduces the short-term memory load, as recommended by the last rule of the interface design rules.

The summary screen presents three tables, providing the user with quick access to important functionality. However, each user is able to individually decide which summary tables should be displayed by clicking the “Add/remove summaries” link at the top right of the screen. All data presented in tables throughout the project component is, if possible, hyperlinked to the source

¹As an example, a project representing the development of the RT tool itself has been created, named *TraceMe RT tool*

The screenshot shows the TraceMe RT tool interface. On the left is a sidebar menu with options: Project home, Pre-RS traceability, Pre-FRS traceability, Post-FRS traceability, Visualisations, Exit project, Search, and Help. The main content area is titled 'Summary' and 'TraceMe RT tool'. It features three tables:

Recent artifacts			
Artifact tag	Artifact type	Traceability stage	Creation date
Formal.Performance.Throughput.ActiveUsers	PerformanceRequirement	Pre-FRS traceability	Fri May 25 10:19:37 CEST 2007
Formal.Attribute.Modifiability.Flexibility	SoftwareSystemAttribute	Pre-FRS traceability	Fri May 25 10:19:37 CEST 2007
SystemComponent.Package.Controller	Package	Post-FRS traceability	Fri May 25 10:19:37 CEST 2007
SystemComponent.Class.MainController	Class	Post-FRS traceability	Fri May 25 10:19:37 CEST 2007
SystemComponent.Class.Dispatcher	Class	Post-FRS traceability	Fri May 25 10:19:37 CEST 2007

Existing artifact categories	
Pre-RS traceability	
Requirement	
Stakeholder	
ChangeProposal	
Rationale	
RequirementDescriptor	
Issue	
Alternative	
Debate	
Decision	
Pre-FRS traceability	
ExternalInterfaceRequirement	
FunctionalRequirement	
PerformanceRequirement	
SoftwareSystemAttribute	
Post-FRS traceability	
Package	
Class	
Design	

Stakeholders	
Name	Position
Thomas Crown	Senior Architect
Clark Kent	Junior Consultant
Peter Parker	Senior Consultant
Aeon Flux	Test manager

Figure 23.9: TraceMe - Project contents

of the information. This allows the user quick access to additional data, and when combined with the adaptable summaries, allows frequent users to create useful shortcuts, which complies with the second rule of the interface design rules.

23.4 Creating and maintaining trace information

Artifacts are sorted according to the traceability stage they belong to. Thus, in order to create an artifact instance, the user must select a traceability stage from the left-hand menu. Doing so triggers the project module to enter the chosen stage, and present a list of the artifact categories of the stage, shown in Figure 23.10. Note that this list can also be reached by clicking the traceability stage headers of the “Existing artifacts categories” summary table on the summary screen, in addition to from many other locations within the system, as will be shown later. This enables users to quickly gain access to the required functionality, as recommended by the second rule of the interface design rules.

By entering a traceability stage, the menus of the project component changes, reflecting the additional functionality available. The user is no able to view artifacts and traces of the chosen stage (although the system automatically displays the artifacts when entering a traceability stage), as well as creating new instances and edit existing instances. In addition, a line of tabs have appeared near the top of the screen, allowing the user to quickly establish the current traceability stage, as well as jumping between stages. The title of the chosen project is displayed next to this line of tabs.

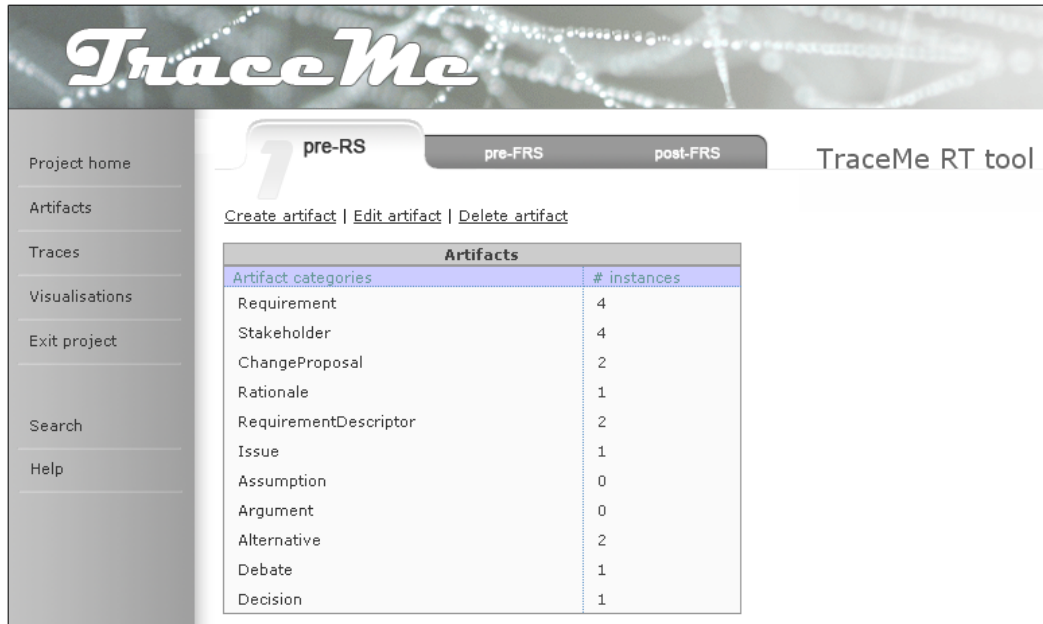


Figure 23.10: TraceMe - Artifact categories

23.4.1 Artifacts

Choosing the “Artifacts” option in the left-hand menu allows the user to work with artifacts. A detailed horizontal menu is displayed, providing the user access to functionality for creating a new artifact, as well as editing and deleting existing artifacts. In addition, the project component presents the list of artifact categories belonging to the chosen stage, along with the number of existing artifact instances within each category. The user is able to click the artifact category, causing the project component to present a list of the artifacts instances, as shown in Figure 23.11.

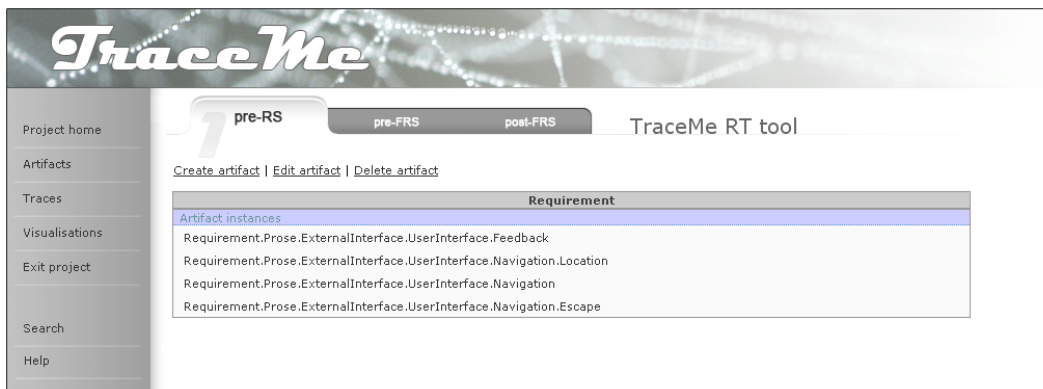


Figure 23.11: TraceMe - Artifact instances

In turn, each of these instances is clickable, linked to a screen presenting the attributes of the specific artifact instance. An example of such an artifact instance presentation is shown in Figure 23.12. The presentation of artifact instances vary, depending on the chosen artifact, and additional examples can be viewed in Figures 23.13 through 23.15.

In general, the artifact instance presentations contains a presentation of the artifact’s attributes (shown to the top left in Figure 23.12), a summary of the version history of the artifact (shown in the bottom left in Figure 23.12), and a summary of artifact instances connected to the chosen artifact as defined in the underlying domain object model (shown to the top right in Figure 23.12).

The screenshot shows the TraceMe web application interface. At the top, there's a navigation bar with 'pre-RS', 'pre-FRS', and 'post-FRS' tabs, and the text 'TraceMe RT tool'. A sidebar on the left contains navigation links: Project home, Artifacts, Traces, Visualisations, Exit project, Search, and Help. The main content area displays details for a requirement artifact: 'Requirement.Prose.ExternalInterface.UserInterface.Navigation.Location'. It includes a 'Requirement specification' table, a table of 'Artifacts connected to this Requirement', a 'Previous versions' table, and a table of 'Accepted change proposals'.

Requirement specification	
Version number	1
Version number	1
Creation date	Wed May 23 19:54:01 CEST 2007
Last change	Never been changed
Title	Location within system
Description	The user must at all times be notified of the current location within the system, also know as the current path.
Underlying rationale	N/A
Owning stakeholder	Clark Kent

Artifacts connected to this Requirement		
Artifact type	# instances	Trace
Stakeholder	2	is affected by
RequirementDescriptor	0	describes
ChangeProposal	1	is accepted by
Decision	N/A	affects
Rationale	N/A	defines
Issue	0	are generated by
FormalRequirement	1	are defined by

Previous versions		
Version number	Valid from	Valid to
N/A		

Accepted change proposals	
Artifact tag	Approval timestamp
ChangeProposal.ExternalInterface.UserInterface.Location	Wed May 23 19:54:01 CEST 2007

Figure 23.12: TraceMe - Artifact instance details

The screenshot shows the TraceMe web application interface for a change proposal artifact: 'ChangeProposal.ExternalInterface.UserInterface.Feedback'. It includes a 'Change Proposal specification' table, a table of 'Artifacts connected to this ChangeProposal', and a 'Previous versions' table. The interface also shows a 'pre-RS' tab and a 'TraceMe RT tool' header.

Change Proposal specification	
Version number	1
Creation date	Thu May 24 10:57:56 CEST 2007
Last change	Never been changed
Affected requirement	Requirement.Prose.ExternalInterface.UserInterface.Feedback
Description	Feedback should be presented to the user through informative messages, both relating what went wrong, and how this could be fixed.
Proposing stakeholder	Aeon Flux
Underlying rationale	Rationale.ChangeProposal.ExternalInterface.UserInterface.Feedback
Approved	No

Artifacts connected to this ChangeProposal		
Artifact type	# instances	Trace
Requirement	Singular	affects
Rationale	Singular	supports
Stakeholder	Singular	proposes

Previous versions		
Version number	Valid from	Valid to
N/A		

Figure 23.13: TraceMe - Artifact instance details - Example 2

The screenshot shows the TraceMe RT tool interface. The top navigation bar includes 'pre-RS', '2 pre-FRS', and 'post-FRS'. The main content area displays details for a Functional Requirement artifact.

Functional Requirement specification

Version number	1
Creation date	Thu May 24 12:09:47 CEST 2007
Last change	Never been changed
Creation date	Thu May 24 12:09:47 CEST 2007
Title	Navigation
Gist	Navigating within the system
Description	The user must be able to switch between the traceability stages of the traceability model, i.e. between the pre-RS traceability, pre-FRS traceability, and post-FRS traceability stage.
Priority	High
Risk	High
Status	Approved

Artifacts connected to this FunctionalRequirement

Artifact type	# instances	Trace
Requirement	Singular	defines
Mandate	0	is supported by
FormalRequirement	0	sub-functions
SystemComponent	0	fulfills
Design	1	is driven by
VerificationProcedure	0	is developed for

Previous versions

Version number	Valid from	Valid to
N/A		

Figure 23.14: TraceMe - Artifact instance details - Example 3

The screenshot shows the TraceMe RT tool interface. The top navigation bar includes 'pre-RS', 'pre-FRS', and '3 post-FRS'. The main content area displays details for a Design artifact.

Design specification

Version number	1
Creation date	Thu May 24 12:21:57 CEST 2007
Last change	Never been changed
Creation date	Thu May 24 12:21:57 CEST 2007
Title	Logical design
Implementation language	UML class diagram
Description	Class diagram depicting the logical design of the application, for use in the logical view of the architectural description.
Location	/root/designs/logical/classdiagram.png

Artifacts connected to this Design

Artifact type	# instances	Trace
FormalRequirement	2	drives
SystemComponent	1	is defined by

Previous versions

Version number	Valid from	Valid to
N/A		

Figure 23.15: TraceMe - Artifact instance details - Example 4

The summary of connected artifact instances is particularly interesting, as it provides easy access to important trace information by creating *predetermined traceability visualisations*. The summary lists all artifact categories directly connected to the chosen artifact, and the number of connected artifact instances (one or more). The summary also describes the relation (referred to as *trace*) between the chosen artifact and the connected artifact, and the user is allowed to click each listing for additional information. This causes the system to present the details of the connected artifact instance (if only one connected instance), or a list of the connected artifact instances (if more than one connected instance). In addition, the summary provides quick access to functionality for creating new artifact instances or traces by listing “create” and “attach” buttons next to each artifact. E.g., by clicking the “create” button next to the Stakeholder listing in Figure 23.12, the system initiates the process of creating a new instance of the Stakeholder artifact.

In addition to the general characteristics of the artifact instance presentation, the individual artifacts occasionally require the presentation of special trace information. The bottom right position in the screen is reserved for this. Often, this position is used for presenting details of a particularly interesting type of connected artifact, thus creating an alternative visualisation of the trace information, as a connected artifact will always be accessible via the summary of all connected artifact in the top right position. Figure 23.12 exemplifies the use of alternative visualisations by providing a summary table accepted change proposals, whilst Figures 23.13 through 23.15 does not make use of this extra space, as it is not required by the presented artifacts.

Create artifacts

When creating a new artifact, the user chooses an artifact category, and fills in a form requesting the necessary trace information (an example of which is displayed in Figure 23.16). The system then instantiates a new domain object corresponding to the chosen artifact category, and saves it in the set of artifact instances held by the project. The forms requesting trace information are built automatically based on the underlying model, which ensures cost-effective addition of new artifacts. Some simple traces are created when creating a new artifact instance, such as “proposing stakeholder” (the user chooses from a list of existing stakeholders). This is often the case when the trace allows only a single artifact instance to be connected to the created artifact. However, the majority of traces are created after the creation of both artifacts included in the trace.

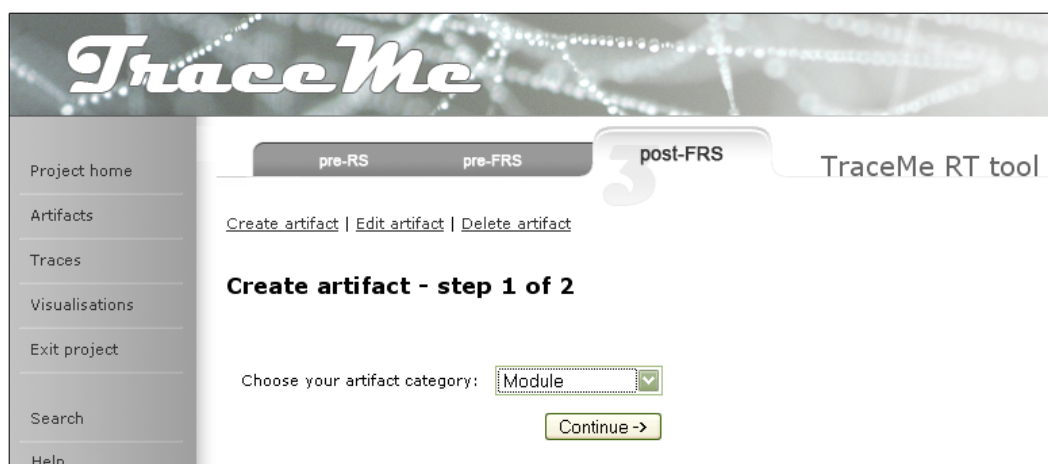
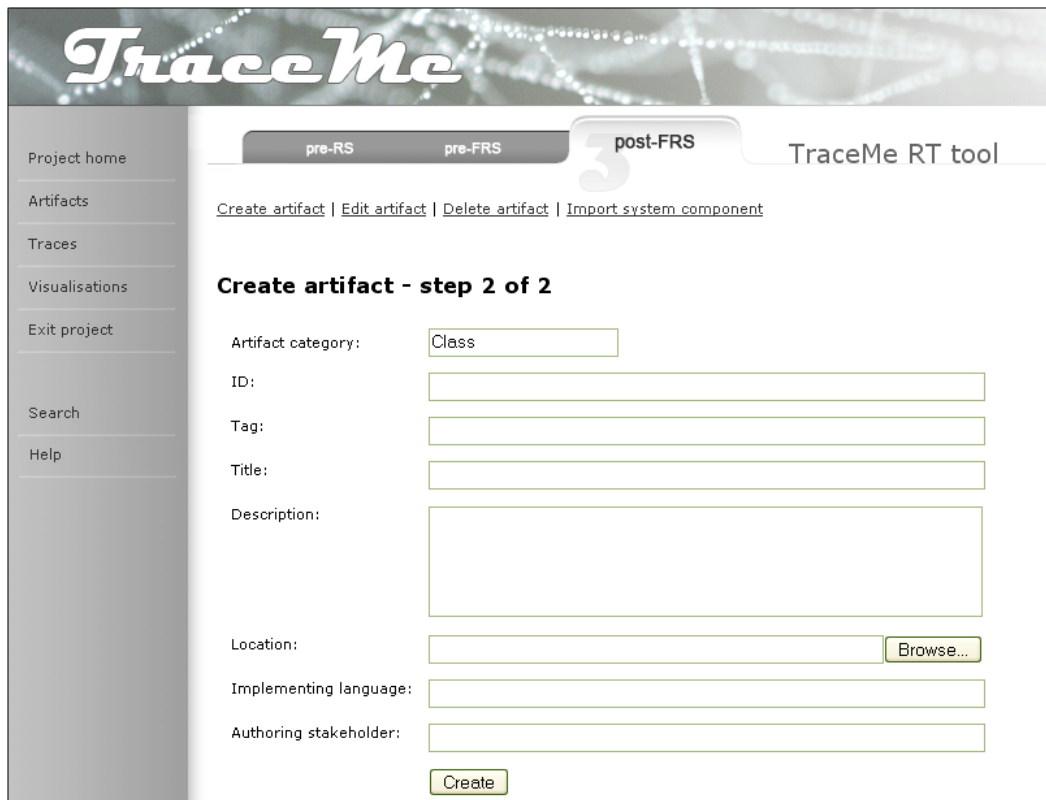


Figure 23.16: TraceMe - Create artifact

In compliance with the fifth rule of interface design, the forms for creating artifacts are designed to prevent errors caused by the user entering incorrect data, e.g. by opting for menu selections rather than fill-in fields whenever possible. Long and complicated forms are divided into steps, not overwhelming the user with information.

Some artifacts are linked to external resources (e.g., RequirementDescriptors and Designs), which requires the user to enter the location of resource. The location can be specified either by browsing the file system, or by providing a URL, allowing different sorts of resources to be added.

When creating SystemComponent artifacts (belonging to the post-FRS traceability stage), the user is allowed to import artifacts from external development environments. For instance, rather than manually creating Class artifacts for an entire development project, the user can import archives of compiled classes, and the system will automatically create Class and Package artifacts based on the contents of the classes, such as its JavaDoc comments [Jav]. The user imports system components by requesting the system to create a SystemComponent artifact, and then choosing “Import system component” from the horizontal detailed menu, as shown in Figure 23.17.



The screenshot shows the TraceMe RT tool interface. The top navigation bar includes 'pre-RS', 'pre-FRS', and 'post-FRS' tabs, with 'post-FRS' selected. The main content area is titled 'Create artifact - step 2 of 2' and contains the following form fields:

- Artifact category:
- ID:
- Tag:
- Title:
- Description:
- Location:
- Implementing language:
- Authoring stakeholder:

A 'Create' button is located at the bottom of the form.

Figure 23.17: TraceMe - Import system components

23.4.2 Traces

By choosing the “Traces” option from the left-hand menu, the system presents a list of the traces within the chosen traceability stage, as defined by the underlying domain model. This screen is shown in Figure 23.18. In addition, the detailed horizontal menu changes, allowing the user to create and maintain traces rather than artifacts.

When clicking the “View” button next to a trace listing, the user is presented with a list of the existing traces, specifying starting and ending artifact instance. Again, this provides an alternative representation of trace information, as the individual artifact instance presentations list connecting artifact instances. However, the list of existing traces summarises all existing traces between two artifact categories, which could spare the user several keystrokes, e.g. when trying to establish whether a certain relationship exists.

Pre-RS traceability traces			
Starting artifact	Trace type	Ending artifact	
Requirement	affects	Stakeholder	View
Requirement	accepts	ChangeProposal	View
Stakeholder	proposes	ChangeProposal	View
Stakeholder	authors	RequirementDescriptor	View
ChangeProposal	is based on	Rationale	View
Requirement	is based on	Rationale	View
Issue	is generated by	Requirement	View
Issue	starts	Debate	View
Decision	is made by	Stakeholder	View
Decision	resolves	Issue	View
Debate	discusses	Alternative	View
Argument	validates	Alternative	View

Figure 23.18: TraceMe - Defined traces

Create traces

When creating new traces, the user is first requested to choose among the possible traces of the chosen traceability stage (shown in Figure 23.19). Traces crossing the boundaries of traceability stages are only included in the starting artifact's traceability stage. After choosing the trace type, the user can choose among artifact instances, choosing those instances he or she wishes to be starting and ending artifact instances (shown in Figure 23.20), and establish a trace between the instances.

23.5 Visualising trace information

The visualisations of trace information comes in two flavours. The *predetermined visualisations* are not configurable by the user, and are amongst others used in the individual presentations of artifact instances. In addition, the project component offers three complex predetermined visualisations (an example of which is shown in Figure 23.21), presenting the three traceability stages as modelled by the underlying domain object model. This enables the user to see how all the artifacts are connected, i.e. providing the big picture, and still provides quick access to details by allowing the user to click each individual domain object.

In addition to these predetermined visualisations, the system allows the user to create custom-made visualisations, an example of which is shown in Figure 23.22.

A custom-made visualisation is created by allowing the user to choose a starting and ending artifact category, and representing existing instances within these categories, connected with an arrow. If the artifact categories are not directly linked, intermediate artifacts are displayed as grey boxes, depicting the route of the traces. These intermediate artifacts are clickable, generating a new custom-made visualisation consisting of the original starting artifact, and the intermediate artifact as the new ending artifact. This is illustrated in Figure 23.23. The displayed artifact instances are also clickable, forwarding the user to the individual artifact instance presentation.

Both the complex predetermined visualisation and the custom-made visualisation can be viewed by choosing "Visualisations" in the left-hand menu, which presents the screen shown in Figure

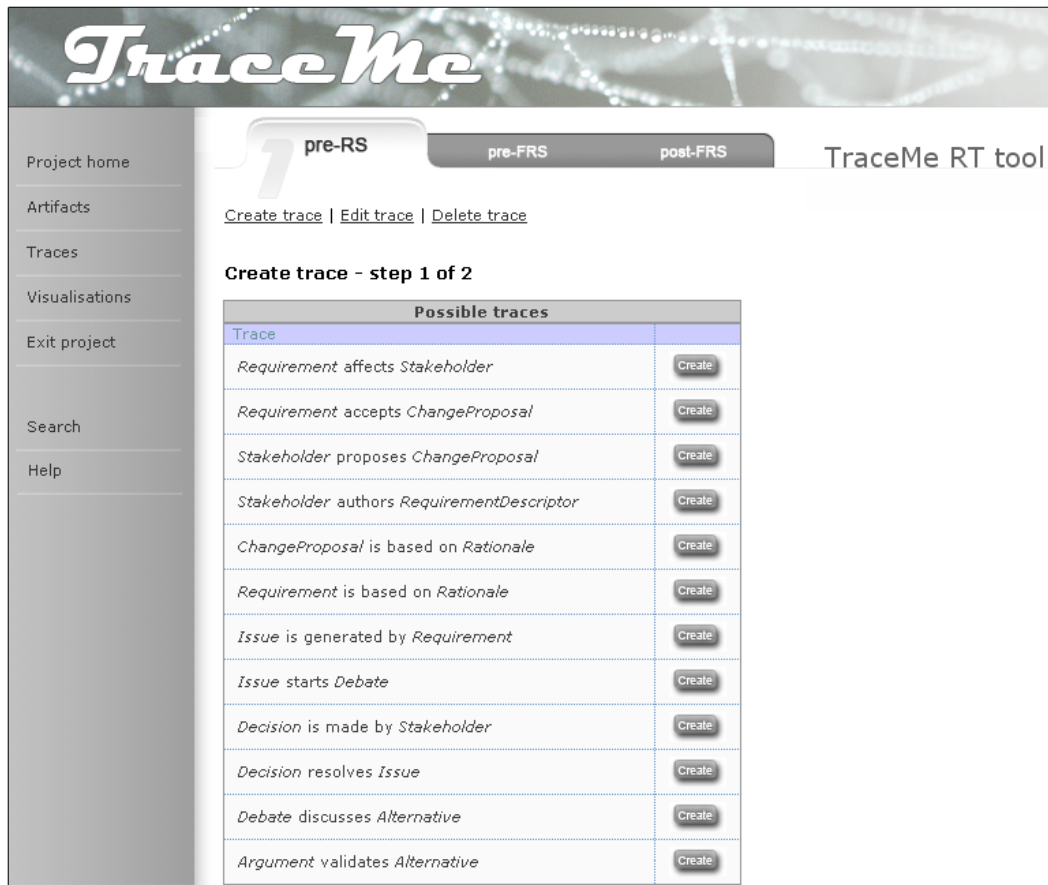


Figure 23.19: TraceMe - Create trace - step 1



Figure 23.20: TraceMe - Create trace - step 2

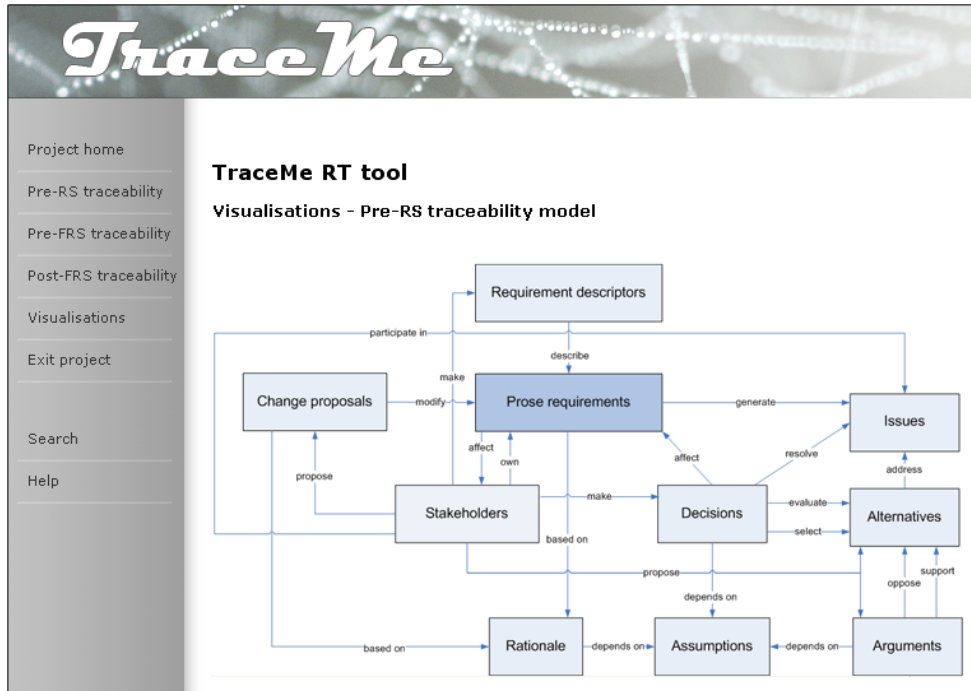


Figure 23.21: TraceMe - Complex predetermined visualisation

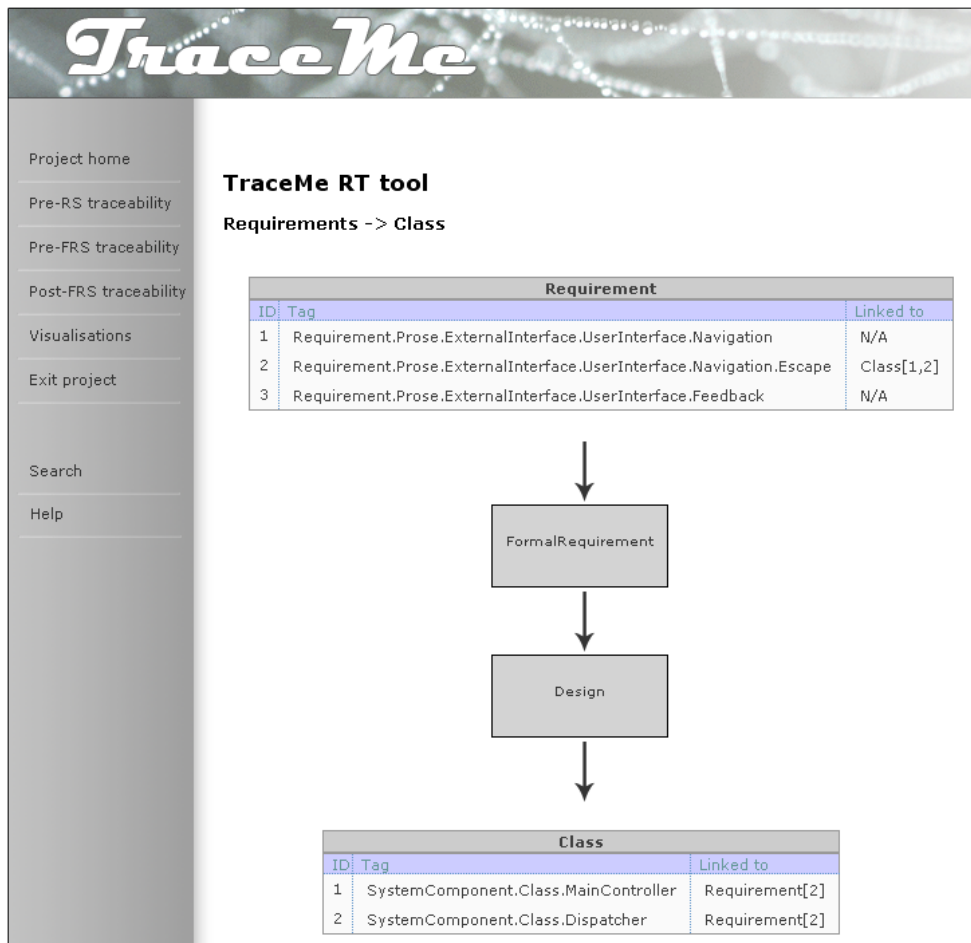


Figure 23.22: TraceMe - Custom-made visualisation

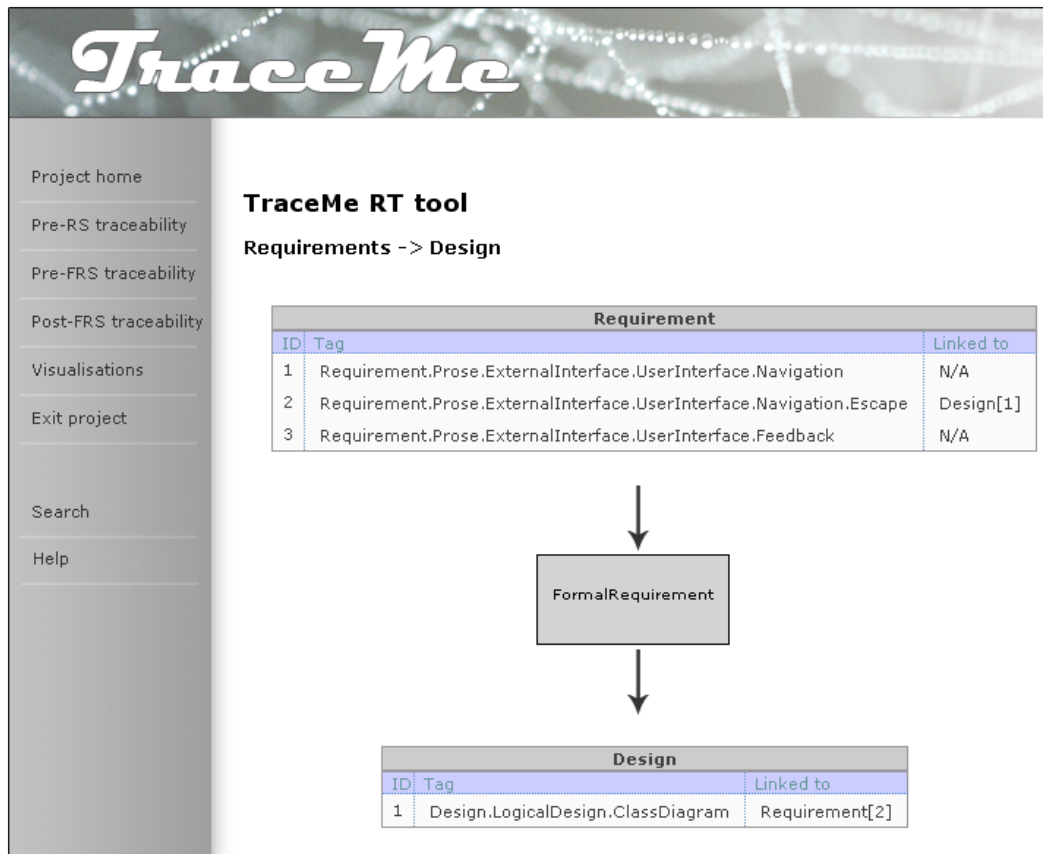


Figure 23.23: TraceMe - Custom-made visualisation - Example 2

23.24. This screen lists both the three complex predetermined visualisations and any existing custom-made visualisations, as well as allowing the user to create new custom-made visualisations.

23.6 Search

In order to enable the user to locate trace information fast and simple, the prototype offers a search functionality (albeit as alpha functionality), located near the bottom of the left-hand menu in all screens. Figure 23.25 shows how the user is able to enter search terms and execute a search (Figure 23.25a), whereupon the system presents the search result (Figure 23.25b).

In addition to the regular search functionality, the prototype offers an advanced search option, which allows the user to specify constraints on the search, for instance specifying which terms are not to be included in the search result.

23.7 Help and assistance

To assist the user if problems arise, the prototype provides a set of help mechanisms. If tasks are performed incorrectly, the user will be informed of what went wrong. I.e., if the user attempts to delete a registered user that also acts as a project manager in one of the existing projects, the system will refuse to delete the registered user until another project manager has been appointed, informing the user of the problem.

In addition, the prototype contains a help menu that lists all tasks of the system, describing them and how they are performed. This menu is shown in Figure 23.26. The help menu allows

TraceMe

Project home
Pre-RS traceability
Pre-FRS traceability
Post-FRS traceability
Visualisations
Exit project

Search
Help

TraceMe RT tool

Visualisations

Predetermined visualisations	
Pre-RS traceability model	View
Pre-FRS traceability model	View
Post-FRS traceability model	View

Custom-made visualisations	
Previous visualisations	
Requirements - System Components - Tests	View
Stakeholders - Requirements - Change Proposals	View
Create new visualisation	Create

Figure 23.24: TraceMe - Visualisations

TraceMe

Project home
Pre-RS traceability
Pre-FRS traceability
Post-FRS traceability
Visualisations
Exit project

Search
Help

New search | Advanced search

Search

Enter your keywords, and the system will search through all artifacts, presenting those that matches with your keywords.

[Search](#)

(a)

TraceMe

Project home
Pre-RS traceability
Pre-FRS traceability
Post-FRS traceability
Visualisations
Exit project

Search
Help

New search | Advanced search

Search results

Presenting search results for term(s) "Clark" + "Requirement"

Search Results	
Term	Top
Stakeholder	Stakeholder.Developer.xent.Clark
Stakeholder	Stakeholder.TestManager.Bogart.Clark
Requirement	Requirement.Prose.ExternalInterface.UserInterface.Navigation.Location
FunctionalRequirement	Functional.WorkingWithProjects.DetermineTraceabilityStage

(b)

Figure 23.25: TraceMe - Search

to user to browse through it contents, but also provide an index of the help contents, and a search function, allowing the user to locate help instructions without having to look through all of them.

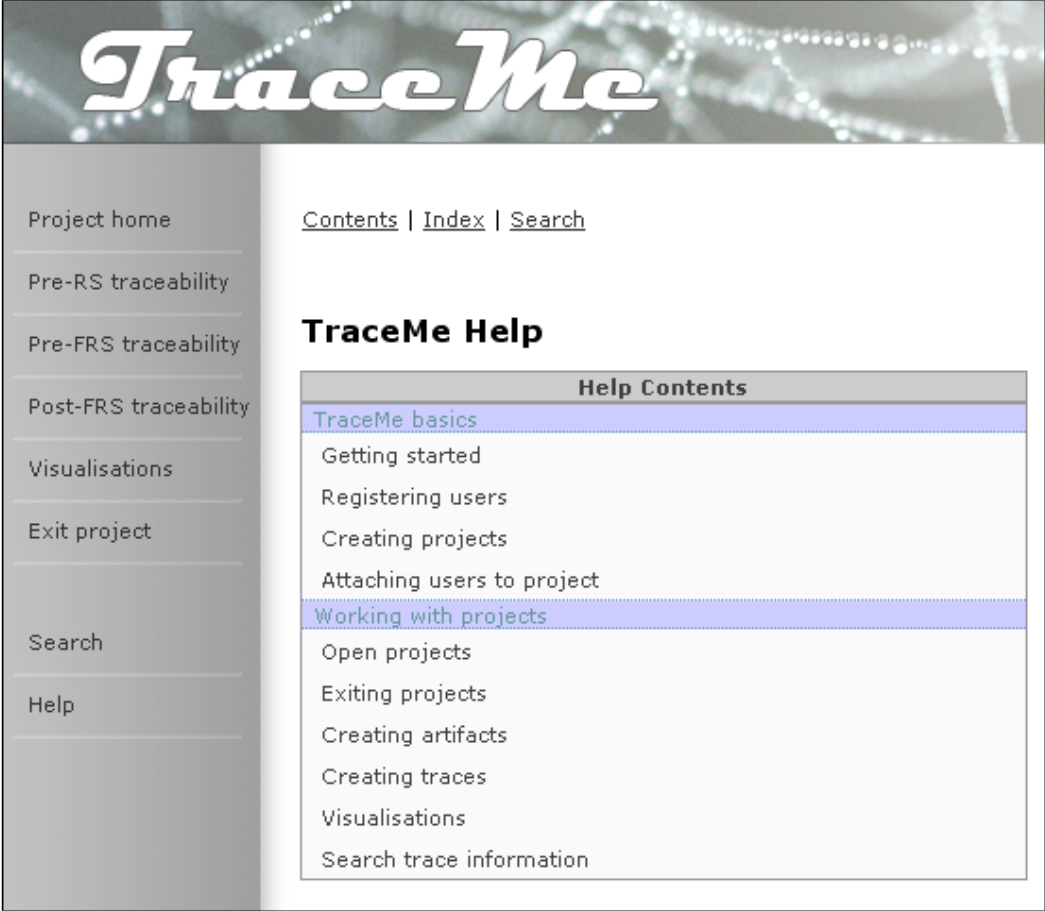


Figure 23.26: TraceMe - Help menu

Chapter 24

Prototype evaluation

This chapter evaluates two different aspects of the prototype. First, the prototype's adaptability and changeability, as discussed in Section 22.2 is evaluated, determining the ease with which the prototype can be evolved into a complete RT tool (i.e., its evolvability). This evaluation focuses on the prototype itself and its implementation. Second, the functionality provided by the prototype is evaluated. This evaluation focuses not on the prototype itself nor its implementation, but its fulfilment of the functional requirements, and its contribution as a prototype of a complete RT tool.

24.1 Prototype evolvability

The evolvability of the prototype is highly dependent on the prototype's ability to conform to the architectural styles presented in the architectural description (see Chapter 16). As mentioned in Section 22.2, the prototype is designed according to the component-based architectural style. This is achieved by compartmentalising the functionality of the prototype with the means of a simplified Model-View-Controller design pattern.

Another important aspect of the prototype is its ability to add new artifacts to the underlying domain model. This enables the prototype to be adapted to the different needs of users, and effort has been invested to ensure that the addition of new artifacts adds little to the work overhead. This is accomplished by describing artifacts with Property objects. The addition of new artifacts is then primarily done by configuring parameters. For instance, the properties of an artifact is used to automatically generate new forms for creating instances of the artifact.

Overall, the prototype's evolvability is satisfactory, both concerning the prototype's encapsulation of the underlying domain model, and its ability to alter its functionality. This allows the prototype to be evolved into a complete RT tool with iterations of user input and continued development. However, evolving the prototype into a complete RT tool is only an option if its contributions satisfies both the requirements specification and the research agenda of this thesis, evaluated in the next section.

24.2 Prototype contributions

In order to evaluate the contributions of the prototype, it's fulfilment of the functional requirements of the requirements specification (see Part III) must first be ascertained. As the prototype does not provide a complete implementation of the functionality, the evaluation of the fulfilment of the functional requirements can only consider whether a requirement has been addressed by the prototype. Testing can be applied to investigate whether a requirement has been fulfilled by the proposed solution. Due to the agile development process, testing have been performed throughout the development of the prototype. Consequently, we have no need for a formal

test plan. Consequently, the tables presenting the fulfilment of the functional requirements are based on informal testing performed in parallel with development.

24.2.1 Fulfilment of functional requirements

The functionality demanded by an addressed requirement can either be implemented by the prototype as a *beta* functionality, i.e., a crude implementation of the functionality, or as a *alpha* functionality, only mimicking the functionality without actually implementing it. Table 24.1 summarises the requirements and states how the prototype addresses them. The column marked α represents alpha functionality, β represents beta functionality, and N/A marks the requirements that are not addressed.

Table 24.1: Fulfilment of functional requirements

Requirement	α	β	N/A
Functional.ManagingProjects.CreateNewProject		X	
Functional.ManagingProjects.RemoveProject		X	
Functional.ManagingProjects.AttachUsers		X	
Functional.WorkingWithProjects.ChooseProject		X	
Functional.WorkingWithProjects.EscapeProject		X	
Functional.WorkingWithProjects.DetermineTraceabilityStage		X	
Functional.WorkingWithProjects.NavigateTraceInformation		X	
Functional.CreatingArtifacts.CreatingAnArtifactInstance		X	
Functional.CreatingTraces.ChoosingStartingArtifactInstance		X	
Functional.CreatingTraces.ChoosingEndingArtifactInstance		X	
Functional.CreatingTraces.RegisteringAuxiliaryTraceInformation			X
Functional.EditingArtifacts.ChooseArtifactToEdit	X		
Functional.EditingArtifacts.SaveChangesToArtifact	X		
Functional.DeletingArtifacts.ChooseArtifactToDelete	X		
Functional.EditingTraces.ChooseTraceToEdit	X		
Functional.EditingTraces.SaveChangesToTrace	X		
Functional.DeletingTraces.ChooseTraceToDelete	X		
Functional.DeletingTraces.SystemInitiatedTraceDeletion			X
Functional.SearchingTraceInformation	X		
Functional.PredeterminedVisualisations.RequestingSimpleVisualisation		X	
Functional.PredeterminedVisualisations.RequestingComplexVisualisation		X	
Functional.PredeterminedVisualisations.RequestingComplexVisualisation		X	
Functional.PredeterminedVisualisations.SimpleVisualisations		X	
Functional.PredeterminedVisualisations.ComplexVisualisations		X	
Functional.CustommadeVisualisations.AdaptingSimpleVisualisations	X		
Functional.CustommadeVisualisations.CreatingComplexVisualisations	X		
Functional.CustommadeVisualisations.CreatingComplexVisualisations. ChooseStartingAndEndingArtifact	X		
Functional.Help.Messages		X	
Functional.Help.Menu	X		
Functional.Help.Menu.Search	X		
Functional.Administrative.MaintainingUserList.AddingUser		X	
Functional.Administrative.MaintainingUserList.RemovingUser		X	
Functional.Administrative.MaintainingUserList.ChangingPassword		X	
Functional.Administrative.MaintainingUserList.ChangingAccessLevel		X	
Functional.Administrative.MaintainingUserList.UserAuthentication		X	

As can be seen from Table 24.1, only two functional requirements have not been addressed by the prototype. *Functional.CreatingTraces.RegisteringAuxiliaryTraceInformation* should, if

implemented, allow the user to add auxiliary trace information to certain traces, as discussed in Section 8.4. However, this functionality is not required to create traces, it only adds value to the trace information. As this is a first version of the prototype, this functionality has been referred to future versions.

System-initiated trace deletion has also been ignored by this version of the prototype. System-initiated trace deletion is used for deleting an artifact instance's traces when the artifact itself is deleted. As the functionality for deleting artifacts is only represented in the prototype as alpha functionality, the system-initiated trace deletion is not required by the system. Consequently, when artifact deletion is implemented as beta functionality, the system-initiated trace deletion should also be implemented.

The use cases of Section 12.3.1 elaborates on the functional requirements of the requirements specification by describing how the functionality is intended to be used. Table 24.2 summarises which use cases are supported by the prototype. As previously, the column marked α denotes alpha functionality, whilst the column marked β denotes beta functionality. The column marked N/A indicates that the prototype does not support the use case. As can be seen in Table 24.2, all use cases are addressed by the prototype, either as alpha or beta functionality.

Table 24.2: Supported use cases

Use case	α	β	N/A
UseCase.UserLogin	X		
UseCase.ChooseProject		X	
UseCase.ChooseTraceabilityStage		X	
UseCase.WorkingWithArtifacts		X	
UseCase.CreateAnArtifact		X	
UseCase.EditAnArtifact	X		
UseCase.DeleteExistingArtifact	X		
UseCase.LocateExistingArtifact		X	
UseCase.AddTrace		X	
UseCase.DeleteTrace	X		
UseCase.VisualiseTraceInformation	X		
UseCase.SearchTraceInformation	X		
UseCase.CreateProject		X	
UseCase.RegisterUser		X	
UseCase.SearchHelpDirectory	X		

Part II concluded that the RT tool must take care to seamlessly integrate Planguage and its attributes. By explicitly specifying the attributes of the artifacts with the use of tables and forms, the prototype does not require the user to have detailed knowledge of Planguage, nor is the user required to remember which attributes an artifact employs in its specification, in compliance with the last rule of the human-computer interface design guidelines of Appendix D.

Otherwise, two-thirds of the functional requirements are implemented as beta functionality. This functionality increases the value of the prototype, bringing it closer to the final version. The remaining one-third of alpha functionality is justified by the system's status as a prototype. We will discuss briefly why selected functional requirements were implemented as alpha functionality rather than beta functionality. Chapter 25 discusses how the alpha functionality can evolve to beta functionality, and eventually, complete functionality.

Functionality for editing and deleting artifacts and traces were implemented as alpha functionality because it is easy for a user to grasp the underlying actions. In addition, functionality for editing and deleting projects and users have been implemented as beta functionality, giving the user an impression of how functionality for editing and deleting is intended to work.

Search functionality has been added to the prototype as alpha functionality due to the sheer complexity of such functionality. The same applies to the creation of custom-made visualisations. Both these functionalities incur a significant development effort, and are consequently referred to future versions of the prototype.

The prototype will primarily be used for testing purposes, for instance in a scenario where a user and a member of the development staff sits down and tests the functionality of the prototype, attemptively discovering new requirements or altering existing requirements. Consequently, the help functionality is implemented as alpha functionality, as the user can turn to the development staff member for help. It is, however, important to signal to the user that help functionality will exist in the final version, consequently it is implemented as alpha functionality rather than ignored by the prototype altogether.

Overall, the prototype addresses the majority of functional requirements, and all use cases, thus giving the user a good impression of how the RT tool is intended to function. This paves the way for further development of the prototype. However, in order to be able to determine whether further development of the prototype is viable, the prototype's current or future ability to fulfil the non-functional requirements must be evaluated.

24.2.2 Fulfilment of non-functional requirements

The in-depth study [Nor06] presented a high-level requirements specification, based on important findings in the study. This high-level requirements specification consisted of both functional and non-functional requirements, and has served as the foundation of the full requirements specification of Part III. The prototype's fulfilment of the functional requirements have been evaluated above, and attention is now shifted to the non-functional requirements. As previously argued (see Section 22.2), the prototype is to a certain degree allowed to ignore the non-functional requirements. However, the non-functional requirements of the high-level requirements specification address fundamental issues for an RT tool that it is important to consider if an RT tool is to be capable of decreasing the work overhead associated with implementing RT in software development projects. As an evolutionary prototype, the system is intended to evolve into a complete RT tool. Consequently, evaluating how the prototype addresses these fundamental issues, and more importantly, how the prototype is able to evolve into a tool that addresses the issues, enables us to determine whether it is worth the effort to perform the necessary iterations evolving the prototype into a complete RT tool.

The high-level requirements specification lists six non-functional requirements, that each addresses an issue an RT tool must address. We choose to evaluate the prototype with respect to the high-level requirements specification rather than the full requirements specification in order to reduce the complexity of the evaluation. The following sections discuss how the prototype, either in its current or future versions, fulfils the non-functional requirements. The rationale behind each functional and non-functional requirement is given in the in-depth study [Nor06]. Consequently, this evaluation does not consider why these requirements are stated, only that they must be fulfilled in order for the prototype to satisfy its goals.

The first non-functional requirement demands persistent storage of all trace information. This is, as discussed in Section 22.2, not implemented by the prototype. However, due to the component-based architecture, adding functionality for persistent storage is not a complicated task, albeit a large one, primarily due to the size of the underlying domain object model. Thus, future versions of the prototype will be able to offer persistent storage of trace information.

Next, the non-functional requirements demand that the tool should be easily adaptable to its environment. The prototype challenges this demand from several angles. First, adaptability is ensured by developing a prototype conforming to a component-based architecture, making it easier to change, remove, or add functionality. Second, the underlying model is designed to accommodate the introduction of changes, such as adding a new artifact as a domain object to the domain model. This ensures a domain model that can be adapted to new environments, i.e., reflect the nature of the software development project that is being modelled. Third and last, the prototype strive for adaptable human-computer interfaces, allowing the user to adapt

the interfaces to the current working situation. The summary screen of each project (shown in Figure 23.9) is an example of this, allowing the individual users to choose which summaries are to be displayed.

The non-functional requirements further states that the tool must be scalable, enabling it to handle small and large amounts of trace information equally well. The RT tool's scalability is determined by the tool's ability to handle many artifact instances without causing latency within the system, as well as the tool's ability to handle an increase in the number of users. These abilities are primarily determined by two factors: the performance of the architecture, and the performance of the hardware, the latter of which is not a concern of the prototype. The prototype conforms, to the degree determined in Section 22.2, to the architecture described in Part IV. This architecture is specifically designed to accommodate the demands of the non-functional requirements to the RT tool, and consequently, is designed to be scalable. However, as discussed in Section 28.4, scalability vs. performance represent a tradeoff point in the architecture, as improving one of these attributes deteriorates the other. Consequently this represents a risk in the future evolution of the prototype. Although the prototype handles an increase in the number of artifact instances (i.e., the size of the underlying model and its contents), the prototype does not consider the issue of concurrency. This causes it to not scale well with regard to an increase in the number of users. For simplicity, the prototype is designed to be used by a singular user, and future versions must address the issue of concurrency.

Further, the non-functional requirements demands the ability to gather the necessary trace information without imposing the user. This is a difficult requirement to fulfil, as the user is required to provide input to the system. However, automating tasks relieves the workload of the user, and renders the tool less intrusive. The prototype provides alpha functionality for importing system components from i.e. Java class files, creating artifact instances based on the contents of the files. This is a simple example of how tasks can be automated to create a less imposing tool.

Finally, the non-functional requirements demand that the user must be able to record and find desired trace information easily and quickly. The usability of the prototype is dependent on the quality of the human-computer interface design, and the eight golden rules of interface design (see Appendix D) were applied to ensure high-quality human-computer interfaces. Throughout Chapter 23, the use of the rules were pointed out. Overall, the majority of the rules were applied, with particular focus on consistency, use of shortcuts, informative feedback, dialogue closure, error-prevention, and reduction of short-term memory load. For example, the prototype provides shortcuts to trace information with the means of hyperlinks, linking references to trace information to the originating source, thus allowing the user quick access to supplementary information. In addition, the prototype provides alpha functionality for trace information searches, which simplifies the task of locating a specific piece of trace information, enabling quick access to the required trace information.

Overall, the prototype addresses all issues that must be amended by an RT tool, with the exception of persistent storage of trace information, concurrency, and scalability. In addition, the prototype addresses all functional requirements of the requirements specification, with few exceptions. Thus, the prototype provides a good foundation for an RT tool attempting to reduce the work overhead associated with implementing RT in software development projects. Consequently, we can conclude that the further development of the prototype can safely commence, giving particular attention to persistent storage of trace information, concurrency, and scalability. Chapter 25 describes the effort required to evolve the prototype into a complete RT tool.

Chapter 25

Prototype evolution

This chapter discusses the remaining work that is required to evolve the prototype into a complete RT tool. As pointed out in Section 22.2, and then again in Section 24.2.2, the prototype ignores issues concerning model persistence and concurrency. This chapter does not discuss in detail how these issues could be addressed, but focuses on how much effort it requires to implement the functionality that amend the issue. The prototype also employs a simplified version of the Model-View-Controller design pattern, combining the Controller and the View components. This chapter describes the work required to make the prototype adhere to the complete MVC pattern. In addition to discussing the MVC pattern, model persistence, and concurrency, the chapter considers the alpha functionality, arguing how much effort is required to implement this functionality as beta functionality.

25.1 Model-View-Controller design pattern

Currently, the prototype has combined the behavioural and presentational components, in the form of JSP files. In order to comply with the full MVC design pattern as described by the architectural description (see Part IV), the prototype is required to split the View component (consisting of JSP pages) from the Controller component (consisting of servlets). Consequently, the actions currently embedded in the JSP pages must be implemented as individual actions in Java classes implementing the Action interface, as explained in Section 18.1. In addition, rather than encoding behaviour in hyperlinks in the JSP files, a FrontController and several dispatchers must be implemented, determining how the system is to handle HTTP requests, depending on the URI of the request. Thus, hyperlinks are altered from linking to a specific JSP file, to specifying the desired action.

Implementing these changes are a relatively complex task, primarily due to the fusion of the Controller and View components. However, as the component-based architecture is attemptively sustained in spite of this fusion, especially by maintaining a clean-cut interface towards the model component, the implementation of the changes is not out of reach.

25.2 Model Persistence

The task of model persistence is to ensure that the data held by the domain objects of the domain model is stored in a database, ensuring persistent storage of the data. As explained in Chapter 16, this can be accomplished with by of Hibernate in combination with a database management system. Due to the compartmentalisation of the domain model, adding functionality for persisting it involves no changes to the model itself, only requiring the implementation of Data Access Objects and mechanisms for accessing the database, a simple, albeit large task.

As the interfaces between the Model component and the fused View-Controller component are clean-cut (they exist in separate layers of the system), inserting a model persistence layer in between as shown in Figure 18.6, is not a task of high complexity. Any implemented actions of the Controller component must add functionality to use the Data Access Objects rather than the underlying domain model, allowing clean layering of the components.

25.3 Concurrency

Presently, the prototype is not able to handle more than a single user at the time. The choice to ignore concurrency issues in the prototype was made, as discussed in Section 22.2, to lessen the complexity of the prototype. Concurrency is most important when considering gathering and maintenance of trace information. When several users are logged on the system, and accesses the same artifact instance simultaneously, a problem with data preservation can occur. Only a single user must be allowed to edit an artifact instance at any time, in order to avoid accidental overwrites of data. This is implemented by placing a read-only lock on the artifact, signalling that it is in use by another user, and can only be read, not altered. Reading trace information contained by an artifact instance is in comparison considered a safe operation, and several users can perform simultaneous read operations.

Implementing read-only locks on artifact instances is a simple task when done in its most crude form, and the real work lies in making the actions of the Controller component use the functionality provided by the locks, which involves checking the status of an artifact instance before accessing it.

25.4 Alpha functionality

Section 24.2.1 discusses how the prototype fulfils the functional requirements of the RT tool, marking functionality as either alpha or beta functionality. This section shortly discusses the work required to evolve from alpha functionality to beta functionality, and eventually, complete functionality.

Functionality for editing and deleting artifacts and traces is easy to implement, and implementation details from editing and deleting projects and users can be reused, saving effort. Likewise, help functionality requires no complex implementation, but significant attention must be given the contents and phrasing of the help instructions, a task that could incur considerable cost.

Implementing search functionality is a complex task. In its most crude form, a search functionality could involve a range of queries to a database. More sophisticatedly, the field of information retrieval [BYRN99] provides a number of methods for searching through large amounts of data. Information retrieval is beyond the scope of this thesis, and will consequently not be addressed further.

The alpha functionality of custom-made visualisation provides a foundation on which the beta functionality can be based, by illustrating how the data can be visualised. What remains, is the implementation of the underlying algorithm. This algorithm must be able to traverse the underlying domain model, looking for a path from the chosen starting artifact to the chosen ending artifact. Once this path has been determined, the View component can build a visualisation as illustrated by the alpha functionality.

Chapter 26

Summary

The primary objective of this thesis is to show how requirements traceability can be provided by the means of an RT tool. The previous chapters have presented an evolutionary prototype of an RT tool, based on the requirements specification and architectural description of Parts III and IV. The prototype is intended to give the user an impression of how the RT tool will function, without investing the effort needed to realise the complete functionality.

The prototype is developed as a web application by using the programming platform Java EE, provided by Sun Microsystems. Due to the evolutionary nature of the prototype, i.e., its ability to evolve into a complete RT tool, the design emphasises a component-based and layered architecture. This simplifies the process of adding or changing functionality as the prototype evolves.

The prototype's functionality is implemented as either alpha or beta functionality. Alpha functionality gives the user an impression of the functionality, without providing the required underlying implementation. Beta functionality extends the alpha functionality to include an implementation, albeit not complete functionality. All functional requirements of the requirements specification have been addressed by the prototype as either alpha or beta functionality, with the exception of a few requirements, that have not been addressed. The majority of the prototype's functionality is beta functionality, forming a solid foundation for further evolution into a complete RT tool.

In addition to the functional requirements, the design and implementation of an RT tool is driven by non-functional requirements. The prototype primarily focuses on the functional requirements, but in order to provide an evolvable foundation for an RT tool, the prototype gives attention to the majority of issues pointed out by the non-functional requirements. These issues are important to address if full requirements traceability is to be provided in a satisfactory manner by an RT tool. Consequently, the prototype encourages the further development of an RT tool by establishing an evolvable foundation that provides the required functionality and focus on key issues.

Part VI

Evaluation & Discussion

Chapter 27

Introduction

The previous chapters have suggested how an RT tool can be implemented, providing mechanisms that reduce the complexity of the task of implementing RT in a software development project, and the work overhead associated with this task. In order to assess the value of the proposed solution, it must be evaluated. Thereafter, its influence on the problem domain can be discussed.

27.1 Purpose

The main purpose of this part is to evaluate to what degree the research agenda put down in Section 2.3 has been realised, and discuss the influence the realised research agenda exerts on the problem domain. The proposed solution is reviewed as a whole to see if it achieves the goals put down by the research agenda. In addition, the individual parts of the solution are evaluated, focusing on their contribution to the solution as a whole. Further, this part of the thesis offers a discussion on the ramifications of the outlined RT tool, looking deeper into how the RT tool will impact the processes of a software development.

27.2 Scope

This part evaluates the preparatory work of the thesis, as well as the requirements specification, architectural description of the prototype, and the prototype itself. However, the prototype was duly evaluated in Chapter 24, and only a summary of this evaluation will be given here. An overall assessment of the realisation of the research agenda will also be given.

27.3 Overview

Here we give an overview of the structure and a quick summary of all chapters in this part.

- **Chapter 28 - Evaluation**

This chapter evaluates the overall realisation of the research agenda, as well as the individual parts of the proposed solution.

- **Chapter 29 - Discussion**

In this chapter, the reader is presented with a discussion of the significance of the proposed solution.

- **Chapter 30 - Summary**

This part summarises the most important findings in this part.

Chapter 28

Evaluation

The following sections evaluate the work of the previous parts, starting with the preparatory work. The validity of the empirical work will be evaluated, in addition to the ascertaining the value of the remainder of the research. Further, the requirements specification is evaluated, followed by the architectural description. Finally, the prototype is evaluated.

However, before any of these evaluations are given, the fulfilment of the research agenda is evaluated, summarising the work of the thesis.

28.1 Fulfilment of research agenda

The research agenda presented a number of goals that should be achieved during the thesis work. The achievements of the thesis has been evaluated in detail in the previous sections, and this section only provide a summary of how the research agenda has been fulfilled.

In the thesis, we have performed the empirical work required to establish the applicability of the results presented in the in-depth study [Nor06], and the validity of the results of the empirical work has been deemed adequate (see Section 28.2). We have performed the required preparatory work, allowing the development of a evolutionary prototype to commence. How to represent stakeholders' organisational hierarchies have been determined, likewise the internal representation of artifacts and traces.

Further, the thesis has presented a requirements specification and an architectural description, outlining an RT tool that assists its users with the tasks of implementing RT in a software development project. Finally, the thesis has presented an evolutionary prototype, describing development details, functionality, and an evaluation of how full forwards and backwards requirements traceability is provided by the prototype.

In all stages of the work of the thesis, the objective has been to design an RT tool that assists with the tasks of requirements traceability and implementing RT in software development projects. How the work overhead associated with these tasks can be reduced to an acceptable level have been the primary focus, in conformance to the research agenda.

28.2 Preparatory work

The preparatory work consists of three major tasks. The first includes empirical work, attempting to ascertain the applicability of the results presented in the in-depth study [Nor06], and the validity of this empirical work will be reviewed. The two remaining tasks are narrow literature studies, focusing on how specific findings in the literature regarding the internal representation of stakeholders, and artifacts and traces can be applied by an RT tool. The proposed use of the findings is evaluated, including a summary of the evaluation of the use of Planguage.

28.2.1 Validity of empirical work

When performing empirical work, it is important to consider the validity of the results. If the results are of *adequate validity*, they are “valid for the population to which we would like to generalize” [WRH⁺00]. The empirical work, in the shape of interviews, were intended to uncover the applicability of the results of the in-depth study by gathering expert opinions. Several factors can cause threats to the validity of the results, and they are categorised according to the type of threat they constitute for the validity. In this context, the *internal validity* and *external validity* of the results from the interviews are of high priority.

Internal validity is concerned with ensuring that the participants of the interview realise the subject of the interview, i.e., what they are discussing. This is important to be able to trust the results of the interview. The interviewees came from two different software development organisations, where one was a large organisation, whereas the other was a small organisation. This gave the interviewees an individual foundation for evaluating the applicability of the results. However, in spite of the differences in foundation, the interviewees points out many of the same aspects of the results. This increases the internal validity of the results, as they are confirmed by two independent sources.

External validity is concerned with ensuring that the results can be generalised to the entire problem domain, i.e. that it is safe to conclude that the suggestions given by the interviewees would also be suggested by any other individual with the same pretences in the problem domain. By choosing two independent interviewees, with experience from several software development projects, the external validity is increased. Their experiences will reflect a wide variety of development projects, allowing generalisation of their suggestions. In addition, by including representatives from both a small and a large software development organisation, the external validity increases further. This is caused by the similar results of the two interviews, indicating that the size of the software development organisation does not matter.

However, the external validity is threatened by the number of conducted interviews. By conducting only two interviews, we cannot be certain that the results can be generalised. Additional interviews should be conducted to remove this threat. However, due to the reduction of other threats to both the internal and external validity, as discussed above, the results of the interview is regarded to be of adequate validity. Thus, the changes to the traceability models caused by the interviews are considered to be valid.

28.2.2 Representation of stakeholders and their organisational roles

Determining a stakeholder’s role in a organisational hierarchy, his or her authority and position, and the context in which decisions are made, were in Chapter 7 found to be a complex task requiring human effort, and consequently excluded from the scope of the RT tool. The tasks of the RT tool was limited to registering the information required to deduce the structure of the organisational hierarchy.

A simple approach was outlined by the findings in theory, as this provided adaptability (by allowing a wide variety of information to be registered). However, the in-depth study [Nor06] discusses briefly a more complex solution to this problem, *Contribution structures*.

The contribution structure separates the modelling of artifacts from the modelling of *contributors* (i.e., stakeholders), mapping the artifacts to a structure of contributors. This structure of contributors reflect the underlying organisation, but as the contribution structure is limited to modelling tangible artifacts only (e.g., documents, memos, etc), it is left insufficient when attempting to map the hidden aspects of the organisational hierarchy, e.g. group dynamics and authority.

The approach outlined in Chapter 7 enables the construction of organisational hierarchies based on both visible and hidden aspects of stakeholder influence, as both a stakeholder’s position and authority can be registered. In addition, the approach enables the traceability models to capture the context of decisions affecting artifacts of the traceability model by means of the

Decision artifact. Such information makes it possible to deduce the underlying group dynamics, thus gaining greater understanding of the outcome of the decision.

Overall, the approach of Chapter 7 is implemented in the RT tool as it provides the ability to represent both the visible and the hidden aspects of the organisational hierarchy.

28.2.3 Internal representation of artifacts and traces

The traceability models specify a set of artifacts interrelated by means of traces. Chapter 8 discusses the internal representation of artifacts and traces, and how this can be accomplished with the means of Planguage.

Primarily, Planguage is used to specifying the requirements in a formal manner, i.e., quantifying them. Planguage provides a set of attributes that is originally intended to be used to describe software requirements. As the traceability models include other artifacts in addition to requirements, the set of Planguage attributes has been extended to suit the included artifacts. The choice of attributes for the individual artifacts is discussed in Chapter 8, and will not be addressed here. Adding additional attributes is a small task, and the artifacts are consequently susceptible for changes, reflecting the dynamism of the problem domain they model. However, in order to ascertain the appropriateness of using Planguage for quantifying and representing artifacts, its use must be evaluated.

28.2.4 Evaluation of Planguage

Planguage is a rich planning language which can be employed within many of the software engineering activities, in particular requirements engineering. The language contains a set of attributes that assist with specifying and quantifying requirements. However, this set of attributes is large, in order to enable the user to express any range of requirements. Planguage provides a full glossary, explaining both every attribute and any employed concepts. Planguage's ability to express all eventualities also becomes its failing, as its users are required to remember large amounts of syntax and semantics. Using Planguage alone implies writing i.e. a requirements specification from scratch, which requires extensive user knowledge of the use of Planguage, and could lead to a dependency on glossaries and templates; a severe problem, as it complicates the process of introducing new users to the world of Planguage.

The primary challenge of an RT tool is to render Planguage an intuitive aid in the process of specifying and quantifying requirements. The RT tool must integrate Planguage seamlessly, so that a user unfamiliar with the concepts of Planguage will still be able to specify requirements within the RT tool. The Planguage concept of templates have been of assistance, helping with the task of determining which attributes should be included in each artifact. A major difference between the internal representation of the artifacts of the traceability models and the original concepts of Planguage, is the dispersion of attributes across several artifacts, rather than gathering them in a single large artifact representing a requirements specification. Doing this join together the two worlds of the traceability models and Planguage, employing key concepts from both. The concept of several artifacts connected by traces is preserved, whilst the attributes of Planguage are employed throughout the traceability models, ensuring the benefits Planguage presents regarding formalisation and quantification of requirements.

However, the attributes of Planguage relates only what information each artifact should hold. Planguage also suggests how some attributes should be expressed, employing a notion referred to as *qualifiers*. Qualifiers represent a formal way of expressing the context of the attribute, i.e. where, if, and when. E.g., when specifying a goal for a non-functional requirement, qualifiers can be employed to express different goals for different context. A good example is the differentiation of novice and expert users. A novice user will often have a lower-level goal than an expert user, and qualifiers can be employed to create separate goals for each type of user. It is important that the RT tool makes the use of qualifiers an implicit and intuitive part of specifying requirement, whenever it is needed. It is likewise important that the RT tool conveys in an easily

understandable manner what information the user is required to state. The desired content of each attribute must be clearly communicated to the user.

Teaching the user how to specify requirements is not the main objective of the RT tool, and could potentially impose severe restrictions on the user if forced, in particular if the user favours his or hers own methods for specifying requirements. The RT tool exists in order to assist with the traceability of requirement rather than their specification, although these two concepts are tightly intertwined. Thus, it is important that the RT tool allows the user some freedom when stating the contents of each attribute. In addition, the employed set of attributes should be dynamic, allowing the user to choose which attributes should be employed. This provides an additional degree of freedom and adaptability to the tool.

28.3 Requirements specification

The full requirements specification of Part III is based on the high-level requirements specification presented in the in-depth study [Nor06]. The requirements of the high-level requirements specification are in turn based on important findings in the literature studies of the in-depth study, ensuring a rationale deeply rooted in the problem domain.

Figure 13.2 shows a tracking matrix, linking each requirement of the high-level requirements specification to at least one requirement of the full requirements specification. Additional requirements have been specified, primarily requirements not classified as functional requirements or software system attributes. Thus, the full requirements specification addresses the issues pointed out in the in-depth study, and includes additional requirements focusing on aspects of the RT tool not considered in the in-depth study.

Due to the evolutionary prototyping technique, the development processes are agile and iterative. Thus, the inclusion of any future requirements is enabled by performing a new iteration of the development process. This helps to ensure an always updated requirements specification,

28.4 Architectural description

Evaluation of software architectures can be done by performing an Architecture Tradeoff Analysis Method (ATAM). This is a thorough and comprehensive method for revealing how well an architecture satisfies specified quality goals, and how quality goals interacts [BCK05b]. As performing a complete ATAM requires substantial effort, this evaluation performs a simplified and informal ATAM. The steps of the ATAM is then reduced to include basic versions of *mapping of architectural decisions to quality requirements*, *identification of tradeoff points*, and *identification of risks and nonrisks*. However, the mapping of architectural decisions to quality requirements is thoroughly discussed in Chapter 16, and will not be addressed further. The results of the remaining two steps are described in the following sections.

28.4.1 Identified tradeoff points

A tradeoff point is defined as an architectural decision that affect more than one software system attribute, some positively and some negatively.

Model persistence ensures that the trace information gathered by the RT tool is persistently stored in a database, which improves the reliability of the system, as the probability of data loss is reduced. However, persisting the model consumes system resources and will consequently affect performance negatively. Consequently, there exists a tradeoff point between model persistence (reliability) and performance.

Adaptability is ensured by designing a component-based architecture extended with the Model-View-Controller design pattern. Combining this with a layered approach creates a layered application, where all communication must pass through the neighbouring layers. In situations

where a component in the top-most layer requires the services of a component in the the bottom layer, a layered approach will cause an increase in the work overhead for the intermediate layers, thus reducing the performance of the application. This is a tradeoff point between adaptability and performance, as increased performance in this case will reduce the adaptability of the application due to complex interfaces.

Scalability is an important attribute of the client-server architectural style. However, when a system upscales, i.e. in the number of simultaneous database clients, the performance could deteriorate, as the number of transactions processed per second is difficult to upscale. In addition, an architecture generating a minimal amount of requests has improved scalability when compared to an architecture that gives no attention to the amount of produced requests. Consequently, there exists a tradeoff point between the scalability and the performance of the architecture.

Usability must be considered when implementing human-computer interfaces. However, these interfaces must be designed to suit both novice and expert users. Expert users will often desire a greater level of control than novice users, implying the need for more complicated user interfaces than those required by novice users. Consequently, there exists a tradeoff point in the design of human-computer interfaces, as the interfaces must be suit the needs of both novice and experienced users.

28.4.2 Identified risks and nonrisks

A risk is defined as an architectural decision that may lead to undesirable consequences in light of system software attributes.

In addition to the risks associated with the tradeoff points discussed in the previous section, every software system attribute of the full requirements specification have been quantified, specifying failure and survival constraints. These constraints specify risks in the system, as they describe values that are unacceptable or close to unacceptable if the software system attribute is to be satisfied. However, this section will limit itself to evaluating the risks and nonrisks among the tradeoff points described in the previous section.

The tradeoff point caused by model persistence is classified as a nonrisk in the architecture, as the reduction of performance is insignificant compared to the loss of reliability. An RT tool is dependent on being able to provide reliable storage of data, as the information it contains could prove vital for its stakeholder. Thus, reduction in performance caused by persisting the model to a database is acceptable.

The same applies to the tradeoff point caused by designing an adaptable architecture. The reduced performance constitutes a minor risk when compared to the benefits harvested by an adaptable architecture. This is primarily due to the dynamism of the problem domain, creating the need for an RT tool able to adapt to its environment.

However, the tradeoff point that exists between the scalability and performance of the system constitutes a risk to the architecture, as the importance of the two attributes are assimilated. A reduction in the scalability of the system is equally detrimental as a reduction in the performance of the system. Consequently, an equilibrium between the two software system attributes must be established.

The final tradeoff point regards the usability of the system, and is also classified as a risk to the architecture, as the usability of the system is equally important to novice and experienced users, even though they have different demands on the usability. Complying to guidelines for good human-computer interface design, established through best practices, mitigates this risk.

28.5 Prototype

The evaluation of the prototype is given in Chapter 24. Only a short summary of this evaluation is given below.

The prototype's functionality is implemented as either alpha or beta functionality, where alpha functionality mimics the functionality without actually providing an implementation, and beta functionality provides a basic implementation of the functionality. All functional requirements of the requirements specification have been addressed by the prototype as either alpha or beta functionality, with the exception of a few requirements, that have not been addressed at all. The majority of the prototype's functionality is beta functionality, forming a solid foundation for further evolution into a complete RT tool.

The prototype primarily focuses on the functional requirements, but in order to provide an evolvable foundation for an RT tool, the prototype gives attention to the majority of issues pointed out by the non-functional requirements. These issues are important to address if full requirements traceability is to be provided in a satisfactory manner by an RT tool, thus reducing the work overhead associated with implementing RT in software development projects. Consequently, the prototype encourages the further development of an RT tool by establishing an evolvable foundation that provide the required functionality and focus on key issues.

Chapter 29

Discussion

It has been established by the evaluation of Chapter 28 that the prototype provides a solid foundation for further development of an RT tool, as it addresses the majority of requirements presently stated. This chapter discusses the ramifications of the prototype to the field of requirements traceability, and gives particular attention to aspects important to consider when introducing the RT tool in software development organisations. We choose to discuss the ramifications of a fully implemented RT tool, rather than the prototype, as the prototype is an outline of the tool, a first step towards a RT tool ready for use in software development projects. Thus, the properties of the prototype will in general apply to the fully implemented RT tool, allowing the discussion of the ramifications of the RT tool to be based on the properties of the prototype.

The RT tool provides mechanisms to control the artifacts of the software development processes, and their evolution as the development processes proceed. When introducing the RT tool to software development projects, the offered control mechanisms can assist with bringing closer the theoretical aspects of requirements traceability and the practical tasks of software development, thus allowing the participants of the software development project to harvest the benefits of implementing requirements traceability without adding unacceptable amounts to the work overhead. These benefits are elaborated by the in-depth study [Nor06], and include amongst others increased ability to handle changes, reduction in the influence of detrimental forces caused by the dynamism of software development, and an improved lifecycle perspective of the software, all adding qualitative value to the project.

However, if an RT tool is to provide these benefits to a software development project, it must first be introduced to the organisation conducting the project. The in-depth study discusses organisational aspects that must be considered when implementing RT, but this discussion will not delve deeper into the issues concerning the implementation of requirements traceability in general. Rather, attention will be given the issues concerning the introduction of the RT tool and how the costs associated with this task can be kept at a minimum. Large costs associated with the introduction of the RT tool will act as an obstacle, impeding its use in the field of requirements traceability. Consequently, the costs associated with introducing the RT tool must be discussed. This requires the consideration of several aspects, focusing on the cost of introduction versus the cost of use and added value to the project.

A prerequisite of introducing the RT tool is the installation of the tool. This includes configuring servers, and deploying the application. The architecture of the RT tool takes into account the *portability* of the tool, including its installability and adaptability. Several architectural strategies were chosen to support these software system attributes, including a client-server architectural style based on a virtual machine. Due to the effort invested in designing a portable architecture, the cost of installation are significantly reduced. Consequently, the cost of installation is not rendered an obstacle to the introduction of the RT tool.

In addition, the adaptability of the RT tool enables it to adjust to its environment. The underlying model is easily extended, and the component-based architecture combined with a layered approach decreases the overhead associated with introducing additional artifacts to the

model. Consequently, even though adapting the RT tool to fit the present environment does not come effortlessly, the size of this cost is still acceptable.

Further, the successful introduction of the RT tool is dependent on the usability of the tool, including its understandability, learnability, and operability. The RT tool emphasises usability, and the belonging software system attributes have been given attention throughout the development processes. High-quality usability reduces the cost of introducing an RT tool, as the need for courses, training, and other user-oriented learning sessions is decreased. Thus, the design of the RT tool encourages a minimal latency between installing the RT tool and full operability of the tool, significantly reducing the cost of introducing the RT tool to software development projects.

Overall, the design of the RT tool facilitates an cost-efficient introduction of the tool to software development projects. The costs of installation, adaptation, and user training can be kept at an acceptable level, and the time and effort spent before harvesting the benefits is equally acceptable.

Chapter 30

Summary

This part has evaluated the work of this thesis, and discussed the ramifications of the proposed RT tool. The research agenda stated at the beginning of this study was found to be satisfied, both when considering the focus of the study, and the presented results.

The presented results of the thesis were evaluated, beginning with the preparatory work, continuing with the requirements specification, architectural description, and finally, the evolutionary prototype of the RT tool.

The preparatory work included empirical work in the shape of interviews, which was evaluated to be of adequate validity. In addition, the applicability of the narrow literature studies were found satisfying, allowing the development of the prototype to commence.

The full requirements specification fulfilled all high-level requirements from the in-depth study, thus providing a good foundation for the continued design of the RT tool. The architecture was evaluated by means of a simplified ATAM, revealing several tradeoff points in the architecture. However, the majority of these tradeoff points were found to be nonrisks, with the exception of the scalability vs. performance tradeoff point, which constitutes a risk to the architecture.

Finally, the prototype was evaluated, concluding that the prototype forms a good foundation for the continued development of the RT tool, and that attention in particular must be given model persistence, scalability, and concurrency.

A discussion was included, regarding the ramifications of the prototype to the field of requirements traceability. The discussion gave particular attention to aspects important to consider when introducing the RT tool in software development organisations, and concluded that the introduction of the RT tool to real development projects comes at an acceptable cost compared to the harvested benefits.

Part VII

Conclusion & Further Work

Chapter 31

Introduction

The following part offers a conclusion on the results presented in this thesis, emphasising aspects of significance. In addition, any further work is presented, outlining the future direction of the RT tool.

31.1 Purpose

The purpose of this part is to conclude the thesis, by summarising the most important results, and reaching a conclusion regarding their significance and applicability. In addition, further work of the thesis will be deliberated.

31.2 Scope

The conclusion will only conclude on the results presented in this thesis, and will not consider the results presented in the in-depth study [Nor06], as a separate conclusion is included in the study.

Further work will not be outlined in detail, as this could place constraints on the future evolution of the RT tool.

31.3 Overview

Here we give an overview of the structure and a quick summary of all chapters in this part.

- **Chapter 32 - Conclusion**
This chapter presents a conclusion, based on the results presented throughout this thesis.
- **Chapter 33 - Further work**
In this chapter, the future work of this thesis is outlined, giving particular attention to the future work of the evolutionary prototype.

Chapter 32

Conclusion

The tasks of implementing requirements traceability in software development projects are challenging, often adding to the work overhead of the project. By using an RT tool designed to assist with the tasks of requirements traceability, this work overhead can be reduced to an acceptable level. The reduction in the work overhead is important, as it allows focus to be shifted from gathering and maintaining trace information to the analysis of the trace information, enabling the harvest of the benefits of implementing RT in software development projects.

The work of the thesis has a firm foundation in an in-depth study conducted preparatory to the thesis. The in-depth study researched the field of requirements traceability, and proposed a set of traceability models, modelling the problem domain. In addition, a high-level requirements specification was presented, listing the issues of requirements traceability an RT tool must address, based on key findings in the literature.

The applicability of the traceability models were confirmed through empirical work, in the shape of interviews. The analysis of the results of the interviews brought some changes to the models, increasing the models' ability to model practical aspects of software projects. Due to the adequate validity of the empirical work, the models' firm foundation in the world of software development is ensured.

Further, the models were extended with the ability to represent visible and hidden aspects of the organisational hierarchy, enabling trace information analysis to deduce the context of important decisions throughout the software development processes, an important tool in understanding how requirements are determined. The internal representation of artifacts and traces was determined, registering trace information with the means of Planguage, seamlessly integrated within the RT tool. This integration enables users to employ Planguage and harvest its benefits without extensive knowledge of or previous experience with the language.

The presented requirements specification and architecture outlines an RT tool that addresses the issues pointed out by the high-level requirements specification of the in-depth study, with a firm foundation in the theories of the field of requirements traceability, giving particular attention to how requirements traceability is to be provided. Consequently, an RT tool based on the presented requirements specification and architecture will be able provide its users with qualitative forwards and backwards requirements traceability.

Based on the requirements specification and architectural description, an evolutionary prototype of an RT tool has been developed. The prototype is considered a first version of a complete RT tool, giving its users the impression of how requirements traceability is provided by the tool. The prototype addresses the majority of requirements to an RT tool, which ensures that the fundamental issues of requirements traceability are addressed. This, in addition to the prototype's focus on facilitating its own evolution, makes the prototype a good foundation for further development of an RT tool.

Chapter 33

Further work

The thesis has presented an evolutionary prototype of the RT tool outlined by the requirements specification and architectural description, giving its users the impression of how a complete RT tool is intended to function, without investing the effort required to realise the complete functionality.

Evolutionary prototyping is an iterative technique, where each iteration is concluded by user-involved testing of the prototype. This testing is conducted by allowing the end-users of the RT tool use the prototype, play around with it, and determine whether the provided functionality covers their need. If new requirements are discovered, or existing requirements altered, a new iteration is required, causing the prototype to evolve. After several such iterations, a complete RT tool can be presented.

This project has only performed a single iteration of the prototyping technique, and the testing concluding the iteration has not been conducted, as additional iterations are assumed to be required. Consequently, it is left to the further work to conduct the concluding tests and perform the additional iterations required to present a complete RT tool. However, Chapter 25 presents an outline of the work required to evolve the prototype into a complete RT tool, pointing out the need for an improved implementation of the Model-View-Controller design pattern, model persistence, concurrency, and the evolution of alpha functionality into beta functionality and beyond.

When the complete RT tool has been presented, its use in software development projects must be investigated. It has previously been argued that the cost of introducing the RT tool in software development projects is acceptable when considered along the value added to the project by implementing requirements traceability. However, this proposition remains to be tested, leaving it to the further work to investigate the cost of introducing the RT tool to software development projects. Further, the applicability of the RT tool must be investigated, giving particular attention to how much (if at all) the RT tool actually reduces the work overhead. Thus, a case study of the tool is required, investigating the tool's impact on the work overhead associated with implementing requirements traceability.

Part VIII

Appendices

Appendix A

A brief overview of the in-depth study

The in-depth study was conducted during the autumn semester of 2006, at the Norwegian University of Technology and Science. The study preceded the Master's Thesis, and its contributions serve as a foundation for the thesis.

A.1 Study Outline

The study consists of four major parts, starting with research of the State of the Art of quantification and traceability of requirements (RT). This part looks closer at software requirements and requirements engineering, and discusses the current situation in the field of quantification of requirements and RT, focusing in particular on how the human element of software engineering can be handled. In addition, existing techniques and solutions are reviewed.

The study continues with an introduction to its own contributions to the field; an RT model, a requirements specification for an RT tool, and a checklist for important focus areas when integrating RT into an organisation.

An evaluation and discussion of these contributions follows, reviewing how and to what degree they affect the field of RT. The study concludes by summarising the findings of the study, and presents further work, which is referred to the Master's Thesis itself.

A.2 Important findings

When investigating the State of the Art of quantification and traceability of requirements, the study focused primarily on the human aspects of RT. A definition of requirements traceability were established, dividing RT into three different stages, as shown in Figure A.1. The pre-RS traceability stage focuses on the elicitation of the first requirements, the pre-FRS traceability stage is concerned with quantifying these requirements, and the post-FRS traceability stage maps requirements to artifacts of the development. The lines represent traces between artifacts.

Due to the focus on human aspects, the study found that many of the problems that lead to the invention of the concept of RT, were brought about by the dynamic human nature, which often caused frequent changes to project specifications, tasks, and processes. Focusing on the human aspects, an RT integration checklist was proposed, pointing out key focus areas that should be maintained in order to successfully integrate RT into an organisation.

An important part of the study was to investigate how RT can be implemented, and several challenges were identified. The *quantification of prose requirements*, i.e., the process of formalising the requirements of the stakeholders, expressing them in a quantifiable manner, was

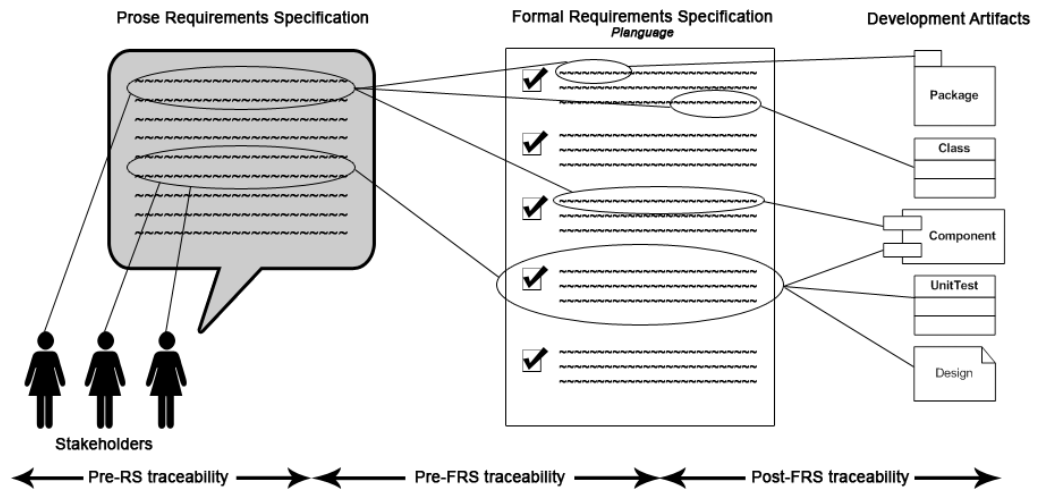


Figure A.1: Requirements traceability

found to be one of the prefatory challenges. The in-depth study proposed the use of Planguage to quantify functional and non-functional requirements. Further, *gathering trace information without creating unnecessary work overhead* for the users, as well as *ensuring that the gathered amount of trace information is satisfactory and of high quality*, were other challenges that must be overcome. As a solution to these challenges, the use of traceability models and automation of repeated tasks is suggested. *Maintenance of trace information* is also one of the challenges that must be overcome, as well as making the information available to the users, i.e., *ensuring accessibility*.

Appendix B

Employed *Planguage* attributes

This appendix alphabetically lists all the attributes employed in the internal representation of the artifacts of the traceability models. The internal representation of the artifacts is discussed in Chapter 8, and is based on the planning language described by Tom Gilb, named *Planguage*. Each attribute is accompanied by a short description of its purpose, and how it should be expressed. Full descriptions of the attributes can be found in the *Planguage Concept Glossary* [Gil05]. Some additional attributes have been added for practical reasons, these are labelled with an asterisk.

Ambition - States the requirement concerned (like “Usability”) and must contain a notion of the kind of level being sought (like “high”).

Authority - Indicates the specific level of Authority, approval, commitment, sanction, or support held by an individual or group.

Budget - A Budget is a resource target allocating a limited resource, implicating a commitment to stay within the limits of the Budget.

Category* - Specifying a Category to which the artifact belongs.

Company* - Specifying a stakeholder’s employing Company.

Context - A Context describes a (system) view from any useful perspective, for instance describing the context in which a decision was made.

Description - A Description is a set of word and/or diagrams, which describe, and partially define, an artifact.

E-mail* - E-mail address specifying an access point to a stakeholder.

Fail - A Fail constraint signals an undesirable and unacceptable system state by specifying the point at which system or attribute failure begins.

Gist - The Gist states the essence or main points of a specification.

Goal - The Goal is a primary numeric target level of performance, implying a commitment to deliver the Goal level.

Location* - This attribute specifies a Location external to the traceability model where additional information can be found.

Meter - A Meter attribute is used to identify, or specify, the definition of a practical measuring device, process, or test that has been selected for use in measuring a numeric value (level) on a defined Scale.

Name* - This attribute states the Name of e.g. a stakeholder, for identification purposes.

Office* - This attribute states the Office address of a stakeholder, for contact purposes.

Owner - A person or group responsible for an object, and for authorising any change to it.

Past - A Past parameter is used to specify historical experience.

Phone* - This attribute states the Phone number of a stakeholder, for contact purposes.

Position* - The Position attribute states the Position of a stakeholder in the organisational hierarchy.

Priority - A Priority is the determination of a relative claim on limited resources, stating the relative right of a competing requirement to the budgeted resources.

Rationale - A Rationale is the reasoning or principle that explains and thus seeks to justify a specification.

Record - A Record attribute is used to inform us about an interesting extreme of achievement.

Risk - A Risk is any factor that could result in a future negative consequence.

Scale - A Scale attribute is used to define a scale of measure, stating the units of measurement, and any required scalar qualifiers. The measuring instrument is specified by the Meter attribute.

Status - Status is the outcome of an evaluation of a defined condition (or set of conditions).

Stretch - A Stretch attribute is used to define a somewhat more ambitious target level than the committed Goal or Budget levels.

Specification - A Specification communicates one or more system ideas and/or descriptions to an intended audience, usually a formally, written means for communicating information.

Survival - Survival is a state where the system can exist, but in nearby existence of “sudden death”. A Survival specification should always have a clearly stated source specified.

Tag - A term that serves to identify a statement, or a set of statements (or artifacts), unambiguously.

Timestamp* - This attribute timestamps a Specification version.

Title* - Specifies a Title of a Specification.

Trend - A Trend attribute is used to specify how we expect or estimate attribute levels to be in the future, and is used as a benchmark.

Type - Type specifies the category of a Planguage concept, and may be defined by both Planguage and by local extensions.

Version - A version is an initial or changed Specification instance.

Wish - A Wish attribute is used to specify a stakeholder-valued uncommitted target level for a scalar attribute.

Appendix C

Internal representation of artifacts

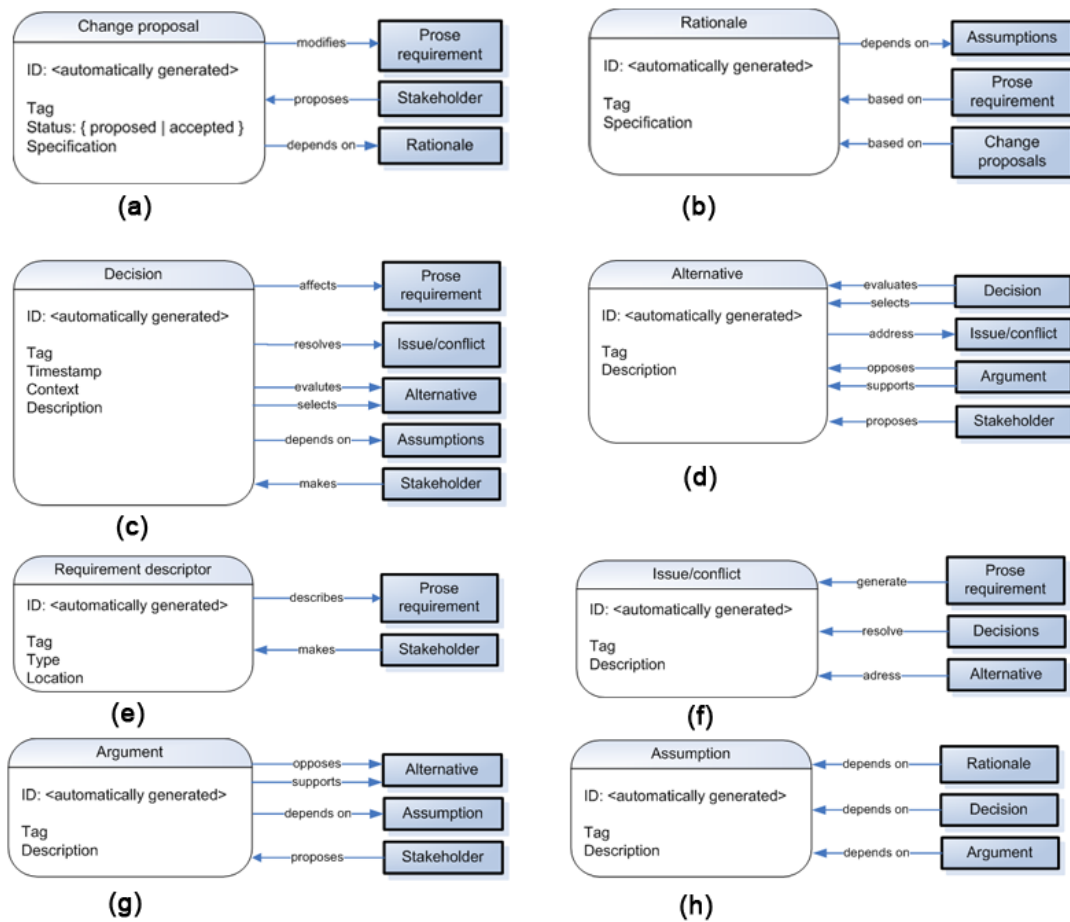


Figure C.1: Internal representation of artifacts - Pre-RS traceability submodel

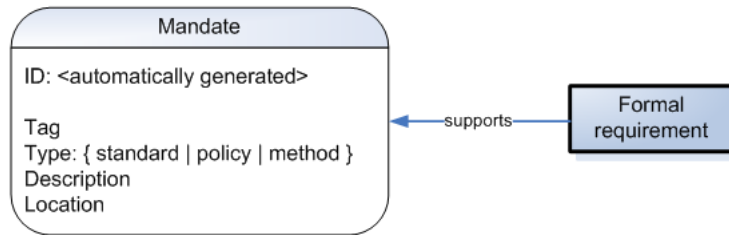


Figure C.2: Internal representation of artifacts - Pre-FRS traceability submodel

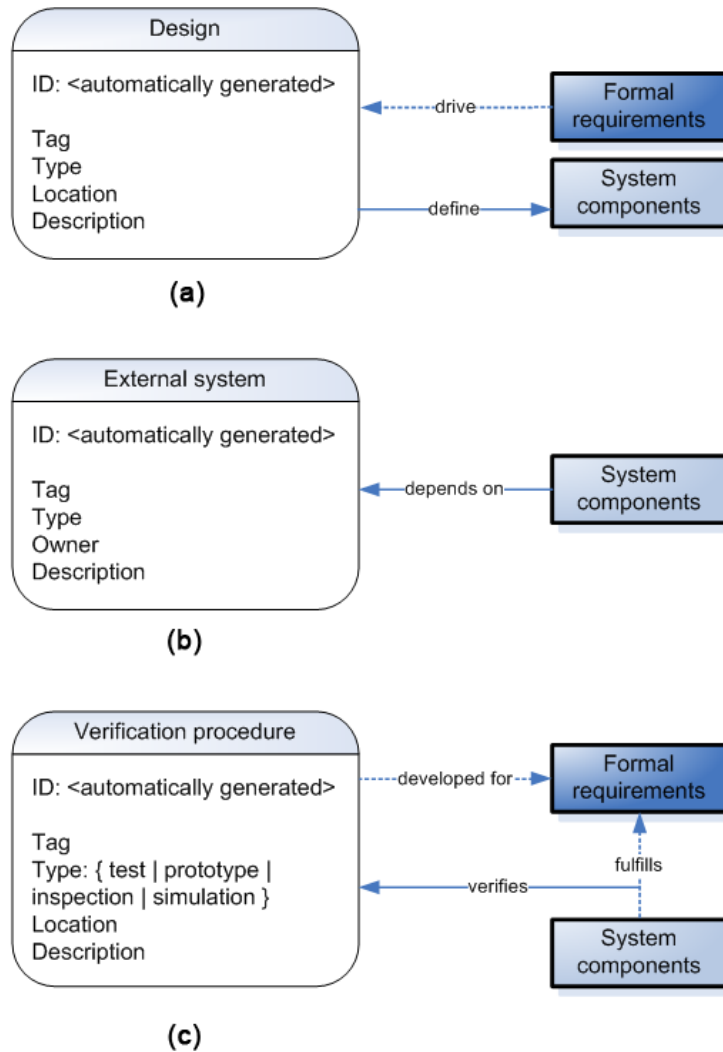


Figure C.3: Internal representation of artifacts - Post-FRS traceability submodel

Appendix D

Eight golden rules of interface design

This section introduces the reader to the subject of human-computer interface design by elaborating the eight golden rules of interface design originated from Ben Shneiderman. These eight golden rules are renowned in the world of graphical user interface design. The specification given below is taken from [Shn98b]. For more information, the reader is referred to the source. Chapter 23 discusses how the developed prototype applies these rules.

1. Strive for consistency.
 - Consistent sequences of actions should be required in similar situations.
 - Identical terminology should be used in prompts, menus, and help screens.
 - Consistent colour, layout, capitalization, fonts, and so on should be employed throughout.
2. Enable frequent users to use shortcuts.
 - To increase the pace of interaction use abbreviations, special keys, hidden commands, and macros.
3. Offer informative feedback.
 - For every user action, the system should respond in some way (in web design, this can be accomplished by DHTML - for example, a button will make a clicking sound or change color when clicked to show the user that something has happened).
4. Design dialogs to yield closure.
 - Sequences of actions should be organized into groups with a beginning, middle, and end.
 - The informative feedback at the completion of a group of actions shows the user that their activity has completed successfully.
5. Offer error prevention and simple error handling.
 - Design the form so that users cannot make a serious error; for example, prefer menu selection to form fill-in and do not allow alphabetic characters in numeric entry fields.
 - If users make an error, instructions should be written to detect the error and offer simple, constructive, and specific instructions for recovery.
 - Segment long forms and send sections separately so that the user is not penalized by having to fill in the form again - but make sure you inform the user that multiple sections are coming up.
6. Permit easy reversal of actions.
7. Support internal locus of control.

-
- Experienced users want to be in charge. Surprising system actions, tedious sequences of data entries, inability or difficulty in obtaining necessary information, and inability to produce the action desired all create anxiety and dissatisfaction.
8. Reduce short-term memory load.
- A famous study suggests that humans can store only 7 (plus or minus 2) pieces of information in their short term memory. You can reduce short term memory load by designing screens where options are clearly visible, or using pull-down menus and icons.

Appendix E

High-level Requirements Specification

Table E.1: High-level Requirements Specification

ID	Requirement
FR1	The tool must allow the creation of artifacts, as specified in TRACY.
FR2	The tool must allow modification of existing artifacts.
FR3	The tool must be able to record traces between artifacts as specified in TRACY, and must support different trace types.
FR4	Automatic detection of traces must be employed when possible.
FR5	The tool must be able to visually represent the recorded trace information to its users.
FR6	The visual representation of the trace information should consist of several views, each representing the trace information differently.
FR7	The tool must implement a search function, enabling users to search for specific traces.
NFR1	The tool must store all trace information persistently.
NFR2	The tool should be easily adaptable to its current environment.
NFR3	The tool must be platform-independent.
NFR4	The tool must be scaleable, enabling it to handle small and large amounts of trace information equally well.
NFR5	The tool must make sure the necessary trace information is gathered, without imposing the user.
NFR6	The user must be able to record and find the desired trace information easily and quickly.

Appendix F

Full Requirements Specification

F.1 External interface requirements

F.1.1 User interface requirements

Table F.1: User Interface Requirement - Window-based application

Attribute	Contents
Tag	ExternalInterface.UserInterface.WindowApplication
Title	Window-based application
Description	The product must be usable through a window-based application.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	Medium

Table F.2: User Interface Requirement - Feedback

Attribute	Contents
Tag	ExternalInterface.UserInterface.Feedback
Title	Provide user with feedback
Description	Each task must, when completed, provide the user with feedback describing the results of the previous user action.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High

Table F.3: User Interface Requirement - Navigation

Attribute	Contents
Tag	ExternalInterface.UserInterface.Navigation
Title	Navigation
Description	The user must be able to navigate within the system.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High
Sub-functions	ExternalInterface.UserInterface.Navigation.Location ExternalInterface.UserInterface.Navigation.Escape ExternalInterface.UserInterface.Navigation.Menu ExternalInterface.UserInterface.Navigation.Menu.Logout

Table F.4: User Interface Requirement - Location

Attribute	Contents
Tag	ExternalInterface.UserInterface.Navigation.Location
Title	Location within system
Description	The user must at all times be notified of the current location (i.e. which task he is currently working on) within the system, also know as the current path.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High
Supra-functions	ExternalInterface.UserInterface.Navigation

Table F.5: User Interface Requirement - Escape

Attribute	Contents
Tag	ExternalInterface.UserInterface.Navigation.Escape
Title	Escaping system tasks
Description	The system must always provide the user with navigation possibilities for escaping tasks in progress.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High
Supra-functions	ExternalInterface.UserInterface.Navigation

Table F.6: User Interface Requirement - Menu

Attribute	Contents
Tag	ExternalInterface.UserInterface.Navigation.Menu
Title	Navigation Menu
Description	The system must always provide the user with a navigation menu enabling the user to select any major navigational choices, consisting of traceability stages, artifacts, traces, and visualisations .
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High
Supra-functions	ExternalInterface.UserInterface.Navigation
Sub-functions	ExternalInterface.UserInterface.Navigation.Menu.Logout

Table F.7: User Interface Requirement - Logout

Attribute	Contents
Tag	ExternalInterface.UserInterface.Navigation.Menu.Logout
Title	User logging off
Description	The navigation menu must contain the choice of logging out of the system.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High
Supra-functions	ExternalInterface.UserInterface.Navigation.Menu

Table F.8: User Interface Requirement - Eight Golden Rules of HCI Design

Attribute	Contents
Tag	ExternalInterface.UserInterface.GoldenRules
Title	The Eight Golden Rules of HCI Design
Description	The design of the user interfaces must comply with the eighth golden rules of humancomputer interface design (as defined in Appendix D).
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High

F.1.2 Hardware interfaces

Table F.9: Hardware Interface Requirement - Hardware Independence

Attribute	Contents
Tag	ExternalInterface.HardwareInterface.HardwareIndependency
Title	Hardware Independence
Description	The system is to be implemented in a hardware-independent fashion, and should not rely on any particular hardware interface.
Version	1
Stakeholder	User Architect Developer Tester
Priority	Medium
Risk	Low

F.1.3 Software interfaces

Table F.10: Software Interface Requirement - Application Server

Attribute	Contents
Tag	ExternalInterface.SoftwareInterface.ApplicationServer
Title	Application Server
Description	The system requires a underlying application server, providing an environment in which the system will run. The application server will serve as a front-end to clients, offering its services and handling all communication, thus shielding the system from the implementation of communication interfaces.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High

Table F.11: Software Interface Requirement - Database Server

Attribute	Contents
Tag	ExternalInterface.SoftwareInterface.DatabaseServer
Title	Database Server
Description	The system requires a back-end database server, providing persistant storage of data.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High

F.1.4 Communications interfaces

Table F.12: Communication Interface Requirement - Underlying Communication

Attribute	Contents
Tag	ExternalInterface.CommunicationsInterface.UnderlyingCommunication
Title	Underlying Communication
Description	No external communication interfaces are required by the system, as all such communication are handled by the underlying application server.
Version	1
Stakeholder	User Architect Developer Tester
Priority	Medium
Risk	Low

F.2 Functional requirements

F.2.1 Traceability projects

Table F.13: Functional Requirement - Create a new project

Attribute	Contents
Tag	Functional.ManagingProjects.CreateNewProject
Title	Create a new project
Description	The administrative user must be able to create a new project with a given name.
Version	1
Stakeholder	Administrative user Architect Developer Tester
Priority	High
Risk	High
Dependencies	Functional.Administrative.UserAuthentication

Table F.14: Functional Requirement - Remove a project

Attribute	Contents
Tag	Functional.ManagingProjects.RemoveProject
Title	Remove a project
Description	The administrative user must be able to remove a project, deleting all trace information associated with the project.
Version	1

Continued on next page

Table F.14 – continued from previous page

Attribute	Contents
Stakeholder	Administrative user Architect Developer Tester
Priority	High
Risk	High
Dependencies	Functional.Administrative.UserAuthentication Functional.ManagingProjects.CreateNewProject

Table F.15: Functional Requirement - Attach users with project

Attribute	Contents
Tag	Functional.ManagingProjects.AttachUsers
Title	Attach users to a project
Description	The administrative user must be able to attach users to a project, and remove users from a project.
Version	1
Stakeholder	Administrative user Architect Developer Tester
Priority	High
Risk	High
Dependencies	Functional.Administrative.UserAuthentication Functional.ManagingProjects.CreateNewProject Functional.Administrative.MaintainingUserList

Table F.16: Functional Requirement - Choose project to work with

Attribute	Contents
Tag	Functional.WorkingWithProjects.ChooseProject
Title	Choose project to work with
Description	The user must be able to choose which project her or she wishes to work with.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High
Dependencies	Functional.Administrative.UserAuthentication Functional.ManagingProjects

Table F.17: Functional Requirement - Exit project

Attribute	Contents
Tag	Functional.WorkingWithProjects.ExitProject
Title	Exit project
Description	The user must be able to quit working with a specific project, thus exiting the project.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High
Dependencies	Functional.Administrative.UserAuthentication Functional.WorkingWithProjects.ChooseProject

Table F.18: Functional Requirement - Determine Traceability Stage

Attribute	Contents
Tag	Functional.WorkingWithProjects.DetermineTraceabilityStage
Title	Determine traceability stage
Description	The user must be able to switch between the traceability stages of the traceability model, i.e. between the pre-RS traceability, pre-FRS traceability, and post-FRS traceability stage.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High
Dependencies	Functional.Administrative.UserAuthentication Functional.WorkingWithProjects.ChooseProject

Table F.19: Functional Requirement - Navigating the trace information

Attribute	Contents
Tag	Functional.WorkingWithProjects.NavigateTraceInformation
Title	Navigating the trace information
Description	The user must be able to navigate through the available trace information, viewing artifacts and traces and their belonging trace information.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High
Dependencies	Functional.Administrative.UserAuthentication Functional.WorkingWithProjects

F.2.2 Gathering trace information

Table F.20: Functional Requirement - Creating an artifact instance

Attribute	Contents
Tag	Functional.CreatingArtifacts.CreatingAnArtifactInstance
Title	Creating an artifact instance
Description	The system must allow the user to create instances of any of the artifacts modelled in TRACY, requesting the required trace information, and thereafter storing it persistently.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High
Dependencies	Functional.Administrative.UserAuthentication Functional.WorkingWithProjects.DetermineTraceabilityStage

Table F.21: Functional Requirement - Choosing a starting artifact instance

Attribute	Contents
Tag	Functional.CreatingTraces.ChoosingStartingArtifactInstance
Title	Choosing starting artifact instance
Description	The system must allow the user to choose a starting artifact instance for a trace, deciding among existing artifact instances in the determined traceability stage.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High
Dependencies	Functional.Administrative.UserAuthentication Functional.CreatingArtifacts

Table F.22: Functional Requirement - Choosing an ending artifact instance

Attribute	Contents
Tag	Functional.CreatingTraces.ChoosingEndingArtifactInstance
Title	Choosing ending artifact instance
Description	The system must allow the user to choose an ending artifact instance for a trace, deciding among a list of existing artifact instances. This list must be made up of only the artifacts that can be connected to the chosen starting artifact, as determined in the traceability model.
Continued on next page	

Table F.22 – continued from previous page

Attribute	Contents
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High
Dependencies	Functional.Administrative.UserAuthentication Functional.CreatingArtifacts Functional.CreatingTraces.ChoosingStartingArtifactInstance

Table F.23: Functional Requirement - Registering auxiliary trace information

Attribute	Contents
Tag	Functional.CreatingTraces.RegisteringAuxiliaryTraceInformation
Title	Register auxiliary trace information
Description	The system must request any required auxiliary traceability information in accordance with the internal representation of the chosen trace, and store the trace.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High
Dependencies	Functional.Administrative.UserAuthentication Functional.CreatingArtifacts Functional.CreatingTraces.ChoosingStartingArtifactInstance Functional.CreatingTraces.ChoosingEndingArtifactInstance

F.2.3 Maintaining trace information

Table F.24: Functional Requirement - Choose artifact to edit

Attribute	Contents
Tag	Functional.EditingArtifacts.ChooseArtifactToEdit
Title	Choose artifact to edit
Description	The user must be able to edit a selected artifact instance.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High
Dependencies	Functional.Administrative.UserAuthentication Functional.CreatingArtifacts

Table F.25: Functional Requirement - Save changes to edited artifact instance

Attribute	Contents
Tag	Functional.EditingArtifacts.SaveChangesToArtifactInstance
Title	Save changes to edited artifact instance
Description	Any changes to an existing artifact instance during edit must be saved by the system when requested by the user.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High
Dependencies	Functional.Administrative.UserAuthentication Functional.CreatingArtifacts Functional.EditingArtifacts.ChooseArtifactToEdit

Table F.26: Functional Requirement - Choose artifact to delete

Attribute	Contents
Tag	Functional.DeletingArtifacts.ChooseArtifactToDelete
Title	Choose artifact to delete
Description	The user must be able to delete any artifact instance, and then be presented with a request for confirmation of deletion, in which the user can either acknowledge or renounce the request for deletion.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High
Dependencies	Functional.Administrative.UserAuthentication Functional.CreatingArtifacts

Table F.27: Functional Requirement - Choose trace to edit

Attribute	Contents
Tag	Functional.EditingTraces.ChooseTraceToEdit
Title	Choose trace to edit
Description	The user must be able to request that a selected trace is to be edited, and then be presented with a list of alternative artifact instances of the same categories as the previous artifact instances. An editable version of the existing auxiliary trace information of the selected trace must also be presented to the user.
Version	1
Stakeholder	User Architect Developer Tester

Continued on next page

Table F.27 – continued from previous page

Attribute	Contents
Priority	High
Risk	High
Dependencies	Functional.Administrative.UserAuthentication Functional.CreatingArtifacts Functional.CreatingTraces

Table F.28: Functional Requirement - Save changes to edited traces

Attribute	Contents
Tag	Functional.EditingTraces.SaveChangesToTrace
Title	Save changes to edited traces
Description	Any changes to an existing trace during edit must be saved by the system when requested by the user.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High
Dependencies	Functional.Administrative.UserAuthentication Functional.CreatingArtifacts Functional.EditingTraces.ChooseTraceToEdit

Table F.29: Functional Requirement - Choose trace to delete

Attribute	Contents
Tag	Functional.DeletingTraces.ChooseTraceToDelete
Title	Choose trace to delete
Description	The user must be able to delete a selected artifact instance, and then be presented with a request for confirmation of deletion, in which the user can either acknowledge or renounce the request for deletion.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High
Dependencies	Functional.Administrative.UserAuthentication Functional.CreatingArtifacts Functional.CreatingTraces

Table F.30: Functional Requirement - System-initiated trace deletion

Attribute	Contents
Tag	Functional.DeletingTraces.SystemInitiatedTraceDeletion
Title	System-initiated trace deletion
Description	The system itself must be able to request a deletion of a trace, when deleting an artifact, without having to acknowledge or renounce the request for deletion.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High
Dependencies	Functional.Administrative.UserAuthentication Functional.CreatingArtifacts Functional.CreatingTraces

Table F.31: Functional Requirement - Searching trace information

Attribute	Contents
Tag	Functional.SearchingTraceInformation
Title	Searching Trace Information
Description	The system must be able to list information that contains part of or all search terms listed by a user, and the hit ratio of the search must be at least 80 %, meaning that at least 80 % of the result set of the search must be related to the search terms listed by the user.
Version	1
Stakeholder	User Architect Developer Tester
Priority	Medium
Risk	Low
Dependencies	Functional.Administrative.UserAuthentication Functional.CreatingArtifacts Functional.CreatingTraces

F.2.4 Visualising trace information

Table F.32: Functional Requirement - Requesting visualisation of single artifacts or traces

Attribute	Contents
Tag	Functional.PredeterminedVisualisations.RequestingSimpleVisualisation
Title	Requesting visualisation of single artifacts or traces
Description	The user must be able to request a visualisation of a single artifact or trace, selected by the user.
Version	1
Continued on next page	

Table F.32 – continued from previous page

Attribute	Contents
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High
Dependencies	Functional.Administrative.UserAuthentication Functional.CreatingArtifacts Functional.CreatingTraces Functional.PredeterminedVisualisations.SimpleVisualisations

Table F.33: Functional Requirement - Requesting complex visualisations

Attribute	Contents
Tag	Functional.PredeterminedVisualisations.RequestingComplexVisualisation
Title	Requesting complex visualisations
Description	The user must be able to request predetermined visualisations of complex relationships within the traceability models, including several artifact instances and traces. The user should choose from a list of predetermined visualisations.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	Medium
Dependencies	Functional.Administrative.UserAuthentication Functional.CreatingArtifacts Functional.CreatingTraces Functional.PredeterminedVisualisations.ComplexVisualisations

Table F.34: Functional Requirement - Simple predetermined visualisations

Attribute	Contents
Tag	Functional.PredeterminedVisualisations.SimpleVisualisations
Title	Simple predetermined visualisations
Description	The system must contain a predetermined set of simple visualisations, corresponding to the artifacts and traces of the traceability models, to be used when the user requests a visualisation of a single artifact or trace.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High
Dependencies	Functional.Administrative.UserAuthentication Functional.CreatingArtifacts Functional.CreatingTraces

Table F.35: Functional Requirement - Complex predetermined visualisations

Attribute	Contents
Tag	Functional.PredeterminedVisualisations.ComplexVisualisations
Title	Complex predetermined visualisations
Description	The system must contain a predetermined set of complex visualisations, visualising an assortment of the artifacts and traces of the traceability models.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	Medium
Dependencies	Functional.Administrative.UserAuthentication Functional.CreatingArtifacts Functional.CreatingTraces

Table F.36: Functional Requirement - Adapting simple visualisations

Attribute	Contents
Tag	Functional.CustommadeVisualisations.AdaptingSimpleVisualisations
Title	Adapting simple visualisations
Description	The system must provide the ability to add filters to the predetermined set of simple visualisations, enabling the user to impose regulations on the displayed information, thus creating an adapted (custom-made) simple visualisation.
Version	1
Stakeholder	User Architect Developer Tester
Priority	Medium
Risk	Low
Dependencies	Functional.Administrative.UserAuthentication Functional.CreatingArtifacts Functional.CreatingTraces Functional.PredeterminedVisualisations

Table F.37: Functional Requirement - Creating custom-made complex visualisations

Attribute	Contents
Tag	Functional.CustommadeVisualisations.CreatingComplexVisualisations
Title	Creating custom-made complex visualisations
Description	The user must be able to create custom-made complex visualisations of a set of user-specified artifacts and traces.
Version	1
Continued on next page	

Table F.37 – continued from previous page

Attribute	Contents
Stakeholder	User Architect Developer Tester
Priority	Low
Risk	Low
Dependencies	Functional.Administrative.UserAuthentication Functional.CreatingArtifacts Functional.CreatingTraces
Sub-functions	Functional.CustommadeVisualisations.CreatingComplexVisualisations. ChooseStartingAndEndingArtifact

Table F.38: Functional Requirement - Choosing starting and ending artifact of complex custom-made visualisations

Attribute	Contents
Tag	Functional.CustommadeVisualisations.CreatingComplexVisualisations. ChooseStartingAndEndingArtifact
Title	Choosing starting and ending artifact of complex custom-made visualisations
Description	When creating a custom-made complex visualisations, the user must be able to specify a starting artifact instance from a set of existing instances. The system must provide a list of possible ending artifact instances based on the traceability models, and the user must be able to choose one of these, causing the system to generate a custom-made complex visualisation.
Version	1
Stakeholder	User Architect Developer Tester
Priority	Low
Risk	Low
Dependencies	Functional.Administrative.UserAuthentication Functional.CreatingArtifacts Functional.CreatingTraces Functional.CustommadeVisualisations.CreatingComplexVisualisations
Supra-functions	Functional.CustommadeVisualisations.CreatingComplexVisualisations

F.2.5 Help and assistance

Table F.39: Functional Requirement - Help Messages

Attribute	Contents
Tag	Functional.Help.Messages
Title	Help Messages
Description	Help must be available for all tasks within the product. If tasks are attempted incorrectly, the product must present the user with a message explaining the problem, shedding light on how the task is to be performed correctly.
Continued on next page	

Table F.39 – continued from previous page

Attribute	Contents
Version	1
Stakeholder	User Architect Developer Tester
Priority	Medium
Risk	Medium
Dependencies	Functional.Administrative.UserAuthentication

Table F.40: Functional Requirement - Help Menu

Attribute	Contents
Tag	Functional.Help.Menu
Title	Help Menu
Description	The user must be able to look through a help menu describing all tasks of the system, and how these are executed.
Version	1
Stakeholder	User Architect Developer Tester
Priority	Medium
Risk	Medium
Dependencies	Functional.Administrative.UserAuthentication
Sub-functions	Functional.Help.Menu.Search

Table F.41: Functional Requirement - Search Help Menu

Attribute	Contents
Tag	Functional.Help.Menu.Search
Title	Help Menu
Description	The user must be able to search through the help menu, in order to locate help instructions without having to look through all of them.
Version	1
Stakeholder	User Architect Developer Tester
Priority	Medium
Risk	Medium
Dependencies	Functional.Administrative.UserAuthentication Functional.Help.Menu
Supra-functions	Functional.Help.Menu

F.2.6 Administrative functions

Table F.42: Functional Requirement - Adding a user

Attribute	Contents
Tag	Functional.Administrative.MaintainingUserList.AddingUser
Title	Adding a user
Description	An administrative user must be able to add a user to the user list, specifying the granted access level, as well as a username and password.
Version	1
Stakeholder	Administrative user Architect Developer Tester
Priority	High
Risk	High
Dependencies	Functional.Administrative.UserAuthentication

Table F.43: Functional Requirement - Removing a user

Attribute	Contents
Tag	Functional.Administrative.MaintainingUserList.RemovingUser
Title	Removing a user
Description	An administrative user must be able to remove a user from the user list.
Version	1
Stakeholder	Administrative user Architect Developer Tester
Priority	Medium
Risk	Medium
Dependencies	Functional.Administrative.UserAuthentication Functional.Administrative

Table F.44: Functional Requirement - Changing user password

Attribute	Contents
Tag	Functional.Administrative.MaintainingUserList.ChangingPassword
Title	Changing password
Description	An administrative user and regular users must be able to change their own password. In addition, an administrative user must be able to change the password of all other users, in case of forgotten passwords.
Version	1
Stakeholder	Administrative user Regular user Architect Developer Tester
Priority	High
Risk	Medium
Dependencies	Functional.Administrative.UserAuthentication Functional.Administrative

Table F.45: Functional Requirement - Changing access level

Attribute	Contents
Tag	Functional.Administrative.MaintainingUserList.ChangingAccessLevel
Title	Changing access level
Description	An administrative user must be able to change the access level of all other users.
Version	1
Stakeholder	Administrative user Architect Developer Tester
Priority	Medium
Risk	Low
Dependencies	Functional.Administrative.UserAuthentication Functional.Administrative

Table F.46: Functional Requirement - User Authentication

Attribute	Contents
Tag	Functional.Administrative.UserAuthentication
Title	User Authentication
Description	A user must log in by providing a username and password for authentication before being given access to system resources. If login succeeded, the user is granted his or her prespecified access level and shown the list of projects he or she is attached to. If login failed, the user is notified and access is not granted.
Version	1
Stakeholder	User Architect Developer Tester
Priority	Medium
Risk	Low
Dependencies	Functional.Administrative

F.3 Performance requirements

F.3.1 Throughput

Table F.47: Performance Requirement - Throughput: Number of simultaneously active users

Attribute	Contents
Tag	Nonfunctional.Performance.Throughput.ActiveUsers
Title	Throughput: Number of simulatenously active users
Description	The system must support at least 50 simultaneously active users, where an active user represents a logged-in user performing tasks within the system.
Continued on next page	

Table F.47 – continued from previous page

Attribute	Contents
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High
Ambition	A high level of simultaneous users, without violating any other performance requirements.
Scale	Number of simultaneously active users
Targets	Goal: 70 Stretch: 100 Wish: 150
Constraints	Fail: 30 Survival: 40
Rationale	The goal was set at 70 simultaneous users so that the system can be employed by both small projects (with perhaps 0-10 participants) and large projects (with perhaps 20-100 participants). The number of supported simultaneous users does not need to equal the number of participants in the project, as not all participants in a software development project partake in the traceability processes.

Table F.48: Performance Requirement - Throughput: Number of requests to system

Attribute	Contents
Tag	Nonfunctional.Performance.Throughput.SystemRequests
Title	Throughput: Number of requests to system
Description	The throughput of system requests must be at least 50 processed requests per second.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High
Ambition	A high throughput in number of processed system requests.
Scale	Number of processed requests per time unit
Targets	Goal: 50 processed requests per second Stretch: 100 processed requests per second Wish: 150 processed requests per second
Constraints	Fail: 20 processed requests per second Survival: 35 processed requests per second
Rationale	The number of requests processed by a system per second is highly fluctuating, depending on the type of requests, and how many other requests are present in the system. A higher load often deteriorates the throughput. The number of processed requests is also dependent upon the application server.

Table F.49: Performance Requirement - Throughput: Number of requests to database

Attribute	Contents
Tag	Nonfunctional.Performance.Throughput.DatabaseRequests
Title	Throughput: Number of requests to database
Description	The throughput of database requests must be at least 15 processed requests per second.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High
Ambition	A high throughput in number of processed database requests, both when storing and fetching trace information.
Scale	Number of processed database requests per time unit
Targets	Goal: 15 processed requests per second Stretch: 30 processed requests per second Wish: 70 processed requests per second
Constraints	Fail: 5 processed requests per second Survival: 10 processed requests per second
Rationale	The number of requests processed by a database per second is highly fluctuating, depending on the type of requests, type of database, and which performance improvements have been implemented on the database server.

F.3.2 Response time

Table F.50: Performance Requirement - Response time: User requests

Attribute	Contents
Tag	Nonfunctional.Performance.ResponseTime.UserRequests
Title	Response time: User requests
Description	The response time for requests from the user to the system, including any system-generated requests to the database, should not exceed 5 seconds, as this increases the chance of user exhaustion.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	Medium
Ambition	A short response time that will help removing the risk of user exhaustion.
Scale	The mean average speed to perform a defined task when requested.
Targets	Goal: 5 seconds Stretch: 2 seconds Wish: 1 second
Constraints	Fail: 10 seconds Survival: 7 seconds
Rationale	User exhaustion occurs after approximately 3-5 seconds, according to [Shn98a].

Table F.51: Performance Requirement - Response time: Database requests

Attribute	Contents
Tag	Nonfunctional.Performance.ResponseTime.DatabaseRequests
Title	Response time: Database requests
Description	The response time for requests from the system to the database should not exceed 3 seconds, as this increases the chance of user exhaustion.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	Medium
Ambition	A short response time that will help removing the risk of user exhaustion.
Scale	The mean average speed to perform a defined task when requested.
Targets	Goal: 3 seconds Stretch: 1 second Wish: 0.5 seconds
Constraints	Fail: 7 seconds Survival: 5 seconds
Rationale	User exhaustion occurs after approximately 3-5 seconds, according to [Shn98a].

Table F.52: Performance Requirement - Response time: Search

Attribute	Contents
Tag	Nonfunctional.Performance.ResponseTime.Search
Title	Response time: Search
Description	The results of a search initiated by the user must be displayed to the user within at least 5 seconds, as this decreases the chance of user exhaustion.
Version	1
Stakeholder	User Architect Developer Tester
Priority	Medium
Risk	Medium
Ambition	A short response time that will help removing the risk of user exhaustion.
Scale	The mean average speed to perform a defined task when requested.
Targets	Goal: 5 seconds Stretch: 2 seconds Wish: 1 second
Constraints	Fail: 10 seconds Survival: 7 seconds
Rationale	User exhaustion occurs after approximately 3-5 seconds, according to [Shn98a].

Table F.53: Performance Requirement - Response time: User authentication

Attribute	Contents
Tag	Nonfunctional.Performance.ResponseTime.UserAuthentication
Title	Response time: User authentication
Description	When the user has requested an authentication, the system response must be provided within 3 seconds.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	Medium
Ambition	A short response time that will help removing the risk of user exhaustion.
Scale	The mean average speed to perform a defined task when requested.
Targets	Goal: 3 seconds Stretch: 1 seconds Wish: 0.5 second
Constraints	Fail: 10 seconds Survival: 5 seconds
Rationale	User exhaustion occurs after approximately 3-5 seconds, according to [Shn98a].

Table F.54: Performance Requirement - Response time: Visualisations

Attribute	Contents
Tag	Nonfunctional.Performance.ResponseTime.Visualisations
Title	Response time: Visualisations
Description	The response time for generating visualisations of artifacts specified by the user must not exceed 7 seconds.
Version	1
Stakeholder	User Architect Developer Tester
Priority	Medium
Risk	Medium
Ambition	A short response time that will help removing the risk of user exhaustion.
Scale	The mean average speed to perform a defined task when requested.
Targets	Goal: 7 seconds Stretch: 3 seconds Wish: 1 second
Constraints	Fail: 10 seconds Survival: 7 seconds
Rationale	User exhaustion occurs after approximately 3-5 seconds, according to [Shn98a].

F.3.3 Storage capacity

Table F.55: Performance Requirement - Storage capacity: Registered users

Attribute	Contents
Tag	Nonfunctional.Performance.StorageCapacity.RegisteredUsers
Title	Storage capacity: Registered users
Description	
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High
Ambition	Providing a system that in normal mode can be employed by an extensive number of users.
Scale	The capacity to store defined units under defined conditions
Targets	Goal: 500 Stretch: 700 Wish: 1000
Constraints	Fail: 70 Survival: 200
Rationale	By allowing the system to register at least 200 users, and preferable at least 500 users, the tool can be employed by large software development organisations, introducing all the benefits of requirements traceability.

Table F.56: Performance Requirement - Storage capacity: Trace information

Attribute	Contents
Tag	Nonfunctional.Performance.StorageCapacity.TraceInformation
Title	Storage capacity: Trace information
Description	The system must be able to store at least 2000 artifacts of each kind and their belonging traces.
Version	1
Stakeholder	User Architect Developer Tester
Priority	Medium
Risk	Low
Ambition	Providing an apparently endless storage facility for trace information.
Scale	The capacity to store defined units under defined conditions.
Targets	Goal: 2000 instances of each artifact category, with trace Stretch: 4000 instances of each artifact category, with traces Wish: 1000 instances of each artifact category, with traces
Constraints	Fail: 500 instances of each artifact category, with traces Survival: 1000 instances of each artifact category, with traces
Rationale	Enabling the system to store large amounts of data is not difficult, as the database can easily be expanded. Ensuring valid response times and throughput when the system grows could prove difficult, this is a potential trade-off.

F.4 Design constraints

F.4.1 Standards compliance

Table F.57: Design Constraint - Architectural description

Attribute	Contents
Tag	NonFunctional.DesignConstraints.StandardsCompliance.ArchitecturalDescription
Title	Architectural description
Description	The architectural description of the system must comply with the IEEE standard 1471-2000 [IEEmla].
Version	1
Stakeholder	Architect
Priority	High
Risk	Medium

F.4.2 Hardware limitations

Table F.58: Design Constraint - Hardware limitations

Attribute	Contents
Tag	NonFunctional.DesignConstraints.HardwareLimitations
Title	Hardware limitations
Description	The system must be able to be deployed, run, and tested on a regular desktop computer with an installation of an application server and a database server, with a minimum of 1 GB internal memory, a CPU of at least 1 GHz, and a minimum of 1 GB of available space on a disk.
Version	1
Stakeholder	Architect Developer Tester
Priority	Medium
Risk	Low
Rationale	The given specifications are the specifications of a modern desktop computer, and will most likely be superdimensioned in this context.

F.5 Software system attributes

F.5.1 Functionality

Table F.59: Software System Attribute - Nonrepudiation

Attribute	Contents
Tag	Nonfunctional.Attribute.Security.Nonrepudiation
Title	Nonrepudiation
Description	Any transactions within the system during normal operation mode storing trace information must be nonrepudiative by any party.
Continued on next page	

Table F.59 – continued from previous page

Attribute	Contents
Version	1
Stakeholder	User Architect Developer Tester
Priority	Medium
Risk	Medium
Ambition	Trace information stored by the system must be nonrepudiative, not allowing any parties to deny the origin of the information, thus avoiding any false trace information representations.
Scale	Probability of detecting individual responsible for unauthorised modification of data.
Targets	Goal: 80 % Stretch: 95 % Wish: 99 %
Constraints	Fail: 50 % Survival: 70 %

Table F.60: Software System Attribute - Confidentiality

Attribute	Contents
Tag	Nonfunctional.Attribute.Security.Confidentiality
Title	Confidentiality
Description	Any data and services within the system must be protected from unauthorised access in both normal and degraded operation mode.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High
Ambition	Trace information stored by the system, and the services provided by the system must be impossible to access for illegitimate users in both normal and degraded system operation mode.
Scale	Probability of detecting individuals attempting unauthorised access.
Targets	Goal: 95 % Stretch: 99 % Wish: 99.9 %
Constraints	Fail: 85 % Survival: 90 %

Table F.61: Software System Attribute - Auditing

Attribute	Contents
Tag	Nonfunctional.Attribute.Security.Auditing
Title	Auditing
Description	The usage of any services within the system in normal system operation mode must be audited, as well as the storage of any trace information, registering the information required to backtrack any actions when necessary.
Version	1

Continued on next page

Table F.61 – continued from previous page

Attribute	Contents
Stakeholder	User Architect Developer Tester
Priority	High
Risk	Medium
Ambition	Any activity within the system must be possible to reconstruct.
Scale	Amount of time required to restore data.
Targets	Goal: 12 hrs Stretch: 6 hrs Wish: 1 hr
Constraints	Fail: 24 hrs Survival: 16 hrs

F.5.2 Reliability

Table F.62: Software System Attribute - Recoverability

Attribute	Contents
Tag	Nonfunctional.Attribute.Recoverability
Title	Recoverability
Description	When experiencing fatal error and subsequent downtime and degraded system operation mode, the downtime of the system must be no more than 1 hour per 24 hours (mean time to recover)
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High
Ambition	A high degree of recoverability, allowing users to quickly regain access to the application after a failure.
Scale	Mean time to recover after failure, measured in hours.
Targets	Goal: 1 hour Stretch: 30 minutes Wish: 10 minutes
Constraints	Fail: 5 hours Survival: 2 hours

Table F.63: Software System Attribute - Availability

Attribute	Contents
Tag	Nonfunctional.Attribute.Reliability.Availability
Title	Availability
Description	In normal system operation mode, and with a mean time to recover as specified in NonFunctional.Attribute.Reliability.Recoverability, the availability rate (as specified in Equation 12.1) must be at least 0.95
Version	1

Continued on next page

Table F.63 – continued from previous page

Attribute	Contents
Stakeholder	User Architect Developer
Priority	High
Risk	High
Ambition	A high degree of availability, allowing users to access the application whenever desirable.
Scale	$\frac{\text{Mean time to failure}}{\text{Mean time to failure} + \text{Mean time to recover}} \quad (\text{F.1})$
Targets	Goal: 0.95 Stretch: 0.98 Wish: 1.00
Constraints	Fail: 0.50 Survival: 0.70

F.5.3 Maintainability

Table F.64: Software System Attribute - Changeability

Attribute	Contents
Tag	Nonfunctional.Attribute.Maintainability.Changeability
Title	Changeability
Description	The system must be designed in a manner that lessens the effort of introducing a change to the system.
Version	1
Stakeholder	Architect Developer
Priority	High
Risk	High
Ambition	Ensuring fast and cost-effective introduction of changes to the system.
Scale	Time spent by a developer (with the specified level of knowledge of the system's architecture and design) introducing a new component to the system (not including time spent implementing the component itself)
Targets	Goal: A developer with medium experience spends 3 hours introducing a new component. Stretch: A developer with some experience spends 3 hours introducing a new component. Wish: A developer with medium experience spends 1 hour introducing a new component.
Constraints	Fail: A developer with extensive experience spends 8 hours introducing a new component. Survival: A developer with medium experience spends 6 hours introducing a new component.
Dependencies	None

Table F.65: Software System Attribute - Testability

Attribute	Contents
Tag	Nonfunctional.Attribute.Maintainability.Testability
Title	Testability
Description	Preparing the system for unit testing of an implemented change, and performing the unit test must not require more resources than those specified below.
Version	1
Stakeholder	Tester
Priority	Medium
Risk	Low
Ambition	A system design that can be easily tested during and after development.
Scale	The amount of effort required for setting up a unit test harness for the system and performing the unit tests, measured in work hours.
Targets	Goal: 3 hours. Stretch: 1 hours Wish: 15 minutes
Constraints	Fail: 16 hours Survival: 8 hours

F.5.4 Portability

Table F.66: Software System Attribute - Installability: Installation costs

Attribute	Contents
Tag	Nonfunctional.Attribute.Portability.Installability.InstallationCosts
Title	Installability
Description	The cost of deploying the system must be acceptable to expert users of the system.
Version	1
Stakeholder	User Architect Developer
Priority	Medium
Risk	Low
Ambition	The system must be deployable at acceptable cost by expert users, i.e. IT professionals.
Scale	Invested user effort, measured in simple user operations (e.g. a mouse click or keystrokes), for deploying the system.
Targets	Goal: Maximum 40 simple user operations Stretch: Maximum 30 simple user operations Wish: Maximum 15 simple user operations
Constraints	Fail: 100 simple user operations Survival: 60 simple user operations

Table F.67: Software System Attribute - Installability: Time to install

Attribute	Contents
Tag	Nonfunctional.Attribute.Portability.Installability.TimeToInstall
Title	Portability
Description	The cost, measured in time, of moving the system from an initial location to a target location must be acceptable to the developing staff and the user group (or customer).
Version	1
Stakeholder	User Architect Developer
Priority	High
Risk	High
Ambition	The system must be able to be relocated to a different environment without disturbing the daily work to an unacceptable degree.
Scale	Amount of time spent porting the system from the initial location to the target location with the assistance of expert users.
Targets	Goal: Maximum 1 hrs Stretch: Maximum 0.5 hr Wish: Maximum 0.25 hrs
Constraints	Fail: More than 4 hrs Survival: Maximum 2 hrs

Table F.68: Software System Attribute - Adaptability: Increase number of simultaneous users

Attribute	Contents
Tag	Nonfunctional.Attribute.Portability.Adaptability IncreaseNumberOfSimultaneousUsers
Title	Adaptability: Increase number of simultaneous users
Description	
Version	1
Stakeholder	Architect Developer Tester
Priority	High
Risk	High
Ambition	Cost-efficient addition of functionality for handling an increased number of users.
Scale	Invested user effort, measured in time, for increasing the number of simultaneous users within the system.
Targets	Goal: Maximum 3 working days, i.e. 24 hrs Stretch: Maximum 2 working days, i.e. 16 hrs Wish: Maximum 1 working day, i.e. 8 hrs
Constraints	Fail: More than 6 working days, i.e 48 hrs Survival: Maximum 5 working days, i.e. 40 hours
Dependencies	NonFunctional.Performance.Throughput.ActiveUsers Functional.Administrative.UserAuthentication

Table F.69: Software System Attribute - Adaptability: Alterations to the traceability model

Attribute	Contents
Tag	Nonfunctional.Attribute.Portability.Adaptability.Improveability.AlterationsToTraceabilityModel
Title	Adaptability: Alterations to the traceability model.
Description	Improving the system by applying changes to the traceability model must be accomplishable at an acceptable cost, measured in time.
Version	1
Stakeholder	Architect Developer Tester
Priority	Medium
Risk	Medium
Ambition	A system that easily handles changes to the core element of the system, i.e. the traceability models.
Scale	Invested user effort, measured in time, for applying changes to the traceability model, where the changes include between 0 and 10 artifacts. For a greater number of artifacts, the allowed amount of time spent applying the changes can be doubled.
Targets	Goal: Maximum 2 working days, i.e. 16 hrs Stretch: Maximum 1 working days, i.e. 8 hrs Wish: Maximum 0.5 working day, i.e. 4 hrs
Constraints	Fail: More than 4 working days, i.e 32 hrs Survival: Maximum 3 working days, i.e. 24 hours

F.5.5 Usability

Table F.70: Software System Attribute - Understandability

Attribute	Contents
Tag	Nonfunctional.Attribute.Usability.Understandability
Title	Understandability
Description	The amount of experience required to understand the overall application and use of the system.
Version	1
Stakeholder	User Architect Developer Tester
Priority	Medium
Risk	Medium
Ambition	The developed system shall not require extensive domain knowledge, i.e. extensive knowledge and/or previous experience with requirements traceability, in order to be used efficiently.
Scale	The defined level of knowledge required to understand and perform the basic operations of the system
Targets	Goal: Novice users with a minimum of 1 year experience with software development, and knowledge of, but no experience with RT. Stretch: Novice users with at least six months experience with software development, but no previous experience with nor knowledge of RT. Wish: Novice users with no previous experience with nor knowledge of software development or RT.

Continued on next page

Table F.70 – continued from previous page

Attribute	Contents
Constraints	<p>Fail: Users with a minimum of 3 years experience with software development and 1 year experience with RT.</p> <p>Survival: Users with a minimum of 5 years experience with software development and 2 years experience with RT, in addition to at least 1 encounter with other RT tools.</p>

Table F.71: Software System Attribute - Learnability

Attribute	Contents
Tag	Nonfunctional.Attribute.Usability.Learnability
Title	Learnability
Description	Novice users (users with no previous experience with the system apart from training) with a maximum of 1 hour one-to-one training (one student, one teacher) are required to execute a certain amount of the tasks of the system with a specified degree of proficiency, as defined in the targets.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	Medium
Ambition	A system that require little user training regardless of the entry level experience of the user.
Scale	The degree of training required for a defined user type to achieve a defined degree of proficiency with the system.
Targets	<p>Goal: 1 hour of training for a novice user results in 90 % successfully accomplished tasks.</p> <p>Stretch: 1 hour of training for a novice user results in 95 % successfully accomplished tasks.</p> <p>Wish: 1 hour of training for a novice user results in 99 % successfully accomplished tasks.</p>
Constraints	<p>Fail: 1 hour of training for a novice user results in 60 % successfully accomplished tasks.</p> <p>Survival: 1 hour of training for a novice user results in 75 % successfully accomplished tasks.</p>

Table F.72: Software System Attribute - Operability: Visualisation flexibility

Attribute	Contents
Tag	Nonfunctional.Attribute.Operability.VisualisationFlexibility
Title	Visualisation Flexibility
Description	The visualisations of trace information must in normal system operation mode be adaptable by the users of the system, allowing enhanced system operability.
Version	1
Stakeholder	User Architect Developer Tester
Priority	Low

Continued on next page

Table F.72 – continued from previous page

Attribute	Contents
Risk	Low
Ambition	The system must be easily operable in order to accommodate users in a wide range of working situations.
Scale	Invested user effort, measured in simple user operations (e.g. a mouse click or a keystroke), for adapting trace information to current working situation.
Targets	Goal: Maximum 6 simple user operations Stretch: Maximum 4 simple user operations Wish: Maximum 2 simple user operations
Constraints	Fail: 15 simple user operations Survival: 10 simple user operations
Dependencies	Functional.CustommadeVisualisations

Table F.73: Software System Attribute - Operability: Change functionality

Attribute	Contents
Tag	Nonfunctional.Attribute.Usability.Operability.SwitchFunctionality
Title	Switch between functionality
Description	A user must, in normal system operation mode, face acceptable costs when changing focus between main system components, in order to work with different functionalities and tasks.
Version	1
Stakeholder	User Architect Developer Tester
Priority	High
Risk	High
Ambition	Changing between the different functionalities of the system must represent an unacceptable cost to the user.
Scale	Invested user effort, measured in simple user operations (e.g. a mouse click or a keystroke), for changing between the different functionalities of the system.
Targets	Goal: Maximum 3 simple user operations Stretch: Maximum 2 simple user operations Wish: Maximum 1 simple user operations
Constraints	Fail: 5 simple user operations Survival: 4 simple user operations
Dependencies	Functional.WorkingWithProjects

Appendix G

Use cases

Table G.1: Use case - User login

	Comment
Use Case	User login
Tag	UseCase.UserLogin
Included functional requirements	Functional.Administrative.UserAuthentication
Actors	User, system
Pre-conditions	The user has opened a web browser and entered the URL of the application.
Main Success Scenario	<ol style="list-style-type: none"> 1: System presents user with a login window, requesting username and password. 2: User provides the required information. 3: System validates the information and checks the user access level. 4: System sets the appropriate access rights and grants the user access. 5: System redirects the user to the main page of the application.
Extensions	3a. User is not registered within the system. .1: System denies the user access, and displays a rejection message to the user. .2: User can retry login, or contact a system administrator for assistance.

Table G.2: Use case - Choose project

	Comment
Use Case	Choose project
Tag	UseCase.ChooseProject
Included functional requirements	Functional.Administrative.UserAuthentication Functional.WorkingWithProjects.ChooseProject Functional.WorkingWithProjects.EscapeProject
Actors	Regular user, system
Pre-conditions	The user is <u>logged in</u> .
Main Success Scenario	<ol style="list-style-type: none"> 1: System presents user with a choice of the projects the user is attached to. 2: User selects the desired project. 3: System presents the traceability stages of the chosen project. 4: User chooses the desired traceability stage.
Continued on next page	

Table G.2 – continued from previous page

	Comment
Extensions	3a. User has chosen the wrong project. .1: User asks system to exit the current project. .2: System exits the current project. .3: Enter MMS at step 1.

Table G.3: Use case - Choose traceability stage

	Comment
Use Case	Choose traceability stage
Tag	UseCase.ChooseTraceabilityStage
Included functional requirements	Functional.Administrative.UserAuthentication Functional.WorkingWithProjects.ChooseProject Functional.WorkingWithProjects.DetermineTraceabilityStage
Actors	Regular user, system
Pre-conditions	The user is <u>logged in</u> and has <u>has chosen a project</u> .
Main Success Scenario	1: System presents user with a choice of the three traceability stages pre-RS traceability, pre-FRS traceability, and post-FRS traceability. 2: User selects the desired stage. 3: System presents the tasks sorting under this traceability stage.
Extensions	None

Table G.4: Use case - Working with artifacts

	Comment
Use Case	Working with artifacts
Tag	UseCase.WorkingWithArtifacts
Included functional requirements	Functional.Administrative.UserAuthentication Functional.WorkinWithProjects.ChooseProject Functional.WorkingWithProjects.DetermineTraceabilityStage
Actors	Regular user, system
Pre-conditions	The user is <u>logged in</u> , has <u>choosen a project</u> , and has chosen a traceability stage.
Main Success Scenario	1: User informs system that he or she wishes to work with artifacts. 2: System presents a list of possible artifact actions. 3: User selects desired artifact action.
Extensions	3a: User cannot find the desired artifact action in the presented list. .1: User uses the help functionality and searches through the help topics, looking for the desired action. .2: System presents the list of related help topics. .3: User selects a suitable topic, or if no suitable topic is found, informs the system of this. .4: System informs the system administrator of the missing functionality. .5: System administrator evaluates the missing functionality, and if deemed necessary, forwards a change proposal to the RT tool development team. .6: The development team implements the missing functionality.

Table G.5: Use case - Create an artifact

	Comment
Use Case	Create an artifact
Tag	UseCase.CreateAnArtifact
Included functional requirements	Functional.Administrative.UserAuthentication Functional.WorkinWithProjects.ChooseProject Functional.WorkingWithProjects.DetermineTraceabilityStage Functional.CreatingArtifacts.CreateAnArtifactInstance
Actors	Regular user, system
Pre-conditions	The user is <u>logged in</u> , has <u>chosen a project</u> , and has chosen the desired traceability stage, then selected to <u>work with artifacts</u> .
Main Success Scenario	1: User selects to create a new artifact. 2: System presents the artifact categories. 3: User selects which artifact type he or she wishes to create. 4: System presents the attributes belonging to the artifact type, prompting the user to provide this information. 5: User provides the desired attributes of the artifact. 6: System saves the artifact. 7: System confirms the creation of the new artifact.
Extensions	6a. System fails to save the artifact due to incorrect user input . .1: System notifies the user of the failure, specifying what went wrong and how it could be corrected. .2: User reattempts to save the artifact by correcting the specified errors. .3: Enter MSS (Main Success Scenario) at step 6. 6b. System fails to save the artifact due to system error. .1: System notifies the user of the failure, specifying that the error was caused by the system and not the user. .2: System notifies the maintenance team of the error, specifying which user experienced the problem . .3: Maintenance team corrects the error, and informs the users that the error have been corrected. .4: Enter MSS at step 1.

Table G.6: Use case - Edit an existing artifact

	Comment
Use Case	Edit an existing artifact
Tag	UseCase.EditAnArtifact
Included functional requirements	Functional.Administrative.UserAuthentication Functional.WorkinWithProjects.ChooseProject Functional.WorkingWithProjects.DetermineTraceabilityStage Functional.EditingArtifacts.ChooseArtifactToEdit Functional.EditingArtifacts.SaveChangesToArtifactInstance
Actors	Regular user, system
Pre-conditions	The user is <u>logged in</u> , has <u>chosen a project</u> , and has chosen the desired traceability stage, then selected to <u>work with artifacts</u> .
Main Success Scenario	1: User <u>locates the artifact</u> that is to be edited. 2: User informs system that this artifact shall be edited. 3: System presents existing details on artifact. 4: User performs desired changes. 5: System saves changes. 7: System confirms a successful save.
Continued on next page	

Table G.6 – continued from previous page

	Comment
Extensions	<p>6a. System fails to save the artifact due to incorrect user input .</p> <p>.1: System notifies the user of the failure, specifying what went wrong and how it could be corrected.</p> <p>.2: User reattempts to save the artifact by correcting the specified errors.</p> <p>.3: Enter MSS at step 6.</p> <p>6b. System fails to save the artifact due to system error.</p> <p>.1: System notifies the user of the failure, specifying that the error was caused by the system and not the user. .2: System notifies the maintenance team of the error, specifying which user experienced the problem .</p> <p>.3: Maintenance team corrects the error, and informs the users that the error have been corrected.</p> <p>.4: Enter MSS at step 1.</p>

Table G.7: Use case - Delete an existing artifact

	Comment
Use Case	Delete an existing artifact
Tag	UseCase.DeleteExistingArtifact
Included functional requirements	Functional.Administrative.UserAuthentication Functional.WorkinWithProjects.ChooseProject Functional.WorkingWithProjects.DetermineTraceabilityStage Functional.DeletingArtifacts.ChooseArtifactToDelete Functional.DeletingTraces.SystemInitiatedTraceDeletion
Actors	Regular user, system
Pre-conditions	The user is <u>logged in</u> , has <u>chosen a project</u> , and has <u>chosen the desired traceability stage</u> , then selected to <u>work with artifacts</u> .
Main Success Scenario	1: User <u>locates</u> the artifact that is to be deleted. 2: User informs system that this artifact shall be deleted. 3: System asks the user for a confirmation of deletion. 4: User provides the necessary confirmation. 5: System deletes the artifact, and any belonging traces. 7: System confirms a successful deletion.
Extensions	4a. User aborts the deletion of the artifact. .1: System confirms that the artifact was not deleted. .2: System forwards user to the view of the artifact.

Table G.8: Use case - Locate an existing artifact

	Comment
Use Case	Locate an existing artifact
Tag	UseCase.LocateAnExistingArtifact
Included functional requirements	Functional.Administrative.UserAuthentication Functional.WorkinWithProjects.ChooseProject Functional.WorkingWithProjects.DetermineTraceabilityStage Functional.WorkingWithProjects.NavigatingTraceInformation Functional.SearchingTraceInformation
Actors	Regular user, system
Pre-conditions	The user is <u>logged in</u> , has <u>chosen a project</u> , and has <u>chosen the desired traceability stage</u> , then selected to <u>work with artifacts</u> .
Continued on next page	

Table G.8 – continued from previous page

	Comment
Main Success Scenario	1: System presents a list of the existing categories of artifacts. 2: User chooses the desired category of artifact. 3: User chooses to <u>search for the desired artifact</u> . 4: User enters keywords into a search field. 5: System presents the search results, limited to artifacts within the chosen category. 6: User selects the desired artifact. 7: System presents the information stored within the artifact.
Extensions	2a. User chooses to <u>search for the desired artifact</u> . .1: User enters keywords into a search field. .2: System presents the search results, limited to artifacts. .3: Enter MSS at step 3.

Table G.9: Use case - Add a trace between artifacts

	Comment
Use Case	Add a trace between artifacts
Tag	UseCase.AddTrace
Included functional requirements	Functional.Administrative.UserAuthentication Functional.WorkinWithProjects.ChooseProject Functional.WorkingWithProjects.DetermineTraceabilityStage Functional.CreatingTraces
Actors	Regular user, system
Pre-conditions	The user is <u>logged in</u> , has <u>chosen a project</u> , has <u>chosen the desired traceability stage</u> , and then indicated that he or she wishes to <u>work with artifacts</u> .
Main Success Scenario	1: User notifies the system that he or she wishes to register a trace. 2: System presents the user with the artifacts that are possible to trace between, according to the model representing the chosen traceability stage (pre-conditions). 3: User selects an artifact. 4: System presents the possible artifacts that the chosen artifact can be linked to. 5: User chooses the desired second artifact. 6: System presents any attributes that requires filling out. 7: User provides any required attribute information. 8: System creates a trace between the two artifacts.
Extensions	2a. No artifacts have yet been registered in the system. .1: System notifies the user. .2: User aborts the registration of a trace. .3: System forwards user to the visualisations of existing traces.

Table G.10: Use case - Delete a trace between artifacts

	Comment
Use Case	Delete a trace between artifacts
Tag	UseCase.DeleteTrace
Included functional requirements	Functional.Administrative.UserAuthentication Functional.WorkinWithProjects.ChooseProject Functional.WorkingWithProjects.DetermineTraceabilityStage Functional.DeletingTraces
Actors	Regular user, system

Continued on next page

Table G.10 – continued from previous page

	Comment
Pre-conditions	The user is <u>logged in</u> , has <u>chosen a project</u> , and has chosen the desired traceability stage.
Main Success Scenario	<ol style="list-style-type: none"> 1: User informs system that he or she wishes to delete a trace. 2: System presents the user a list of traces belonging to the current traceability stage, sorted after the artifacts they connect. 3: User selects a trace. 4: System requests a confirmation that this trace is to be deleted. 5: User provides the necessary confirmation. 6: System deletes the trace. 7: System confirms a successful deletion. 8: System forwards the user to the visualisation of existing traces.
Extensions	<ol style="list-style-type: none"> 4a. User aborts deletion of the trace. <ol style="list-style-type: none"> .1: System confirms that the trace was not deleted. .2: Enter MSS at step 8. 6a. System fails to delete trace. <ol style="list-style-type: none"> .1: System informs user that the deletion failed, and that the trace still exists. .2: Enter MSS at step 8.

Table G.11: Visualise trace information

	Comment
Use Case	Visualise trace information
Tag	UseCase.VisualiseTraceInformation
Included functional requirements	Functional.Administrative.UserAuthentication Functional.WorkinWithProjects.ChooseProject Functional.WorkingWithProjects.DetermineTraceabilityStage Functional.PredeterminedVisualisations Functional.CustommadeVisualisations
Actors	Regular user, system
Pre-conditions	The user is <u>logged in</u> , has <u>chosen a project</u> , and has chosen the desired traceability stage.
Main Success Scenario	<ol style="list-style-type: none"> 1: User selects to visualise traceability data. 2: System asks the user to choose between a pre-defined visualisations or creating custom-made visualisations. 3: User selects to create a custom-made visualisation. 4: System presents a list of starting artifact categories. 5: User selects one of these starting categories. 6: System presents the artifacts belonging to the chosen category. 7: User selects the desired start artifact. 8: System presents a list of potential ending artifact categories. 9: User selects one of these ending categories. 10: System presents the visualisation.
Extensions	None

Table G.12: Use case - Search through trace information

	Comment
Use Case	Search through trace information
Tag	UseCase.SearchTraceInformation
Continued on next page	

Table G.12 – continued from previous page

	Comment
Included functional requirements	Functional.Administrative.UserAuthentication Functional.WorkinWithProjects.ChooseProject Functional.WorkingWithProjects.NavigateTraceInformation Functional.SearchingTraceInformation
Actors	Regular user, system
Pre-conditions	The user is <u>logged in</u> , and has <u>chosen a project</u> .
Main Success Scenario	1: User enters keywords into the search field. 2: System searches through trace information, building a result set containing references to the keywords 3: System presents the result set to the user.
Extensions	3a. The result set is empty. .1: System notifies user that his or her keywords didn't have any hits within the trace information .2: System allows the user to enter a new query into the search field.

Table G.13: Use case - Creating a project

	Comment
Use Case	Create a project
Tag	UseCase.CreateProject
Included functional requirements	Functional.Administrative.UserAuthentication Functional.ManagingProjects.CreateNewProject Functional.ManagingProjects.AttachUsers
Actors	Administrative user, system
Pre-conditions	The user is <u>logged in</u> .
Main Success Scenario	1: User notifies system that a project is to be created. 2: System asks user for the name of the new project. 3: User states the name of the new project. 4: System creates a new project with the name specified by the user, and informs the user of the results of its action. 5: User tells system to attach users to the project. 6: System presents a list of users registered within the system. 7: User selects the users that are to be attached to the system. 8: System attaches the users, and informs the user of the results of its action.
Extensions	6a. No users are registered within the system other than the administrative user performing the task. .1: System informs user that no other users exist and allows user to register a user within the system. 8a: User discovers the wrong users have been attached to the project. .1: User asks system to remove the incorrect user(s). .2: System removes the specified user(s). .3: System presents user with the list of currently attached users.

Table G.14: Use case - Register a user

	Comment
Use Case	Register a user within the system
Tag	UseCase.RegisterUser
Continued on next page	

Table G.14 – continued from previous page

	Comment
Included functional requirements	Functional.Administrative
Actors	Administrative user, system
Pre-conditions	The user is <u>logged in</u> .
Main Success Scenario	<p>1: User notifies system that a new user is to be registered.</p> <p>2: System asks user to specify the username, password and access level of the new user.</p> <p>3: User states the required information and asks system to store the information.</p> <p>4: System registers a new user with the attributes specified by the user and presents the stored results to the user.</p>
Extensions	<p>4a. System cannot register a new user because a user with the specified attributes already exists.</p> <p>.1: System informs the user of the problem, and asks the user to enter a new username, password and access level.</p> <p>.2: Enter MSS at step 3.</p>

Table G.15: Use case - Search through help directory

	Comment
Use Case	Search through help directory
Tag	UseCase.SearchHelpDirectory
Included functional requirements	Functional.Help
Actors	User, system
Pre-conditions	The user is <u>logged in</u> , and have a problem with figuring out how to register new artifacts.
Main Success Scenario	<p>1: User asks system for the help directory.</p> <p>2: System displays the help directory, allowing the user to look through the topics, as well as searching for interesting topics by entering key words.</p> <p>3: User enters the key word “artifact”, and requests system to search for topics including this key word.</p> <p>4: System searches through topics and lists the results.</p> <p>5: User look through topics and finds a topic concerning the creation of artifacts.</p>
Extensions	<p>4a. System finds no topics including the given key word.</p> <p>.1: System displays an error message to the user, stating that no topics were found.</p> <p>.2: User asks system to perform another search.</p> <p>.3: Enter MSS at step 2.</p>

Appendix H

Prototype development

H.1 Prototype design

The prototype can be described as two abstract components, where one encapsulates the other. The outer component, referred to as the administrative component, provides functionality for creating and managing projects within the RT tool, whilst the inner component, referred to as the project component, provides functionality for working with specific projects. Complying with a simplified Model-View-Controller design pattern, the administrative component uses the Tool Model of Figure 18.3 as its underlying model, whilst the project component uses the Domain Model shown in the same figure. The views and controller are combined, implemented in a number of JSP files, organised according to Figure H.1.

Figure H.1 primarily shows the organisations of the JSP files into folders, but some select JSP files have been included in the file hierarchy. JSP fragments (fragments of JSP code meant to be included in complete JSP files) are used to include banners and navigation functionality on every page, these files are stored in the WEB-INF folder, which ensures that they cannot be accessed unless they are included by a regular JSP file. JSP files acting as the controller and views for the Tool Model are stored in the *project* and *user* folder, whilst the controller and views of the Domain Model are stored in *tool* folder. The login screen (`login.jsp`) and welcome screen (`welcome.jsp`) are stored directly under the root of the prototype, and are shown in the bottom of Figure H.1. In addition, the style sheet determining the layout of the graphical user interfaces is stored directly under the root (`traceme_styles.css`).

Figure H.2 presents the structure of the prototype, using the notation of UML class diagrams (presented in Section 17.1). In the model shown in the figure, the boundaries of the project component is illustrated by heavy dotted lines. The boundaries of the administrative component is not shown, as the component constitutes the entire model. However, the Project object have a special role, as shown in Figure H.2, the object exists in both components, aggregated by the administrative component by the `ProjectList`, and displaying its content in the project component.

The administrative component contains a list of projects, and provides functionality for manipulating this list and the properties of each project, and opening the contents of projects. In addition, the administrative component keeps a list of registered users within the system, and provides functionality for changing the attributes of a user, such as name password, and access level. Users and projects are implemented as individual objects, and the web container of the application server controls the administrative component by instantiating the `ProjectList` and `UserList` object when the prototype is first deployed (using the `ResourceManagerListener` shown in Figure 18.8). Only a single instance of these objects exist. The `preRS`, `preFRS`, and `postFRS` packages contain the domain model, sorted according to traceability stage. The detailed specification of the domain model is given in Figure H.3 through H.5.

An instance of the `Project` object contains, in addition to a list of users attached to the project, a set of artifacts instantiated from the domain objects of the domain model, sorted according

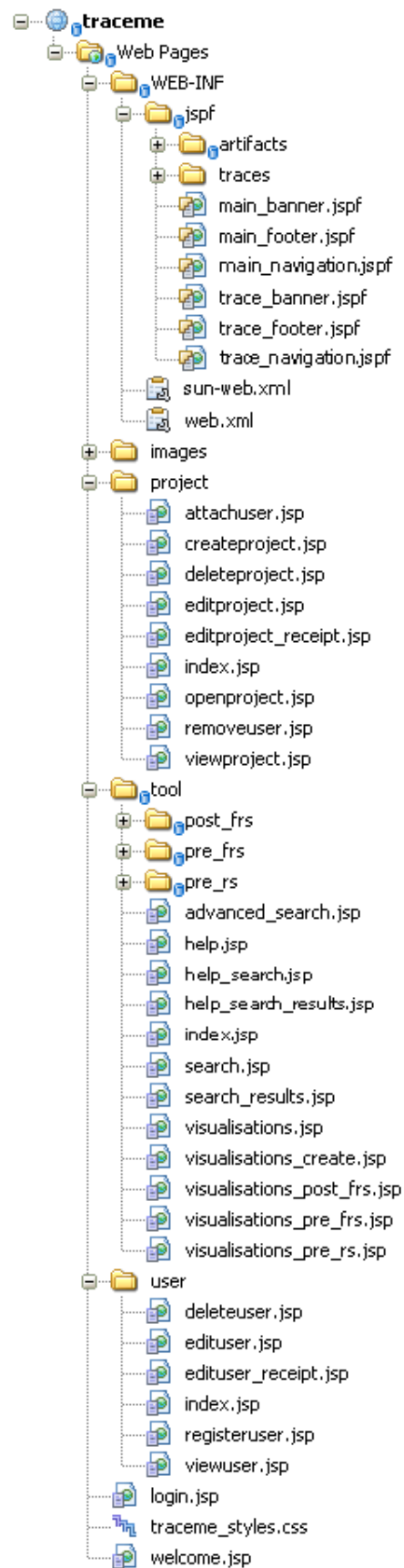


Figure H.1: Organisation of JSP files

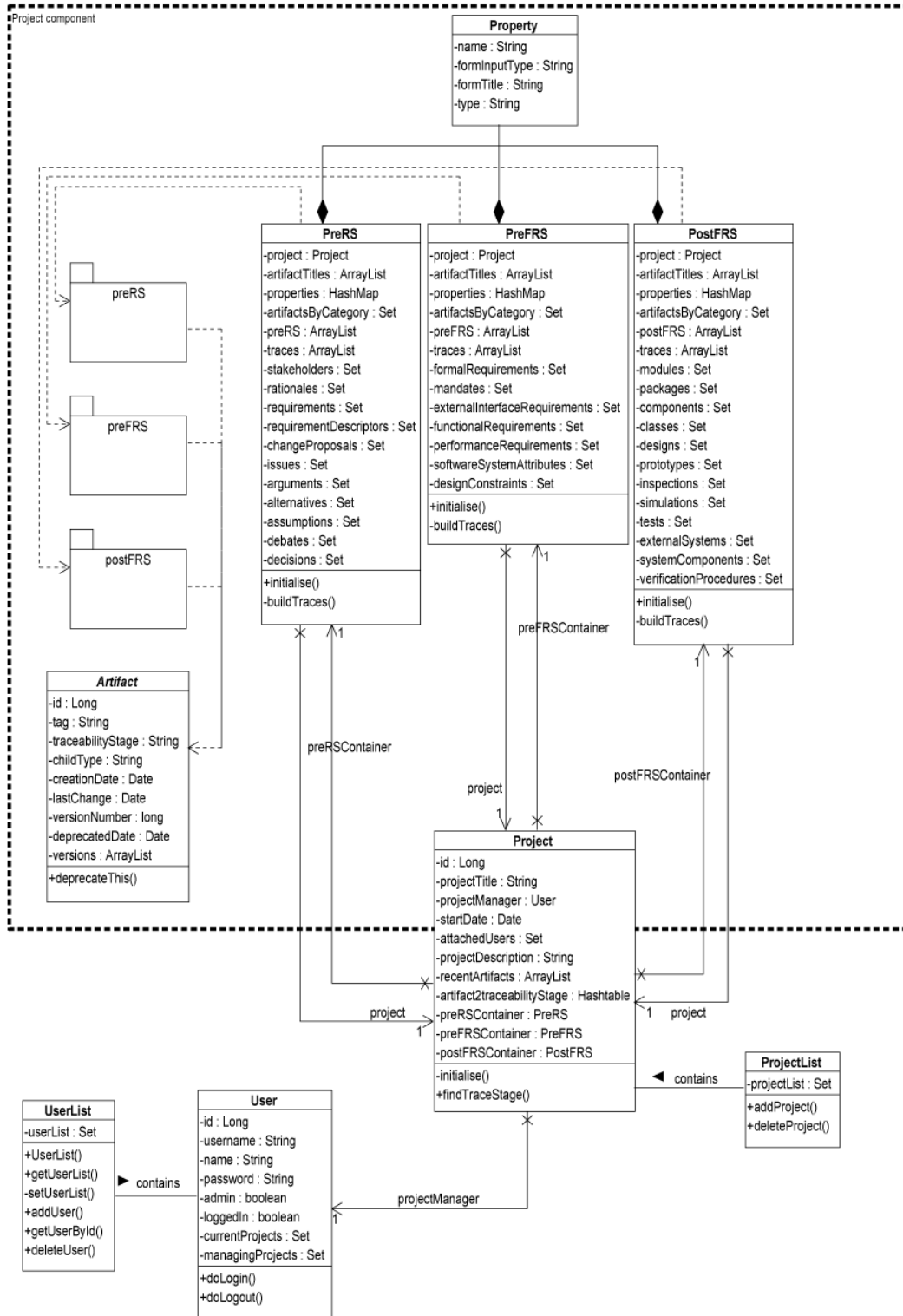


Figure H.2: Prototype design

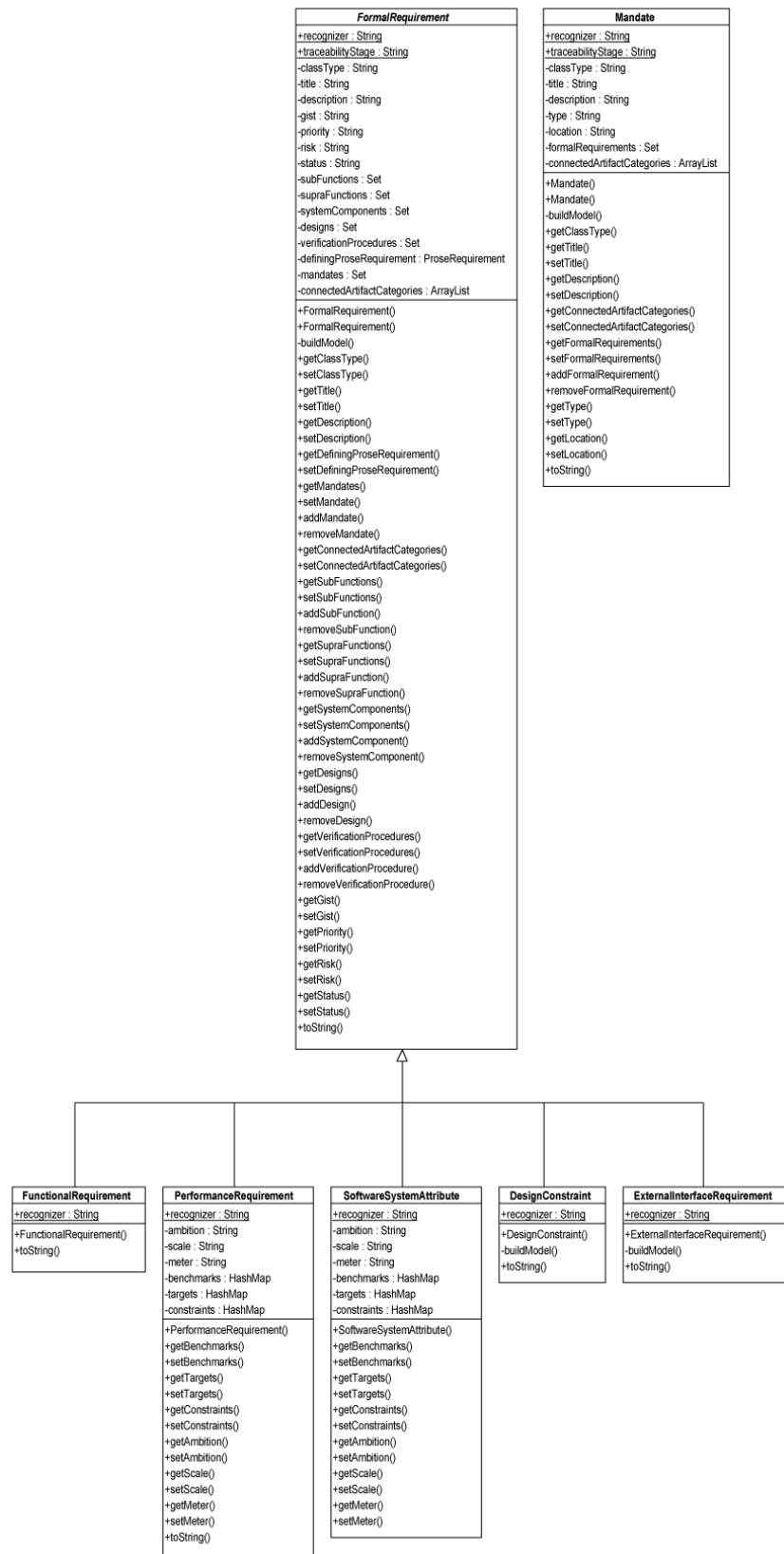


Figure H.4: Detailed specification of the domain object model - pre-FRS

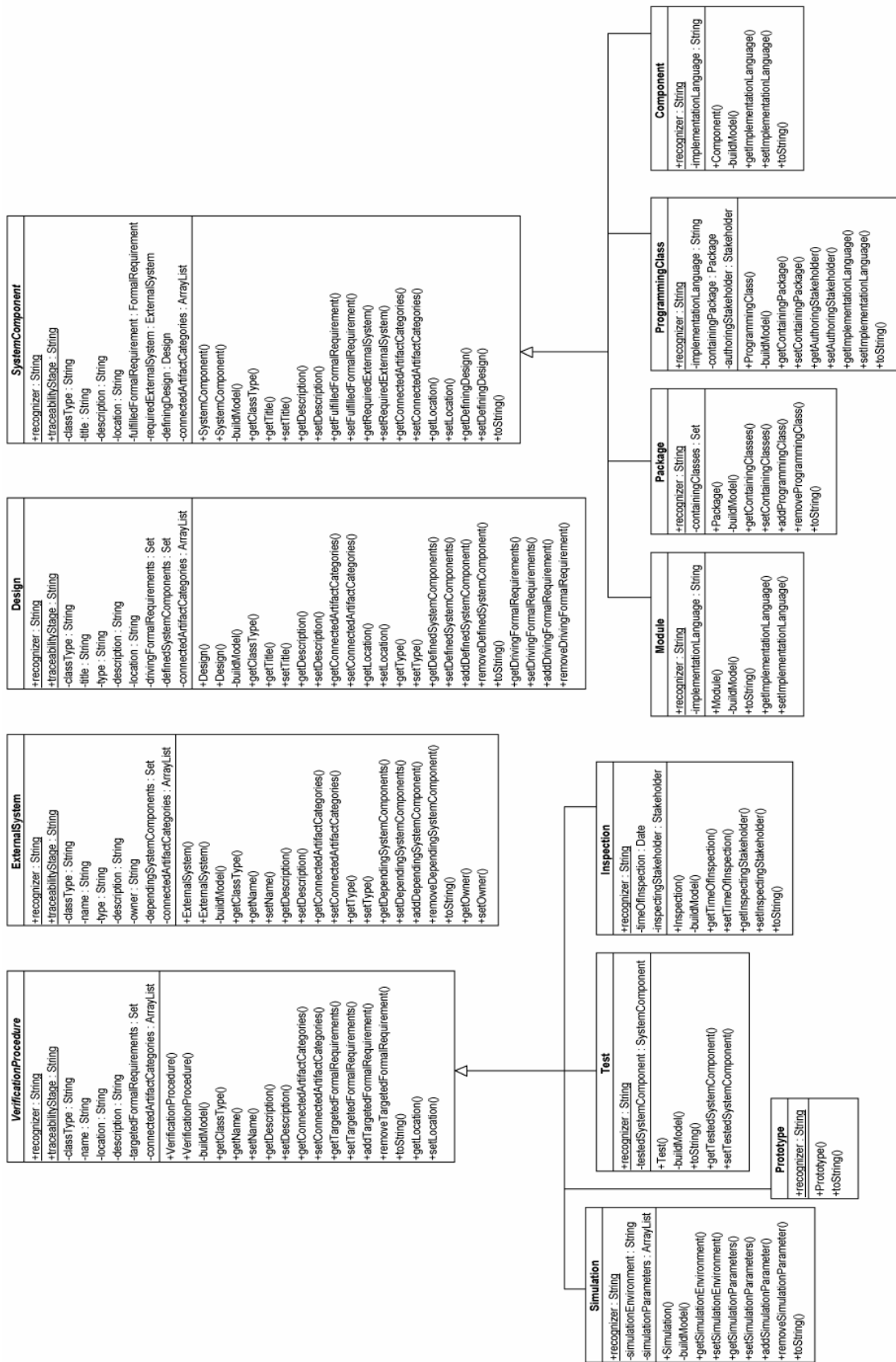


Figure H.5: Detailed specification of the domain object model - post-FRS

to the traceability stage they belong. This is accomplished by the artifact containers. Each `Project` instance contains three containers, one for each traceability stage. The containers are tailored to suit their traceability stage, and contains the sets of artifact instances, and functionality for adding and removing instances. The containers are a part of the underlying model, and keeps a record of the properties of contained artifact categories, as well as existing traces between artifact categories. When adding new artifacts, an implementation of the domain object is required, in addition to specifying the properties to the container, and any traces including the new artifact. The container uses this information to automatically build new views, e.g. forms for creating new artifact instances.

H.2 Prototype installation

The prototype is packaged as a WAR file, and must be deployed to an application server to be accessed. Instructions for installing the Sun Java System Application Server, and then deploying the prototype is given below (only for Windows). Detailed installation guides is provided by Sun Microsystems, and can be found in the same location as the download location.

H.2.1 Downloading the Application Server

The Java EE bundle distributed by Sun Microsystems can be downloaded from

<http://java.sun.com/javaee/downloads/index.jsp>

To deploy and test the prototype, it is only required to download and install the Sun Java System Application Server Platform. Individual download of the components can be found at the bottom of the page, amongst them the application server.

H.2.2 Installing the Application Server

1. Navigate to the directory where you downloaded the .exe file.
2. Double-click the .exe file to start the installation program.
3. Follow the instructions on the wizard screens of the installation program.
4. In the Admin Configuration page (or at the command line), enter the following:
 - Admin User Name - Name of the user who administers the server.
 - Password - Admin user's password to access the Admin Server (8-character minimum).
 - Prompt or Don't Prompt for Administrator User Name - The user name can be stored in a preferences file so that you do not have to provide it to perform administrative tasks.
 - Admin Port - Administration port number for initial server instance.
 - HTTP Port - Port number to access the default server instance.
 - HTTPS Port - Secure port number to access the initial server instance.
5. Enter the directory where you want to install Application Server.
6. In the Installation Options page, select the options that you want.
7. Follow the instructions on the wizard screens of the installation program.
8. On the Ready to Install page choose Install Now.
9. After the installation completes, if you did not select the corresponding option on the Installation Options screen, set the PATH environment variable to include the Application Server install-dir/bin directory.

H.2.3 Starting the Application Server

You may have already started the server when you installed, using the Start Server button on the last installation screen. However, if your server is not running, start it using the steps below.

1. From the Start menu, choose Programs ⇒ Sun Microsystems ⇒ Application Server PE 9 ⇒ Start Default Server.
2. A command window appears showing initialization messages. When the server has started, a message appears stating that the server has been started and is ready to receive requests. The output also includes information on ports used by the Application Server.
3. Press any key to dismiss the command window.

To verify that the server is running on your system, open a web browser and visit the URL listed below.

`http://localhost:8080`

You should now see the server Welcome page.

H.2.4 Deploying the prototype

The thesis includes a ZIP file that contains the distributable prototype, packaged in a WAR file, and the source code of the java libraries employed by the prototype (the implementation of the underlying domain model).

1. Find the directory where the prototype WAR file is stored (named `traceme.war`)
2. Copy `traceme.war` to the `install-dir/domains/domain1/autodeploy/` directory.

The prototype can now be accessed through a web browser by entering the URL listed below.

`http://localhost:8080/traceme/`

A note of caution

The prototype uses Cascading Style Sheets for controlling the layout of the application, i.e., the rendering of the HTML elements of the JSP pages. Web browsers from different vendors interpret the styles of CSS differently, causing the prototype's layout to be rendered differently, depending on the used browser. Consequently, the prototype is optimised for viewing in the Mozilla Firefox web browser [Fir], and should not be viewed in any other browser. The Firefox web browser was chosen because it complies with the standards of CSS as defined by W3C [sta].

Bibliography

- [BCK05a] Len Bass, Paul Clements, and Rick Kazman. Software Architecture in Practice, chapter 2, pages 38–40. Addison Wesley, 2005.
- [BCK05b] Len Bass, Paul Clements, and Rick Kazman. Software Architecture in Practice, chapter 11, pages 271–288. Addison Wesley, 2005.
- [BDS⁺00] Mike Beedle, Martine Devos, Yonat Sharon, Ken Schwaber, and Jeff Sutherland. SCRUM: A pattern language for hyperproductive software development. In Neil Harrison, Brian Foote, and Hans Rohnert, editors, Pattern Languages of Program Design 4, pages 637–652. Addison Wesley, 2000.
- [BYRN99] Ricardo A. Baeza-Yates and Berthier A. Ribeiro-Neto. Modern Information Retrieval. ACM Press / Addison-Wesley, 1999.
- [CK03] William Crawford and Jonathan Kaplan. J2EE Design Patterns, chapter 8, pages 143–152. O’Reilly, 2003.
- [Coc05] Alistair Cockburn. Writing Effective Use Cases. The Agile Software Development Series. Addison Wesley, 2005.
- [CS99] P. Carstensen and K. Schmidt. Computer supported cooperative work: New challenges to systems design, 1999.
- [da] Problem domain analysis.
http://en.wikipedia.org/wiki/problem_domain_analysis - last accessed 26.04.07. Wikipedia.
- [dom] Problem domain.
http://en.wikipedia.org/wiki/problem_domain - last accessed 26.04.07. Wikipedia.
- [EE] Java EE.
<http://java.sun.com/javaee/index.jsp> - last accessed 26.04.07. Sun Microsystems.
- [Env] Integrated Development Environment.
http://en.wikipedia.org/wiki/integrated_development_environment - last accessed 12.04.07. Wikipedia.
- [Fir] Mozilla Firefox.
<http://www.mozilla.com/firefox> - last accessed 25.05.07.
- [For06a] Donelson R. Forsyth. Group Dynamics, chapter 6. Thomson Wadsworth, fourth edition, 2006.
- [For06b] Donelson R. Forsyth. Group Dynamics, chapter 1. Thomson Wadsworth, fourth edition, 2006.
- [For06c] Donelson R. Forsyth. Group Dynamics, chapter 7. Thomson Wadsworth, fourth edition, 2006.
- [Fow03] Martin Fowler. UML Distilled. Addison Wesley, third edition, 2003.
- [Gil05] Tom Gilb. Competitive Engineering - Planguage Concept Glossary, pages 321–438. Elsevier Butterworth Heinemann, 2005.
- [Gog93] J. Goguen. Social issues in requirements engineering, 1993.

BIBLIOGRAPHY

- [Hem06] Anil Hemrajani. Agile Java Development with Spring, Hibernate and Eclipse, chapter 1, page 15. Sams Publishing, 2006.
- [HQ95] Frederick G. Hillmer and James Brian Quinn. Strategic outsourcing. The McKinsey Quarterly, (1), 1995.
- [IEEmla] IEEE. Std 1471-2000: Recommended practice for architectural description of software-intensive systems. "http://standards.ieee.org/reading/ieee/std_public/description/se/1471-2000_desc.html". last accessed April 12th, 2007.
- [IEEmlb] IEEE. Std 830-1998: Recommended practice for software requirements specification. http://standards.ieee.org/reading/ieee/std_public/description/se/830-1998_desc.html. last accessed April 11th, 2007.
- [Jav] JavaDoc.
http://java.sun.com/j2se/javadoc/ - last accessed 26.05.07.
- [Kin04] Gavin King. Hibernate in Action. Manning Publications, 2004.
- [Kru95] Phillippe Kruchten. Architecture blueprints - the "4+1" view model of software architecture. In TRI-Ada '95: Tutorial proceedings on TRI-Ada '91, pages 540–555, New York, NY, USA, 1995. ACM Press.
- [LB03] Rick Kazman Len Bass, Paul Clements. Software Architecture in Practice, chapter 4. Addison Wesley, 2nd edition, 2003.
- [MyS] MySQL.
http://www.mysql.com/ - last accessed 01.05.07.
- [Nor06] Gyrđ Norvoll. Quantification and traceability of requirements, an in-depth study . Technical report, NTNU,
http://folk.ntnu.no/norvoll/forprosjekt_Gyrđ_Norvoll.pdf, 2006.
- [O'K02] Daniel J. O'Keefe. Persuasion - Theory & Research. Sage Publications, second edition, 2002.
- [out03] Global IT Outsourcing: Software Development Across Borders. Cambridge University Press, 2003.
- [RJ00] Linda Rising and Norman S. Janoff. The Scrum software development process for small teams. IEEE Software, 17(4):26–32, /2000.
- [Sch98] Ben Schneiderman. Designing the User Interface - Strategies for Effective Human Computer Interaction. Addison Wesley Longman, third edition, 1998.
- [Shn98a] Ben Shneiderman. Designing the User Interface, chapter 10, page 367. Addison Wesley Longman, Inc., third edition, 1998.
- [Shn98b] Ben Shneiderman. Designing the User Interface - Strategies for Effective Human Computer Interaction, chapter 2, pages 74–76. Addison Wesley Longman, third edition, 1998.
- [SK06] Gerd Melteig Stalheim and Margrethe Adde Kjeøy. Use cases as a tool for communication: An empirical study of the understandability of use cases. Technical report, NTNU, 2006.
- [SPLW02] R. Spears, T. Postmes, M. Lea, and A. Wolbert. When are net effects gross products? the power of influence and the influence of power in computer-mediated communication. Journal of Social Issues, (58):91–107, 2002.
- [sta] CSS standards.
http://www.w3.org/style/css/ - last accessed 25.05.07. W3C.
- [Tec] Java Technology.
http://www.sun.com/java - last accessed 26.04.07. Sun Microsystems.
- [Tom] Apache Tomcat.
http://tomcat.apache.org/ - last accessed 01.05.07.

- [Tuc65] B.W. Tuckman. Developmental sequences in small groups. Psychological Bullentin, (63):384–399, 1965.
- [Vli00a] Hans Van Vliet. Software Engineering - Principles and Practice, chapter 3.1, pages 49–50. Wiley, 2nd edition, 2000.
- [Vli00b] Hans Van Vliet. Software Engineering - Principles and Practice, chapter 3.1, pages 52–56. Wiley, 2nd edition, 2000.
- [W3C04] W3C. Web services architecture. W3C Working Group Note, 2004.
- [WRH⁺00] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. Experimentation in Software Engineering - An Introduction. Kluwer Academic Publishers, 2000.