Quality Goal Oriented Architectural Design and Traceability for Evolvable Software Systems

Dissertation

zur Erlangung des akademischen Grades

Doktor-Ingenieur (Dr.-Ing.)

vorgelegt der

Fakultät für Informatik und Automatisierung

der Technischen Universität Ilmenau

von

Dipl.-Inf. Stephan Bode

geboren am 5. Februar 1983

in Mühlhausen/Thüringen, Deutschland

1. Gutachter: PD Dr.-Ing. habil. Matthias Riebisch, TU Ilmenau
2. Gutachter: Prof. Dr.-Ing. Detlef Streitferdt (JP), TU Ilmenau
3. Gutachter: Prof. Dr. Wilhelm Hasselbring, Universität zu Kiel

Datum der Einreichung: 2011/04/20

Datum der Verteidigung: 2011/09/22

urn:nbn:de:gbv:ilm1-2011000313

Zusammenfassung

Softwaresysteme werden heute z. B. aufgrund sich ändernder Geschäftsprozesse oder Technologien mit häufigen Veränderungen konfrontiert. Die Software und speziell ihre Architektur muss diese Änderungen zur dauerhaften Nutzbarkeit ermöglichen.

Während der Software-Evolution können Änderungen zu einer Verschlechterung der Architektur führen, der Architekturerosion. Dies erschwert oder verhindert weitere Änderungen wegen Inkonsistenz oder fehlendem Programmverstehen. Zur Erosionsvermeidung müssen Qualitätsziele wie Weiterentwickelbarkeit, Performanz oder Usability sowie die Nachvollziehbarkeit von Architekturentwurfsentscheidungen berücksichtigt werden. Dies wird jedoch oft vernachlässigt.

Existierende Entwurfsmethoden unterstützen den Übergang von Qualitätzielen zu geeigneten Architekturlösungen nur unzureichend aufgrund einer Lücke zwischen Methoden des Requirements Engineering und des Architekturentwurfs. Insbesondere gilt dies für Weiterentwickelbarkeit und die Nachvollziehbarkeit von Entwurfsentscheidungen durch explizite Modellabhängigkeiten.

Diese Arbeit präsentiert ein neues Konzept, genannt Goal Solution Scheme, das Qualitätsziele über Architekturprinzipien auf Lösungsinstrumente durch explizite Abhängigkeiten abbildet. Es hilft somit, Architekturlösungen entsprechend ihrem Einfluss auf Qualitätsziele auszuwählen. Das Schema wird speziell hinsichtlich Weiterentwickelbarkeit diskutiert und ist in ein zielorientiertes Vorgehen eingebettet, das etablierte Methoden und Konzepte des Requirements Engineering und Architekturentwurfs verbessert und integriert. Dies wird ergänzt durch ein Traceability-Konzept, welches einen regelbasierten Ansatz mit Techniken des Information Retrieval verbindet. Dies ermöglicht eine (halb-) automatische Erstellung von Traceability Links mit spezifischen Linktypen und Attributen für eine reichhaltige Semantik sowie mit hoher Genauigkeit und Trefferquote.

Die Realisierbarkeit des Ansatzes wird an einer Fallstudie einer Software für mobile Serviceroboter gezeigt. Das Werkzeug EMFTrace wurde als eine erweiterbare Plattform basierend auf Eclipse-Technologie implementiert, um die Anwendbarkeit der Konzepte zu zeigen. Es integriert Entwurfsmodelle von externen CASE-Tools mittels XML-Technologie in einem gemeinsamen Modell-Repository, wendet Regeln zur Linkerstellung an und bietet Validierungsfunktionen für Regeln und Links.

Abstract

Today software systems are frequently faced with demands for changes, for example, due to changing business processes or technologies. The software and especially its architecture has to cope with those frequent changes to permanently remain usable.

During software evolution changes can lead to a deterioration of the structure of software architectures called architectural erosion, which hampers or even inhibits further changes because of inconsistencies or lacking program comprehension. To support changes and avoid erosion, especially quality goals, such as evolvability, performance, or usability, and the traceability of design decisions have to be considered during architectural design. This however often is neglected.

Existing design methods do not sufficiently support the transition from the quality goals to appropriate architectural solutions because there is still a gap between requirements engineering and architectural design methods. Particularly support is lacking for the goal evolvability and for the traceability of design decisions by explicit model dependencies.

This thesis presents a new concept called Goal Solution Scheme, which provides a mapping from goals via architectural principles to solution instruments by explicit dependencies. Thus it helps to select appropriate architectural solutions according to their influence on quality goals. The scheme is discussed especially regarding evolvability, and it is embedded in a goal-oriented architectural design method, which enhances and integrates established methods and concepts from requirements engineering as well as architectural design. This is supplemented by a traceability concept, which combines a rule-based approach with information retrieval techniques for a (semi-) automated establishment of links with specific link types and attributes for rich semantics and a high precision and recall.

The feasibility of the design approach has been evaluated in a case study of a software platform for mobile robots. A prototype tool suite called EMFTrace was implemented as an extensible platform based on Eclipse technology to show the practicability of the thesis' concept. It integrates design models from external CASE tools in a joint model repository by means of XML technology, applies rules for link establishment, and provides validation capabilities for rules and links.

Danksagung

An dieser Stelle möchte ich die Gelegenheit nutzen, um all denjenigen meinen Dank auszudrücken, die mich während meines Promotionsvorhabens begleitet haben und ohne die diese Dissertation nicht geschrieben worden wäre.

Der größte Dank gebührt meinem Doktorvater, Matthias Riebisch, der mich ermutigt und mir die Möglichkeit gegeben hat unter seiner Betreuung zu promovieren. Er hatte immer einen ergiebigen Rat auf fachliche und persönliche Fragen und hat mich die ganze Zeit aufschlussreich geleitet.

Weiterer Dank geht an Professor Armin Zimmermann und Professor Wilhelm Hasselbring für ihre Kommentare und für die Begutachtung meiner Arbeit.

Ich möchte auch Professor Ilka Philippow danken für die Unterstützung durch sie und ihr Fachgebiet an der Technischen Universität Ilmenau. In diesem Zusammenhang gehört auch all meinen Kollegen für ihre Unterstützung besonderer Dank, insbesondere Heiner für seine Mühe bei all meinen technischen Wünschen und für seine aufmunternden Bemerkungen.

Verschiedene Studenten haben an unterschiedlichen Aspekten in Bezug auf mein Dissertationsthema und den Prototypen gearbeitet und dazu beigetragen. Vielen Dank an Steffen Lehnert für sein außerordentliches Engagement für das EMFTrace Projekt. Ich möchte auch Matthias Rumpf, Ralf Stollberg, Philipp Wagner und Daniel Motschmann für ihre Arbeit danken.

Weiterhin, bedanke ich mich bei Professor Horst-Michael Groß und seinen Mitarbeitern für die Roboterfallstudie zur Evaluierung meiner Arbeit. Insbesondere bedanke ich mich bei Andrea Scheidig, Erik Einhorn und Jens Kessler für ihre geopferte Zeit für Diskussionen.

Darüber hinaus gebührt mein aufrichtiger Danke meiner Familie für die enorme Unterstützung und Geduld während meines gesamten Studiums.

Acknowledgment

At this place I would like to take the opportunity to express my thanks to all the people who accompanied me during my Ph.D. project and without whom this dissertation would not have been written.

The biggest thanks goes to my advisor, Matthias Riebisch, for encouraging me and giving me the opportunity to do my Ph.D. under his supervision. He always had a fruitful advice for professional and personal questions and insightfully guided me during the whole time.

Further thanks go to professor Armin Zimmermann and to professor Wilhelm Hasselbring for their comments and for examining my work.

I also would like to thank professor Ilka Philippow for her support through her department at the Ilmenau University of Technology. In this regard special thanks also belong to my colleagues for all their support and especially to Heiner for his effort with all my wishes regarding technology and for his encouraging remarks.

Several students worked on different aspects related to my work and contributed to the approach and the prototype tool. Many thanks go to Steffen Lehnert for his extraordinary dedication to the EMFTrace project. I also would like to thank Matthias Rumpf, Ralf Stollberg, Philipp Wagner and Daniel Motschmann for their work.

Furthermore, I thank professor Horst-Michael Groß and his staff for providing the case study of the robot software platform for the evaluation of my approach. In particular I thank Andrea Scheidig, Erik Einhorn and Jens Kessler for their time for discussions.

Moreover, I sincerely thank my family for the tremendous support and patience during my whole studies.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Goals of the Thesis . . . 3
1.3 Scope . . . 5
1.4 Contribution . . . 6
1.5 Outline of the Thesis . . . 7

2 Evaluation of State-of-the-Art Methods and Concepts **11**
2.1 Description of Functional and Quality Requirements . . . 11
2.1.1 Goal-Oriented Approaches . . . 12
2.1.1.1 NFR Framework . . . 12
2.1.1.2 i* (i-star) . . . 14
2.1.1.3 GRL and URN . . . 15
2.1.1.4 Evaluation of the Goal-Oriented Approaches . . . 16
2.1.2 Use Cases and Scenarios . . . 17
2.1.3 The EMPRESS Framework for Requirements . . . 19
2.2 Evolvability Support by Architectural Design Methods . . . 21
2.2.1 QASAR . . . 22
2.2.2 Attribute-Driven Design (ADD) . . . 24
2.2.3 The Siemens Four Views Approach . . . 25
2.2.4 EMPRESS Pattern-Based Architectural Analysis and Design . 27
2.2.5 Quasar . . . 29
2.2.6 Evaluation and Relation to Further Works . . . 30
2.3 Design Traceability . . . 32
2.3.1 Origin, Definition, and Benefits of Traceability . . . 33
2.3.2 Classification of Traceability . . . 34

2.3.3 Traceability Schemes and Metamodels . . . 36
2.3.4 (Semi-)Automated Traceability Approaches . . . 37
2.3.4.1 Information Retrieval-based Approaches . . . 37
2.3.4.2 Rule-based Traceability . . . 39
2.3.5 Tool Support . . . 41
2.3.6 Evaluation and Challenges . . . 42

3 Evolvability and Related Terms 45
3.1 Maintenance . . . 45
3.2 Evolution . . . 46
3.3 Maintainability . . . 48
3.4 Evolvability . . . 49

4 Overview of the Approach 53
4.1 Refinement of the Goals of this Thesis . . . 53
4.1.1 Summary of the Benefits and Limitations of Existing Approaches 54
4.1.2 Research Goals for the Proposed Approach . . . 55
4.2 Proposed Approach . . . 56

5 The Goal Solution Scheme 61
5.1 General Concept . . . 61
5.1.1 Structure of the GSS . . . 62
5.1.2 Transitions of the GSS . . . 63
5.1.3 Contribution of the GSS . . . 64
5.2 Establishment of the GSS . . . 65
5.2.1 Quality Goal Refinement and Elaboration . . . 66
5.2.2 Filling the GSS with Principles . . . 69
5.2.3 Filling the GSS with Solution Instruments . . . 72
5.3 Establishment of the GSS for Evolvability . . . 74
5.3.1 Evolvability Model . . . 75
5.3.1.1 Refinement of Evolvability to Subcharacteristics . . . 75
5.3.1.2 Relating Design Principles to Evolvability Subcharacteristics . . . 77
5.3.2 Evaluation of Architectural Solution Instruments . . . 81
5.3.2.1 The Set of Evaluated Patterns . . . 82

5.3.2.2 Determination of the Impact on the Principles 83
5.3.2.3 Calculation of the Impact on Evolvability Subcharacteristics 87
5.3.2.4 Determining the Impact on Evolvability 88
5.3.3 Discussion of the Results 88

6 Goal-Oriented Architectural Design 93
6.1 Introduction of the Case Study 93
6.2 The Goal-Oriented Design Method (GOAD) 95
6.2.1 Requirements and Architectural Analysis 97
6.2.1.1 Quality Goal Elaboration and Refinement 97
6.2.1.2 Goal Prioritization 100
6.2.1.3 Scenario Description 103
6.2.1.4 Global Analysis 103
6.2.2 Architectural Synthesis 108
6.2.2.1 Structuring the Architecture 109
6.2.2.2 Top-down vs. Bottom-up Structuring 117
6.2.2.3 Detailing the Architecture 118
6.2.2.4 Conflict Resolution and Trade-offs 120
6.2.3 Architectural Evaluation 121

7 The Traceability Concept 123
7.1 Models, Model Elements, Dependencies, and Traceability Links . . . 123
7.2 Overview and Classification of the Concept 125
7.3 Traceability Metamodel and Hypertext Concept 127
7.3.1 The Traceability Link Metamodel 127
7.3.2 The Hypertext Concept 129
7.4 Traceability Link Types 131
7.4.1 The Link Type Clusters 132
7.4.2 Application and Utilization of the Link Types 136
7.5 Traceability Rules 137
7.6 Ontology Definition 140

8 Tool Support by EMFTrace 143
8.1 Requirements and Core Concept 143

8.2 Architecture . . . 146
8.2.1 EMFStore . . . 146
8.2.2 EMFTrace . . . 147
8.2.3 EMFfit . . . 151
8.3 Model Integration . . . 154
8.4 Usage Scenario . . . 156

9 Evaluation of the Approach 161
9.1 Way of Evaluation . . . 161
9.2 Evaluation of GSS and GOAD . . . 163
9.3 Evaluation of the Traceability Concept . . . 167
9.4 Limitations of the Approach of this Thesis . . . 171

10 Conclusions and Outlook 173
10.1 Contributions . . . 173
10.2 Future Work . . . 176

A Case Study Artifacts 179
A.1 Factor Tables . . . 179
A.2 IssueCards . . . 194
A.3 UML Diagrams . . . 200

B Traceability Artifacts 205
B.1 Traceability Rule Catalog . . . 205
B.2 Ontology for the Traceability Approach . . . 223

Chapter 1

Introduction

"Nothing endures but change."
Heraclitus of Ephesus (c. 535 – c. 475 BC)

1.1 Motivation

Today software systems usually are major investments for companies, and they have a high business value. Furthermore, adaptation to the market is an essential quality in today's business, and software systems have to remain usable for a long time when they are built once. Moreover, the software systems are faced with frequent requests for changes due to necessary adaptations to the market, the optimization of processes, new technologies, or because of the integration of existing systems in the development of new architectures. The displacement of existing software systems, however, is generally not applicable because of the risks involved. For this reason, the software and especially its architecture has to cope with frequent requests for changes to permanently remain usable.

Software changes often have to be implemented with low effort and in short time frames during software maintenance. Therefore, these changes can lead to a deterioration in the structure of the software, which hampers or even inhibits further changes. This effect is called architectural decay, drift, or erosion [RSN09]. Replacing an affected system with a completely new developed one to prevent architectural decay usually is not an option because of time or budget constraints and to avoid risks.

The difficulties that arise during software maintenance or evolution, as for ex-

ample architectural decay, program comprehension, or lacking traceability due to scarce documentation, lead to several challenges for today's software development:

C1 The fact of frequent changes is a constant issue that has to be supported in all development phases.

C2 Software architectures have to be evolvable in a controlled way without violating integrity. Further, they should be partitioned into independently evolving subsystems. [BR00]

C3 Program understanding has to be supported, for example, by impact analysis, and unnecessary complexity should be removed through restructuring. [BR00]

C4 A better understanding is necessary of the influence of architectural design decisions and choices on the prioritization and evolution of current and future requirements and their satisfaction, especially for quality requirements. [NE00]

C5 For tracing design decisions a mapping between requirements and design as well as between design and code has to be established [MJS+00], which inevitably leads to the request for the implementation of suitable traceability techniques.

C6 With the aging of the software [Par94] its quality properties as, for example, traceability or maintainability have to be preserved or even improved. [MWD+05]

In this regard a demand beyond software maintenance arises to keep the software and its architecture in a state that allows quick and easy changes for the long-term. The quality property evolvability denotes this state. Maintenance and maintainability—in contrast to evolvability—are closely related to the term legacy system [BDP06]. Legacy systems are old software systems, the structure of which cannot be developed further and which resist quick and easy changes in consequence of many changes and progressive architectural decay. High evolvability is the only way to avoid turning a software system into a legacy system [MM98]. However, the effort, a strategic procedure, and the complexity for software evolution are considerably more demanding than for software maintenance.

These are all arguments to differentiate between software maintenance and software evolution (see also Chapter 3). From a perspective of long-term development

evolvability rather aims at introducing new features than at correcting errors. In this regard a differentiation between evolution and maintenance is possible according to development activities. For example, cyclic implementations or the architectural evaluation of new features typically have to be assigned to evolution and not to maintenance. Maintenance activities are better characterized as patchwork to keep a legacy-system operating. Furthermore, maintenance starts with the deliverance of the software to the customer, whereas evolution usually refers to the whole life cycle of a software system.

Software systems mostly are exceedingly complex by nature. This is one reason why software architectures are necessary in large projects to coordinate development and to assure that quality goals of the development are met. Unfortunately, during architectural development quality aspects, as for example performance, security, usability, or maintainability, far too often are considered insufficiently, although they are critical for success. Quality requirements (often called non-functional requirements (NFR)) as evolvability are at least as, if not even more, important as functional requirements during architectural design. Their realization has to be integrated in the software development process [FL93]. Quality requirements, especially evolvability, however, are often intangible, incomplete, and conflicting, hence, often realized insufficiently because of lacking methodical support [Eur07, CdPL09]. Therefore, appropriate architectural design methods and tools are necessary. Moreover, trade-offs between various quality aspects have to be made explicit and usable so that they can be considered in the development process itself and during evolutionary changes.

1.2 Goals of the Thesis

This thesis is targeted on one particular phase of software development—the architectural design phase. This phase starts for a new software system to be developed with the transition from specified requirements to the software architecture. For a definition of software architecture and related terms see IEEE Std 1471-2000 [Ins00].

Based on the challenges for today's software design the overall goal of this thesis is to *provide appropriate means for an architectural design method that allows to develop evolvable software by especially considering quality goals and supporting software changes.*

These means and procedure shall enable to systematically realize evolvability in a software's architecture akin as it is usual and desirable for other quality goals as for instance security. To realize this, a transformation of the quality goals and requirements into functional concepts and solutions is necessary.

Unfortunately, the systematical support for the above mentioned transformation and the means to develop evolvable software are insufficient today. Consolidated technical solutions for quality requirements, especially for evolvability are lacking. Moreover, there is still a gap between requirements engineering and architectural design methods [GEM06] that reduces the applicability of detailed design instructions.

This thesis shall overcome these limitations with a structured, systematical architectural design method. Therefore, research goals are to:

G1 Evaluate existing analysis and design methods regarding their support of a systematical design for quality goals and especially evolvability.

G2 Establish suitable means and procedures for a systematical realization of quality goals during architectural design.

Next to evolvability there are of course other important quality goals that have to be considered as requirements during architectural design. Some of the quality goals are in conflict with each other or influence one another. Therefore, this thesis shall:

G3 Examine how the intangible quality goal evolvability is related to other quality goals and how it can be refined to ease understanding and realization in architectural design.

G4 Find and possibly reuse appropriate means to model and utilize such a refinement during architectural design.

G5 Examine how goal influences can be represented and how goal conflicts can be resolved, for example, by so-called trade-offs.

The systematical application of well-known design principles and solution instruments, such as architectural styles, patterns, or frameworks, can help with these trade-off decisions. For this purpose further goals are to:

G6 Determine the influence of well-known design principles on evolvability and related goals.

G7 Analyze solution instruments according to their support for evolvability.

G8 Find a possibility to express these influences and establish a catalog for the selection of suitable solution instruments to conserve the knowledge for further use in specific software projects.

During architectural design the knowledge about dependencies between different models is important for the realization of changes. The same holds for the knowledge about design decisions that are made, when solutions are assigned to goals and goal conflicts are resolved. The concept of traceability, to explicitly model dependencies as links between software artifacts or models during development, promises advantages for impact analysis of changes especially if traceability links are established with a certain degree of granularity and consistency. However, this leads to a high effort, especially for quality requirements, which usually are related to a multitude of design elements. Thus, another goal of this thesis is to:

G9 Develop an appropriate concept for the implementation of traceability that limits itself to a justifiable effort for link establishment and maintenance and integrate it with the other architectural means discussed above.

1.3 Scope

The scope of this thesis is on the architectural design of business critical software systems that demand for a long lifetime. Such software systems can be, for example, business information systems. Embedded systems are not especially considered although the developed concepts might be applied or adopted for them.

The focus of this thesis is further on the structuring of the software architecture of a system and on the transition from requirements to design. Although some requirements engineering approaches are discussed and influenced the work, this thesis is not about elicitation and specification of functional and non-functional requirements. Instead, it is about a methodical way for their realization in software architectures.

Furthermore, for the design-time quality goal evolvability no formal techniques for specification are considered. This is due to its rather intangible nature, which is reluctant to formalization. However, for the execution-time quality goal performance, for example, this would be imaginable. For the same reason the architectural design method in this thesis is not concerned with model-driven design techniques that rely on automated transformations and the generation of a solution. But the method is model-based as it relies on the explicit modeling of the design artifacts and their dependencies.

Moreover, the thesis concentrates on the analysis and synthesis part of designing the architecture from a forward engineering perspective. The reengineering perspective and a sophisticated architectural evaluation method especially for evolvability based on appropriate metrics and assessment techniques are left out of scope. A methodical approach with a reengineering perspective, which is based on metrics for architectural evaluation, is the one of Brcina [Brc11].

1.4 Contribution

The contribution of this thesis regarding the overall goal is threefold. As the first and most significant contribution of this thesis, a concept called Goal Solution Scheme is introduced for a systematical guidance during architectural design to realize quality goals and especially evolvability. The scheme combines ideas from goal-oriented requirements engineering with architectural design principles to ease the transition from the goals and requirements to architectural design artifacts. An innovation of the scheme is the explicit modeling of the dependencies between goals, principles, as well as solution instruments. Furthermore, it helps to quantitatively evaluate the solution instruments, such as design patterns, regarding their influence on quality goals to support the selection of the solution instruments. In this way a scheme for evolvability is established, which provides a consolidated view on evolvability and a catalog of solution instruments with an evaluated impact on this goal.

Secondly, a goal-oriented architectural design method is described. It is based on established methods and concepts. As a contribution the method targets on the development of evolvable software systems by especially considering the quality goals. For this reason existing methods and concepts are enhanced, combined, and integrated. The novelty of the design method is the integration of goal modeling

with architectural design methods such as Global Analysis and the Attribute-Driven Design method (ADD). Moreover, it utilizes the Goal Solution Scheme for the resolution of goal conflicts and a systematic selection of architectural solutions.

Thirdly, as a major contribution a traceability concept for the whole design method is described to explicitly represent model dependencies with traceability links for further analyses. The traceability concept spans the different artifacts from goal models over analysis artifacts to design models. It integrates all models in a repository. As an important feature it uses standard modeling languages, such as the Unified Modeling Language (UML), as far as possible. This enables the coupling to established CASE tools. For a justifiable effort the traceability links are established in a (semi-) automated way by applying rules. As a novelty, these rules are combined with n-gram matching as an information retrieval technique. This approach results in higher precision and recall than purely retrieval based approaches. Simultaneously, through the rules it allows to attach types to the links that provide a richer semantics of the links. A further innovation is the usage of an ontology to facilitate more explicit dependencies between the design models. The complete traceability concept is implemented with a tool called EMFTrace together with a tool called EMFfit supporting Global Analysis activities.

1.5 Outline of the Thesis

The reminder of the thesis is structured as follows.

Chapter 2: Evaluation of State-of-the-Art Methods and Concepts In this chapter relevant methods and concepts from the state-of-the-art of requirements engineering, architectural design, as well as traceability are evaluated regarding their suitability for the goals of this thesis. This provides the foundation for the refinement of the goals and for the proposed approach as discussed in Chapter 4.

Chapter 3: Evolvability and Related Terms This chapter discusses important terms regarding evolvability based on existing definitions in the literature. It explains the meaning of the rather complex quality goal evolvability in comparison to maintainability, maintenance, and evolution. For other quality goals this is not necessary because they are already standardized and sufficiently discussed in the

literature.

Chapter 4: Overview of the Approach This chapter provides an overview of the approach of this thesis. First, the goals of the thesis are refined according to the results from the evaluation of the state-of-the-art. Then, an introduction to the proposed approach that shall overcome the existing limitations is given.

Chapter 5: The Goal Solution Scheme In this chapter the main idea of the Goal Solution Scheme for a systematic transition from the quality goals to the architectural solutions is explained. First, the concept of the Goal Solution Scheme is presented in general for several quality goals. Afterwards, it is specifically applied for evolvability.

Chapter 6: Goal-Oriented Architectural Design This chapter explains in detail the goal-oriented architectural design method. It explains how the Goal Solution Scheme is utilized together with existing approaches evaluated from the state-of-the-art and how they are combined for an advanced treatment of quality goals during architectural analysis and synthesis.

Chapter 7: The Traceability Concept This chapter describes a concept for the establishment of traceability links during the application of the beforehand developed architectural design method to support evolvability. The concept supports the (semi-) automated establishment of links between the different models spanning the development phases of the method based on a defined ruleset.

Chapter 8: Tool Support by EMFTrace This chapter describes a prototype software system called EMFTrace, which implements the approach introduced in this thesis. Information is provided about the system's architecture, its components and features. EMFTrace is complemented with a tool called EMFfit, which supports the activities of architectural analysis.

Chapter 9: Evaluation of the Approach For proof of feasibility of the approach, which is introduced in this thesis, a case study was applied in an evolution scenario of a real world software system. The chapter reports on the applicability

of the approach for the case study, evaluates the approach, and discusses its limitations. Moreover, in this chapter the quality of the traceability link establishment is determined by means of precision and recall.

Chapter 10: Conclusions and Outlook The final chapter gives a summary on the results of this thesis and clearly states the contribution. Furthermore, it discusses possible directions for future work.

Appendix A: Case Study Artifacts This appendix lists additional design artifacts of the case study that cannot be presented completely in Chapter 6.

Appendix B: Traceability Artifacts This appendix lists artifacts from the implementation of the traceability approach, such as the XML Schema Definition for traceability rules and the rules themselves.

Chapter 2

Evaluation of State-of-the-Art Methods and Concepts

This section describes state-of-the-art methods and concepts and their evaluation regarding their support for the main goal of this thesis to provide a quality goal oriented architectural design method for evolvable software. Because this is a broad scope of research ranging from requirements engineering to architectural design and evaluation, only those existing works are discussed that are closely related and mainly influenced or were integrated into this thesis.

Approaches that deal with the description and specification of functional and especially quality requirements are analyzed in Section 2.1 because they are needed for the transition from the requirements to the design. Well-known architectural design methods are discussed regarding their contribution to the thesis' goal in Section 2.2 because they shall be integrated in the goal-oriented design method. Finally, the current state-of-the-art of traceability is evaluated regarding the traceability of design artifacts in Section 2.3 because this is important for the evolvability of the architectural design.

2.1 Description of Functional and Quality Requirements

In requirements engineering there are several approaches that deal with functional requirements and quality requirements or non-functional requirements as they are often called. A comprehensive introduction to requirements engineering, for example,

can be found in [vL09]. Here only a limited number of approaches that are directly related to the approach of this thesis are discussed. Although in requirements engineering a classification of requirements into functional and non-functional ones does exist, most available approaches mainly deal with functional requirements and lack an appropriate treatment of quality requirements [CdPL09]—which by the way is the same for the architectural design approaches. In this section the scope is mainly on the specification of quality requirements. It will be analyzed, how goal-oriented and scenario-based techniques and concepts can contribute to the specification of quality goals and in particular evolvability and how the approaches can be integrated into architectural design to reduce the gap between requirements engineering and architectural design.

2.1.1 Goal-Oriented Approaches

The works on goal-oriented requirements engineering (GORE) have a key strength in an appropriate treatment of non-functional requirements or quality goals. A good overview and recent presentation of the state-of-the-art on this topic is given in [CdPL09]. The GORE approaches are used to elicit and model requirements, whereas they deal with functional and quality requirements in parallel. Important representatives are the NFR framework of Chung et al. [CNYM00], the i* framework of Yu [Yu95] and the Goal-oriented Requirements Language (GRL) [GRL08], which was standardized as part of the User Requirements Notation (URN) [Amy03]. Such GORE approaches are quite popular in trying to bridge the gap between requirements and architecture [GEM06]. In the following they are examined concerning their suitability for the refinement and modeling of the quality goal evolvability and concerning their integrability with architectural design.

2.1.1.1 NFR Framework

The non-functional requirements (NFR) framework of Chung et al. [CNYM00] represents one of the GORE approaches that deal with modeling quality requirements. The framework uses a concept of so-called softgoals to represent quality requirements. Softgoals are goals without a clear definition and criteria for their fulfillment as for example maintainability or evolvability. Design decisions on solutions often only partially contribute to the satisfaction of such goals within acceptable lim-

its or even have a negative influence. Therefore, softgoals are seen as fulfilled—or "satisficed"—if an acceptable degree of the goal's criterion is achieved. The interdependencies between softgoals are categorized according to the influence on each other. The influence can range from strongly positive (++) or weakly positive (+) over unknown (?) to weakly negative (−) or strongly negative (−−). This is expressed by the corresponding contribution types *make*, *help*, *unknown*, *hurt*, and *break*, respectively.

The NFR framework describes some interlaced and iterative activities to arrange the softgoals and their interdependencies in a so-called Softgoal Interdependency Graph (SIG). First, the softgoals are established. Then, they can be refined by AND or OR decomposition into subgoals. They also can be connected by contribution links. As a next step architectural solutions are established and mapped to the softgoals as so-called operationalizations. Additionally, the softgoals can be refined by argumentations that correspond to domain-specific characteristics or expert knowledge. Moreover, during the refinement there is the opportunity to consider the priority of certain goals by annotating a criticality statement. Finally, solutions for the softgoals are selected and their contribution to the goals is evaluated bottom-up along the contribution links.

Evaluation By performing the activities of the NFR framework quality goals can be refined to ease their comprehension and realization. Goal conflicts can be elaborated in a well-grounded way. Implicit dependencies between the softgoals can be detected and documented together with rationale for design decisions. Alternative solutions can be evaluated regarding their contribution to the quality goals. Finally, suitable solutions can be chosen. A resolution of conflicts is enabled by prioritization. In this regard the NFR framework can help to design software architectures that explicitly consider quality goals. During refinement of quality goals into subgoals existing quality models or the catalogs provided by the NFR framework itself can help. However, Chung et al. concentrate only on certain quality goals like performance, accuracy, and security. Unfortunately, less tangible, "softer" goals as maintainability or evolvability, the specification and formalization of which is even harder, are not considered. Furthermore, since it is a requirements engineering approach, the authors do not consider architectural design principles during the refinement into operationalizations. These principles, however, provide impor-

tant hints for appropriate solutions during architectural design. Besides, the NFR approach does not consider architectural constraints that result from the selection of certain technical solutions. The reference to the technical realization is lacking. Moreover, the traceability of chosen solutions is not explicitly considered. Indeed alternative solutions are mapped to goals via contribution links. However, there is no possibility to ensure the traceability between chosen solutions of the goal graph to corresponding solutions modeled in architectural design artifacts.

2.1.1.2 i* (i-star)

The i* framework of Yu [Yu95] is another GORE approach very similar to the NFR framework. It is a comprehensive framework for modeling the organizational context and the rationale that led to the various requirements for a software system. The framework knows two types of models for different abstraction levels—the Strategic Dependency (SD) model and the Strategic Rationale (SR) model. SD models describe the intention of a process and the relations between different actors, which cooperate to fulfill certain goals. SR models describe the rationale behind it. They contain goals, softgoals, tasks, as well as resources that are of importance for an actor. An SR graph represents the interdependencies between these elements using different types of links: *contribution* for positive or negative influence, *decomposition* for refinement, and *means-end* for characterizing the fulfillment of a goal. Decomposition and means-end correspond to AND and OR decomposition of the NFR framework. Additionally, i* supports the selection of alternatives by an evaluation process that uses the contribution links similar to the NFR framework [Hor06].

Evaluation Akin to the NFR framework the i* notation uses goals and softgoals to represent functional and non-functionals requirements. It equally allows to model the interdependencies between goals using contribution links. Additionally, i* provides the model elements actor, resource, and means-end links. Moreover, in the SD model it allows to describe the context of the quality aspects by modeling the dependencies between goals and different stakeholders. i* alike the NFR framework allows to refine quality goals and to examine alternative solutions for the goals. Not included in i* is the concept of criticality for prioritization, which has a negative impact on the potential to resolve conflicts. However, all in all i* is richer in its expressiveness because it does not only consider system requirements but also can

represent the organizational context. Anyhow, as i* is also a concept from requirements engineering, it has the same disadvantages regarding architectural design as the NFR framework. Architectural design principles and patterns are not directly considered when exploring the solution space. Further, there is also no concept for traceability between i* models and architectural artifacts.

2.1.1.3 GRL and URN

The Goal-oriented Requirements Language (GRL) [GRL08] is a language for goal- and agent-oriented modeling and for the reasoning about requirements with a focus on quality requirements. It provides means for the description of different concepts of the requirements engineering process. The GRL is part of the User Requirements Notation (URN) [ITU08, Amy03], which was approved as ITU-T Recommendation Z.151 as an international standard in 2008. The GRL as part of the URN is the newest of the presented goal-oriented modeling notations and is based on i* and the NFR framework. The main concepts of the GRL are intentional elements, intentional links, and actors. A GRL model consists of a global goal model or of several goal models that are distributed to different actors. The elements are as already known from the i* framework: goals, softgoals, tasks, resources, and additionally *beliefs*, which represent design assumptions and relevant environmental conditions. The links are of the types contribution, decomposition, or means-end with the same semantics as with i*. Further types are *correlation* for side effects, and *dependency* for relations between actors. GRL also supports a mechanism for the evaluation of alternatives regarding their influence on the satisfaction of goals on a higher level in the graph.

Evaluation The concepts of GRL for modeling quality requirements, which are relevant for this thesis, largely correspond to those of i* and the NFR framework. There are little differences in the graphical notation as well as in additional model elements. A detailed comparing analysis of i*, GRL, as well as TROPOS, which is another variant of i*, can be found in [ACC+05]. An important concept that GRL adopts from the NFR framework, which is not included in i*, is the concept of criticality. Criticality provides the opportunity to prioritize goals for the support of conflict resolution. Regarding the application of a GORE approach for architectural design GRL has the same properties as its predecessors. The treatment of quality

goals with refinement, prioritization, and contribution links is advantageous as well as the evaluation mechanism for the selection of alternative solutions. Disadvantages are the missing support for architectural styles, design principles, architectural constraints due to technical solutions, and the lacking support for traceability.

2.1.1.4 Evaluation of the Goal-Oriented Approaches

The presented GORE approaches are adequate means for the treatment of quality goals and the modeling of functional and quality requirements during the development of software systems. The explicit consideration of quality aspects right from the beginning and the support for an evaluation of alternatives can be very useful for architectural design. The goal-oriented techniques are valuable for dealing with interdependencies between functional and quality requirements as well as with the refinement of quality goals. Furthermore, to a certain extent they enable the mapping of solutions to goals on a higher abstraction level.

However, the approaches clearly focus on the analysis and refinement of quality goals from a perspective of requirements engineering. Indeed, there are works that try to apply the goal-oriented approaches for architectural design. Van Lamsweerde [vL03] argues that requirements engineering and architectural design have to be intertwined. He derives an architectural draft from system goals. Afterwards this draft is transformed according to architectural styles and patterns to meet the constraints and quality goals. Although this approach is valuable for its systematic manner, only the older KAOS framework is utilized, which not explicitly targets on quality goals, and it is assumed that all conflicts have been resolved on the requirements level. Moreover, his formal approach to directly derive UML classes from requirements without considering architectural principles seams to be unrealistic for real world scenarios and for such an intangible quality goal as evolvability.

Further works deal with integrating goal models with UML use case and class diagrams [CdPL04], expressing design patterns in goal graphs [GY01], or combining the goal-oriented approaches with aspect-orientation [dSdSdC03, YNGB+09]. Grau and Franch [GF07] or Liu and Yu [LY01] try to generate and evaluate architectural alternatives using goal-oriented techniques. Subramanian and Chung [SC03] even relate metrics to the quality goals.

Summing up, however, from an architectural point of view these approaches on their own are not sufficient for bridging the gap to software architectural design.

Although there is some overlapping of the proposed development steps, further activities and means are necessary to support architectural design especially for evolvability. A refinement of evolvability using the GORE approaches has not yet been done in a sophisticated way considering relevant works on software evolution.

There are several weaknesses when applying the GORE approaches to bridge the gap to architectural design as already mentioned during the evaluation of the individual approaches. For example, architectural constraints regarding the system's environment, or interdependencies with organizational factors or certain technical solutions are not considered. Therefore, just applying GORE approaches to find suitable architectural solutions cannot guarantee that a certain solution can be implemented using a specific technology. Furthermore, the GORE approaches do not consider architectural design principles when relating goals and solutions. However, these principles are important means for an architect in choosing appropriate solutions even if this is done only based on experience. Moreover, it is not sufficient in a development process to just handover quality requirements descriptions in a goal-oriented notation to the architect. For architectural design the comprehension of the rationale and design decisions is very important. That is why the architect has to be involved in the establishment of quality goals and the GORE approaches have to be intertwined with architectural analysis.

2.1.2 Use Cases and Scenarios

Next to the GORE techniques, which provide the most extensive treatment of quality goals, there are of course other complementary approaches, some of which are discussed in the following.

One possibility for a structured description of requirements is given, for example, by the so-called "snow card" of the Volere template [RR99]. According to this a requirement description is an informal, textual information comprising: an identification number, a requirement type, a related use case, a description, a rationale, the originator, a fit criterion, a degree for customer satisfaction and dissatisfaction, a priority, conflicts with other requirements, supporting material, and a history. However, this concept is not a specific one for quality requirements but rather general.

Use cases are a form of specifying the behavior of a software system and its elements from the user perspective. Cockburn [Coc00] defines: *"A use case is a description of the possible sequences of interactions between the system under dis-*

cussion and its external actors, related to a particular goal". He provides a template for use case description consisting of: number, name, characteristic information such as context of use, scope, actor, stakeholders, and conditions, further a main success scenario, extensions, as well as input and output variations. The interdependencies of use cases can be visualized with use case diagrams from the Unified Modeling Language (UML), however, both are useful to specify steps of action but rather are not very effective for specifying quality requirements. Nevertheless, use cases can correspond to tasks modeled in the goal graphs of i* or GRL.

Use cases are strongly related to scenarios. Although the exact definition of scenario can vary, in general, a scenario can describe a part of the system usage as seen by its users, which corresponds to a possible flow of actions of a use case. In [AE03] a discussion of different scenario notations can be found, among them, for example, use cases, UML use case diagrams, UML sequence diagrams, UML activity diagrams, state charts, petri nets, or use case maps.

Use Case Maps (UCMs) [Amy03, Amy99] are a scenario notation, which next to GRL also belongs to the URN. UCMs focus on the description of causal relations between responsibilities of abstract components related to use cases. They graphically illustrate behavioral and structural aspects of a system in one view. Use case maps represent a complementary notation for the GRL of the User Requirements Notation. However, they are restricted to functional requirements and behavioral aspects. Quality aspects are left for the GRL. A disadvantage is also the lack of a well-defined semantics and the large human input that is required for modeling UCMs [GEM06].

In [dPLHDK00] do Prado Leite et al. present a structured scenario description that consists of title, goal, context, resources, actors, episodes, exceptions. A *constraint* attribute is used to characterize non-functional requirements as restrictions applicable to context, resources, and episodes.

Furthermore, Rozanski and Woods [RW05] define an architectural scenario as *"a crisp, concise description of a situation that the system is likely to face, along with a definition of the response required of the system."* In the context of architectural evaluation, scenarios are used for describing non-functional requirements [KC99]. For the scenario-based architecture evaluation method ATAM [KKC00] three types of scenarios are distinguished: *a*) use case scenarios, involving typical uses of the software, *b*) growth scenarios, covering anticipated changes to the software, and

c) exploratory scenarios, covering extreme changes that might strain the system. More generally, they can be divided into functional (e.g., use case) scenarios and quality (e.g., modifiability, usability) scenarios. The textual scenario description should be formulated in a way that the considered quality is well observable and assessable. Therefore, a structure as in Table 2.1 is proposed.

Table 2.1: The structure of an architectural scenario based on [KKC00, BCK03]

Attribute	**Description**
Name	The name of the scenario.
Quality	The related quality attribute.
Stimulus	The event or condition arising from this scenario.
Stimulus source	The entity (e.g., a human or software system) that generated the stimulus.
Environment	The context applying to this scenario.
Artifact	The artifact affected by the stimulus.
Response	The reaction of the system (the affected artifact) to the scenario event.
Response measure	The measurable effects showing if the scenario is fulfilled by the architecture.

In this way, the scenarios represent another valuable means next to the GORE approaches to specify intangible quality goals, such as evolvability, in more detail. Scenarios are input to architectural design, can be used for evaluating the architecture as performed by ATAM, or can be used to communicate with stakeholders to find (non-functional) requirements as for example in [SM98].

2.1.3 The EMPRESS Framework for Requirements

In conjunction with the European research project EMPRESS[1] a framework for dealing with requirements was established. In this project a method for eliciting, documenting, and analyzing quality requirements was developed [Heu04], which is of special interest for this thesis.

[1]EMPRESS: Evolution Management and Process for Real-Time Embedded Software Systems, `http://www.empress-itea.org`

The method describes a refinement process for quality goals and artifacts supporting this process. The EMPRESS approach is experience-based, which means it provides knowledge gained by experience in form of best practices and guidelines, such as checklists or questionnaires to capture non-functional requirements [PvKD+03]. It provides a quality model based on ISO 9126 [Int01] with common quality attributes as maintainability, portability, or usability, and in this way extends the catalog provided by the NFR framework by Chung et al. In the quality model these quality attributes are related to so-called means, which are principles, techniques, or mechanisms for the realization of the quality attributes (see Figure 2.1). Moreover, they related metrics for evaluation to the quality attributes. In the EMPRESS project four types of quality attributes that constrain non-functional requirements are distinguished, which should guide the requirements elicitation: organizational, and system specific attributes, as well as attributes that pertain to user tasks or system tasks.

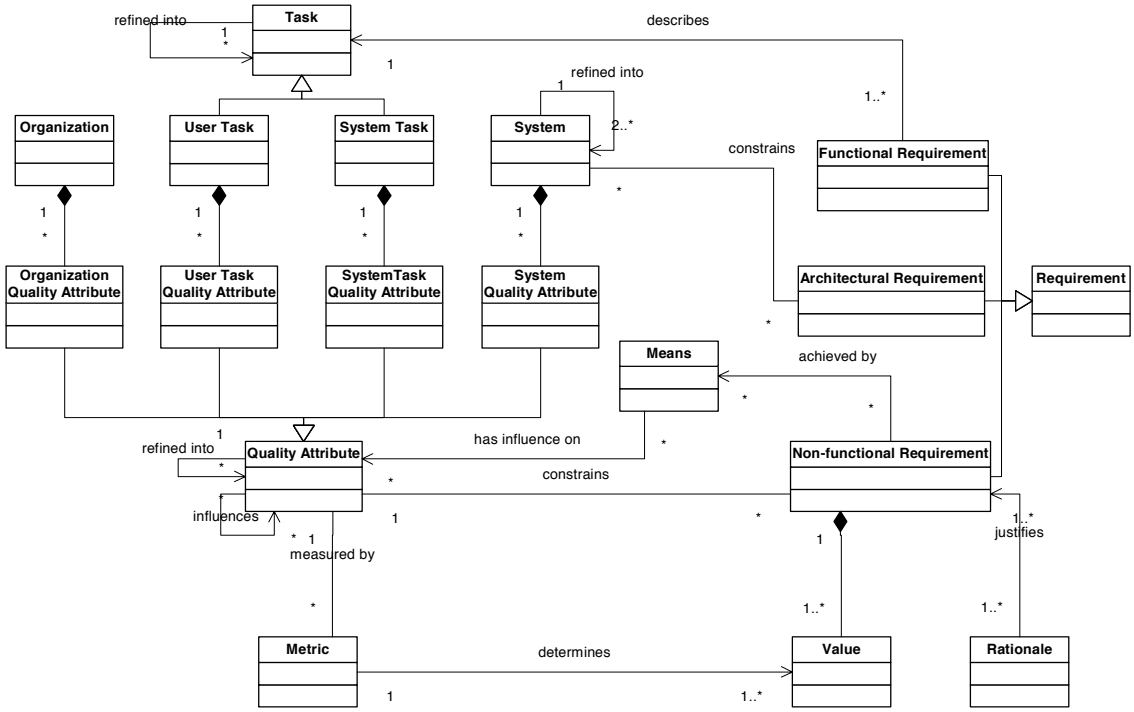


Figure 2.1: The EMPRESS metamodel for NFR by [Heu04]

During the elicitation of non-functional requirements in workshops with stakeholders first the quality attributes are determined and prioritized utilizing questionnaires. Following, the attributes are refined using refinement graphs based on the

NFR framework by Chung et al. For this purpose the EMRESS quality model can be tailored for project specific needs. Afterwards, measurable non-functional requirements are specified using checklists and templates for documentation. Finally, means for realization have to be identified.

Evaluation The EMPRESS approach provides an appropriate methodical way for the elicitation and specification of non-functional requirements using its questionnaires, checklists, templates, and the refinement of quality goals in a quality model. The described questionnaire for prioritization, however, does only consider maintainability, efficiency, usability, and reliability. Here an extension and completion is necessary. Moreover, only one exemplary checklist for efficiency is provided.

The quality model contains attributes from ISO 9126 and their refinements, for which the GORE approach of the NFR framework was utilized. The description of the refinement is comprehensive, however, does not go beyond the quality attributes of the ISO standard. Therefore, the quality goal that is most important for this thesis—evolvability—is not considered and, hence, not sufficiently supported. A further improvement could be achieved by a description of the refinement with the newer URN standard using the GRL instead of the NFR framework.

The EMPRESS approach has a clear focus on requirements elicitation and documentation for the communication with the users. Means and metrics are related to the quality attributes in the metamodel, which is destined for requirements analysis and for the support of architectural design and its evaluation. Therefore, it represents a good starting point for architectural design, but of course must be followed by an appropriate architectural design approach that further analyzes the constraints for the transformation from the requirements to the architecture.

2.2 Evolvability Support by Architectural Design Methods

As already motivated above, the satisfaction of quality goals and appropriate options for future changes are among the major goals of software architectures and even more important than functional requirements [BM07]. According to the importance of quality goals for architectural design, a high risk is related to them. As a consequence, an effective guidance is needed during the development, especially for

the implementation of goals, such as evolvability, performance, and security. From an architectural point of view there are only few methods that lead to a design process explicitly considering quality requirements. This includes, for example, the QASAR method [Bos00], the Siemens 4 Views approach comprising Global Analysis [HNS00], the Attribute-Driven Design method (ADD) [BKB02], and the Quasar method [Sie04], which are evaluated below regarding their suitability for architectural design considering quality goals and especially regarding evolvability.

2.2.1 QASAR

The QASAR (Quality Attribute-oriented Software ARchitecture) method is a design method that is often at least implicitly applied for the design of a software architecture. QASAR is a method that explicitly considers functional and quality requirements. The latter are realized in a software architecture by architectural transformations.

The method comprises three phases. In the first phase functional requirements are realized by functional components according to the sub-method Functionality-based Architectural Design (FAD). FAD uses so-called archetypes as core abstractions for functional concepts, which are used to create architectural components by functional decomposition.

In the second phase the designed architecture is evaluated regarding if it fulfills the quality requirements or not. For this evaluation different techniques are used, such as scenario-based assessment, simulation, mathematical modeling, or objective reasoning, which can constitute the starting point for the other techniques.

When the quality properties of the architecture are evaluated, the architecture is transformed to meet the remaining quality requirements in the third phase. The architecture transformation leads to a new version of the architecture with the same functionality but other functional structures and components that better realize the quality requirements. Bosch proposes five categories of transformations:

Imposing an architectural style An architectural style in general cannot be merged; changing the style results in a complete reorganization of the architecture.

Imposing an architectural pattern An architectural pattern is applied for a certain aspect of the whole architecture, such as concurrency or persistence.

Applying a design pattern A design pattern has a limited impact on the architecture, hence, multiple patterns can be applied.

Converting quality requirements to functionality Primarily some functionality not related to the problem domain is added to realize some non-functional requirements as for example exception handling to increase fault tolerance.

Distributing requirements The divide-and-conquer principle is applied, which means that quality requirements on the system level are transformed to quality requirements on subsystem or component level to be realized there, or the quality requirements themselves are divided.

All transformations in the third phase result in the realization of all remaining quality requirements that were not realized in the first phase. This is achieved by changing already designed components. Therefore, especially this last transformation type leads to a scattering of the implementation of quality requirements over the whole system that was designed by functional decomposition before.

Evaluation The author considers the QASAR method to be very valuable for an architectural design regarding quality requirements. The core concept of fulfilling quality requirements by functional solutions is of central importance. Every quality goal as especially evolvability finally has to be broken down into functional concepts to be realized by the architecture of the software system to-be. The architectural evaluation in the meantime of design makes sense to early estimate the degree of quality goal fulfillment. A disadvantage of QASAR is the scattered implementation of the remaining quality attributes during the third phase, after all components have already been designed by FAD in the first phase. This is due to the fact that the method treats functional requirements as primary and quality requirements as secondary. The scattering of the third phase is related to a high number of dependency relations resulting in a high number of traceability links for tracking the realization of the quality requirements. This hampers the maintainability and evolvability of a software system because with a changing quality goal the changes have to be traced to a lot of components in the architecture and implementation.

2.2.2 Attribute-Driven Design (ADD)

The Attribute-Driven Design (ADD) method by Bass et al. [BKB02, WBB+06] is a step-by-step instruction for the design of software architectures. ADD considers quality requirements, called quality attributes, to be at least as important as functional ones. Quality attributes and functional requirements together with other constraints represent the required input for the method. Quality attributes are not considered as "soft"-goals as by the NFR framework, which can be treated simoultaneously. Instead, quality attributes beforehand have to be prioritized and specified precisely with criteria for their satisfaction using scenarios that are similar to the ones discussed in Section 2.1.2 for ATAM. During the application of the ADD method the requirements are transformed into a so-called conceptual architecture. This conceptual architecture represents the coarse-grained structure of the software to-be without detailed, specific components and interfaces. The architecture is designed by a recursive refinement in several steps, which are shown in Figure 2.2. The architecturally relevant requirements become so-called architectural drivers that are realized by applying design concepts, as architectural tactics, styles, or patterns.

Evaluation The ADD method is an interesting approach regarding the fulfillment of quality goals in architectural design. ADD, just as QASAR, achieves the fulfillment of quality goals by allocating functional components. Its strengths are the precise specification of the quality attributes with scenarios and its focus on the most significant architectural drivers. Regardless of which type the drivers are used for choosing appropriate functional solutions by utilizing suitable architectural tactics, styles, and patterns. However, ADD also has its drawbacks. First, the method outputs only a conceptual architecture. ADD does not provide decisions for specific components or classes; hence, the design process has to be continued by other means. Because of its recursive nature ADD is only top-down oriented; a bottom-up approach for some parts of the architecture is not intended. Furthermore, if the focus is put too much on quality attributes, the influence of the functional requirements for the software is neglected. If evolvability is the most important quality goal, the application of ADD can get difficult as examined in [Rum09]. ADD requires design concepts, such as patterns and tactics, that have a positive influence on the quality goal for a proper decomposition. Unfortunately, there is not yet a well-established set of styles, patterns, and tactics for evolvability, which could be applied.

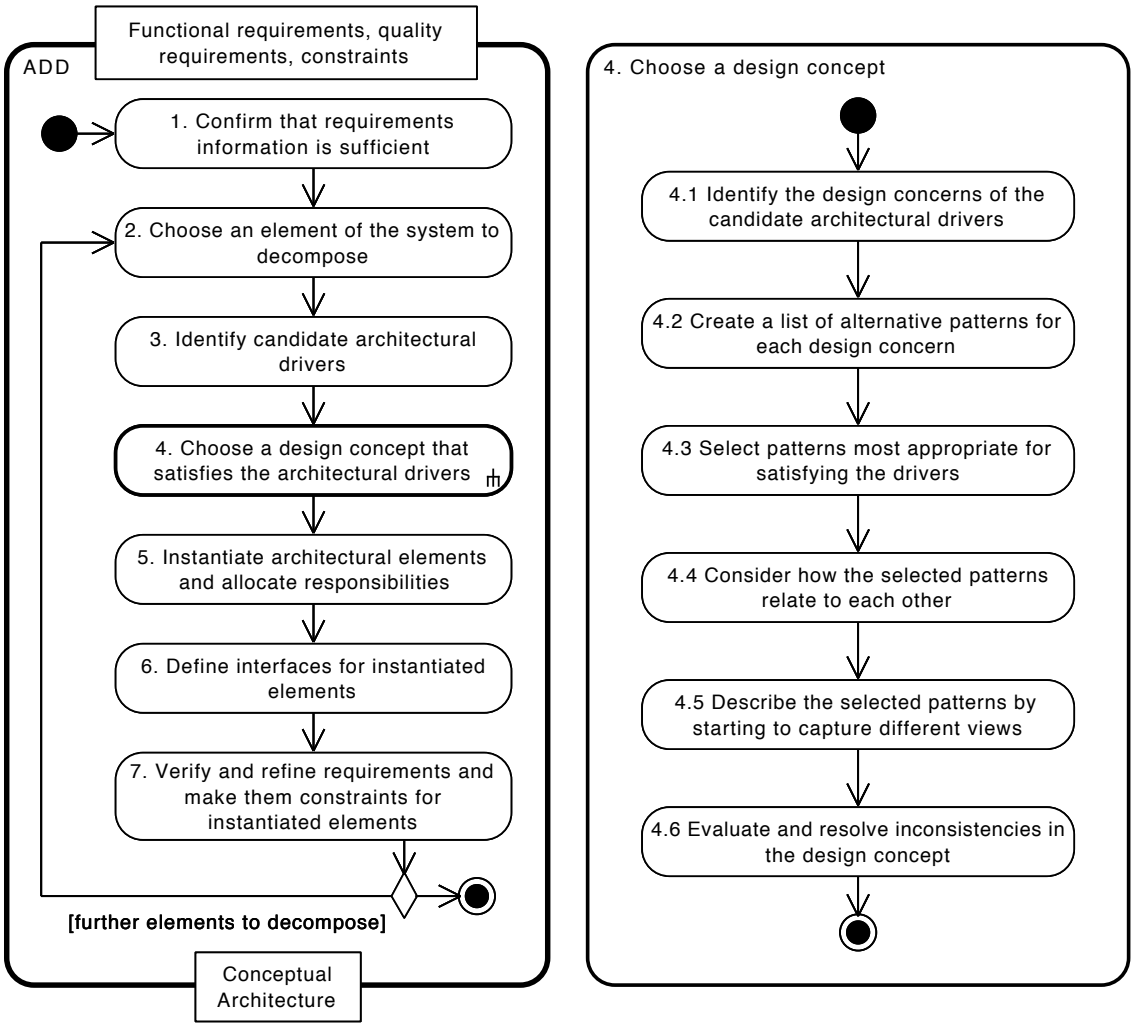


Figure 2.2: Steps of ADD adapted from [WBB+06]

2.2.3 The Siemens Four Views Approach

Siemens Four Views (S4V) is an architectural design approach by Hofmeister et al. [HNS00, HNS05], which uses four views to document an architecture: a conceptual, a module, a code, and an execution view. The methodical steps of the approach, which are used to develop each of the views, are depicted in Figure 2.3. S4V tries to bridge the gap between requirements and architecture with the *Global Analysis* method.

The purpose of Global Analysis is to analyze the factors that influence the architecture and to develop strategies for its design considering these factors. Influene factors are the key issues that drive architectural design. They are distinguished according to three categories with increasing impact on the architectural design:

- *organizational factors* constraining design choices,
- *technological factors* limiting the design regarding hardware, software, or standards, and
- *product factors* covering functional features and quality attributes.

The Global Analysis method consists of two parts each with three sequential steps as shown in Figure 2.3. Accordingly, it is concerned with two main artifacts: *factor tables* and *issue cards*. Three factor tables, one for each factor category, are used to record influence factors. The issue cards discuss important design topics that are related to different factors in order to develop strategies for an accommodation of the factors in the architecture. The activities *central design tasks* and *final design task* complete the S4V approach by creating design artifacts, such as layers, components, and interfaces, according to the different views.

Evaluation The strength of the S4V approach is the Global Analysis method, which complements requirements analysis with the influence factors. It provides a set of factor categories and subcategories, which help the designer to focus on key design issues and which can be used as a kind of checklist. Furthermore, architectural rationale can be recorded with strategies as a part of issue cards.

In this way Global Analysis provides systematic means for bridging from requirements to an architectural viewpoint. It supports quality requirements especially by product factors. The product factors can directly be connected to non-functional requirements or quality goals. Thus there is a need for traceability of factors to requirements. The GORE approaches could be used to support the identification of factors. Furthermore, Global Analysis explicitly considers flexibility and changeability of the analyzed factors, which can be helpful for evolvability. Flexibility covers the negotiability of factors. Changeability can cover both the stability and variability of factors.

However, Global Analysis also has its drawbacks. It does not answer the question how to find the solutions and strategies described by the issue cards starting from the influence factors. In this regard Global Analysis should be complemented with activities for selecting appropriate means such as tactics and patterns as proposed by ADD. Moreover, a prioritization of influence factors is only considered implicitly through the importance of the categories but could be considered explicitly. Besides,

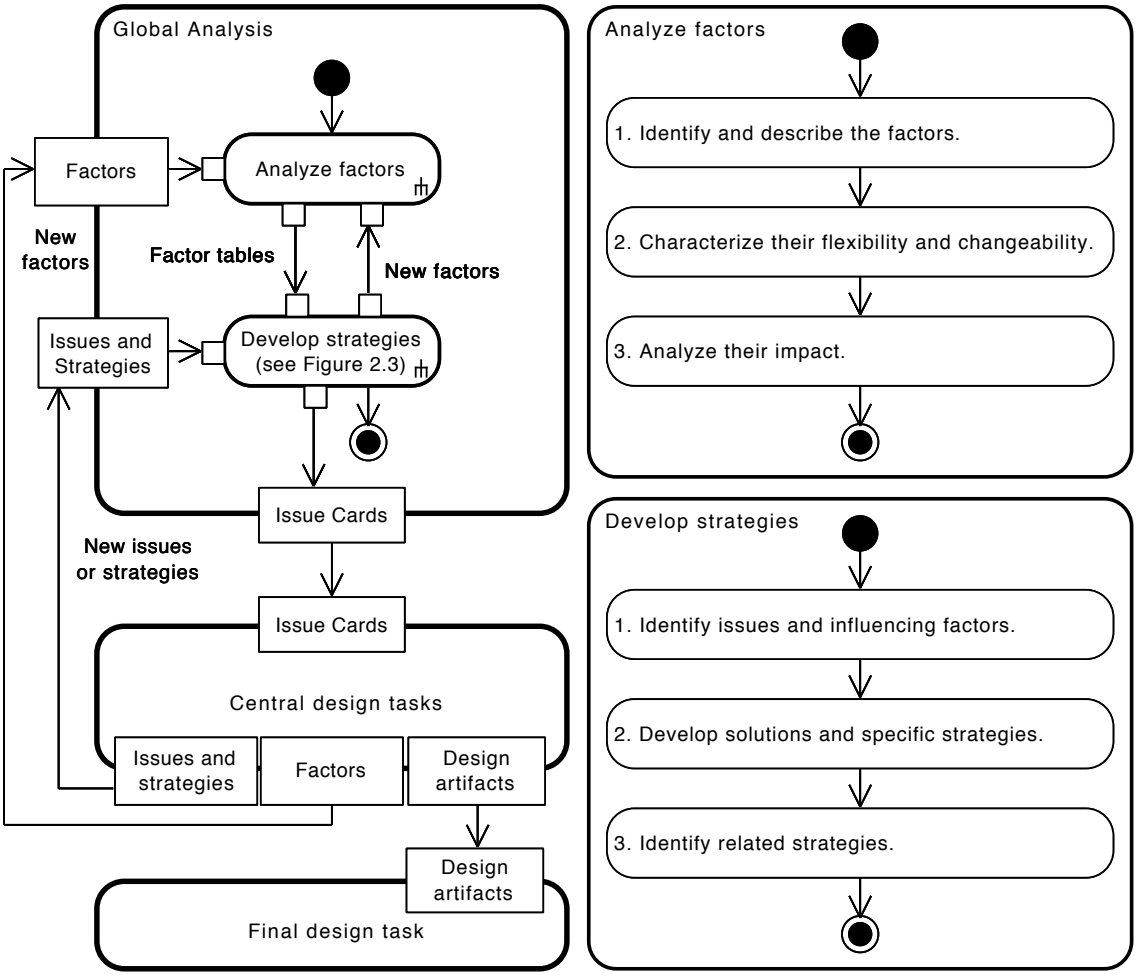


Figure 2.3: The Siemens Four Views approach according to [HNS00]

for the practical applicability of the approach, which deals with a larger number of factors, tool support is needed. Moreover, traceability of influence factors and the issue card's solution strategies should be supported not only inside the artifacts of the Global Analysis but also from and to other artifacts of the design process.

2.2.4 EMPRESS Pattern-Based Architectural Analysis and Design

In the context of the EMPRESS project next to the framework for requirements (cf. Section 2.1.3) also a method for pattern-based analysis and design of software architectures was developed [KB03]. The EMPRESS requirements framework already relates quality attributes to non-functional requirements, metrics, and means. Be-

yond this, in the architectural design method, scenario descriptions and patterns are related to the means and thus to the quality attributes. Abstract scenarios comprise a number of stimulus/response pairs. Concrete scenarios belong to one alternative of the abstract scenario and describe the expected behavior that is guaranteed by a pattern. Patterns are related to a certain problem, which is solved by the pattern, a rationale why the pattern solves the problem, and a description of the actual solution. The corresponding metamodel is depicted in Figure 2.4.

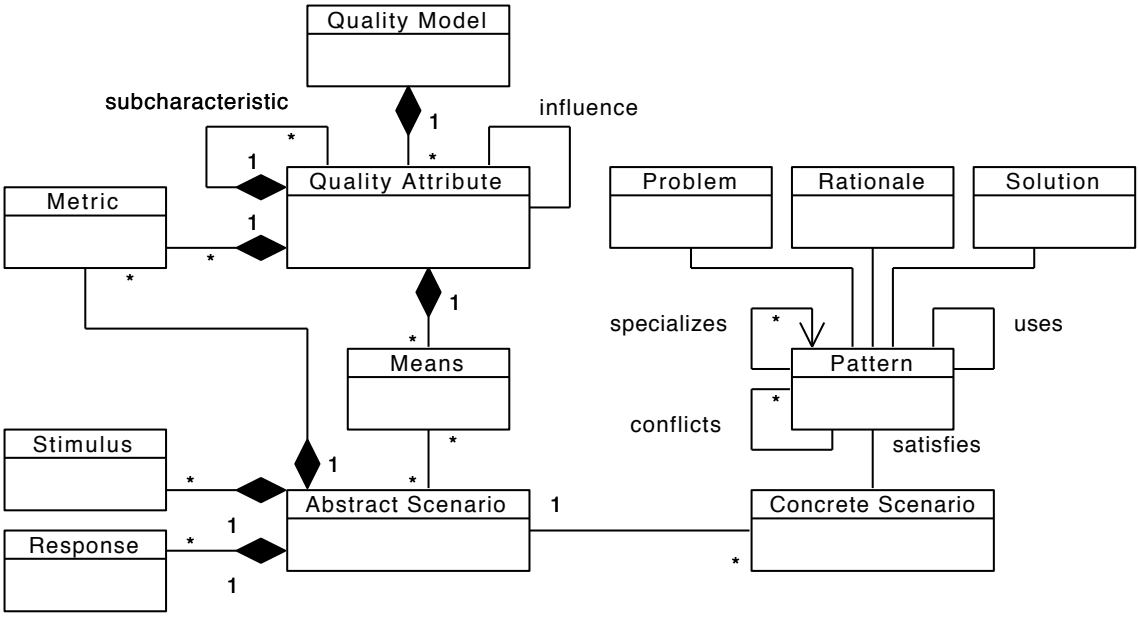


Figure 2.4: The EMPRESS metamodel for design by [KB03]

The method description includes quality models for the quality attributes performance, maintainability, and reliability with related metrics, means, as well as a description of patterns for reliability and safety. Besides, the EMPRESS design method describes an iterative process with seven steps:

1. Selection of scenarios and planning of the next iteration,
2. Definition of evaluation criteria,
3. Selection of means and patterns,
4. Instantiation of patterns,
5. Documentation of the architecture using views,

6. Evaluation of the architecture; and if the architecture is not already ok,

7. Analysis of the problem and return to step 1.

Evaluation Many design methods consider architectural design as a creative process, which is for the most part based on experience. The EMPRESS method for pattern-based design is a good approach to improve this process with defined activities. It is especially concerned with quality attributes. However, the several design activities of the method are described rather sketchy in comparison to the established metamodel. An open question, which remains when applying the method, is, how conflicts between quality attributes are resolved. The method implicitly contains the assumption that an appropriate pattern can be found by selecting means and comparing the related scenarios with the requirements. Trade-offs for quality attributes are not considered. With its steps the method has similarities to ADD combined with ATAM but without the restriction of the recursive nature. Nevertheless, it does not contain guidance for the specification of concrete components and interfaces either. Unfortunately, only three quality attributes are described with subcharacteristics, metrics, and means in the quality model; evolvability is not included. Furthermore, there are inconsistencies with the EMPRESS framework for non-functional requirements, for example concerning maintainability. This demands for a revision and extension regarding evolvability.

2.2.5 Quasar

Quasar—an acronym for QUAlity Software ARchitecture—is an architectural design method that was developed in industrial practice [Sie04]. The method combines best practices and principles for a good component-based software architectural design. It uses so-called software categories for the identification and structuring of software components. All components are based on the standard categories 0, A, T, and R. For a specific application design, these categories are refined in a category model.

A-components are application specific but independent of technical issues. They contain the application logic and entity classes for the realization of the domain functionality. T-components cover technical knowledge about a system, and they frequently provide an application programming interface (API), for example, for database connectivity or for graphical user interface (GUI) elements. They are in-

dependent of specific application functions. Software of the category 0 is neutral concerning the application's functionality and independent of technical aspects, hence, it creates no undesired dependencies. Modules, classes, and interfaces with a high degree of reusability belong to 0-software, for example, class libraries. R-software refers to representation; it establishes a connection between A- and T-components, however, minimizing the dependencies between them. This is achieved by transformation, for example, to external data presentation formats like XML. Other ways of directly connecting or even mixing A and T—the so-called AT-software—are prohibited, because they would re-introduce stronger dependencies.

In contrast to other methods, as for example ADD, Quasar defines a fine-grained sequence of steps for the detailed specification of components and their interfaces. It was developed further in [Bod08] to establish three basic steps: component identification, interface specification, and inner structuring. These steps help to move from analysis to architectural design in a more precise way than other design methods.

Evaluation Quasar can be considered as a good architectural design method for component-based design. The use of software categories helps to structure the requirements, and thus to bridge the gap between analysis and architectural design. The identification of components with the help of software categories results in the reduction of dependencies by a separation according to the responsibilities and the knowledge covered by the components. Moreover, the specification activities for interfaces are strongly concerned with decoupling. In this way the design according to Quasar, especially the distinction between application-specific and technology-specific software, leads to good modularization, decoupling, and separation of concerns. Therefore, it provides high architectural quality with some benefits for evolvability. However, the method does not especially consider the implementation of non-functional requirements. Nevertheless, the fine-granular design steps for component and interface specification can complement other design methods as ADD and facilitate a precise definition of rules for the establishment and maintenance of traceability links [BR09].

2.2.6 Evaluation and Relation to Further Works

The methods' activities can be classified to the phases *architectural analysis*, *architectural synthesis*, and *architectural evaluation* in a general model [HKN+07]. Archi-

tectural analysis identifies the architecturally significant requirements. Architectural synthesis creates the candidate solutions addressing the requirements, and architectural evaluation ensures that the right architectural decisions are made. This general model can be extended according to [TAJ+10] for the management of architectural design knowledge by further phases, which are: *architectural implementation* and *maintenance*.

For the architectural analysis phase the Global Analysis of the S4V approach is an appropriate method. The influence factors represent architecturally significant requirements and can be identified in a goal-oriented way utilizing the goals from the GORE approaches. Besides, the influence factors drive the architectural design process, and have a direct correspondence to the architectural drivers of the ADD method. Scenarios can be used to specify quality attributes in detail.

In the architectural synthesis phase the utilization of styles and patterns as in QASAR or as proposed by Harrison and Avgeriou [HA07a], as well as the tactics from ADD constitute an effective way for balancing between functional and quality requirements and for structuring the conceptual architecture. This can be complemented with Quasar to derive specific components and interfaces with a good separation of concerns.

A distinction and classification of the terms design patterns, architectural patterns, and architectural styles as general architectural constraints concepts can be found in [GHR07]. The patterns necessary for synthesis can be taken from pattern catalogs (e.g. [GHJV94, BMR+96]). A good introduction on patterns is given in [AZ05]. Unfortunately, there is not yet a catalog of patterns especially for evolvability, as it does exist for security [YB97], for example. Architectural tactics represent further means guiding the design and are discussed in several works by the SEI (e.g. [BBK03, BBN07]).

Architectural decisions between the several instruments, such as patterns, have to be made according to their impact on the quality goals. This should result in a goal-oriented way of selecting patterns and tactics. One related approach is the MidArch design method [GRH08]. It supports a valuable systematic selection of particularly middleware-oriented architectural styles from a repository to attain a system's quality requirements. An integration with the GORE approaches, an explicit consideration of design principles, and a classification of styles regarding quality goals as intended by the approach of this thesis however is not discussed.

A general repository of solution instruments can follow the concept of the so-called architect's toolbox. This toolbox contains the architect's stock of solution instruments and represents a knowledge base of design knowledge. According to [PBG04] the toolbox can be structured into two parts:
a) a catalog of approved methods and solution templates (e.g., patterns), as well as
b) a catalog of fundamental technologies and tools (e.g., frameworks).

2.3 Design Traceability

An important issue in designing for evolvability is to support comprehension of the design and the dependencies of the development artifacts. During the development artifacts depend upon each other as the creation of new artifacts is influenced by already existing artifacts. For example, a component depends on a requirement that specifies the desired functionality, which the component realizes. The detection of these dependencies and the documentation of design decisions form the basis for analyses that are performed to enable changes during the evolution and reengineering of software systems. The utilization of design dependencies for change impact analysis and coverage analysis can prevent incomplete or inconsistent changes.

Traceability is a concept for the detection and documentation of dependencies during software development to enable utilization of the dependencies and thus to support evolutionary changes. An appropriate comprehensive approach of traceability should support the development approaches analyzed in the preceding sections regarding the evolvability of a software system. Supporting traceability during software development can increase other quality goals, such as understandability or changeability. For this thesis the dependencies from quality goals to design artifacts and between design artifacts are of special importance because this relates to the scope of architectural design. In this regard the term design traceability is used.

In this section the traceability concept is explained with definitions, advantages, and challenges. Furthermore, classification criteria for existing traceability approaches are presented. A comprehensive recent survey of traceability can be found in [WvP10], which builds upon former ones, as e.g., [vKP02, Pin04, SZ05, ARNRSG06, GG07]. Meanwhile an enormous amount of traceability approaches emerged from research. Therefore, only some approaches can be discussed in detail that are relevant to architectural design and had a major influence on this the-

sis for tracking the realization of quality goals and their applicability during the architectural design process.

2.3.1 Origin, Definition, and Benefits of Traceability

The concept of traceability in software engineering refers back to requirements traceability. Requirements traceability is defined by Gotel and Finkelstein [GF94] 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.)"*. It enables to track requirements to their source during elicitation and to document all changes made to them.

Important works in the area of requirements engineering, where the largest part of traceability research has been done so far, are, for example, those of Gotel and Finkelstein [GF94], who introduced the distinction between pre-requirements specification and post-requirements specification traceability, Ramesh and Jarke [RJ01], who introduced a reference model for requirements traceability, or Letelier [Let02], who utilized UML for a metamodel for traceability between textual requirements and models. Other important approaches for traceability link establishment between requirements and test cases, for example, are the scenario-driven approach of Egyed [Egy01] and the approach of Olsson and Grundy [OG02].

Traceability can be applied not only to requirements engineering but on the whole software development life-cycle. In this context the IEEE Standard Glossary of Software Engineering Terminology [Ins90] defines traceability as: *"The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another. [...]"* Generally, traceability is a concept for linking together software artifacts (e.g., documents, models, code) or entities (e.g., model elements, paragraphs) from different development phases. Lago et al. [LMvV09] concisely define: *"Traceability is the ability to describe and follow the life of a software artifact and a means for modeling the relations between software artifacts in an explicit way."* The concept of *traceability links* is used to express this explicit way of modeling relations or *traces* [Ins90] between artifacts spanning different development phases. In this way traceability enables or supports for example [Mäd09, SZ05, Poh96]:

- coverage analysis regarding the implementation of all requirements;
- impact analysis of changes;
- software maintenance and evolution;
- documentation and reasoning of design decisions;
- the increase of system comprehension;
- the reuse of software components;
- software testing, verification, and validation.

Summing up, traceability has advantages in understanding, testing, changing, and evolving a software system. It improves the software quality by supporting different development activities. The utilization of traceability provides several advantages even if it requires additional effort. However, the benefit-effort ratio should be reasonable for the practical applicability of traceability approaches [AR05, CH06].

2.3.2 Classification of Traceability

Important aspects of traceability, which are discussed by different approaches, can be categorized by specific activities [SEW09, vKP02]:

Defining traceability information refers to determining entity types to be traced and relationships in between (also referred to as "planning and preparing" in [WvP10]).

Recording traces refers to physically representing traceability relations in data structures.

Identifying refers to discovering dependencies and making them explicit as traceability links.

Maintaining means to update, modify, or delete existing traceability links according to changes in the traced entities.

Retrieving addresses querying or gathering traceability information, which has already been identified and recorded.

Utilizing or using the traceability information can be part of several development activities, for which it provides benefits (compare Section 2.3.1) and is also related to means of visualization.

Typically the traceability approaches only cover one or two and seldom more aspects of traceability. A well-defined general and comprehensive approach spanning all activities is not yet available and needs more work to be done [ARNRSG06, SEW09].

In addition to a classification of the existing approaches according to activities several authors discuss further classifications and properties of traceability.

The ANSI/IEEE Standard 830-1984 [Ins84] introduced the terms *backward traceability* for following a traceability link from a certain artifact to its source and *forward traceability* for following it to other derived artifacts. This results in a generally bidirectional characteristic of traceability links although they might be technically represented in a model as unidirectional.

Gotel and Finkelstein [GF94] discuss the terms *pre-requirements specification (pre-RS)* and *post-requirements specification (post-RS)* traceability referring to activities that lead to a requirements specification document and activities concerned with the later realization of the requirements, respectively.

Ramesh and Edwards [RE93] distinguish *horizontal* and *vertical* traceability. These terms refer to traceability links established between artifacts belonging to the same abstraction level or development phase and between artifacts of a different abstraction level or development phase, respectively. Therefore, also the terms *intra-level* and *inter-level* traceability apply [GG07]. In model-driven development this correlates to *intra-model* and *inter-model* traceability [KPP06].

Regarding the storage of model-related links *internal* and *external* traceability can be distinguished [KPP06]. Internal or embedded storage of traceability links is realized with specific model elements inside the model they refer to. It is only possible for intra-model links and results in model pollution. External traceability relates to a separate traceability model, requires unique and persistent identifiers of the model elements that are linked, and can cope with intra and inter-model links.

The recording of traceability links can be done *on-line* as a by-product of the actual development activity or *off-line* after the development activity [vKP02]. In the latter case, the recording should be performed as soon as possible after the development activity, because the later the recording is performed, the more imprecise the links can get [CHCC03]. On-line recording usually is done during model transfor-

mations, nevertheless recording of additional traces that are not a direct by-product of the transformations might be necessary.

Moreover, the identification activity for traceability links can be distinguished to be *manual*, *automatic*, or *semi-automatic* [SEW09]. Purely manual identification approaches strain the developer with a lot of additional effort and are error-prone, which hampers traceability to be accepted in industry. A complete automation seems illusive regarding completeness and correctness of the links. However, semi-automated traceability approaches, which demand only for user interaction in questionable or conflicting situations or as a basis for automation, are a compromise to reduce effort and cost while producing high-quality traceability information. Automation can be achieved using techniques as *model transformation*, *information retrieval*, or *rule-based traceability*.

2.3.3 Traceability Schemes and Metamodels

The definition of the traceability-related terms helps to understand the concept. However, in order to be able to utilize traceability reasonably and efficiently in the development process, a traceability metamodel is necessary, which defines how traceability links are structured and what is their syntax and semantics [WvP10]. One of the first traceability models has been proposed by Ramesh and Jarke [RJ01] for requirements traceability. However, this model is rather a conceptual one. In the context of model-driven design and its model transformations more formal metamodels were established, a lot of which were published in the ECMDA traceability workshop series.

The core of such a metamodel, the specification of a traceability link that connects a source and a target model element, frequently is the same with some potential variation points regarding the cardinality of the links and the modeled metadata. But often the model is defined in an ad-hoc manner. In contrast, Drivalos et al. [DKPF08] propose a "traceability metamodeling language" (TML) which could be used for the definition of traceability metamodels.

Moreover, Aizenbud-Reshef et al. [ARNRSG06] argue that a traceability metamodel should predefine link types and allow customization as well as extensibility for new link types. Espinoza et al. [EAG06] analyzed several traceability approaches and proposes a so-called *traceability meta-type* that comprises: description, purpose, objects to link, linkage rule, subtype classification, uses, and examples. One

extensive classification of link types is presented by Spanoudakis and Zisman [SZ05]. They distinguish links of the types: dependency, refinement, evolution, satisfiability, overlap, conflict, rationalization, and contribution. Unfortunately, there is no standardized set of traceability link types. A universal semantically rich traceability model has still to be found [LG05, RJ01, vKP02]. The advantage of links with well-defined semantics is to provide more accurate impact analysis results [LS96].

2.3.4 (Semi-)Automated Traceability Approaches

Many existing traceability techniques and tools (e.g. DOORS[2]) expect manual effort for the creation of traceability links [SZ05]. However, as already mentioned above, automated or semi-automated approaches for traceability link establishment are desirable because manual traceability is error-prone, difficult, time consuming, and hence rather less effective. Therefore, a major goal for traceability research is to find a reliable and effective way for automatic traceability link establishment.

Of course, some approaches that support automated or semi-automated traceability have already been proposed. However, they are mostly considered with requirements [Poh96, CHCC03] or code and documentation [ACC+02, MM03]. Traceability of architectural design artifacts is considered to a lesser extent.

2.3.4.1 Information Retrieval-based Approaches

One approach to achieve fully automated traceability is to use text mining and information retrieval (IR) techniques. Typical traceability approaches use probabilistic and vector-based IR techniques [ACC+02, HDO03] as well as Latent Semantic Indexing [MM03]. A commonality of these approaches is their use of similarity scores or distance measures based on the frequency and dispersion of terms to determine the similarity of documents or models. As a result, candidate links between pairs of models are identified, the similarity score of which is above a certain threshold.

A first processing step in these approaches is to analyze the documents or models and extract relevant terms, which then are inserted into intermediary auxiliary means as word matrices or vector spaces. For text-based analysis, differences in syntax and semantics are resolved for example through [IK06]:

[2]IBM® Rational® DOORS®, former Telelogic DOORS, `http://www-01.ibm.com/software/awdtools/doors/`

- thesaurus replacement for treating synonyms,
- word stemming,
- expansion of abbreviations, or
- stop-word elimination for words with no semantic meaning, such as articles.

Afterwards, a comparison step follows to identify interrelated pairs of words for example with:

- direct, complete word matching,
- partial matching, such as containment,
- approximate n-gram matching, or
- vector-based distance measures.

Goal-Centric Traceability An interesting IR-based traceability approach regarding quality goals is the one of Cleland-Huang et al. [CHSB+05]. Their goal-centric traceability (GCT) approach establishes traces between functional or quality goals and UML class diagrams based on a probabilistic network model. Goals are modeled using the Softgoal Interdependency Graphs from the NFR framework (see Section 2.1.1.1). The trace relations are dynamically retrieved with a query algorithm but not explicitly recorded. The results are assessed in a user evaluation step.

Evaluation The processing step of GCT that integrates user evaluation of traceability links is valuable for treating incorrect links. Unfortunately, the manual evaluation is additional effort. Moreover, they only support a limited range of the whole software development process because they cover only requirements and UML class diagrams. Therefore, they can identify only those traceability links between quality goals and design artifacts that are based on incidental matching of terms in the processed artifacts. However, goal graphs as requirements descriptions and source code generally are not created considering traceability. Consequently, the completeness and correctness of the traceability information is limited and partly relies on naming conventions for the artifacts. Nevertheless, this approach inspired the work of this

thesis by considering information retrieval techniques and especially goal graphs for traceability link establishment regarding quality goals.

One of the main advantages of IR-based approaches is their high recall factor, which can be up to 90%. This is due to the fact that all artifacts are equally considered for traceability analysis without especially considering their semantical meaning. However, these approaches typically suffer from a low precision from 10% to 50% because they detect pretended dependencies based on incidental similarities of terms. An important disadvantage of these approaches is that they are lacking a defined semantics of the links. Due to the IR techniques it is hard to automatically create links with categorized link types, which is important for change impact analysis as explained above.

2.3.4.2 Rule-based Traceability

Another possibility for automated traceability next to IR-based approaches is to specify the relevant development artifacts and to utilize rules for link establishment. Rule-based approaches can be distinguished further by *structural* and *linguistic* rule-based approaches [SEW09], though they can be combined. Structural rules are based on the existing relations between the considered artifacts, e.g., a generalization relationship between classes. They further can determine transitive relationships between artifacts. Linguistic rules are applied on the syntax of natural language texts. An easy application of linguistic rules is to establish traceability links based on the occurrences of keywords. Natural language processing techniques can accomplish this technique.

The approach proposed by Spanoudakis et al. [SdGZ03, SZPMK04] uses rules for an automatic creation of both intra-requirements traceability links and links between requirements statements, use cases, and analysis object models. With these rules syntactically related terms are identified in the different artifacts. The rules comprise three parts:

Head defines the relevant elements for the rules and an informal description,

Query defines a search description or conditions to which the rule applies,

Action formulates the result processing, e.g., the link creation.

The query part uses constructs akin to query languages for databases, such as SQL, which can be used to navigate in the artifacts. A rule-engine processes the rules

defined in a rule catalog. In this way different types of links can be established. From a technical point of view, the eXtensible Markup Language (XML) is used to specify the traceability rules. Furthermore, the approach requires to export all artifacts into XML files on which the rules are applied to create hyperlinks between the artifacts based on the XML Linking Language (XLink).

Jirapanthong and Zisman [JZ09] extend this approach by a specific traceability metamodel and a rule set for the establishment of links between different kinds of artifacts, such as feature models and UML diagrams. They utilize the XQuery language for querying related model elements. However, the disadvantage of this is that all models have to be materialized as XML files and the rules cannot operate directly on a model repository.

Another extension of the approach, which also provided valuable input for this thesis, is the one of Filho et al. [FZS03]. The authors relate i* models to UML models and establish traceability links of different types between the model elements. In this thesis this approach is adapted to URN models as a means for relating functional and quality requirements models to design artifacts.

Evaluation The rule-based approach around the group of Spanoudakis and Zisman is very promising for achieving high quality traceability information. With the consideration of goal graphs and UML models they make an important contribution regarding the traceability of quality goals and design artifacts. An important advantage of rule-based approaches is their high precision. If traces are inferred according to a (correct) predefined rule set, the created traceability links can hardly be wrong [ARNRSG06]. Moreover, due to the exact specification of source and target elements in the element part of a rule, it is possible to specify the type of the traceability relationship explicitly. Hence, for the utilization of the links not only a dependency between two model elements is known, but also the more interesting "how" and "why" they are related. Therefore, rule-based approaches are evaluated better than pure IR-based approaches [SZ05]. Recall and precision range between 50% and 95%. However, they require human effort in establishing an appropriate rule set as a basis. Furthermore, there is no consensus on what are acceptable levels of recall and precision that create trustworthiness with the users of traceability. Nevertheless, compared to manual traceability the (semi-) automated approaches reduced the effort.

2.3.5 Tool Support

For a proper usage of traceability the links have to be established on a certain level of detail. The resulting high number and complexity of the traceability relations leads to a high effort to manage them. Therefore, tool support is essential. Today tool support for traceability often is provided by requirements management tools, such as IBM's Requisite Pro® or DOORS®. The traceability capabilities of UML modeling tools are very limited [Mäd09] as well as for other design artifacts. Spanoudakis and Zisman [SZ05] discuss five dominant types of tool support:

The single centralized database approach The database typically co-operates with the tool used to maintain the traceability relations (e.g., DOORS). With this approach the linking of artifacts managed by different tools is difficult, as well as a differentiation of link types.

The software repository approach A repository provides flexibility for a traceability metamodel definition. This approach also typically provides an API that can be used for querying and integrating with other tools, but it needs tool integration effort in advance.

The hypermedia approach The utilization of hypermedia technology can be helpful for the integration of artifacts from heterogenous tools. In this way the effort for integration is decreased compared to a repository approach.

The mark-up approach For enabling traceability in distributed and heterogenous environments mark-up languages can be used to store the relations separately from the artifacts as discussed by Spanoudakis et al. with XML (see Section 2.3.4.2). The approach also uses translators for the transformation of the original models to XML in order to support any kind of models.

The event-based approach With all approaches except the centralized database the maintenance of traceability relations is difficult but can be addressed with event-based approaches. They establish a notification mechanism for change events to enable event subscribers to be informed about changes and to update the traceability links accordingly.

Evaluation For industrial settings the applicability of traceability relies on the ability to provide tool support for all types of artifacts from the development pro-

cess. The mark-up and repository approaches together with event-based traceability seem promising for the integration of heterogenous tools. A repository might be superior regarding performance and might enable versioning of traceability relations. The event-based approach facilitates maintenance. Mäder [Mäd09] addresses the maintenance issue for traceability links with his tool traceMaintainer that uses a semi-automatic event-based, and rule-based approach but is limited to static UML models. Most of the existing tools lack a customizable set of traceability link types and have limited support for standardized input or output representations. Because of variety in artifact types, the technical coupling of different models and especially support for design artifacts is an important issue. Additionally to the interoperability problem also effective, robust, automated establishment of traceability links has to be supported.

2.3.6 Evaluation and Challenges

For an evolvable architectural design an adequate traceability approach should be utilized. Then benefits such as improved comprehension and support for the impact analysis of evolutionary changes are achieved. The establishment of traceability links should be integrated with the usual activities for creating and changing development artifacts. This would lead to increased quality of traceability information, for example, regarding the semantically important types of traceability links. However, there is some research to do to overcome the challenges still existing regarding traceability:

- Traceability currently is not applied continuously over the whole development process. Therefore, comprehensive approaches spanning several development phases are needed.

- Especially design artifacts and activities are still not considered to the same extent as requirements. Therefore, different approaches from requirements engineering and model-driven design should be integrated.

- The linking of heterogenous artifacts from different development phases requires appropriate tool-support, which integrates different CASE tools in regular development environments.

- Information retrieval-based, rule-based, and event-based approaches should be combined to further enhance automated traceability for effort reduction.
- Traceability approaches should provide semantically rich traceability with a standardized set of link types, which is necessary for analyses performed on traceability information.
- Standardization of traceability vocabulary as well as ontologies might provide another way to improve the semantic meaning of traceability relations [SZ05]. Ontologies can formalize common aspects of the specification of software systems.
- The maintenance of traceability links needs further consideration, for example, by versioned traces [WvP10].
- Trace utilization is not an easy task as well. Therefore, an appropriate, easy, and usable access to traceability information including visualization should be provided [WvP10].

For this thesis a combination of existing approaches explained in this section is of interest and will be explained in detail later (see Chapter 7). Especially the rule-based approach has potential to be combined with other techniques for automated traceability supporting different types of links regarding quality goals and design artifacts. Together with some information retrieval techniques the comparison of model elements and properties can be improved to identify as much as possible of the implicitly existing artifact dependencies. A combination with XML technology should enable the integration of heterogenous models from different CASE tools and the versioning in a repository. Finally, an event-based mechanism could be applied for maintaining the traceability links.

Chapter 3

Evolvability and Related Terms

The discussion about evolvability and the distinction from maintainability was already introduced in Chapter 1. The term evolvability is closely related to the terms evolution, maintenance, as well as maintainability. The following sections describe these terms in more detail. Subsequently, evolvability is explained in its differences to the related terms, a definition of evolvability is provided, and concepts for its support are examined. Parts of this chapter were already published in [RB09] and [Bod09].

3.1 Maintenance

Software maintenance according to the IEEE Standard 1219-1998 [Ins98] is the *"modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment"*. Accordingly, the term maintenance is generally used for changes to a software system in the phase after its delivery to the customer. Furthermore, the standard provides a distinction between different types of changes, that is:

- *corrective maintenance* for fault correction often called bug-fixing,
- *adaptive maintenance* for modifications to a changing environment,
- *perfective maintenance* to improve quality attributes.

The IEEE Standard Glossary of Software Engineering Terminology [Ins90] further adds:

- *preventive maintenance* for measures performed against future problems.

Maintenance activities mostly involve small changes, because time and cost restrictions play an important role. Conversely, time and cost pressure lead to incomplete changes and subsequent faults. Moreover, maintenance activities not necessarily contribute to an improved maintainability of a software system. Exemplary activities are extensions for dealing with additional logical conditions, which lead to worse maintainability if a complicated distinction of cases or the multiple usage of parameters decrease the understandability. Maintainability also decreases if extensions lead to code duplicates, very long methods or building blocks, or to many local variables. This results in subsequent faults, which finally lead to architectural decay.

The newer definition of software maintenance from ISO 14764:2006 [Int06] incloses the *"totality of activities required to provide cost-effective support to a software system. Activities are performed during the pre-delivery stage as well as the post-delivery stage"*. Pre-delivery activities for example refer to planning post-delivery operations. Post-delivery activities are modifications or operating a help desk for the customers. This definition is more extensive than the former one and is closer to the conditions of reality.

Moreover, maintenance often is related to the term legacy system, which denotes an old system that can hardly be developed further in consequence of various modifications and an advanced architectural decay. In this case maintenance activities just keep the system operating. Far-reaching innovations normally require the application of reverse and re-engineering activities [Arn93].

3.2 Evolution

The term software evolution was introduced by Lehman with his *"laws of software evolution"* [Leh80]. However, so far there is no standard definition for software evolution. Lehman and Ramil [LR06] describe evolution as *"a process of* progressive*, for example beneficial, change in the attributes of the evolving entity or that of one or more of its constituent elements. What is accepted as progressive must be determined in each context. It is also appropriate to apply the term evolution when long-term change trends are beneficial even though isolated or short sequences of changes may appear degenerative. Thus it may be regarded as the antithesis of decay. For example,*

an entity or collection of entities may be said to be evolving if their value or fitness is increasing over time; Individually or collectively they are becoming more meaningful, more complete or more adapted to a changing environment."

Evolution can refer to the whole life cycle of a software system from its development to its closedown. Therefore, it includes the maintenance activities. In contrast to revolution it emphasizes the character of stepwise, continuous changes. Evolution includes subsequent versions of a software system, which thus also spans several maintenance phases. In their staged model of the software lifecycle (see Figure 3.1) Rajlich and Bennett [RB00] distinguish *evolution* as a stage, which follows *initial development* and allows evolutionary changes, from the stage *servicing*, which is characterized by the loss of evolvability. In this regard evolvability means to preserve the possibility for evolutionary changes of the software system for the long-term.

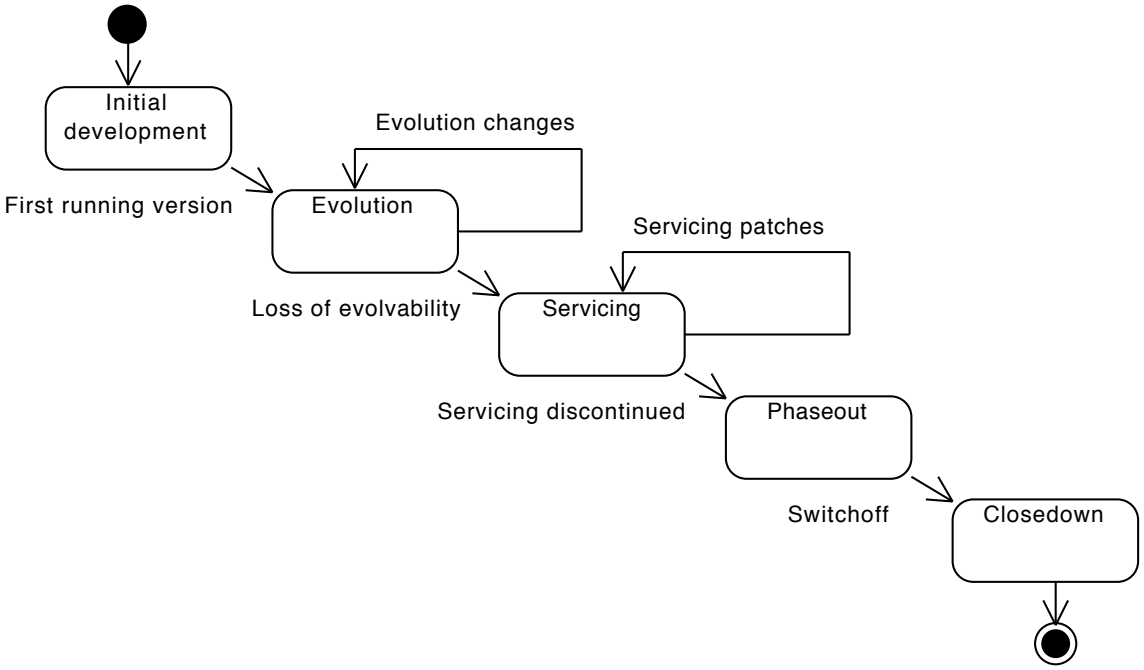


Figure 3.1: The Staged Model of the software lifecycle according to [RB00]

Next to maintenance activities, evolutionary changes typically comprise structural modifications as well. Evolution activities are, for example, the integration of new technologies, such as wrapping an existing system as a service in a service-oriented architecture, the realization of new requirements like an increase in scalability and multilingualism for global operation, or changing an architectural style, such

as the introduction of a new abstraction layer for persistence. The emphasis is not on keeping the system operating but on evolving its architecture, for example, because of a changing environment with changing or new requirements and standards. This often is connected to re-design, re-structuring, and re-factoring. Evolution is not limited to legacy systems. But it is also related to systems that still have a high business value and therefore are subject of enhancements for an adaptation to a changing domain. The demand for software evolution and a long life-time of complex software systems goes beyond software maintenance because effort, strategic activities, and complexity are considerably more challenging.

Sometimes, software maintenance and evolution are used synonymously, especially if the maintenance definition is a broader one not only restricted to post-delivery activities as in the ISO 14764. However the author of this thesis argues for a deliberate usage with keeping some possible differences in the meaning of the terms in mind. Further discussions about the terms evolution and maintenance can be found in [MD08] and [MFRP06].

3.3 Maintainability

The term maintainability is related to the effort for changes and is standardized in the ISO 9126 as *"the capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications"*. In the quality model of the ISO 9126 maintainability is refined into the subcharacteristics analyzability, changeability, stability, testability, and maintainability compliance.

Maintainability is a desired property of software, which denotes how much effort is related to maintenance activities. If software shall be maintainable, then it must be easy to analyze and allow changes. For the validation of modifications, for example, the stability of interfaces is important, as well as the testability of the system as a whole or of its parts.

With the refinement of the quality model, the quality attribute maintainability gets a bit more tangible. The subcharacteristics of the ISO standard however should not be considered as the only ones. An essential issue with maintenance of software is program comprehension, which can account for more than 50 percent of the maintenance effort [BR00]. That is why understandability is also considered

as a subcharacteristic of maintainability [BBL76]. Understandability in turn depends on quality attributes of the code and documentation, such as structuredness, consistency, or conciseness. Furthermore, traceability plays an important role for maintenance because traceability links support tracing design decisions and hence understandability.

Broy et al. [BDP06] provide another two-dimensional quality model for maintainability where they relate factors of the maintenance context, such as infrastructure, system properties, or organisation, to maintenance activities in order to derive controlling measures. This so-called activity-based quality model shall encounter difficulties that arise in practice when the software quality is evaluated via metrics that are related to the quality goals [DWP+07].

High maintainability increases the lifetime of software by a reduction of development risks. Nevertheless, this property akin to maintenance emphasizes the recent short-term effort for changes and does neglect the evolutionary changes for the long-term upkeep of software. Therefore, evolvability should be considered as a separate quality goal [Bod09], which is discussed below.

3.4 Evolvability

In contrast to maintainability there is no standardized definition for evolvability. Many of the definitions for evolvability hardly differ from that for maintainability. However, a distinction between maintainability and evolvability can be made for the same reasons as with maintenance and evolution.

Maintainability and evolvability have a different focus on related activities that should be supported. Activities related to maintainability are bug-fixing and small modifications, furthermore maybe sometimes but not regularly the implementation of new requirements or extended modifications. However, evolvability has a broader focus. It additionally and deliberately covers activities as the integration of new technologies, large structural changes of the architecture, or regular refactorings because of cyclically new requirements. Moreover, a software system is not necessarily easy to evolve if it is easy to maintain. Maintainability, for example, can be improved via code quality without considerable impact on the software's ability to evolve.

Several authors already dealt with the definition of evolvability, for example, Rowe

et al. [RLL98], Cook et al. [CJH01], as well as Ciraci and van den Broek [CvdB06]. A definition that especially considers typical evolutionary aspects, such as structural changes and architectural integrity, is the one of Breivold et al. [BCE07]. In this thesis the following definition based on Breivold et al. [BCE07] and Rowe et al. [RLL98] is used.

Definition *Evolvability is the ability of a software system throughout its lifespan to accommodate to changes and enhancements in requirements and technologies, that influence the system's architectural structure, with the least possible cost while maintaining the architectural integrity.*

High evolvability increases the lifetime and decreases development risks, which enables the progressive realization of business goals. Thus high evolvability can avoid turning a software system into a so-called legacy system [MM98]. Activities for the increase of evolvability also improve maintainability, the inverse does not hold. Examples are the development of a plug-in interface that enables the exchange of components but not the evolution as a result of changes, or the introduction of an abstraction layer for distribution or platform change.

For evolvability, similar to maintainability, subcharacteristics can be defined. Breivold et al. [BCE08] in this regard discuss analyzability, integrity of the architecture, changeability, portability, extensibility, testability, as well as domain-specific attributes. For evolvability, in contrast to maintainability, for example, extensibility plays a very important role, not only changeability. Reusability should not be neglected for evolvability as well. Furthermore, architectural integrity should be ensured for the prevention of architectural decay by considering flexibility and variability. Moreover, akin to maintainability, traceability is important. Documented and traceable design decisions are inevitable for the comprehension of the previous development and, for example, facilitate the decision on how to integrate a new feature.

The subcharacteristics named so far are mainly focused on artifacts as source code, models, and descriptions, but quality attributes regarding the development process have to be considered for evolvability as well. Such attributes are, for example, the maturity of the development process regarding planning, analyzability, or self-optimization, the productivity of the process, and further properties concerning

the degree of feedback control, knowledge management, or error prevention.

A refinement of evolvability into subcharacteristics similarly to the quality model of ISO 9126 significantly contributes to the comprehension of the term. However, the refinement of Breivold et al. is considered to be not sufficient by the author of this thesis. Besides, in the current revision of the standard ISO 25010:2011 [Int11] as the successor of ISO 9126 there is no improvement regarding evolvability. Therefore, one of the contributions of this thesis is the consolidation of a set of subcharacteristics for evolvability (see Chapter 5).

Chapter 4

Overview of the Approach

Based on the challenges listed in Chapter 1, the evaluation of the state-of-the-art approaches from Chapter 2, and the discussion of the terms related to evolvability in Chapter 3 it can be argued that the systematical support for the design of evolvable software is still insufficient. Therefore, the remainder of this thesis describes a comprehensive architectural design approach for evolvability-oriented software to overcome the limitations of the existing methods and concepts. This chapter presents an overview of the approach developed in this thesis and deals as an anchor for the following chapters that explain its concepts in more detail.

Section 4.1 refines the goals of this thesis from Chapter 1 based on the limitations of the existing approaches, which are summarized first. In Section 4.2 an overview of the concepts in this thesis is given as an introduction to the details that follow in the next chapters.

4.1 Refinement of the Goals of this Thesis

In Chapter 1 several challenges of today's software development were listed that motivated the general goal of this thesis to *provide appropriate means for an architectural design method that allows to develop evolvable software by especially considering quality goals and supporting software changes*. To achieve this, further goals (**G1** to **G9**, see pp. 4–5) were established. This section first summarizes the limitations identified during the evaluation of the state-of-the-art. Afterwards, the goals for the proposed approach of this thesis are refined based on this knowledge.

4.1.1 Summary of the Benefits and Limitations of Existing Approaches

During the state-of-the-art analysis several valuable approaches regarding the analysis and realization of quality goals were identified. However, also limitations of the approaches were determined.

The goal-oriented modeling approaches from requirements engineering (GORE) are very important means for dealing with quality goals. But they do not go much beyond requirements engineering. From an architectural point of view they do not consider relevant design principles and technical constraints to find appropriate solutions. To specify a software architecture further means and activities for architectural analysis are necessary. The GORE approaches and architectural analysis should be intertwined because some requirements can constitute constraints and hence restrict the design space at an early time. Vice versa technological aspects can influence requirements. Just to handover the goal models to an architect for the design is not enough.

The EMPRESS approach is valuable for treating quality goals due to its questionnaires and checklists. Moreover, it provides some guidance for finding solutions by relating quality attributes to scenarios and further to architectural means, such as patterns. However, it does not explain how to resolve conflicts. The proposed design activities are rather sketchy, and guidance for the specification of concrete components and interfaces is not provided.

Unfortunately, especially evolvability is not considered in an advanced way in the analyzed approaches. EMPRESS discusses maintainability but has inconsistencies in this regard. At least there are works that discuss a refinement of evolvability akin to quality models (see Chapter 3). However, this is not related to design approaches and appropriate design principles or solutions to support decision-making.

The examined architectural design approaches QASAR, ADD, S4V, and Quasar of course somehow deal with quality goals and design quality, though either not especially with evolvability. QASAR expresses the concept of fulfilling quality goals by functional solutions, but treats quality goals as secondary after functional requirements. ADD concentrates on quality goals but does only provide guidance for a conceptual architecture and therefore has to be continued by other means. Moreover, if applied rigorously, it requires design patterns and tactics especially for

evolvability, which are not available as a well-established set, yet. Furthermore, its recursive nature forces strict top-down design and hampers bottom-up experimentation. The Global Analysis of the S4V approach is an important contribution for bridging from requirements to architectural design. But there is further potential for improvement. For example, a coupling with the GORE approaches could improve the identification of influence factors. Moreover, Global Analysis leaves open, how to find the solutions and strategies described by the issue cards. The Quasar method does not concentrate on quality goals but particularly supports certain design principles, such as separation of concerns, and can complement other architectural design approaches for detailed component and interface design. All in all the design approaches can be improved in their goal-orientation and guidance for choosing solutions and resolving conflicts.

Regarding traceability a comprehensive approach spanning several development phases and heterogenous artifacts, for example, considering goal modeling and particularly design artifacts, is an unresolved problem. Besides traceability has to be efficient with as less additional effort as necessary. This is the reason why automation has to be enforced. For automated traceability link establishment rule-based approaches and information retrieval techniques should be combined as well as with an event-based approach for maintaining links. Appropriate semantics for traceability links with specific link types is another issue to be considered to a greater extent. Moreover, adequate tool support that integrates with regular CASE tools and development environments is needed for practicability and useability. Only if all this is realized, change impact analysis can be enabled in an easy and well-established way.

4.1.2 Research Goals for the Proposed Approach

Knowing the limitations of the existing approaches the goals for the approach of this thesis can be refined. Regarding the goals **G1** and **G2** it can be said that a completely new methodical approach is not necessary. The refined goal is:

RG1 Combine the existing architectural analysis and design approaches and integrate them with the GORE approaches to follow a goal-oriented way for a comprehensive and systematic design regarding evolvability. Provide means and activities for its support.

Regarding goal **G3** a discussion about the term evolvability was already started in Chapter 3. However, it is necessary to:

RG2 Provide a consolidated view on the quality goal evolvability with its characteristics and relations to other quality goals as well as a strategy for the realization of evolvability during architectural design.

The goals **G4** and **G5** are partly addressed by the GORE approaches regarding modeling goals and their influences on each other. However, in terms of design they are not sufficient, which requires to:

RG3 Establish a concept to integrate and adopt the GORE approaches for architectural design to provide systematic guidance for the treatment of quality goals and their interdependencies, for conflict resolution, and for design trade-offs.

This concept can benefit from the utilization of design principles and solution instruments that shall be analyzed according to the goals **G6** and **G7** and collected in a catalog (**G8**). In order to provide effective guidance for decision-making it is necessary to:

RG4 Determine a quantitative rating of the impact of the design principles and solution instruments on the quality goal evolvability as a decision criterion, and collect this information in a catalog that can be utilized in the concept from **RG3**.

Regarding the traceability concept of goal **G9** the challenges identified in Chapter 2 should be addressed. Therefore, the refined goal is to:

RG5 Develop a concept for design traceability, which enables comprehensive (semi-) automatic identification, recording, and maintenance of intra and inter model dependencies, includes goal models as well as design artifacts, provides a defined semantics based on specific link types, and hence facilitates change impact analysis and evolutionary development.

4.2 Proposed Approach

The approach of this thesis is explained as follows. This section describes how the approach addresses the refined goals from Section 4.1.2. First, an overview of the

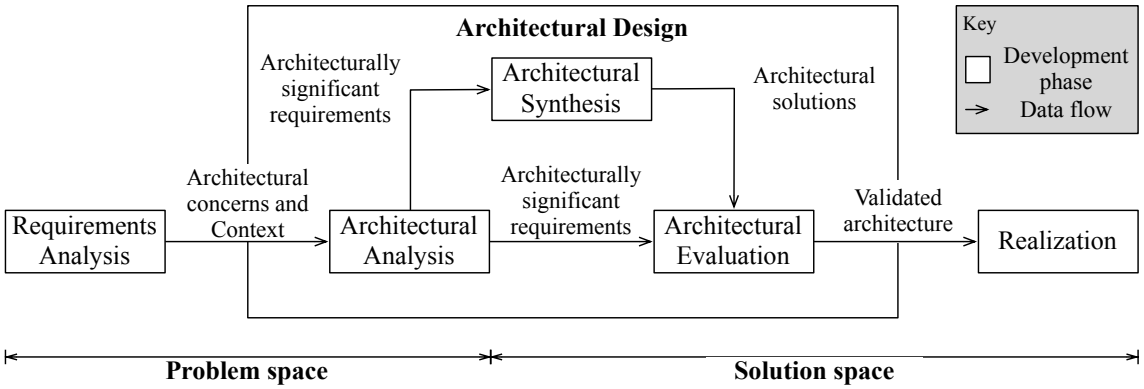


Figure 4.1: Overview of development phases and data flow

phases, artifacts, and the methods that are integrated in this approach is given. Then, the concepts that are described in the following chapters are introduced.

Figure 4.1 gives an overview of the development phases related to architectural design and the data flow between the phases. The architectural design is refined into architectural analysis, synthesis, and evaluation, which represents a general model determined from several design methods as already mentioned in Chapter 2.

Architectural analysis takes as input architectural concerns, such as functional and non-functional requirements, as well as context information, e.g., business goals of the enterprise, and defines the architecturally significant requirements. Architectural synthesis performs the transformation from the problem space to the solution space from the perspective of an architect. The activities during synthesis create architectural solution elements (e.g., architectural building blocks) from the architecturally significant requirements utilizing design principles and patterns. These architectural solutions are then subject of the architectural evaluation phase. The output, a validated architecture, can be realized afterwards. Table 4.1 shows some details of the development phases relevant for this thesis, such as typical activities, artifacts, entities, and standardized modeling notations.

The data flow in Figure 4.1 might indicate a sequential flow of the development phases and activities. However, it rather shows only the forward engineering direction. Instead, the activities of the design phases are typically performed in small leaps and bounds as architects move between them. This apparently random process is driven by an implicit or explicit backlog comprising small issues and problems that need to be solved to get from the problem space to the solution space (cf. [HKN+07]). The backlog concept is mainly known from agile methods as Scrum [SB01].

Table 4.1: Development phases with activities, artifacts, and standard modeling notations

		Architectural Design			
Development Phase	**Requirements Analysis**	**Architectural Analysis**	**Architectural Synthesis**	**Architectural Evaluation**	**Realization**
Input	Stakeholder input, business goals, quality models	Artifacts from previous phase Factors, analysis patterns	Artifacts from previous phases Principles, patterns, tactics, guidelines, heuristics	Artifacts from previous phases	Specified architecture
Activities	Requirements elicitation and specification, goal modeling	Identification of architectural drivers or influencing factors, Identification of issues	Creation of candidate architectural solutions, identification of subelements, instantiation of patterns, documentation of views	Ad-hoc evaluation, scenario-based assessment, discovery review, simulation	Implementation, configuration, deployment
Artifacts or means	Use case description and diagrams, scenarios, goal models, checklists, questionnaires	Factor tables, issue cards, list of strategies	Architectural views, prototypes	Utility tree, metrics	Programming model, coding style guide
Entities	Functional and non-functional requirements, constraints, goals, softgoals	Architectural drivers or key requirements, influence factors, strategies, scenarios	Architectural building blocks (subsystems, interfaces, components, packages, classes), libraries, frameworks, strategies, software categories	Analysis questions, scenarios	Source code, configuration items
Integrated approaches	GORE, EMPRESS	GORE, Global Analysis, Scenarios, Use cases	QASAR, ADD, EMPRESS, Quasar	ATAM	
Standard modeling notations	URN		UML		
	Problem space		**Solution space**		

The goal-oriented architectural design method of this thesis focuses on a forward engineering perspective to initially create a software architecture. However, it can be extended by a reengineering perspective. More specifically, the method targets on the transition from requirements analysis to architectural analysis. It emphasizes the analysis and synthesis phases to perform the transition from the problem space to the solution space from an architect's perspective. The range of the problem and the solution space is illustrated in Figure 4.1 and Table 4.1.

To achieve the refined goal **RG1**, the method adapts and integrates the concepts from existing approaches as follows. The quality goals are analyzed using the GORE approaches and specified with the standardized URN. The guiding theme for the transition from the problem space to the solution space is the transformation of quality goals into functional solutions as proposed by Bosch's QASAR method. For the architectural analysis phase mainly the Global Analysis of the Siemens Four Views approach with its factor tables and issue cards is used to identify the architecturally significant issues. The goals from the goal models can be used as input for the influence factors as well as for the identification of candidate solutions during synthesis. Scenarios are used to specify quality goals in more detail. Besides, the influence factors of architectural analysis can impact the goal-modeling. The architectural synthesis phase combines the EMPRESS approach and its architectural means, the ADD method and its architectural tactics, as well as further ideas to create a conceptual architecture. This is complemented with concepts of the Quasar method, such as the software categories, for the design of detailed components and interfaces. The details of the goal-oriented design method by means of activities and artifacts is described in Chapter 6.

As a major contribution this thesis introduces a guiding scheme for goal-oriented architectural design—the Goal Solution Scheme. The scheme spans several phases: requirements analysis (partly), architectural analysis, and architectural synthesis. It serves as a systematical concept guiding the transition from the problem space to the solution space. The scheme is inspired by and integrates the GORE approaches with architectural design. It provides guidance for conflict detection and resolution as well as for decision-making on solution instruments as required by the refined goal **RG3**. Furthermore, the scheme is used to provide a consolidated view on evolvability and its subgoals as demanded by **RG2**. Moreover, the scheme is the base for rating solution principles and instruments regarding their influence on evolvability

to establish a catalog of solution instruments according to the refined goal **RG4**. The Goal Solution Scheme is explained in detail in Chapter 5. The utilization of the scheme during the goal-oriented architectural design is explained together with the design method in Chapter 6.

The remaining refined goal **RG5** is addressed in Chapter 7. This chapter discusses the concept for tracing the dependencies between the various artifacts from the goal-oriented architectural design method. Inspired from state-of-the-art approaches of Cleland-Huang et al. and Spanoudakis et al. a (semi-) automated, rule-based approach is used together with the information retrieval technique n-gram matching to establish traceability links. Linked artifacts are *a*) the goal models noted with the GRL of the URN standard, *b*) the factor tables and issue cards from Global Analysis, *c*) UML models for design specification, as well as *d*) ontologies modeled with the Web Ontology Language (OWL) as a semantic net of important terms from different phases. The dependencies between the entities of those artifacts (URN models, factor tables, issue cards, UML models, OWL ontologies) are represented as links of specific types to increase the semantic meaning of the links. For this purpose a traceability metamodel is created and link types are clustered and collected in a catalog.

Additionally, in Chapter 8 the tool support for this combined design and traceability approach is presented. A tool called EMFTrace is used to integrate the models from the different development phases in a repository and to establish the traceability links that are identified from the dependencies between the models. EMFTrace is extended by EMFfit to support the architectural analysis with factor tables and issue cards as well as the linking between the goals of the problem space and the design artifacts of the solution space.

Chapter 5

The Goal Solution Scheme

This chapter introduces the Goal Solution Scheme (GSS) as a concept that drives the goal-oriented architectural design and guides the developer from the problem space to the solution space. Section 5.1 describes the general concept of the GSS and its contribution. In Section 5.2 the establishment of the Goal Solution Scheme with its transitions leading from goals to solutions is explained in more detail and on an exemplary case study regarding several quality goals. Furthermore, Section 5.3 describes the GSS for the quality goal evolvability as a central example and how the GSS leads to a catalog of reusable solution instrument's for the architect's toolbox. Parts of this chapter were already published in several papers [BBR09, BFKR09, BR10, BR11].

5.1 General Concept

The Goal Solution Scheme (GSS) was developed to guide the software architect in appropriately dealing with quality goals during architectural design. The scheme indicates and guides how the architectural design steps should be performed for the transition form the problem space to the solution space. The GSS represents a mapping between elements of both problem space and solution space by explicit dependencies. If traceability links are established according to these dependencies and if they are managed in a repository, the organization of that repository reflects the Goal Solution Scheme. The GSS has some similarities with and was partly inspired by goal graphs from the GORE approaches (see Chapter 2). However, it extends them by an explicit consideration of design principles and a quantitative evaluation

of the influence of functional and technical solutions on effort-related quality goals. The Goal Solution Scheme gives information about how to deal with quality goals or how to resolve conflicts between competing goals using design principles, and it facilitates decision making for solution instruments. The means mentioned by the EMPRESS method as well as the architectural tactics of ADD are both covered by the design principles. Beyond, the scheme is related to the software architect's toolbox, where previous solutions are collected and stored according to principles and goals.

5.1.1 Structure of the GSS

The structure of the GSS is shown in Figure 5.1. The layers and transitions of the scheme correspond to the artifacts in the development process and the phases requirements analysis, architectural analysis, and architectural synthesis as introduced in Chapter 4. The scheme maps quality goals covered by layer I to their subcharacteristics represented by layer II. Together with project constraints they represent the problem space. Instead, the layers III and IV represent the solution space. The GSS maps the subcharacteristics from layer II to supporting solution or design principles covered by layer III. Furthermore, the principles are mapped to solution instruments covered by layer IV. Layer IV represents the design space or the architect's toolbox including solution instruments on different levels of abstraction. These can be functional and technical solutions as architectural building blocks, patterns, or existing frameworks implementing the design principles.

The Goal Solution Scheme has a graph structure similar to a tree, however, in reality due to scattering and tangling there are not always one-to-one relations between the elements of the different layers. The crossing arrows in the figure indicate the existence of many interdependencies and trade-offs. The relations between the layers on the one hand represent the positive or negative influence on the quality goals that is utilized during decision-making. Consequently, the scheme shows the propagation of quality goals, e.g., security, evolvability, or usability, to a software architecture along the design process. On the other hand the relations of the scheme are dependencies between development artifacts, such as refinements. These dependencies can be represented as traceability links established during design. These traceability links then carry the design decisions. The establishment of the traceability links is discussed later in Chapter 7.

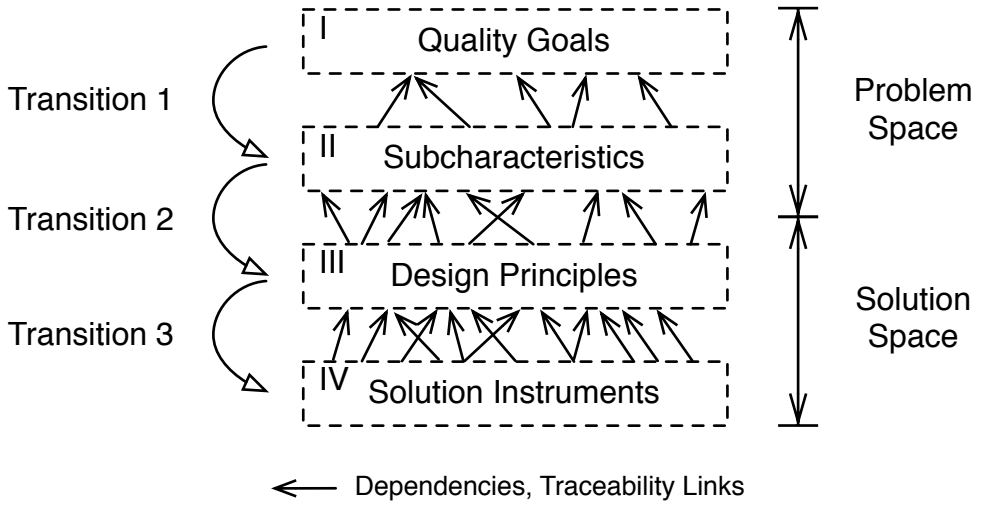


Figure 5.1: Structure of the Goal Solution Scheme

5.1.2 Transitions of the GSS

Transition from Quality Goals to Subcharacteristics The upper transition in the GSS represents a refinement of the soft and intangible quality goals into subgoals or also called subcharacteristics. With this refinement conflict detection and resolution are enabled, because further interdependencies and, hence, conflicts are discovered and can be resolved on the next levels. This transition step in practice is usually performed by requirements engineers but rarely by software architects if it is performed at all (compare the evaluated state-of-the-art design methods). The understanding of this transition, however, is important for the software architect because he has to contribute to the prioritization of the quality goals. Later he is responsible for the selection of proper design principles and solutions for the realization of the quality goals. Therefore, this transition step is included in the goal-oriented design approach of this thesis and utilizes the concepts of softgoals and contribution links from the GORE approaches.

Transition from Subcharacteristics to Design Principles The second transition of the scheme guides the developer from the quality goals and their subcharacteristics to the architectural design principles representing the transition from the problem to the solution space. This constitutes a novelty step regarding the design progress towards a solution in comparison to the GORE approaches. For this transition the influence of the design principles on the quality goals' subcharacteristics is

determined. This transition corresponds to the architectural analysis phase, which facilitates the selection of appropriate solutions in the next transition step.

Transition from Design Principles to Solutions Instruments In the lower transition of the GSS solution instruments from the architect's toolbox, such as architectural patterns or existing frameworks as well as technical components, are related to the design principles. This transition represents the actual architectural synthesis phase. The solution instruments of layer IV are evaluated quantitatively regarding their influence on the design principles. For each instrument the impact values are stored in a catalog together with information about preconditions for their applicability. During decision-making the preconditions of the instruments are compared with architectural constraints for pre-selection. Then, together with the influence of the principles on the subcharacteristics from the preceding transition the influence of the instruments on the quality goals can be determined, which facilitates an appropriate choice of solution instruments. When choosing solution instruments, decisions are made about how design principles and quality goals are realized.

5.1.3 Contribution of the GSS

The Goal Solution Scheme has several advantages that constitute its contribution.

1. It facilitates the prioritization of quality goals for decision-making during the architectural design process.

2. The scheme eases the comprehension of quality goals by their refinement into subcharacteristics utilizing the concepts of the GORE approaches.

3. Furthermore, the GSS supports the resolution of conflicting quality goals by an identification of potential trade-offs or synergies and by prioritizing the alternatives.

4. It proposes a fine-grained sequence of design steps and, therefore, guides the software architect during the design process and enables tool-supported decisions and automation.

5. Moreover, the scheme facilitates the establishment and maintenance of traceability links related to the design activities. The traceability links then carry the information about the design decisions.

6. A reduction of the traceability link complexity is achieved because one or a few principles and solution instruments implement one subcharacteristic. In an ad-hoc design quality goals would be implemented in a more scattered manner leading to a much higher number of links.

7. Besides, the established mapping between goals, principles, and solution instruments simplifies checks for accuracy, completeness, and consistency of the traceability links by a comparison between chosen solution instruments and relations within the Goal Solution Scheme.

8. As a further result, the GSS provides an alignment of principles and functional solutions; it classifies solution instruments and components according to their contribution to quality goals, and it provides a stock of reusable solution instruments to the architect and the designer. The solutions serve as a source of proposals for design alternatives during decision-making.

5.2 Establishment of the GSS

This section describes the establishment of the Goal Solution Scheme. It explains in more detail the transitions that were introduced in the preceding section. For simplicity in this section the relations between the elements of the different layers are only weighted qualitatively. An approach for the quantitative evaluation of the relations is discussed in the next section. The description in this section is based on a case study from a reengineering project for a Manufacturing Execution System (MES) that is restructured according to the principles of Service-Oriented Architecture (SOA) [Erl07]. In general, the information that helps with the establishment of the scheme can be gathered from literature review and from the experience of experts.

Service-oriented MES Case Study A Manufacturing Execution System manages the manufacturing in modern flexible plants [Ver07]. It is connected to Enterprise Resource Planning (ERP) systems, which handle manufacturing plans and actions, and represent a business perspective, while the MES is able to manage the manufacturing actions on a more fine-grained level. A MES has access to the abilities and the limitations of the real manufacturing processes and, therefore, it is

able to optimize them, and simultaneously it provides an increased flexibility. The MES covers tasks like detailed scheduling and process control, the management of machines, material, and personnel, etc. In the case study the focus lies on the integration between ERP and MES. The requirements to the interface between both are defined by the ANSI standard ISA-95 [ANS]. As the platform for the MES interface the Enterprise Service Bus (ESB) [Cha04] has been chosen, in a style similar to a middleware. Figure 5.2 shows the integration interface and its environment.

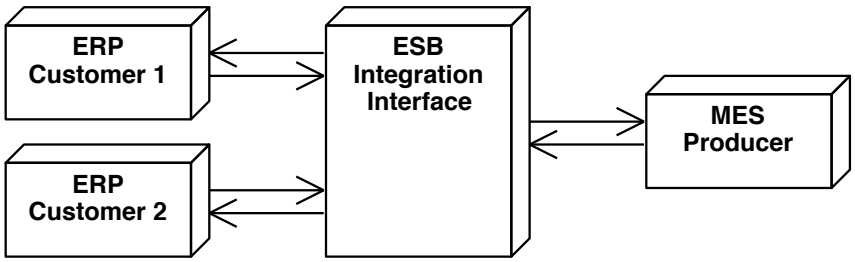


Figure 5.2: Overview of the integration interface

Some of the major quality goals a MES has to fulfill are high flexibility and scalability, time behavior, as well as security. As an example, security is especially important for the information flow control. In this case, it is important to protect the business-critical private information of the customer C_1 from an unauthorized access by its competitor customer C_2. Even if both give manufacturing tasks to the producer P, no details about the order must be disclosed to the competitor through the ESB or the MES, for example, details concerning number of units, delivery terms, and technological process. Flexibility is necessary regarding different planning algorithms, control principles, and regarding the integration with a variety of machines and ERP systems. The requirements for scalability arise from the need for mastering complex manufacturing tasks with a high number of variants and elements, and for the interoperation with multiple different ERP systems because of outsourcing.

5.2.1 Quality Goal Refinement and Elaboration

Quality goals are often vague and intangible. At first sight for an architect it is to hard to know how they interact with each other or how to realize these goals. The several quality goals of a software system often are competing and conflicting in their interdependencies. Such conflicts have to be resolved by prioritization or a

trade-off. If this is not possible on the top-level, a resolution is attempted after a refinement of the goals. By refining the quality goals, they get more tangible, vague interdependencies can be concretized, and previously hidden dependencies can be made explicit. The mutual impact of the relations as well as conflicts and synergies can be detected.

The refinement of the quality goals is covered by the first transition in the Goal Solution Scheme. This eases the comprehension of the goals and makes them more specific. The refinement and specification of quality goals can be performed with the GORE approaches evaluated in Chapter 2, such as the NFR framework, i*, or the URN. For the refinement standards as the ISO 9126 [Int01] can help. Furthermore, the Goal Question Metrics (GQM) method can be applied to identify subcharacteristics. This structured querying technique helps to analyze influences on a goal [BCR94].

According to the NFR framework, quality goals have a type and a topic. As an example, the requirement *security of integration interface* has the type *security*, which indicates the specific quality goal, and the topic *integration interface*, which targets at the subject. Quality goals (or non-functional requirements) can be refined regarding type or topic. The refinement of maintainability mentioned above is a refinement regarding the type of the quality goal. For the case study all refinements are regarding type as well. The integration interface is the target.

Discussion for the Case Study In our case study the top-level quality goals are flexibility, scalability, and security. A consideration of their relationships on this level leads to the assumption that flexibility and scalability are in a rather synergetic relation to each other because they both deal with change. On the other hand security might be conflicting to flexibility and scalability because it implies restrictions of the information flow and the data access. This guess has to be checked in the next steps. For a precise analysis and a solution for the MES project, a refinement, more specifically an AND-decomposition, has been performed. Figure 5.3 shows the refinement. It is presented using the URN for softgoals and links.

For flexibility, there is a definition in the IEEE standard glossary of software engineering terminology [Ins90], although a detailed discussion of the subcharacteristics is missing. Regarding flexibility some discussion can be found in the literature [ZZ02, NNG97, EM06, Mor06], which led to the elaboration of the subcharacteris-

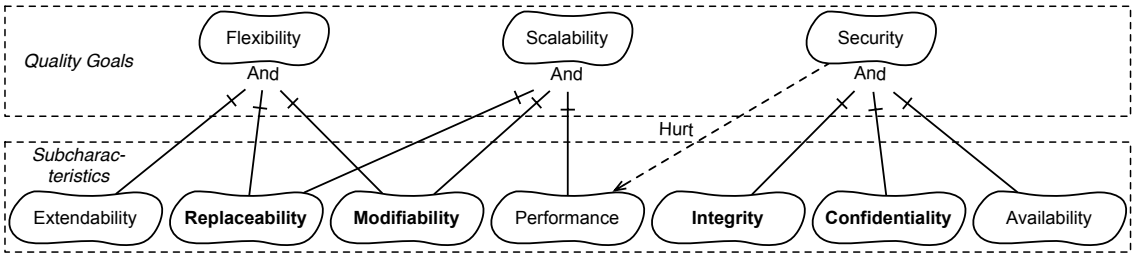


Figure 5.3: First transition of the Goal Solution Scheme for the case study

tics extendability [Ins90], replaceability [Int01], and modifiability [BLBvV04] for the example. The latter two got a higher priority and, hence, are in the focus of the further description.

Scalability is lacking a definition by a standard; however, some works discuss this quality attribute [Bon00, Hil90, DRW06, DRW07]. Scalability is always concerned with performance, or efficiency in terms of the ISO 9126 [Int01], and how well a solution to a problem will work when the size of the problem increases. However, if a system performs well it is not necessarily scalable, too. Therefore, replaceability and modifiability are considered as subcharacteristics of scalability as well. If an MES has to face changes, for example, due to an increasing complexity of the manufacturing tasks, modifications are necessary. Moreover, it should be easy to replace parts of the whole system with more efficient ones if this is necessary to scale up and retain a high performance.

For security there are several definitions from the International Organization for Standardization (ISO), e.g., [Int05]. Chung et al. [CNYM00] comprehensively discuss its refinement in their NFR framework. In the case study, business critical data is not only processed by a shared cooperation platform but also transferred via a public, insecure communication medium. Thus, it is essential first to guarantee confidentiality and integrity of data within the borders of the shared cooperation platform and secondly to realize confidentiality and integrity of transferred data. In general, the most important subcharacteristics of security are confidentiality, integrity, and availability [And01].

Before the refinement, there already has been the assumption for a synergetic relation between flexibility and scalability, as well as a possible conflict between scalability and security. The conflict could neither be verified nor resolved on the top level, because both scalability and security are essential. However, after the

refinement of the quality goals illustrated in Figure 5.3, a resolution of the conflict and verification of the synergetic interdependency between flexibility and scalability can be tried. The latter can be verified by the common refined goals replaceability and modifiability. Beyond, a negative interdependency between performance and security is detected, because security mechanisms, as for example encryption, require extra operations, often are time consuming, and can hamper performance. This confirms the conflict; however, for a resolution a further refinement to principles is necessary. For the further design process of the case study the attention is drawn to the higher prioritized subcharacteristics replaceability, modifiability, as well as integrity, and confidentiality.

In the figure the above mentioned refinements are expressed using decomposition links of the type AND according to the URN. In the Goal Solution Scheme they are represented by traceability links of the link type `refine` (cf. the discussion of link types in Section 7.4). The HURT-contribution can be traced with links of the type `contribution` if necessary.

5.2.2 Filling the GSS with Principles

After the top-level goals have been refined into subcharacteristics in the first transition of the Goal Solution Scheme, they are more specific. But, they are still of a qualitative nature and still cannot be implemented directly. In the next step, the transition from the subcharacteristics to the design principles is performed, as introduced with the general concept of the GSS.

As a step from the problem space to the solution space, in the second transition, properties and principles and guidelines for good architectural design are assigned to the subcharacteristics. These principles and guidelines give hints or advice for the functional solutions. Of course, lots of principles exist and even more relations between quality goals and these principles are imaginable. Therefore, the designer has to analyze the subcharacteristics and to decide on suitable principles. It is always the case that there are different quality goals having symbiotic relations or, in contrary, competing with each other. In order to resolve conflicts, knowledge about the interdependencies between the different subcharacteristics is important. A goal model contains these dependencies and the trade-offs.

For illustration an example for a decision is discussed here. The principle of high encapsulation supports changeability. On the other hand, a strong encapsulation has

a negative influence on testability, because inaccessible attributes are hard to control. Because of the refinement from the first step, both changeability and testability are known to be subcharacteristics of maintainability and contribute to it. Now, by assigning encapsulation to these subcharacteristics the conflict becomes visible and can be considered. Frequently, multiple different principles contribute to the same subcharacteristic. In these cases a decision can be made, which principle is applicable or how to prioritize them.

Conflicting interdependencies between different quality subcharacteristics and architectural principles often are still not tangible enough. Then, they have to be elaborated further on the solution instrument level of the Goal Solution Scheme. This is necessary to be able to decide with clear rationale, which principle to choose to achieve the highest degree of goal fulfillment. Therefore, the principles are mapped further to solution instruments and the decision-making on how to resolve the conflict is postponed to the next step, when the criteria for adequate solution instruments are more precise than those for the principles. Based on the contributions of the solution instruments to the principles and the quality goals, the different alternatives can be weighed and the decisions, which alternatives to choose, can be made. For the GORE approaches also some evaluation techniques exist, which could be applied. Anyway, it is always reasonable to decide as soon as possible to reduce further effort.

Discussion for the Case Study For the cut-out from the case study, the second transition of the Goal Solution Scheme is shown in Figure 5.4. The subcharacteristics result from the refinement in the previous transition step. Starting from the higher prioritized subcharacteristics, appropriate principles are chosen. For the subcharacteristics replaceability and modifiability, we decide in favor of the architectural design principles *encapsulation*, *modularization*, and *loose coupling*. These principles are well known to support changes. Already Parnas [Par72] discussed the importance of modularization for changeability and flexibility, which is one of the most important quality goals here. Moreover, *service orientation* was identified to support encapsulation, modularization, and loose coupling. A service-oriented architecture obviously can help in this scenario, because loose coupling is one of its core principles. It further helps encapsulation and modularization. In Figure 5.4 the mentioned principles are related to the subcharacteristics replaceability and modi-

fiability by `contribution` links, denoting a positive influence. For those principles explicitly chosen by the architect, traceability links of the type `realize` can be established as well. This type of links is established, because the principles represent a step towards the solution of the quality goals, and to document the design decisions for choosing service orientation.

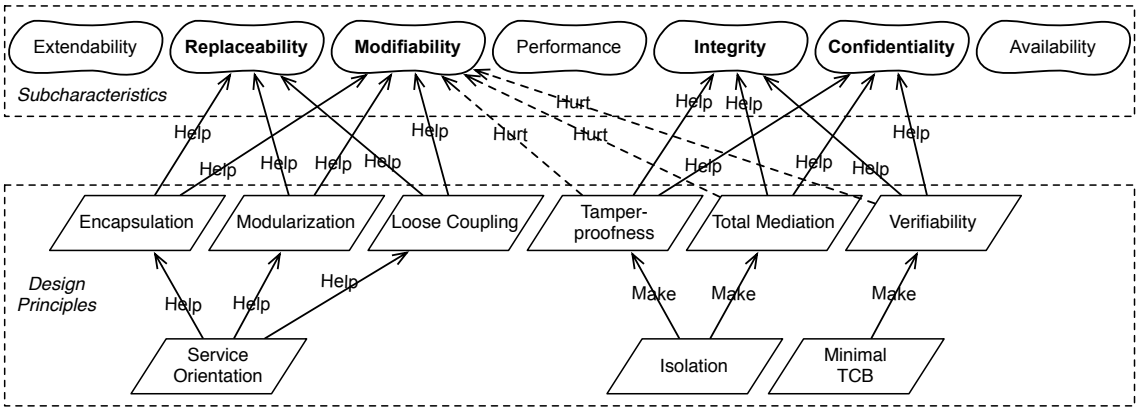


Figure 5.4: Second transition of the Goal Solution Scheme for the case study

The security subcharacteristics integrity and confidentiality are discussed as another example. To achieve the system's security goals, security policies have to be applied, as a comprehensive set of rules that are designed [GM82]. Security policies are applied to determine a so-called trusted computing base (TCB) [LABW92]. The TCB comprises the functional parts of a system that enforce and protect the security policy. For the implementation of a security policy and a trusted computing base, there are fundamental principles that refer to the so-called reference monitor concept [And72]. A reference monitor must be tamperproof, always invoked and small enough to be analyzable and verifiable, which is represented by the principles *tamperproofness*, *total mediation*, and *verifiability*. These reference monitor principles are further supported by *isolation* and a *minimal TCB* as principles for the architectural design. Isolation of the security relevant functions in the security architecture of a system is a necessary consequence to be able to realize a tamperproof reference monitor that cannot be bypassed [Gas88]. Correctness and completeness are additional necessary properties not further discussed here [Dep85]. These decisions and the causes again can be documented by traceability links of the type realize.

However, in this design step, conflicting relations between the security principles

and the subcharacteristic modifiability were identified as well. They are shown as HURT-contribution links. Modifications in the software architecture can negatively influence the minimality of the trusted computing base and vice versa. The other security principles are affected by changes as well. Tamperproofness can easily be breached if a modification is performed in a wrong way. Therefore, changes should only be made on those architectural parts that have not to be isolated due to security reasons.

These conflicting relations confirm the earlier assumption that security is in conflict with flexibility and scalability. However, at the principles level their interdependencies have been clarified and a much better understanding of the conflict is achieved than on the goal or the subcharacteristic level. Anyway, the conflict between the fundamental security principles and the subcharacteristic modifiability cannot be resolved in this transition of the Goal Solution Scheme. The conflict resolution has to be postponed to the next design step, when related solution instruments can be analyzed more precisely than the principles.

5.2.3 Filling the GSS with Solution Instruments

In the third transition step of the GSS the actual transformation of the quality goals to a functional solution is performed. This step is closely related to activities of the architectural design methods ADD and QASAR. A similar mapping of solutions to goals can also be found in the NFR framework and the i* framework, where operationalizations, or tasks respectively, are assigned to decomposed softgoals. However, in the GORE approaches, the architectural design principles are not considered as an intermediary means.

In the goal-oriented design method of this thesis solution instruments or solutions can be functional concepts or even existing technical components, which either support the realization of quality goals or completely fulfill some of them. In this third transition step of the Goal Solution Scheme a large number of solutions is possible. In order to find the most adequate ones, the designer weights the different alternatives, akin to the last transition step.

The explicit mapping from goals to principles and solution instruments classifies the latter ones according to their contributions to the quality goals. The GSS facilitates to build up a knowledge-base or catalog, which enables the incremental collection and the reuse of the solution instruments in a goal-oriented way.

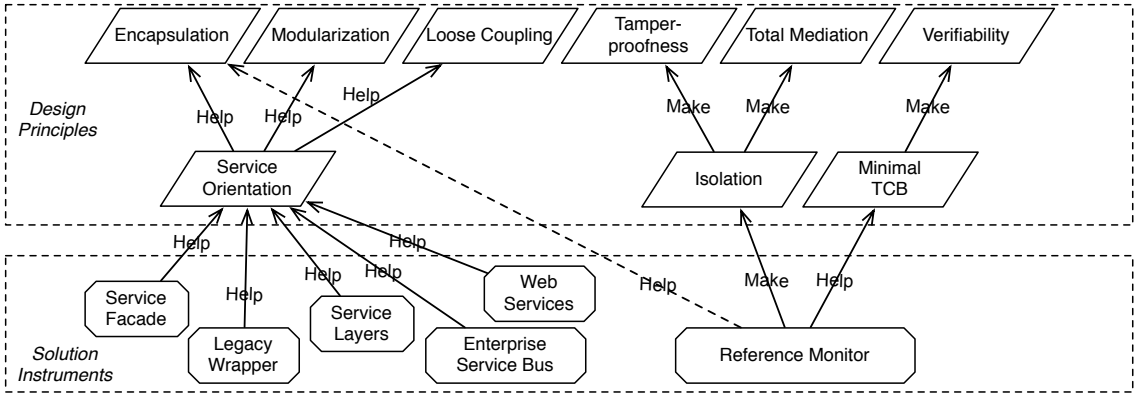


Figure 5.5: Third transition of the Goal Solution Scheme for the case study

Discussion for the Case Study Figure 5.5 illustrates a part of the Goal Solution Scheme for the transition from principles to solution instruments for the case study. To realize service orientation, and thus the principles encapsulation, modularization, and loose coupling, the solution of architectural components as *Web Services* and an *Enterprise Service Bus* (ESB) for the integration of the MES and ERP are chosen. The reason for these decisions is that well-defined web services according to the Service-Oriented Architecture (SOA) paradigm inherently reinforce those principles [Erl07]. A component-based Common Object Request Broker Architecture (CORBA), for example, could have been an alternative for a service-oriented architecture. However, for the case study a CORBA infrastructure was not available.

As an example from the case study, one realized service shall be mentioned. The service `MachineAvailability` can be used for the interaction of the detailed planning of an MES and the general planning of an ERP. Using this service the ERP can request status information about machines, like their availability. When the web services are implemented, architectural and design patterns, such as *Service Layers* [Fow02, Erl08], *Service Facade*, or *Legacy Wrapper* [Erl08], contribute to the realization of the principles, and hence, to the quality goals.

The application of a *reference monitor* solves the integration of the security aspects. The security principles are integrated with the help of the ESB to gain control of the communication between the MES and ERP system and to isolate the security-relevant architectural parts. Aside, it must be considered to keep the TCB as small as possible. A discussion on the integration of security with web services can be found in [FK08]. With this kind of solution the conflict between the security principles and the subcharacteristic modifiability, which has been detected in the

last transition step, cannot be completely resolved. However, as a trade-off, it can be implemented in a controlled way by controlling access to the security-relevant functionality. Hence, the realization of a reference monitor not only positively contributes to the reference monitor principles. But it also has a helping correlation to encapsulation, and therefore, even modifiability despite the conflicts.

As an alternative to the reference monitor, in an ad-hoc approach or according to the discussion by Chung et al. [CNYM00], one could have considered only multiple single solution instruments for security purposes, such as encryption mechanisms or roles and rights. Of course, these solution instruments can contribute to confidentiality and maybe integrity. However, as a drawback, without considering the reference monitor principles the system would be much more vulnerable.

In this transition step of the GSS again all decisions about solution instruments for the design principles can be made explicit by traceability links. As shown in Figure 5.5 all solution instruments are related to the corresponding architectural principles. The chosen solution instruments, such as the patterns Service Layers, Service Facade, and Legacy Wrapper, can be traced with links of the type `realize`. Additionally, elaborated alternatives not discussed here can also be linked as `contribution` and may be reused later. Figure 5.5 depicts only the mentioned and chosen solution instruments. Actually, much more solution instruments are available from the toolbox, and the architect can easily extend them by additional ones.

5.3 Establishment of the GSS for Evolvability

So far the Goal Solution Scheme was discussed for several quality goals. This section presents how the GSS is applied in more depth for the goal evolvability. In Section 5.3.1 an evolvability model is described representing the first and second transition of the GSS. To establish this model, first, subcharacteristics for evolvability are elaborated based on existing approaches and the discussion from Section 3.4. Second, architectural design principles are evaluated regarding their impact on the subcharacteristics of evolvability. Then, in Section 5.3.2 the impact of architectural solution instruments on evolvability is determined via the evolvability model. This step represents the third transition of the GSS and results in a catalog of reusable solution instruments for architectural design.

5.3.1 Evolvability Model

The evolvability of a software system is a property referring to the effort concerning different aspects of its evolution (see also Section 3.4). This effort can be determined by the help of several subcharacteristics as refinements of evolvability, which can be defined by a quality model. This model was partly introduced in [BBR09] and already published in [BR10].

5.3.1.1 Refinement of Evolvability to Subcharacteristics

The refinement of the quality goal evolvability into subcharacteristics according to the first transition of the GSS is an extension of existing works. Contrary to maintainability only a few works deal with quality subcharacteristics for evolvability. Ciraci and van den Broek [CvdB06] discuss modifiability, maintainability, and evolvability, whereby modifiability is seen with a broader scope depending on its definition. Cook et al. [CJH01] suggest few subcharacteristics taken from the ISO 9126 standard [Int01], which were included in the thesis' model. Breivold et al. [BCE07] discuss evolvability subcharacteristics regarding their importance for software developing organizations, and regarding a cost-effective evolution of software. In [BCLL08] the subcharacteristics are related to attributes of the software system to facilitate measurability. However, for quantification purposes of the relations some inconsistencies in categorization and granularity had to be refined for this thesis. Moreover, traceability must be considered as a subcharacteristic because it is an important quality factor for evolutionary development [DG02]. The presented subcharacteristics also strongly correlate to what Matinlassi et al. [MN03] call evolution qualities and additional characteristics (e.g., traceabiliy, variability) for specifying the quality goal maintainability.

The subcharacteristics of the evolvability model are described in Table 5.1. Extensibility, variability, and portability are seen as subordinated to changeability, which is expressed in the table by indentation. Changeability and modifiability are used synonymously. The presented evolvability model might not be completely congruent with other classification schemes in the literature, which is not possible in any case because there are many viewpoints on this topic. At least it is based on clear definitions, tries to consolidate existing works, and can be tailored according to individual needs in specific projects akin as done with quality models.

Table 5.1: Evolvability subcharacteristics

Subcharacteristic	Description
Analyzability, Ease of comprehension, (Understandability)*	The capability of the software product to be diagnosed for deficiencies or causes of failures in the software and to enable the identification of influenced parts due to change stimuli (based on [Int01] and [BCE07]).
Changeability/ Modifiability*	The capability of the software product to enable a specified modification to be implemented quickly and cost-effectively (based on [Int01] and [MN03]).
Extensibility*	"The capability of a software system to enable the implementation of extensions to expand or enhance the system with new capabilities and features with minimal impact to existing system." [BCE07]
Variability*	The capability of a software system or artifact to be efficiently extended, changed, customized, or configured for use in a particular context by using preconfigured variation points (based on [SvGB05]).
Portability*	"The capability of the software product to be transferred from one environment or platform to another." [Int01]
Reusability*	"The system's structure or some of its components can be reused again in future applications." [MN03]
Testability*	"The capability of the software system to enable modified software to be validated." [Int01]
Traceability*	The capability to track and recover in both a forwards and backwards direction the development steps of a software system and the design decisions made during on-going refinement and iteration in all development phases by relating the resulting artifacts of each development step to each other (based on [GF94]).
Compliance to standards*	The extent to which the software product adheres to standards or conventions relating to evolvability (based on [Int01]).
Process qualities	Additional process quality characteristics are, for example, Project Maturity and Community Quality, which are recognized as characteristics that influence the evolvability of open source software projects [DMCS07].

The * denotes the subcharacteristics used for the later evaluation as explained at the end of Section 5.3.1.2.

5.3.1.2 Relating Design Principles to Evolvability Subcharacteristics

For a goal-oriented way of decision-making during architectural synthesis, the impact of a decision on the quality goal has to be determined or predicted. As explained with the general concept of the Goal Solution Scheme (Section 5.1) architectural design principles were introduced as a means for selecting appropriate architectural solution instruments and determining their impact on evolution effort. In the evolvability model the design principles are related to the evolvability subcharacteristics according to the second transition of the GSS. The design principles used for impact evaluation are listed in Table 5.2. Subordination of principles is expressed by indentation in the principle column. In this regard, for example, low complexity should be an overall design principle that helps a lot of quality goals. Other principles such as abstraction or modularity help to achieve low complexity. Modularity in turn can be achieved by proper coupling and cohesion.

The relations between subcharacteristics and design principles have been developed in an iterative way, starting with hypotheses [BBR09] and steps of revision during application in case studies [RB09, Sto10]. Meanwhile, the relations and the way of calculating impact values can be considered as rather mature. They might also be tailored for individual or project-specific needs. For brevity's sake, not all relations are discussed here, but some explanation is given below.

The subcharacteristics of evolvability usually can be influenced positively by reducing the *complexity* of the design. A *simple* and *consistent* design, which also follows, for example, the principles *modularity, encapsulation*, and *separation of concerns*, leads to clear responsibilities of components and interfaces in between. This can ease changeability of the system by reducing the effort for the developer if changes have to be implemented. A proper level of *abstraction* and *granularity* helps if, for example, a certain feature can be realized by just extending or exchanging a single component. *Conceptional integrity* and a *coherent mapping to concepts* in the design with adequate usage of terms help to identify the architectural elements relevant for the changes and inhibit ripple effects. This also is the case if these principles are followed to comply to standards.

For traceability particularly the *consistency* and *completeness* as well as a *low complexity* of the design are important to support analysis of the artifacts that are impacted by a change. A good *separation of concerns* and a *coherent mapping to concepts* help to reduce the number of traceability links by reducing cross-cutting

Table 5.2: Architectural design principles

Principle	Description
Low complexity*	The extent to which the amount of elements and their interdependencies are reduced.
Abstraction*	The extent to which unnecessary details of information are hidden to build an ideal model and the extend to which a solution is generalized (based on [Boo93]).
Modularity*	The property of a software system to be decomposed into a set of coherent and loosely coupled elements with subsumption of abstractions (based on [Boo93]).
Cohesion*	The strength of the coupling between the internals of an element (based on [Boo93]).
Loose coupling*	The extent to which the interdependencies between elements are minimized (based on [Boo93]).
Encapsulation*	The extent of hiding the internals of an element, for example, by separation of interface and implementation (based on [Boo93]).
Separation of Concerns*	The extent to which different responsibilities are mapped onto different elements with as little as possible overlap, at which ideally one responsibility is assigned to exactly one specific element. The violation of this property is called tangling and scattering.
Hierarchy*	The arrangement or classification of related abstractions ranked one above the other according to inclusiveness and level of detail (based on [Boo93] and [McK05]).
Simplicity*	"The quality or condition of being easy to understand or do" [McK05].
Correctness	The property of an element to be complete and consistent resulting in a fulfillment of its responsibilities.
Consistency	The absence of contradictions and violations between related elements.
Completeness	The coverage of all relevant responsibilities by an element without lacking any necessary detail.
Conceptual integrity	The continuous application of ideas throughout a whole solution, preventing special effects and exceptions (based on [Bro95]).
Proper granularity*	The size and complexity of an element is appropriate to its responsibilities and to the particular situation.
Coherent mapping to concepts*	The way to map elements to ideas and mental pictures so that they are easy to understand, for example, by proper names.

concerns as, for example, with the implementation of quality goals.

Another important subcharacteristic of evolvability is the testability of new features as a prerequisite for regression tests. The testability of a system benefits from a good *hierarchical* structure, a *consistent, complete*, and *modular* design with good *separation of concerns*. However, a strong *encapsulation* can hinder testability if the operations to test are hidden behind a restrictive interface, for example, because of a facade, which is an example for a negative influence relation.

All results regarding the correlation are shown in Table 5.3 and Figure 5.6. In the figure some direct dependencies are left out and only the aggregated ones are shown for a better visualization. For the same reason the interdependencies are not shown using the URN notation of contribution links although this would have been appropriate. The positively influencing relations correspond to HELP-contribution links of the URN and are represented by the value 1 in Table 5.3; the same holds for a negative influence, a HURT-contribution link, and -1, respectively. If no certain positive or negative influence could be determined, there is a 0 entry in the table. This corresponds to an UNKNOWN-contribution in URN. For aggregating subcharacteristics as changeability and principles as modularity the influence will later be calculated using the influence of the subordinated concepts. The indentations in Table 5.3 correspond to the refinement links in Figure 5.6.

Many evolvability subcharacteristics can be influenced by the architectural structure and behavior. However, there are important influence factors on evolvability, as for example qualification and motivation of the team members, process maturity, or quality management activities. The development process with roles, phases, communication paths, and traceability has a large influence as well. Architectural structures cannot control these factors; they will be considered partly in the calculation scheme for the influence of solution instruments on evolvability in Section 5.3.2.3. For the subcharacteristics (Table 5.1) and the design principles (Table 5.2) the ones that can have a direct influence from architectural patterns are marked by an * in the first columns. They are applied as evaluation criteria for the architectural patterns in the sequel.

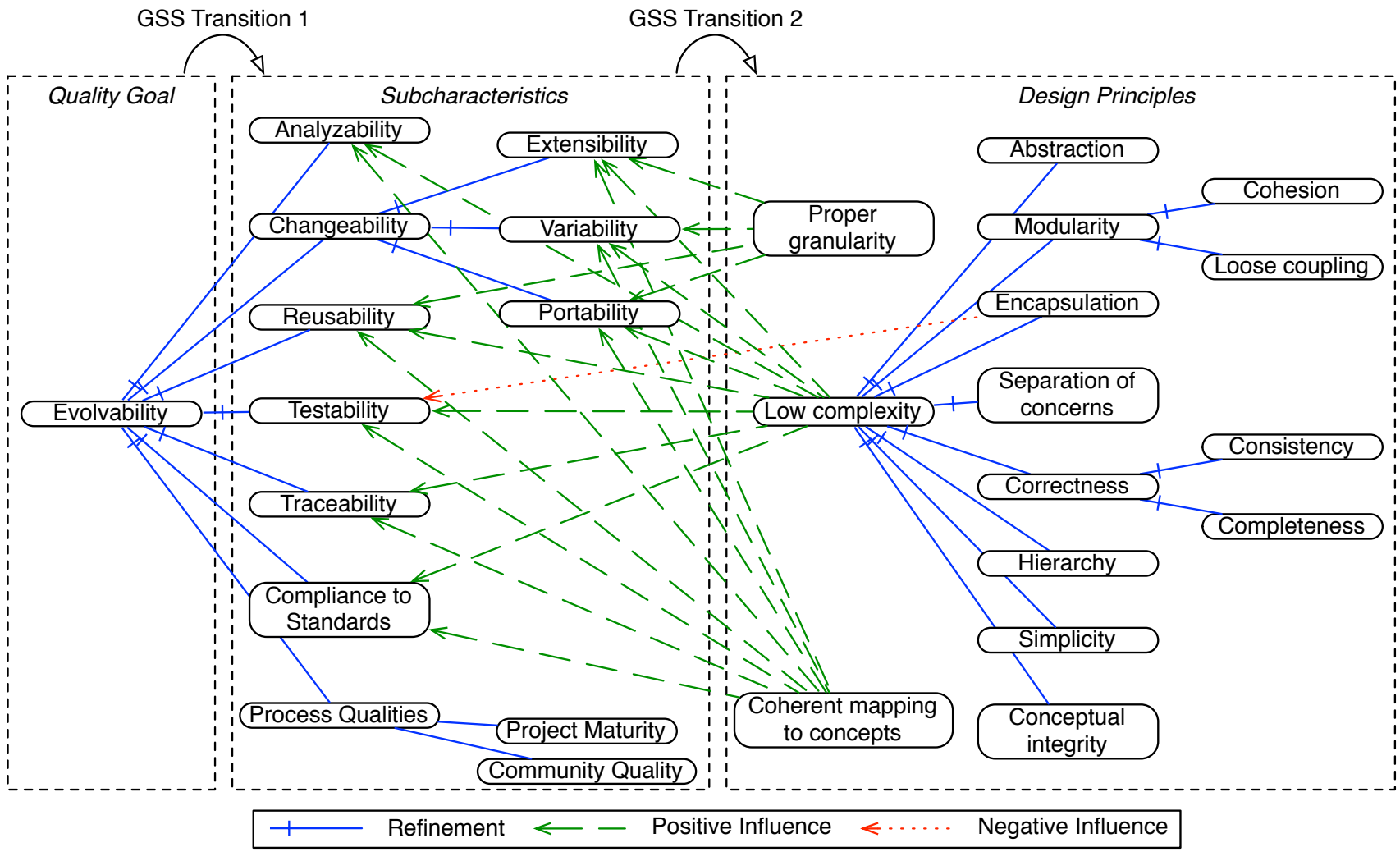


Figure 5.6: Graphical representation of the evolvability model

Table 5.3: Mapping of subcharacteristics to principles

Principle ╲ Subcharacteristic	Analyzability	Changeability	Extensibility	Variability	Portability	Testability	Reusability	Traceability	Compliance
Low complexity*								1	
Abstraction*	1		1	1	1	0	1	0	1
Modularity*									
Cohesion*	1		1	1	1	1	1	0	0
Loose coupling*	1		1	1	1	1	1	0	0
Encapsulation*	1		1	1	1	-1	1	0	0
Separation of concerns*	1		1	1	1	1	1	1	0
Hierarchie*	1		1	1	1	1	0	0	1
Simplicity*	1		1	1	1	1	1	0	0
Correctness									
Consistency	1		1	1	1	1	1	1	0
Completeness	0		0	1	0	1	1	1	0
Conceptional integrity	1		1	1	1	0	0	0	1
Proper granularity*	0		1	1	1	0	1	0	0
Coherent mapping to concepts*	1		1	1	1	1	1	1	1

1 – Positive influence; 0 – unknown/no influence; -1 – Negative influence

5.3.2 Evaluation of Architectural Solution Instruments

In this section an approach for the evaluation of the impact of architectural solution instruments on quality goals is described. The concept of the approach is explained with the evolvability model and the evaluation criteria presented in the last section. The evaluation itself is presented with a case study of a software system for collective orderers of mail order companies that was performed and published in [Sto10] and [BR10].

Case Study: Collective Ordering System *Mail order companies prefer to work together with collective orderers, who accumulate orders of several customers and submit them as a collective order to the mail order company. The mail order company delivers the goods in one shipment to the collective orderer, who in turn*

distributes them to the customers. There are several advantages: The collective orderer knows the formalities and processes for rare procedures such as reshipment, complaint, deferred payment, etc. better than the average customer. The personal and familiar contact to customers has positive effects on the business volume. The mail order company can delegate communication activities with customers to the collective orderer. These procedures belong to the core business in the domain and are affected by frequent changes. Therefore, they were chosen for this case study.

The software system of the company shall enable collective orderers to submit orders, manage their customers, and deal with complaints. The evaluation approach explained in this section is applied for the task of enhancing this system. First, in Section 5.3.2.1 architectural patterns are presented as an initial set of solution instruments for an architect's toolbox. Second, the impact of the patterns on the architectural design principles (Table 5.2) is determined in Section 5.3.2.2. For evaluation purposes, a suitable part of the collective ordering software was designed for each of the considered architectural patterns. This architectural design was used for the impact determination. Based on the results, the impact on the subcharacteristics of evolvability (Table 5.1) was determined as discussed in Section 5.3.2.3. Finally, the values to determine the impact on the quality goal evolvability are aggregated in Section 5.3.2.4. The resulting values are stored together with the patterns in the architect's catalog. They can be used for future design decisions regarding evolvability according to the third transition of the Goal Solution Scheme.

5.3.2.1 The Set of Evaluated Patterns

Software architectures are designed according to certain structural and behavioral principles, which can be supported by patterns. A pattern describes a particular design problem that arises in a specific context and provides a concept for a solution that proved well. Thus, it can be part of the catalog of approved methods and solution templates of the architectural toolbox. Buschmann et al. [BMR+96] distinguish several categories of patterns according to the level of abstraction or range of scale:

- Architectural patterns with fundamental influence on the structure of the software system,
- Design patterns with less range of scale on subsystems or components (see Gamma et al. [GHJV94]), and

- Idioms with low-level impact specific to a programming language.

Bosch [Bos00] additionally distinguishes between architectural styles that have a global impact on the architecture and architectural patterns that are considered with cross-cutting aspects of the architecture. A discussion about the different viewpoints on architectural styles or patterns can be found in [AZ05], a distinction and classification of the concepts in [GHR07]. For the evaluation approach of this thesis just such patterns are interesting that impact the architecture and not only detailed design or code. That's why for the rest of the thesis the term architectural pattern or pattern is used.

There are thousands of pattern descriptions available. For the impact determination regarding the quality goal evolvability, those patterns with an influence on the software architecture constitute interesting candidates. The patterns have to fulfill the constraints of the software system of the case study. Therefore, those patterns of the entirety were chosen that have an influence on the architecture, that are well documented, and that are expected to have an impact on evolvability. In the set of patterns, for example, *Facade* and *Adapter*, which can be called design patterns, are included as well. But they have an architectural influence if they are used.

The set of patterns used for impact evaluation is listed in Table 5.4. At this place it is refrained from explaining every pattern. Instead, literature references are given for the interested reader.

5.3.2.2 Determination of the Impact on the Principles

This section explains the determination of the impact values for the selected patterns in the case study. First, the patterns were applied in an exemplary architectural design for the case study. The resulting pattern-based design was rated regarding the impact on the architectural design principles. The ratings were gained through an assessment of the impact for every principle by expert opinion. The value of the impact is expressed by values between -2 and 2 according to the rating scheme in Table 5.5.

Case study example for impact discussion *A collective orderer has to enter orders into the software system, and then the orders have to be transmitted to the mail order company. Usually this is done via phone but should be supported by the new software system. As a possible solution in the case study, the Client-Server*

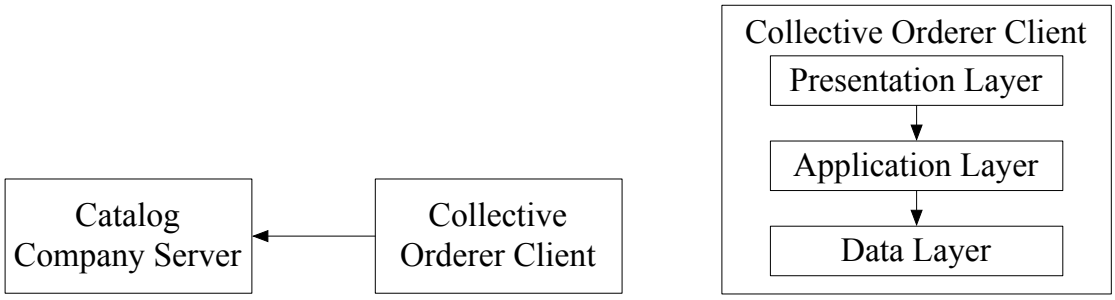
Table 5.4: The set of architectural patterns for the impact evaluation

Name	Sources
Client-Server	see Avgeriou&Zdun [AZ05]
Layers/Tiers	
Repository	
Blackboard	
Pipes and Filters	
Model View Controller (MVC)	
Presentation Abstraction Control (PAC)	
Event-Based, Implicit Invocation	
Broker	
Micro Kernel	
Reflection	
Facade	Gamma et al. [GHJV94]
Adapter	
Proxy	
Plug-in	Manolescu et al. [MVN06]

Table 5.5: Rating scheme

Rating value	Impact
2	strong positive
1	weak positive
0	neutral
-1	weak negative
-2	strong negative

pattern was utilized (see Figure 5.7(a)). The server at the mail order company is connected to the collective orderer's client via internet. It provides an interface for the transmission of orders. The client is structured in three layers as shown in Figure 5.7(b). The presentation layer is responsible for the graphical user interface (GUI). It uses the application layer, which provides functions as calculations for deferred payment, a search for ordered but not delivered goods, or a reminder for the deadline for returning the goods. The data layer is responsible for the data persistence in a databank.



(a) Client-Server (b) Layers

Figure 5.7: Pattern application in the case study example

Now the evaluation of the patterns *Client-Server* and *Layers* regarding their impact on the architectural design principles is discussed. Both patterns have a strong positive impact on several principles. For example, they provide a good *abstraction* of internal details (rating 2). The resulting architecture is *simple* to understand (2). They provide good *modularity* because of high *cohesion* inside the layers, client, and server (2), as well as a *loose coupling* between the elements (2), for example, for the deferred payment. Unnecessary details are hidden behind interfaces between the layers. Therefore, the *encapsulation* is improved (2). Regarding *separation of concerns* *Client-Server* and *Layers* have a positive impact, but they cannot completely prevent mixing different concerns (rating 1). Regarding the *hierarchy* criteria the two patterns differ in their impact. The *Layers* pattern supports the ranking and grouping of abstractions on different levels very well due to the different layers (2). For the *Client-Server* pattern this cannot hold to this extend, which results in a lower rating (1). The same applies for the *coherent mapping to concepts* criterion. The *Layers* pattern has a weak positive impact on a *proper granularity* of an architectural design by structuring into layers instead of one big structural element (1). Overall, the *Client-Server* pattern and the *Layers* pattern reduce the *complexity* of an architecture through structuring.

Inside the client's presentation layer the *Model View Controller (MVC)* pattern can be used to separate the data to be presented (e.g., a customer or an order) from the different views and control mechanisms. For example, there are views for editing the customers' contact information or for collecting and managing the orders.

The support for *abstraction* and *cohesion* as well as *separation of concerns* of *MVC* is very good (rating 2) as a result of the strict separation of model, view,

and controller. This improves the *simplicity* of the design as well (1), although it is not so easy to use *MVC* with modern GUI libraries. The *encapsulation* is also good because the internals of each element are hidden behind interfaces (1). Building a *hierarchy* with *MVC* is not so well supported (0)—here *Presentation Abstraction Control (PAC)* would be better. Regarding *coupling MVC* is evaluated slightly negative (-1). Of course, the views can be decoupled from the model via a change-propagation mechanism, however, view and controller are coupled very tightly. Summed up, the *complexity* resulting from *MVC* is good but not excellent. The *granularity* that results from MVC can be quite good (1) if the models and views are properly designed. However, *MVC*'s real strength is to provide a *coherent mapping of concepts* for the user interaction through the GUI (2).

Table 5.6: Values for the patterns' impact on the properties

Property ╲ Pattern	Client-Server	Layers/Tiers	Repository	Blackboard	Pipes & Filters	MVC	PAC	Impl. Invoc.	Facade	Adapter	Broker	Proxy	Micro Kernel	Reflection	Plug-in
Low complexity	1.7	1.8	0.8	0.3	1.7	1.1	1.4	0.8	1.3	1.4	1.5	1.5	2	0.3	1.6
Abstraction	2	2	0	0	2	2	2	1	2	2	2	2	2	2	2
Modularity	2	2	0	0.5	0	0.5	0.5	1	1.5	1.5	2	2	2	1	1.5
Cohesion	2	2	1	1	2	2	2	0	1	1	2	2	2	0	2
Loose coupling	2	2	-1	0	-2	-1	-1	2	2	2	2	2	2	2	1
Encapsulation	2	2	1	2	2	1	1	1	2	2	2	2	2	0	2
Separation of concerns	1	1	2	0	2	2	2	0	0	1	1	1	2	0	2
Hierarchy	1	2	0	0	2	0	2	0	0	0	0	0	2	0	0
Simplicity	2	2	2	2	2	1	1	2	2	2	2	2	2	-1	2
Proper granularity	0	1	0	0	2	1	1	0	2	1	0	0	1	0	2
Coherent mapping to concepts	1	2	1	0	2	2	2	0	1	1	2	0	2	0	2

For the rest of the patterns the evaluation concerning the properties was done in the same way based on the findings of a bachelor thesis [Sto10]. Table 5.6 shows the determined impacts of all selected patterns on the properties for good architectural design. The ratings of the aggregated properties modularity and low complexity are calculated by arithmetic mean of the subordinates.

5.3.2.3 Calculation of the Impact on Evolvability Subcharacteristics

The patterns' impact on the quality subcharacteristics is primarily determined from the impact on the principles, as discussed above. They are considered in the first step of the calculation. Additional influences on the subcharacteristics—for example from efforts not related to the design principles—are considered by introducing adjustments in a second step.

The results were calculated in the following way. Let $\boldsymbol{R}$ be the matrix of the impact ratings for the properties (Table 5.6) and $\boldsymbol{r}_p$ be a column vector of this matrix for one element p of the set of patterns P. Let $\boldsymbol{M}$ be the mapping matrix of Table 5.3 and $\boldsymbol{M}^*$ be $\boldsymbol{M}$ reduced by the rows for which the properties were not evaluated (and are not marked with *). Further, let $\boldsymbol{m}_s$ be a column vector of $\boldsymbol{M}^*$ for one element s of the set of subcharacteristics S. Moreover, let $\boldsymbol{V}$ be the matrix with the impact values of the patterns on the subcharacteristics. Then, each element v_{sp} of $\boldsymbol{V}$ is calculated by

$$v_{sp} = \boldsymbol{r}_p \cdot \boldsymbol{m}_s / \left\| \boldsymbol{m}_s \right\|_1 .$$

Finally, the matrix $\boldsymbol{V}'$ in the top of Table 5.7 is obtained from $\boldsymbol{V}$ by calculating the impact values for changeability in row two by the arithmetic mean of the values for extensibility, variability, and portability. In this way the patterns' impact values on the subcharacteristics is determined from the direct ratings for the principles (Table 5.6) by evaluating and normalizing the influences of the interdependencies described by the mapping in Table 5.3.

However, through the calculation there is no discrimination regarding the subcharacteristics of changeability. The Reflection pattern, for example, contributes to extensibility and variability but reduces portability if the base technology does not support reflection. Furthermore, testability is decreased due to possible dynamic changes at runtime.

These effects are not represented by the aggregated impact values in $\boldsymbol{V}'$ of the first step. Therefore, offset values o_{sp} were considered that are shown in the middle of Table 5.7 for the determination of the patterns' impact on the subcharacteristics. In the literature (e.g., Buschmann et al. [BMR+96]) also knowledge about consequences of the pattern application regarding quality goals is mentioned. This knowledge was incorporated for the determination of the offset values. The final impact values are calculated as follows. Let $\boldsymbol{F}$ be the matrix for the adjusted impact values. Then

each element f_{sp} of $\boldsymbol{F}$ is calculated by

$$f_{sp} = \begin{cases} v_{sp} & \text{if } o_{sp} \text{ is undefined} \\ (v_{sp} + o_{sp})/2 & \text{otherwise.} \end{cases}$$

The final impact values $\boldsymbol{F'}$ for the subcharacteristics including changeability shown in the bottom of Table 5.7 again are obtained as for $\boldsymbol{V'}$ before.

5.3.2.4 Determining the Impact on Evolvability

As the last step the overall impact of the patterns on the quality goal evolvability was determined by aggregating all subcharacteristics with equal weights. The resulting values of the patterns' impact on evolvability are shown in Figure 5.8 and in the lowermost row of Table 5.7. The changes in the rating resulting from the offset values can be seen by comparing the unadjusted and the final values shown in Figure 5.8. As a consequence of the evaluation, in the case study a plug-in-based architecture was selected as the best solution.

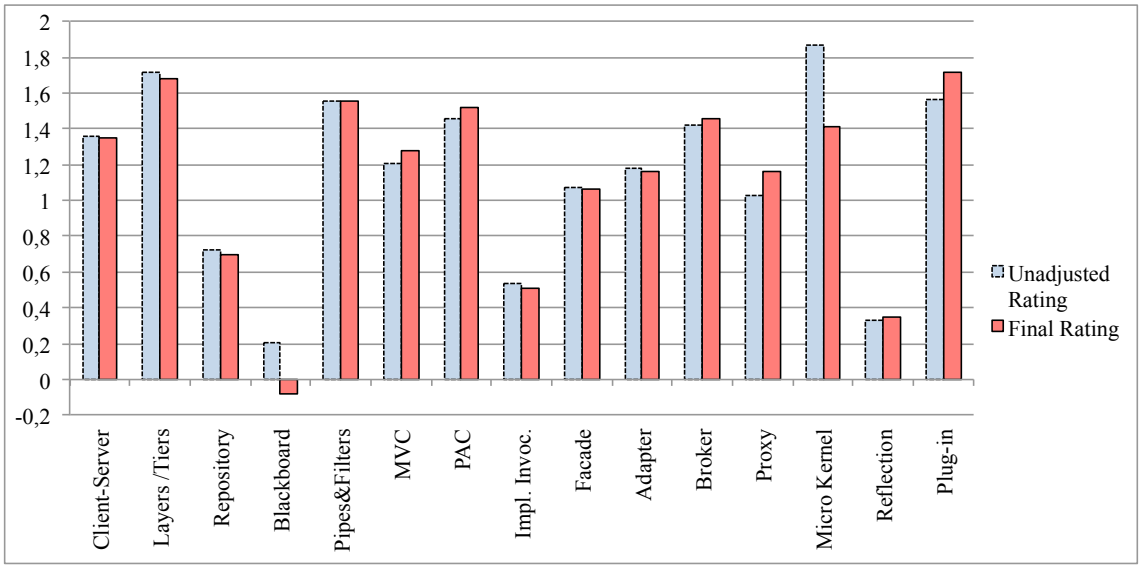


Figure 5.8: Resulting impact of patterns on evolvability

5.3.3 Discussion of the Results

The final results of the impact evaluation are illustrated by Figure 5.8. The chart shows that the patterns in general have a positive impact on the quality goal evolvability. This corresponds to the fact that patterns in general positively influence the

Table 5.7: Impact values for subcharacteristics and evolvability

Subcharacteristic ╲ Pattern	Client-Server	Layers/Tiers	Repository	Blackboard	Pipes & Filters	MVC	PAC	Impl. Invoc.	Facade	Adapter	Broker	Proxy	Micro Kernel	Reflection	Plug-in
Calculated Rating															
Analyzability	1.6	1.9	0.8	0.1	1.5	1.1	1.4	0.8	1.3	1.4	1.6	1.4	2.0	0.4	1.6
Changeability	1.4	1.8	0.7	0.2	1.6	1.1	1.3	0.7	1.3	1.3	1.4	1.2	1.9	0.3	1.7
Extensibility	1.4	1.8	0.7	0.2	1.6	1.1	1.3	0.7	1.3	1.3	1.4	1.2	1.9	0.3	1.7
Variability	1.4	1.8	0.7	0.2	1.6	1.1	1.3	0.7	1.3	1.3	1.4	1.2	1.9	0.3	1.7
Portability	1.4	1.8	0.7	0.2	1.6	1.1	1.3	0.7	1.3	1.3	1.4	1.2	1.9	0.3	1.7
Testability	1.0	1.3	0.6	-0.1	0.9	0.7	1.0	0.4	0.6	0.7	1.0	0.7	1.4	0.1	1.0
Reusability	1.5	1.8	0.8	0.3	1.5	1.3	1.3	0.8	1.5	1.5	1.6	1.4	1.9	0.4	1.9
Traceability	1.2	1.6	1.3	0.8	1.9	1.7	1.8	0.3	0.8	1.1	1.5	0.8	2.0	0.1	1.9
Compliance to standards	1.3	2.0	0.3	0.0	2.0	1.3	2.0	0.3	1.0	1.0	1.3	0.7	2.0	0.7	1.3
Offset															
Analyzability			0	-1				-1	2	0	1		0		
Changeability															
Extensibility	1	1	2	-1		2	2	2	2	2	2			2	2
Variability	1	1				2	2	2		2				2	2
Portability	2	2				2	2			2	2	2		-1	
Testability				-2				-2	-1			2	0	-1	2
Reusability								2	2	2				1	
Traceability								2							
Compliance to standards											2		0		2
Final Values															
Analyzability	1.6	1.9	0.4	-0.4	1.5	1.1	1.4	-0.1	1.6	0.7	1.3	1.4	1.0	0.4	1.6
Changeability	1.4	1.6	0.9	0.0	1.6	1.6	1.7	1.1	1.4	1.7	1.6	1.4	1.9	0.7	1.8
Extensibility	1.2	1.4	1.3	-0.4	1.6	1.6	1.7	1.3	1.7	1.7	1.7	1.2	1.9	1.2	1.8
Variability	1.2	1.4	0.7	0.2	1.6	1.6	1.7	1.3	1.3	1.7	1.4	1.2	1.9	1.2	1.8
Portability	1.7	1.9	0.7	0.2	1.6	1.6	1.7	0.7	1.3	1.7	1.7	1.6	1.9	-0.3	1.7
Testability	1.0	1.3	0.6	-1.1	0.9	0.7	1.0	-0.8	-0.2	0.7	1.0	1.4	0.7	-0.4	1.5
Reusability	1.5	1.8	0.8	0.3	1.5	1.3	1.3	1.4	1.8	1.8	1.6	1.4	1.9	0.7	1.9
Traceability	1.2	1.6	1.3	0.8	1.9	1.7	1.8	1.1	0.8	1.1	1.5	0.8	2.0	0.1	1.9
Compliance to standards	1.3	2.0	0.3	0.0	2.0	1.3	2.0	0.3	1.0	1.0	1.7	0.7	1.0	0.7	1.7
Evolvability	1.3	1.7	0.7	-0.1	1.6	1.3	1.5	0.5	1.1	1.2	1.5	1.2	1.4	0.3	1.7

quality properties of a software system if they are used wisely. Some patterns turned out to be excellent, for example, Layers, Plug-in, or Pipes and Filters; others are less supportive. However, the impact of the patterns on evolvability and on software quality in general is limited if process aspects are not taken into account. Process qualities were considered only partly by the adjustments. Traceability constitutes another aspect important for evolvability which depends on the development process rather than on patterns.

The impact values for the patterns have been derived from a concrete case study. The quantitative values should be seen as an additional hint helping the architect with the uncertainty of a decision, which otherwise would be based purely on experience. It must be admitted that the values are subjective by nature because they were determined from expert opinion. Such results also depend on the application conditions as the experience of the development teams. However, an improvement regarding objectivity is possible by including several experts. The values shall be applicable and applied in further projects. The degree of universality of the impact values can be improved by revising them in a series of projects. Further improvements could be made by incorporating the uncertainty of the subjective ratings into the calculation. Maybe an interval of values for the influence might be more appropriate than absolute values.

The presented results of the evolvability model and the impact evaluation have been developed as hypotheses, later were revised, and evaluated with a case study. They can be considered as rather mature but might be biased by the case study. Even if forthcoming revisions might result in smaller modifications of the impact values, the revisions of the relations for refinement and mapping (Figure 5.6) can be expected to be minor ones. At least these calculated values are an improvement in comparison to the direct evaluation of the impact of patterns on quality goals from other approaches (cf. [HA07b, GEM10]), which do not consider principles.

Buschmann et al. [BMR+96] argue that a classification of patterns into groups is necessary to help the architect to use a system of patterns. The author agrees to this argument for general categories like architectural patterns or design patterns, structural or behavioral patterns, or regarding problems like concurrency or distribution. However, for effort-related quality goals a quantitative evaluation is more effective than a categorization because the impact on effort varies within an interval. The presented evolvability model and the evaluated patterns represent a catalog of

possible solution instruments for architectural design. A filtering step according to project-specific constraints applied to solution instruments from the catalog results in a very similar effect as Buschmann's categories. A ranking of a second step supports the architect in selecting the most appropriate solution instrument for the quality goals (cf. the decision procedure presented in Section 6.2.2.1 in page 111). Besides, the catalog can be easily extended by new patterns and with evaluations for further quality goals.

Chapter 6

Goal-Oriented Architectural Design

As already mentioned in the overview of Chapter 4 this chapter describes the goal-oriented architectural design method (GOAD) by means of activities and artifacts. It explains how the existing approaches from the state-of-the-art are combined in the different development phases. Moreover, it describes how the concept of the Goal Solution Scheme from Chapter 5 is utilized for the refinement and realization of the quality goals and for the decision-making on architectural solution instruments. The whole approach is illustrated with a case study example from a real-world development project for several quality goals.

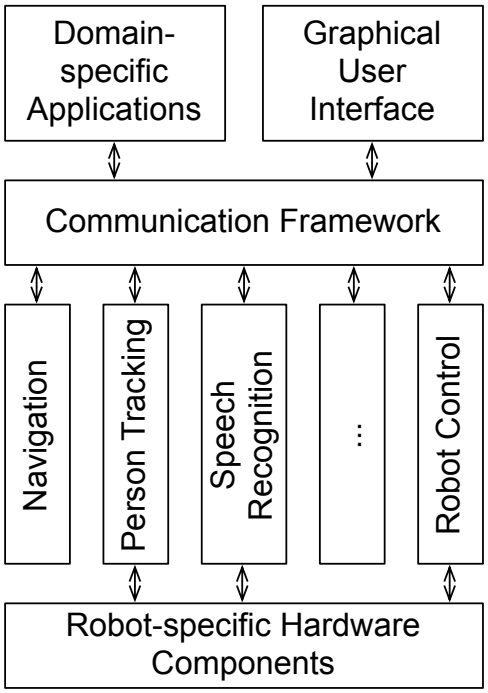
Section 6.1 introduces the case study example. Then Section 6.2 explains the development activities of the goal-oriented architectural design method with the analysis and synthesis phases. Because the method is not especially concerned with requirements analysis but with design and because the activities are iteratively interweaved, all related analysis activities are described in Section 6.2.1. The architectural synthesis is discussed in Section 6.2.2. Finally, Section 6.2.3 gives some hints for architectural evaluation.

6.1 Introduction of the Case Study

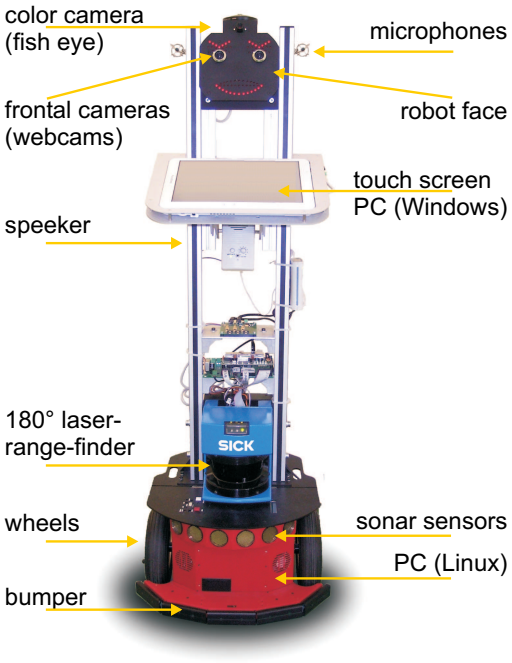
The case study to illustrate the goal-oriented design method is an academic software engineering project from a real world scenario. The project is run by the department of Neuroinformatics and Cognitive Robotics of the Ilmenau University of Technology to provide a controlling software system for mobile interaction-robots. This complex software system and its architecture is developed in an evolutionary scenario

over several years to provide a uniform platform for different robots. The mobile interaction-robots can be used in several environments, for example, as shopping assistance robots, for an office information and guiding system for employees, or even in the care for elderly people. For this purpose these robots have to provide several features, as for example human-robot interaction including person tracking or face and speech recognition, as well as navigation including path planning, collision avoidance, or self localization.

Figure 6.1 gives an overview on the software system for the robots and illustrates the HOme RObot System HOROS. It shows the basic framework of the software system. It is responsible for the communication of the different components controlling the robot's hardware as well as the so-called skill components that provide the features named above. Moreover, the framework is the base for the graphical user interface and domain-specific applications. This software system is subject of several design goals and constraints that are discussed in the curse of the development activities in the following sections. The focus of the case study for the illustration of the design activities is put especially on the framework part of the software system.



(a) Software architecture overview



(b) Mobile interaction robot HOROS [MSW+05]

Figure 6.1: Overview of a mobile interaction robot and its architecture

6.2 The Goal-Oriented Design Method (GOAD)

In Section 4.2 the phases of the development process were already introduced. This section gives more details on the activities and artifacts of the different phases and regarding the utilization of the Goal Solution Scheme. The focus lies on a forward perspective, and one path through the different activities is shown on the example case study. Nevertheless, as already mentioned the activities are not purely sequential. The procedure is rather incremental and iterative following a backlog concept as depicted in Figure 6.2. This is necessary to parallelize and intertwine the activities in a development team.

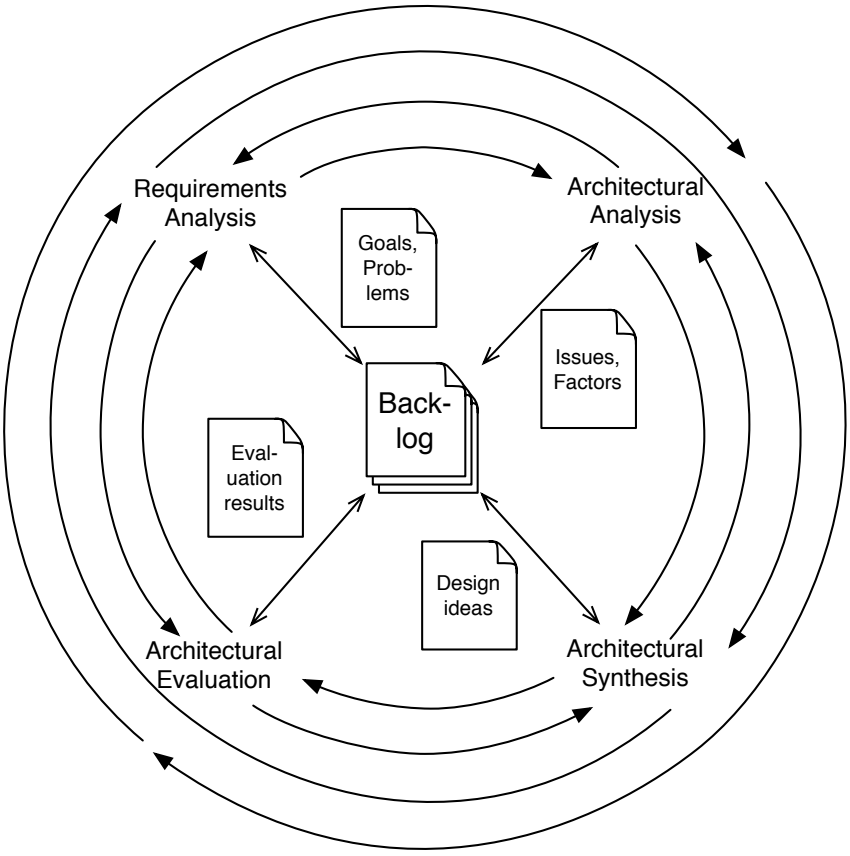


Figure 6.2: Backlog concept and iterations between the development phases

The backlog is kind of an issue list with small issues or problems to tackle and ideas to follow, which drive the architectural design process [HKN+07]. The backlog's items stem from the various development activities of the different phases. It is fed, for example, by architectural concerns or architecturally significant requirements from the architectural analysis, or by feedback from an architectural evaluation, as

well as by ideas for the design.

The architect can iterate between all phases to deal with the items and achieve his design task. For the iteration between design activities different dimensions can be identified:

Transformation from problem space to solutions space One reason to iterate between design activities is that the architect wants to go from the problem to the solution space. This corresponds to the transitions of the Goal Solution Scheme from goals and subgoals to principles and finally to solution instruments. In this regard the architect has to follow the different development phases for each problem to find a solution.

Refinement A second reason for iterating between activities is refinement. Refinement means decomposition or specialization, or in other words adding details. In this regard the architect, for example, refines goals to subgoals, analyzes the related influence factors, and again refines another goal. Moreover, he can, for example, add influence factors or add components to the design of the system.

Confidence for evaluation A third way to iterate can be triggered by the request for an improvement of a certain design artifact considering feedback from a "later" phase to increase the confidence for further evaluations or tests. For example an architect identifies a new requirement during analysis of the influence factors, or he revises a design decision based on the feedback from an evaluation activity.

During all the design activities the backlog constantly is subject to changes, but moreover also external demands for changes occur during the evolution of the software system. To be able to follow these changes during all activities, traceability links should be established between the different models or artifacts of the development process. A traceability concept for the goal-oriented design method is discussed in Chapter 7. In the following sections the activities and artifacts of the different phases are discussed in more detail based on the case study example. According to the Goal Solution Scheme it is discussed, how quality goals are modeled and related to architectural design principles, and how they are transformed into functional and technical solutions. The transformation is performed by using architectural means, such as styles and patterns, and following the ideas of the different design methods. Special emphasize is put on the selection of appropriate solution instruments.

6.2.1 Requirements and Architectural Analysis

During requirements and architectural analysis several activities have to be performed. These activities should be tightly intertwined to specify typical artifacts as use cases, scenarios, goal models, factor tables, and issue cards. For the elicitation and documentation of requirements the EMPRESS approach proposes means as checklists and questionnaires that can be used. However, since this is not the focus of the goal-oriented design method the related activities are not discussed here, neither is the use case specification. Instead, the activities *Goal Modeling* and *Global Analysis* as depicted in Figure 6.3 are discussed in more detail.

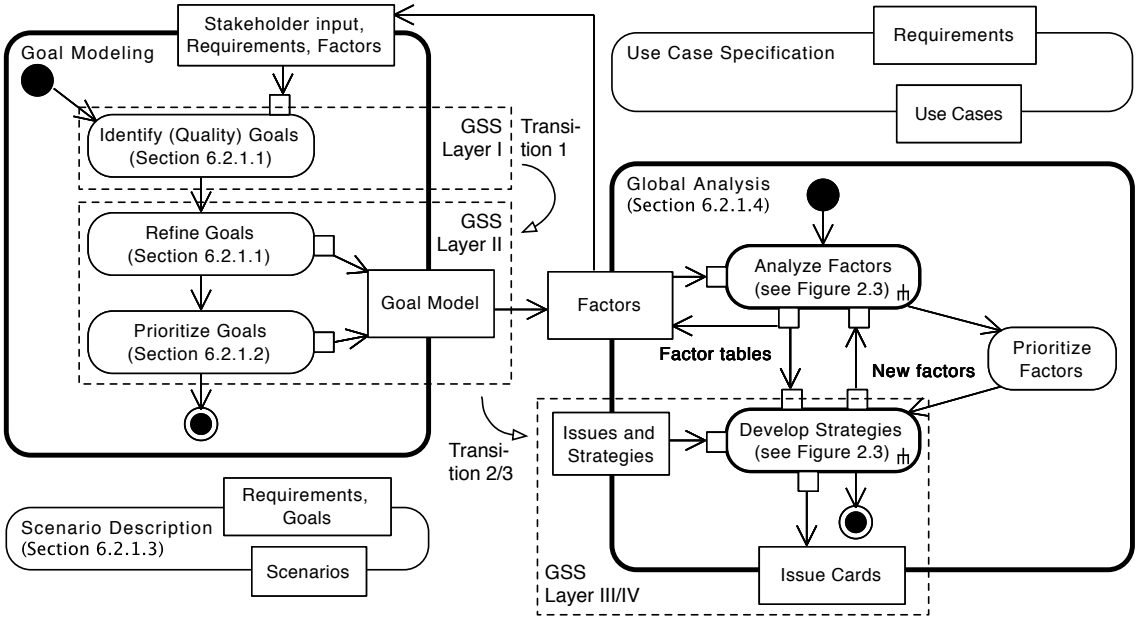


Figure 6.3: Activities of the analysis phase

6.2.1.1 Quality Goal Elaboration and Refinement

For the case study of the robot software system the following quality goals were determined to be of interest by the development team:

- Modifiability (Changeability),
- Testability,
- Reusability,
- Efficiency,
- Reliability,
- Distributability,
- Usability,
- Security.

These quality goals form the basis, the layer I of the Goal Solution Scheme. In the following, the goal-oriented modeling technique known from i* and the NFR framework is utilized for the first transition of the GSS. See Section 5.2.1 for details regarding this transition. As additional means for the specification and refinement of the quality goals of the case study, the quality model of the ISO 9126 and the evolvability model from Section 5.3.1 were used. Table 6.1 shows the specified quality goals of the case study with their description after a refinement. It should be noted that with evolvability also a generalized quality goal was included. The result of the goal specification is a goal model specified with the Goal-oriented Requirements Language (GRL) of the User Requirements Notation (URN), which is depicted in Figure 6.4.

Table 6.1: Quality goals of the case study

Quality Goal	**Description**
Evolvability	The ability of a software system throughout its lifespan to accommodate to changes and enhancements in requirements and technologies, that influence the system's architectural structure, with the least possible cost while maintaining the architectural integrity.
Testability	"The capability of the software product to enable modified software to be validated." [Int01]
Modifiability (Changeability)	The capability of the software product to enable a specified modification to be implemented quickly and cost-effectively (based on [Int01] and [MN03]).
Extensibility	"The capability of a software system to enable the implementation of extensions to expand or enhance the system with new capabilities and features with minimal impact to existing system." [BCE07]
Portability	"The capability of the software product to be transferred from one environment or platform to another." [Int01]
Variability	The capability of a software system or artifact to be efficiently extended, changed, customized, or configured for use in a particular context by using preconfigured variation points (based on [SvGB05]).
Reusability	"The system's structure or some of its components can be reused again in future applications." [MN03]

Table 6.1: Quality goals of the case study (continued)

Quality Goal	Description
Efficiency	"The capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions." [Int01]
Time behavior	"The capability of the software product to provide appropriate response and processing times and throughput rates when performing its function, under stated conditions." [Int01]
Resource utilization	"The capability of the software product to use appropriate amounts and types of resources when the software performs its function under stated conditions." [Int01]
Reliability	"The capability of the software product to maintain a specified level of performance when used under specified conditions." [Int01]
Maturity	"The capability of the software product to avoid failure as a result of faults in the software." [Int01]
Fault tolerance	"The capability of the software product to maintain a specified level of performance in cases of software faults or of infringement of its specified interface." [Int01]
Recoverability	"The capability of the software product to re-establish a specified level of performance and recover the data directly affected in the case of a failure." [Int01]
Distributability	The capability of a software system to express concurrency so that it can be mapped onto effectively concurrent heterogenous resources.
Security	"The capability of the software product to protect information and data so that unauthorized persons or systems cannot read or modify them and authorized persons or systems are not denied access to them." [Int01]
Integrity	The property of information to be protected from unauthorized manipulation.
Confidentiality	The property of information to be restricted to certain users.
Availability	The property of information to be accessible to authorized users in an appropriate time limit.

Table 6.1: Quality goals of the case study (continued)

Quality Goal	Description
Usability	"The capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions." [Int01]
Understandability	"The capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use." [Int01]
Learnability	"The capability of the software product to enable the user to learn its application." [Int01]
Operability	"The capability of the software product to enable the user to operate and control it." [Int01]
Attractiveness (User satisfaction)	"The capability of the software product to be attractive to the user." [Int01]

6.2.1.2 Goal Prioritization

Accompanying the goal specification also a goal prioritization should be performed. A typical prioritization can be performed, for example, following a rating scheme of high, middle, and low priority. This, for example, is known from the architectural evaluation method ATAM for the prioritization of the utility tree. However, in practice it often can be difficult for stakeholders to prioritize their goals and absolutely rate them because all goals are seen to be important. For this reason in this thesis for the goal-oriented design method a relative goal prioritization strategy is proposed—the $100 approach. Often effort and time are the limiting factors for the development of the software. This can be expressed in monetary value. In the $100 approach money is used as a metaphor for limited development capacities and this facilitates the prioritization.

The $100 Approach To rate the goals regarding their importance, each relevant stakeholder is virtually given $100 as a limited budget for addressing a set of goals in one round of rating. With this money the stakeholders can decide how much money to spend on each of the goals. The goals that are subject to one round of rating should be of the same granularity. Optionally, in a further round the prioritization

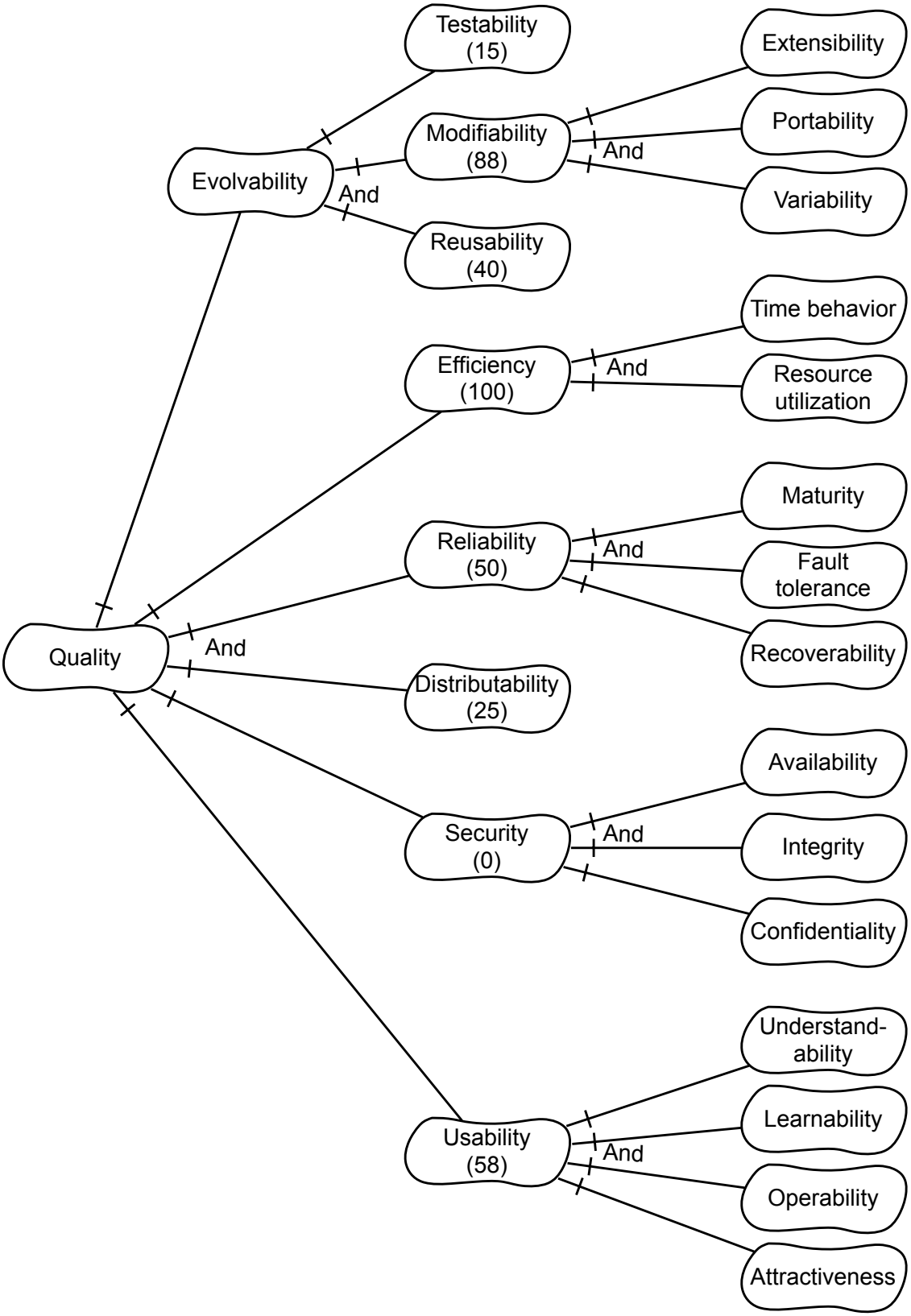


Figure 6.4: URN model for the quality goals of the case study with prioritization

can be proceeded as before by again splitting up the money on the refined goals as illustrated in Figure 6.5. Otherwise, the refined goals can be treated as equally weighted. In this way all quality goals are weighted relatively with each other according to their importance. As a result an absolute prioritization of all goals can be determined by summing up the ratings from all stakeholders.

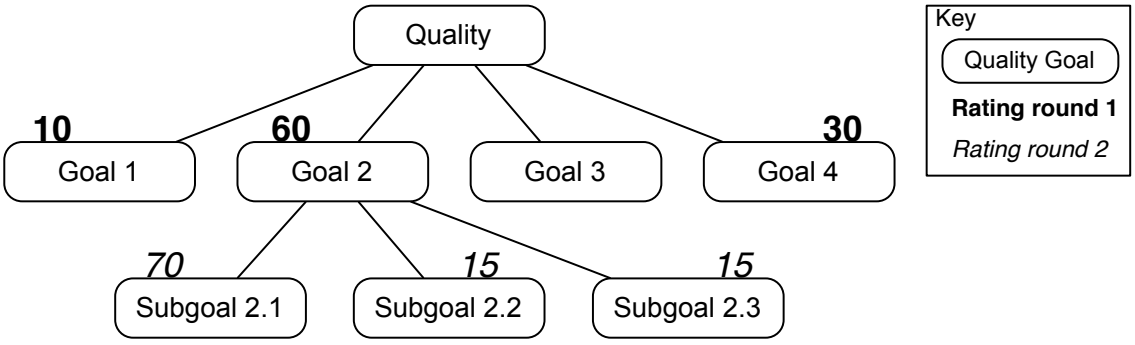


Figure 6.5: Illustration of the rating procedure of the $100 approach

Table 6.2 shows the result of the $100 approach for the quality goals of the case study. In the case study the prioritization was performed only on the higher-level goals. In a final step the summed up ratings were normalized to an interval between 0 and 100. In this way they could be managed by the Tool jUCMNav[1] which was used to model the goal graph using URN as depicted in Figure 6.4.

In the result the highest ranked quality goals are efficiency and modifiability followed by usability and reliability. Moreover, the quality goals testability, modifiability, and reusability, which are subcharacteristics of evolvability, together make up more than 37%. Hence, evolvability is an important quality goal for the case study project. The quality goals distributability and security—yet hardly considered—might get more important in an intended future growth scenario when the system should run on more than one computer.

[1] http://jucmnav.softwareengineering.ca/ucm/bin/view/ProjetSEG/WebHome

Table 6.2: Results of the goal prioritization with the $100 approach

Quality Goal ╲ **Stakeholder**	J. K.	C. S.	R. S.	C. V.	M. V.	S. M.	T. L.	E. E.	A. S.	Total	Total in %	Normalized [0,100]	Priority
Testability	5		20	10						35	3.89	15	
Modifiability	40	30	10	20	20	40	20	20	10	**210**	**23.33**	**88**	**H**
Reusability				20	10	30		5	30	95	10.56	40	
Efficiency	20	50	30	30	30	20	20	40		**240**	**26.67**	**100**	**H**
Reliability	20		20	10	10		20	20	20	**120**	**13.33**	**50**	**H**
Distributability	10			10		10	20	10		60	6.67	25	
Security										0	0.00	0	
Usability	5	20	20		30		20	5	40	**140**	**15.56**	**58**	**H**
	100	100	100	100	100	100	100	100	100	900	100.00		

6.2.1.3 Scenario Description

As a further step after refinement and prioritization of the goals detailed scenario descriptions can be created for the highest-ranked quality goals. This shall enable to further analyze the implications of the quality goals and to evaluate the quality of the architecture to be developed against the goals in a later scenario-based assessment.

In the case study scenario descriptions were created for the highly-prioritized quality goals. Table 6.3 shows an example scenario for the quality goal modifiability. The scenario description follows the template presented by Table 2.1 in Section 2.1.2.

6.2.1.4 Global Analysis

Next to the goal specification, the Global Analysis with its activities known from Hofmeister et al. (cf. Section 2.2.3 and Figure 6.3) is performed for architectural analysis.

Factor Tables The activity of identifying the influence factors according to the categories organizational, technological, and product factors can be combined with the goal modeling. Global business goals can be input for the organizational factors

Table 6.3: An example scenario for modifiability from the case study project

Attribute	Description
Name	Integration of a new hardware driver module.
Quality	Modifiability.
Stimulus	A new driver module is integrated to assure operation of the hardware.
Stimulus source	The laser scanner component is exchanged.
Environment	The system has been configured with a certain laser scanner and a corresponding driver model to enable communication with the laser scanner. The system is not in operation.
Artifact	The communication framework needs to interoperate with the new driver component.
Response	The integration of the new driver component works with the existing interfaces, and communication with the new component is possible even if new datatypes are introduced.
Response measure	The integration of the new component is possible with less than 50 lines of code.

and functional goals as well as quality (soft-)goals can be input for the product factors. The other way around also technological factors that are identified by an architect during Global Analysis can be input for the goal modeling activity. In this way the activities should be intertwined, and hence both contribute to the first transition of the Goal Solution Scheme.

The influence factors were specified for the communication framework of the case study. Table 6.4 shows a cutout of the product factor table from the case study example. The complete factor tables can be found in Appendix A.1.

Issue Cards Issue cards are specified as a step after the specification of the factor tables in Global Analysis. They discuss architecturally important topics to find appropriate solution strategies knowing typical principles and solution instruments. In this way they already correlate to the architectural synthesis phase as they contribute to the transition from the problem space to the solution space.

An exemplary issue card from the case study for the *Serialization* issue is shown

in Table 6.5. After describing the issue itself the relevant influencing factors from the factor tables are collected, as for example factor *P1.2 Serialization*, which is also shown in Table 6.4. Even new factors could be identified during the description of the issue card, which then are inserted in the factor table. They even might be added to the goal model. In the solution section of this issue card one proposed strategy is the usage of known libraries as solution instruments from the architect's toolbox. The strategies for the issue card's solution section can occupy layer III and IV of the Goal Solution Scheme. The strategies for the solution should be selected considering both the influence factors and the impact dependencies of the solution instruments on the quality goals according to the GSS. The selection procedure will be detailed in Section 6.2.2.

As a result of the case study, the demand for an extension of the original factor table concept arose, which though is not yet completely represented in the case study example. One extension is that factor categories can have sub-categories and not only subordinated factors. In this way especially the hierarchical structure of quality and functional goals can be better represented in the product factor table. To improve the clearness of the analysis of the influence factors, the flexibility and changeability of a factor should not be combined in one field in the tables. Instead, they should be separated out (which is not yet represented in the example). Furthermore, an additional column in the tables for a prioritization of the factors should be introduced. This would help to focus first on dealing with the most important factors when starting to create issue cards and synthesizing the architecture. Moreover, a priority and ranking of the issue cards could also be calculated based on their referenced influencing factors.

Table 6.4: Product factors of the case study

	Product Factor	***Flexibility & Changeability***	***Impact on Architecture***
P1	**Functional features**		
...			
P 1.2	*Serialization*		
	Data (objects) have to be (de-)serialized with a unified strategy, in different data format types (XML, binary, plain text) and to different targets (file, console, TCP, STL streams).	New data formats can be introduced every few months, but all are covered by the specified types.	This feature affects all components that deal with saving and loading data as well as transfer data via network connections. Serialization is influenced by portability.
...			
P2	**Quality features**		
P 2.1	*Evolvability*		
...			
P 2.1.2	*Modifiability*		
	Hardware components (e.g., laser, camera) should be replaceable by new ones without changing anything but the driver components.	There are many platforms with different combinations of hardware. New data types may emerge.	There is a major influence on the communication and serialization interfaces and components.
P 2.1.2.1	*Extensibility*		
	Integrating a new component (e.g., a hardware driver) should only need few overhead effort in the configuration with less than 50LoC.	New components regularly appear due to new hardware or due to experimental software components that are target of research.	There is a major influence on the configuration and communication components.
...			

Table 6.5: Issue card example of the case study

Serialization
In the old system there are several different inconsistent ways to save and load data: plain text files for parameters or for configuration or XML files. Serialization to files and data streams or to the console is necessary. **Influencing Factors:** **T3.4** With C++ it is difficult to save types because there is no built-in reflection mechanism. **T5.3** XML is used as a data format for configuration files. **T5.4** Communication via TCP needs serialization. **P1.1.2** Data communication needs serialization for configuration files. **P1.2** The serialization feature includes textural, binary, or XML files as well as output to the network via TCP, to the console or into STL data streams. **P1.3** "Log files" for data should be stored using the serialization component. **P2.2.1** Performance is important for binary data that are transmitted via inter process communication. **P2.1.2.2** Portability: Serialization has to deal with components that are working together on different operation systems.
Solution: There has to be a common and unified serialization component that is accessible and used in every part of the system where loading, saving, or transferring data in a serialized form is necessary. *Strategy: Use existing serialization library.* There are free and open source serialization libraries such as "libs11n" and "boost serialization" that could be used. They support object serialization and can serialize to XML files. It has to be considered if they are flexible enough for all demands.

Table 6.5: Issue card example of the case study (cont.)

Strategy: Build an own serialization component. If the available libraries are not flexible enough, an own serialization component could be built that can serialize to files, streams, the console in textual, binary or XML format. For XML serialization the "libXML2" could be used. The reflection pattern could be used for a non-intrusive approach for object serialization.
Related Issues: **O1.1** Build vs. buy: There is a preference to use mature free and open source software or to build.

6.2.2 Architectural Synthesis

The input of the architectural synthesis phase comprises the results of the previous phases: the refined and prioritized quality goals and scenarios corresponding to layer I and II of the GSS as well as the influence factors and issue cards. Further important input is architectural knowledge about design principles, architectural tactics, or patterns, as well as existing solution instruments as frameworks, libraries, or tools, which can be used.

During architectural synthesis the transition from the problem space to the solution space is performed. The quality requirements are transformed into functional solutions as described by QASAR (cf. Section 2.2.1). Together with the functional requirements an initial structure of the architecture has to be found, which is the most creative and difficult part of the architectural synthesis. This is supported by the principles layer (III) and the solution instruments layer (IV) of the Goal Solution Scheme. Afterwards the design can be iteratively refined and improved or revised. Regarding evolvability this is important to be considered because during the whole lifetime of the software demands for changes will arise.

Figure 6.6 shows an overview of the synthesis activities *Architectural Structuring* and *Detailing the Architecture*. The architectural structuring can be coarsely described with the abstract activities *planning*, *constructing*, and *evaluating*, thus being tightly interweaved with the analysis and evaluation phases. It is discussed in more detail in Section 6.2.2.1 with an adapted ADD method. Special emphasis is put on *constructing* to show the integration of the layers III and IV of the Goal Solu-

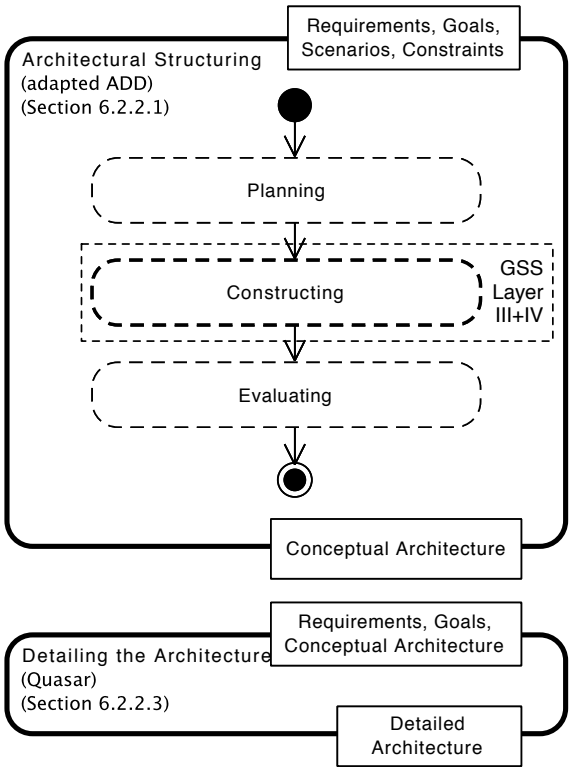


Figure 6.6: Overview of the activities of the synthesis phase

tion Scheme. The resulting conceptual architecture can be concretized and detailed with activities of the Quasar method which is subject of Section 6.2.2.3.

6.2.2.1 Structuring the Architecture

The Attribute-Driven Design (ADD) method was evaluated to be valuable for architectural design concerning the consideration of quality goals (cf. Section 2.2.2). For architectural structuring the steps of the ADD method (cf. Figure 2.2) are reused and adapted. The predominant part of ADD is utilized as explained in detail in the original literature [WBB+06, Woo07], which provides enough information. However, some parts are adapted in this thesis to integrate Global Analysis as well as the Goal Solution Scheme concept and need further explanation. Moreover, the applicability regarding the quality goal evolvability is discussed. Figure 6.7 illustrates, how the activity *Architectural Structuring* is refined into steps (called "actions" in terms of UML speech for activity diagrams).

Referring back to Figure 6.6, the steps 1 to 3 of *Architectural Structuring* in Figure 6.7 are *planning* actions and partly related to analysis, especially step 3.

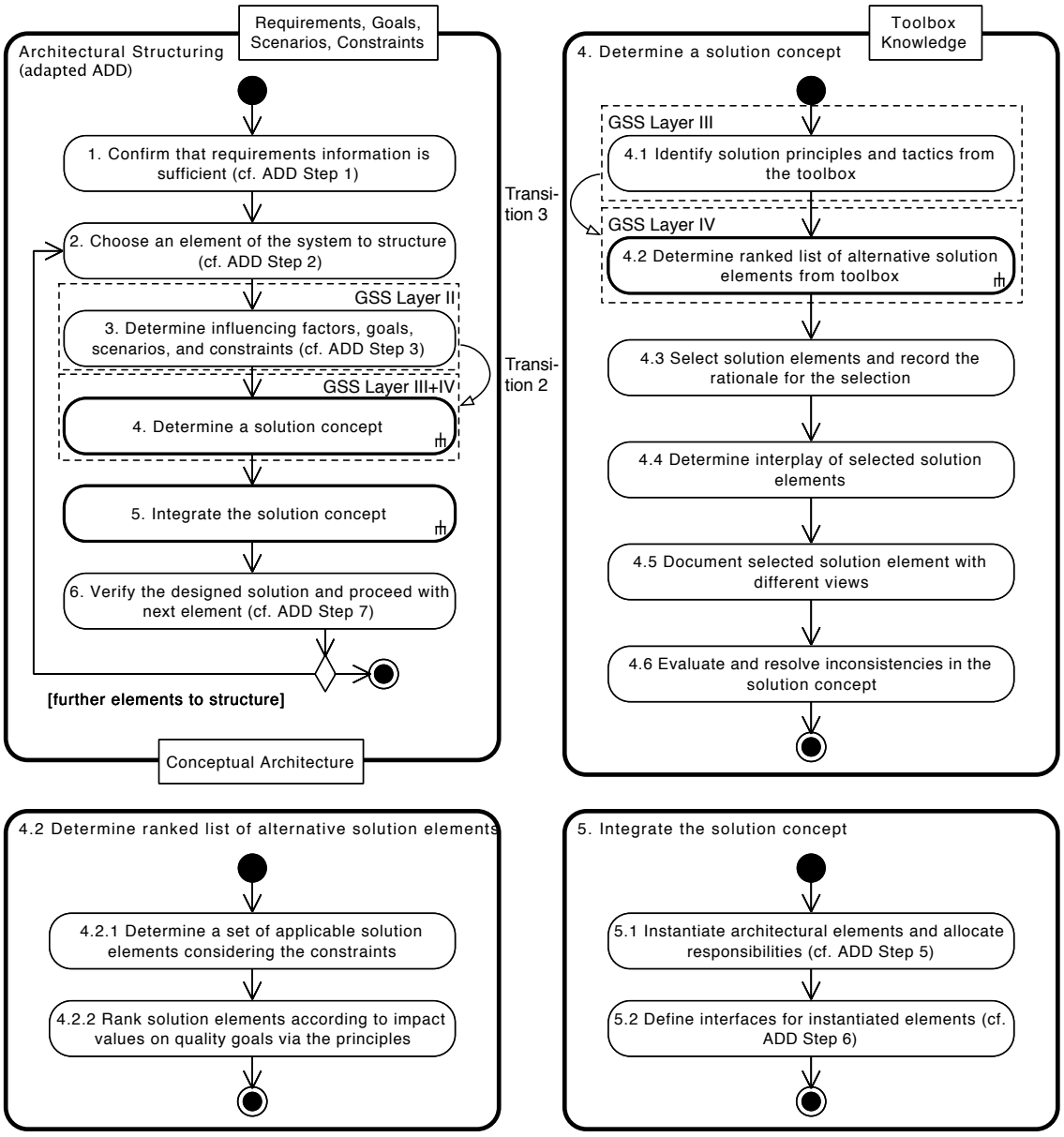


Figure 6.7: Activities of the architectural structuring

Whereas steps 4.1 to 4.5 and 5 refer to *constructing* or the actual synthesis of the design. Step 4.6 together with step 6 refer to *evaluating*, although they should rather be seen as smaller checks than as larger evaluations. All steps not necessarily have to be performed strictly in a sequence as the figure might indicate. Instead the steps can be performed in small leaps and bounds or iterations as discussed above with the backlog concept. For example, if an architect chooses a certain element to structure and recognizes that additional information about goals or influence factors is needed,

a jump back to activities of goal modeling or Global Analysis can be performed.

Determine a Solution Concept Step 4 of architectural structuring is to determine a solution concept. It resembles step 4 of ADD but is adapted to represent the terminology and to use the concept of the GSS. Especially regarding evolvability the selection of patterns as proposed by ADD cannot directly be applied because there is not yet a set of applicable tactics, styles, and patterns for evolvability. Instead, an alternative way with finding subcharacteristics of evolvability and supporting principles as proposed in this thesis is necessary (see also [Rum09]).

The constructing step 4 (a "design step" in ADD) performs the second transition of the Goal Solution Scheme from the problem space to the solution space and is the most important and difficult part of the architectural structuring. The solution concept ("design concept" in ADD) balances between the functional and quality goals of the architecture using design principles and solution instruments. A solution concept can comprise several solution instruments that address a certain issue. It refers to the strategies of the issue card's solution section (cf. Table 6.5). The design principles and solution instruments are the architectural means as proposed by the EMPRESS approach. Handling means is concerned with issues of identifying proper ones, evaluating them compared to the quality goals, as well as making trade-offs between them. These issues are discussed with the following substeps.

Step 4.1 The determination of a solution concept starts with the identification of solution principles performing the second transition of the Goal Solution Scheme. For details of this transition see Section 5.1.2 and 5.2.2. This step is similar to ADD's identification of so-called design concerns using architectural tactics. Regarding evolvability, ADD does not directly provide architectural tactics. However, with the refinement of evolvability into subgoals, such as modifiability and testability, as discussed in Section 5.3.1 the architectural tactics for modifiability [BBN07, BCK03] and testability [BCK03] can be utilized anyhow.

Step 4.2 Based on the identified principles appropriate solutions instruments have to be determined. This requires a considerable amount of creativity. However, experience, as well as a systematic approach can help. Both is provided if the third transition of the Goal Solution Scheme is performed by mapping the principles to solution instruments of the architect's toolbox, as for example patterns or technical

components. Details of this mapping are described in Section 5.1.2 and 5.2.3. This step constitutes an important contribution of this thesis for a systematic guidance from the problem to the solution space.

For the determination of appropriate solution instruments from the architect's toolbox a decision procedure with two refined sub-steps is proposed:

Step 4.2.1 Architectural constraints are used to determine a set of applicable solution instruments by eliminating all unsuitable ones.

This first step is necessary, since of course the decisions on architectural solution instruments are not only influenced by the quality goals but also by other technical or organizational influence factors that may represent constraints for the architecture. As an example consider the cutout of the Goal Solution Scheme in Figure 6.8. The technical component *JGoodies* [JGo10] explicitly supports principles such as *follow platform* as well as *balance and symmetry* by following the functional solution elements *platform support* and *alignment of visual elements* (see also [Bod08, BR09]). These principles facilitate usability via some of its subcharacteristics, which are highly-prioritized quality goals of the case study (cf. Table 6.2). However, *JGoodies* cannot be chosen if the project demands for the C++ programming language, as analyzed with the influence factors (cf. Table A.2 in the Appendix), because it is based on Java and Swing. In this way, step one restricts the set of applicable solution instruments by preselection. This reduces the number of instruments to be ranked in the next step, and thus reduces the complexity of the decision support task.

Step 4.2.2 All solution instruments in the remaining set of the previous step are evaluated and ranked regarding their influence on the relevant design principles and quality goals.

For the ranking of the architectural solution instruments remaining from step one a quantitative rating of these solution instruments is needed regarding their positive or negative influence on quality goals. A rating of patterns, for example, can be taken from the literature if suitable [Woh08, HA07b, TKM03]. But usually pattern catalogs provide only qualitative information about the influence on quality goals. In this case quantitative impact values can be determined as explained in Section 5.3 for evolvability.

To enable not only a ranking for one main quality goal but also to consider prioritization of several quality goals, multi-criteria decision methods, such as the

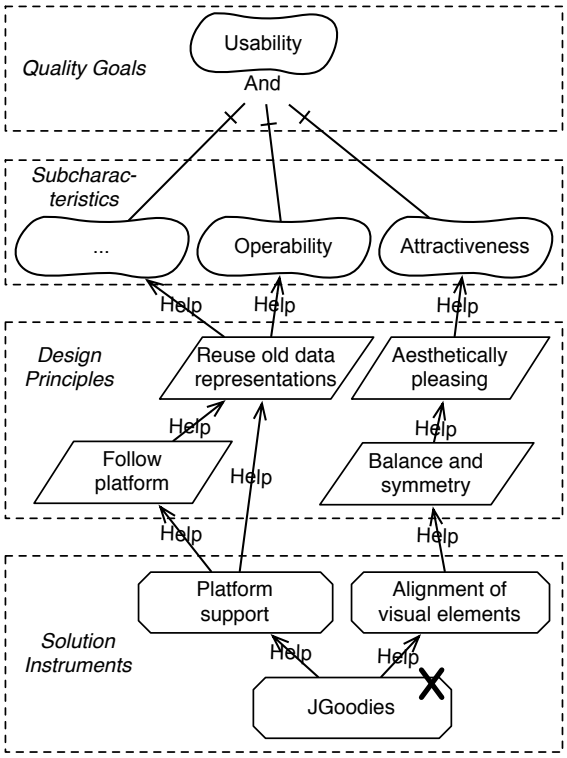


Figure 6.8: Cutout of GSS concerning usability and JGoodies (adapted from [BR09])

Analytic Hierarchy Process (AHP) can be applied. An advanced method based on AHP is the Systematic selection of Software Architecture Styles (SYSAS) [GEM10]. Applying AHP has also proven well in the ArchDesigner approach [ANGB+05].

Step 4.3 to 4.6 The structuring steps 4.3 to 4.6 complete the determination of a solution concept. According to the ranking the architect selects the most suitable solution instruments and records the rationale of the decisions. The important decisions should be documented in a structured way. A template for architectural decisions, for example, is suggested in [TA05]. It comprises the attributes issue, decision, status, assumptions, alternatives, rationale, implications, related decisions, related requirements, related artifacts, related principles, and notes. Regarding the issue cards from Global Analysis at least the chosen strategy for the solution should be noted. Furthermore, a decision should be represented with a traceability link.

After selecting the solution instruments also their interplay should be determined. Moreover, the solution instruments, such as patterns, should be documented in different architectural views to represent aspects as their structure as well as their

internal and external behavior. Architectural views are defined, for example, by the Siemens' 4 Views approach or the 4+1 View Model of Kruchten [Kru95], which was extended by a decision view in [KCD09].

Regarding the case study example, for serialization the two strategies presented in the issue card of Table 6.5 were determined. From the architect's toolbox the solution instruments *libs11n* and *boost serialization* were identified. They are alternative candidate solution instruments next to an own serialization component that uses a *reflection* mechanism and utilizes the *libXML2* library for handling XML files. Finally, the second alternative with creating an own serialization component was selected. This solution was preferred, because it was flexible enough to meet all requirements. For example, the solution had to be lightweight with low overhead for its utilization and the serialized data should be human-readable to provide good usability. It needed to be portable in a way that serialized data can be deserialized on different types of systems. These special requirements inhibited the usage of existing libraries. For the own solution, although difficult to implement, the known pattern *reflection* was implemented. This provides a non-intrusive approach for serialization without the need to adapt existing classes, which in turn supports evolvability.

As an example for an architectural view, Figure 6.9 shows a structural view for the communication framework and its relation to the `Serialization` component. Additional diagrams for the behavioral view can be found in Appendix A.3. To meet the quality goals and functional requirements for the communication framework an architectural style with *implicit invocation* based on the *publisher-subscriber* pattern was chosen. So-called `Channel`s store and manage the data of the system. So-called `Unit`s are the processing components, which subscribe themselves to certain `Channel`s. Then, they are informed about data changes in the `Channel`s. Furthermore, the `Unit`s can act as publishers on the `Channel`s. This style was chosen to overcome the limitations of the legacy *blackboard* architecture, which was not manageable any longer. Further solution instruments chosen from the architect's toolbox are the patterns *facade* (for the `Authority` component in Figure 6.9), *client-server* (for remote communication), *reflection* (for realizing the serialization features), as well as certain *adapters*, for example, for encapsulating the hardware drivers.

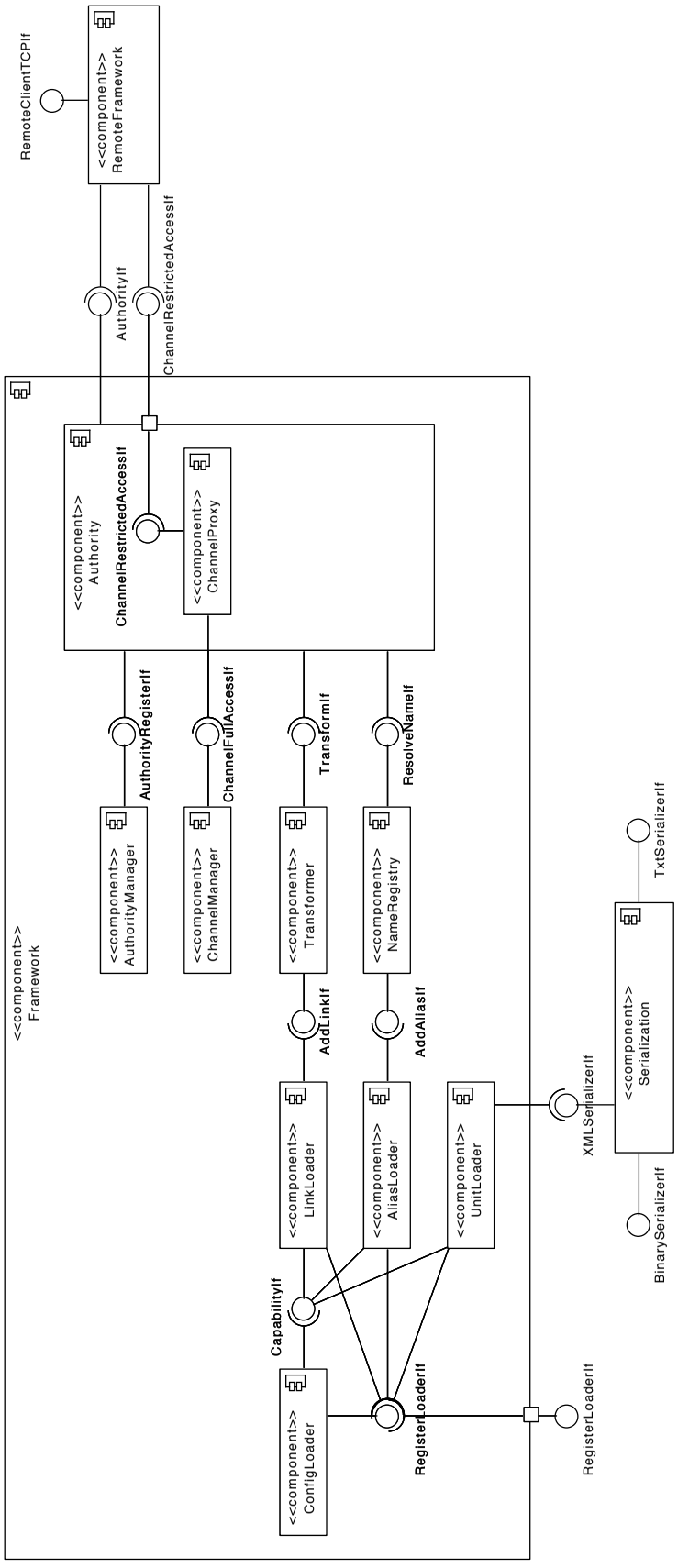


Figure 6.9: Structural view of the communication framework from the robot case study example

Integration of the Solution Concept Step 5 of the architectural structuring comprises the steps 5 and 6 of ADD. They are combined in one activity to emphasize that the integration of a solution concept not necessarily has to be performed directly after the determination and not only for one solution concept. This was a limitation raised from practical application of ADD in [Woo07]. ADD weakly implies a sequential development of the architecture element by element. However, in a larger setting several architects can be assigned to different elements of the architecture to work in parallel. This requires the architects to cooperate and integrate or merge their solution concepts together.

For integrating the solution instruments of the selected solution concept the transformations of QASAR can be applied. The transformations can be chosen according to the abstraction level of the considered architectural element to be structured (system, subsystem, component, etc.) and the type of the solution instruments. QASAR proposes five transformations (cf. 2.2.1): *a*) impose an architectural style, *b*) impose an architectural pattern, *c*) apply a design pattern, *d*) convert a quality requirement into functionality, or *e*) distribute requirements.

For integrating and merging the solution concepts, a sequence of change operations of different types has to be applied. As change operations often *add*, *delete*, and *modify* are mentioned. However, *modify* is ambiguous and can, for example, mean to replace or split an architectural element. A set of six more elementary change operations are the "modular operators" of Baldwin and Clark [BC00]:

1. *Augmentation* - Add an element.
2. *Exclusion* - Remove an element.
3. *Splitting* - Break an element into two sub-elements.
4. *Substitution* - Replace an element, e.g., with an improved or refined version.
5. *Porting* - Move an element up in the design hierarchy by providing specific adapters to other environments.
6. *Inversion* - Move an element up in the design hierarchy by loosening it from its initial context, e.g., a subroutine for printing, which exists in several variants in different components, is generalized and exposed as a separate printing component.

As the result of the merge, the solution concept and its instruments have been integrated into the whole architecture as functional elements. All responsibilities that are due to functional and quality requirements are assigned to these elements, which might be components.

6.2.2.2 Top-down vs. Bottom-up Structuring

The last section specified steps for dealing with elements or building blocks of a software architecture during structuring. However, there is still an open question on how to proceed regarding the abstraction level. In principle there are two ways to proceed during architectural structuring—top-down and bottom-up. A top-down procedure decreases the level of abstraction from system-level requirements at the top to fine-granular technical building blocks at the bottom. A bottom-up procedure increases the level of abstraction, for example, by starting from existing technical components. They can be combined to higher level services and components, which meet the requirements.

The conventional development processes suggest a top-down decomposition of a software system. For example QASAR uses the Functionality-based Architectural Design (FAD) method for a functional decomposition of the software. Unfortunately, FAD neglects the quality requirements in the first place. ADD treats quality requirements, functional requirements, and other constraints as equally important. But it is restricted to a purely top-down-oriented decomposition applying its steps in a recursive manner.

Concerning quality attributes sometimes an adapted way to proceed seems to be necessary. There might be a pre-dominant type of decomposition in certain circumstances. For example, regarding security it is necessary to separate the security-relevant parts of a system from the non-relevant parts as discussed in detail in an earlier publication [BFKR09].

Regarding evolvability Pizka and Bauer [PB04] argue that the process employed during initial development has a major influence on the subsequent evolvability of the resulting software system. They discuss that a bottom-up approach seems to support increased independence from changing requirements because evolution is more independent from technical aspects than from requirements. While purely top-down developed systems are likely to match the (functional) requirements, there is no running system until finishing development, and there is a high impact of

changing requirements. On the other hand bottom-up development quickly produces a rudimentary running system that might not fully fit the requirements at the end. But it is built on a more stable technical platform, which is hardly influenced by requirements and provides increased stability of the derived concepts.

As a consequence, the architectural structuring, which is usually performed top-down according to traditional software processes, should not completely disregard the bottom-up approach to support evolvability. An approach balancing top-down with bottom-up structuring might be useful. Also Graham et al. [GKW07] argue for the consideration of agile concepts to concurrently work both top-down for designing architectural structures that meet the quality goals as well as bottom-up with experimentation and refactoring for questions that are difficult to answer analytically.

6.2.2.3 Detailing the Architecture

Structuring the architecture by following an ADD-oriented approach as described above results in a conceptual architecture that needs to be detailed before implementation. As evaluated in Section 2.2.5 the Quasar method is an appropriate means for detailed design because it results in well separated application-specific and technology-specific components. The general activities of Quasar are depicted in Figure 6.10.

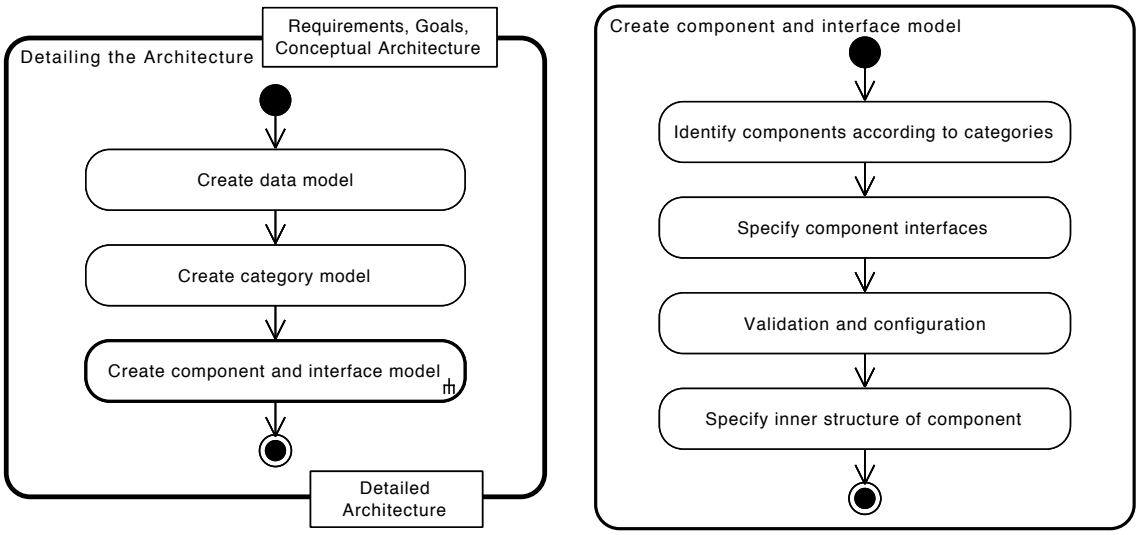


Figure 6.10: The activities and steps of Quasar for detailing the architecture

Following Quasar, first a data model is created with entities of the domain, their attributes, and relationships. A data model can be created using UML or with

classic Entity-Relationship diagrams. Second, the software categories of the software architecture are modeled in a graph to determine the communication paths of the software system between application-specific and technology-specific parts using explicit interfaces.

In the case study the modeling of categories actually was not applied directly because it would have been additional effort to train the developers in applying the Quasar method. Nevertheless, a separation of application-specific and technology-specific components was considered, for example, by the encapsulation of hardware drivers in specific adapters. Figure 6.11 shows a category graph as it can look like for parts of the robot case study using a UML class diagram notation (cf. [Bod08]). The communication between application-specific and technology-specific categories should only use interfaces stereotyped as «0-Software» according to the paths in the graph. As an exception the software category *RobotGUIQt* for the GUI-specific architectural components is stereotyped with «R-Software» because it needs to communicate with both the application-specific categories and the technical category for the Qt framework[2]. However, it should transform the data to be visualized with widgets provided by the Qt framework in a controlled way.

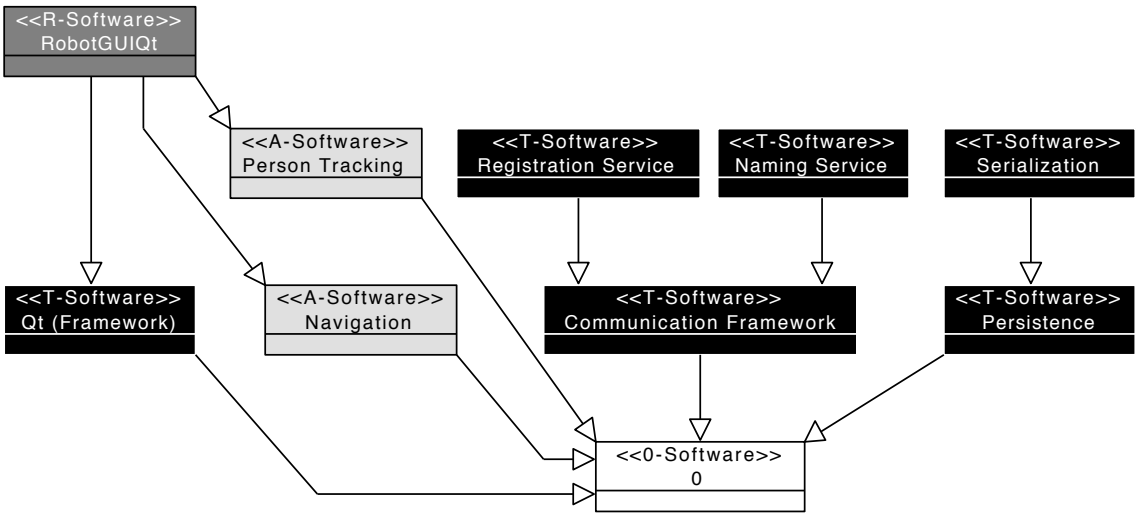


Figure 6.11: An exemplary category graph for the robot case study

In a third step of Quasar, the components and interfaces are mapped on the categories and specified in detail following a four step procedure as depicted in the right of Figure 6.10. As a result of the Quasar activities, a detailed architecture

[2] `http://qt.nokia.com/`

is specified, which can be evaluated and afterwards implemented in the realization phase of the development process.

6.2.2.4 Conflict Resolution and Trade-offs

During the whole architectural design phase conflict resolution and making trade-offs can be necessary at different levels of abstraction of the design and spread over the various activities. Regarding requirements engineering Robinson et al. [RPV03] identified six clusters of general means for conflict resolution, which in this thesis are adapted to the goal-oriented architectural design method:

Relaxation, Generalization Goals and requirements are relaxed for a lesser restriction of the solution space, which allows more alternative solutions to be chosen from. Generalized solution concepts can replace a conflicting concept to satisfy a broader range of goal values in the problem space.

Refinement, Specialization Goals or architectural elements are refined into specialized sub-goals or sub-elements. This enables the identification of conflicts and an easier resolution for the specialized elements. The contribution dependencies of the goal models can help to identify conflicts during refinement.

Compromise To resolve a conflict over a certain value with a compromise relies on the identification of finding another substitute value. During design such values can be quality properties of certain architectural solution instruments. Selecting specific patterns usually makes a compromise or trade-off. Prioritization of the conflicting goals or influence factors can ease the selection.

Restructuring Restructuring refers to an attempt to change the context of a conflict. Restructuring means, such as architectural refactorings, can improve the quality properties of a solution with a better trade-off by enhancing the structure of the architecture to better follow design principles. This in turn has a positive influence on the satisfaction of the goals or requirements that might be conflicting.

Re-enforcement, Re-planning Re-enforcement is an attempt to avoid a conflict by restructuring its precondition. For example, sending multiple notifications to a component can avoid a non-response conflict, which might be due to a too

high workload of the component at run-time. Re-planning refers to choosing an alternative requirement or solution concept.

Postponement, Abandonment An identified conflict can be postponed for a resolution to a later phase or lower level of abstraction according to the layers of the Goal Solution Scheme. In this way a trade-off might be easier with more detailed information about the conflict. Alternatively, conflicting goals can be abandoned.

6.2.3 Architectural Evaluation

The several design decisions made during architectural synthesis should be evaluated before the realization of the architecture. Evaluations can be performed with different measures at different points in the development process. Evaluation measures are, for example, spontaneous ad-hoc evaluations of the architect because of uncertainty for a decision, lightweight discovery reviews in a small team, specific checks for single defined risks, as well as full-fledged architectural assessments in a larger team with all important stakeholders. Relevant techniques are checklists, questionnaires, scenario-based assessment, as well as the utilization of metrics, prototypes or mathematical models for simulation [PBG04].

Regarding evolvability there is no specific evaluation method. As an alternative a general technique such as the Architecture Trade-off Analysis Method (ATAM) or maybe the Architecture-Level Modifiability Analysis (ALMA) [BLBvV04] can be applied, since modifiability is a subcharacteristic of evolvability. Specific metrics for the identification of reengineering measures regarding evolvability are in the focus of the work of Brcina [BBR09].

In the robot case study the scenario-based assessment based on ATAM was performed to evaluate the architecture regarding the satisfaction of the quality goals. For this purpose the goal model from the *goal modeling* activity (cf. 6.2.1.1) was reused for the utility tree required by ATAM. Then, the scenarios for detailing the highest ranked quality goals efficiency, modifiability, reliability, and usability from the *scenario description* activity (cf. 6.2.1.3) were reused and discussed to determine if the designed architecture can fulfill the quality goals. This resulted in an additional documentation of the realizing solution concepts and the identified risks.

Chapter 7

The Traceability Concept

This chapter introduces the traceability concept of this thesis. It describes how the different artifacts of the goal-oriented architectural design method from Chapter 6 are connected with traceability links and how the concept can be categorized based on the state-of-the-art evaluation regarding traceability. Section 7.1 explains some general facts for comprehension about models, dependencies, and traceability links. Section 7.2 states the general idea of the concept and classifies it concerning the state-of-the-art approaches (cf. Section 2.3.2). Section 7.3 introduces the metamodel for traceability links, which is necessary for an exact definition of links and link types as well as for tool support. Sections 7.4 and 7.5 discuss the traceability-link types and the rules for link establishment, respectively. Parts of the ideas of this chapter were already discussed in [BR11, BLR11, RBFL11] and [RPB11].

7.1 Models, Model Elements, Dependencies, and Traceability Links

The general goal of the traceability approach of this thesis is to provide a comprehensive coverage of traceability spanning all models and activities of the goal-oriented architectural design method. This should facilitate evolvability by explicit dependency relationships with traceability links as the basis for further analyses. The links can be used for checks of coverage and completeness regarding requirements, for the evaluation of the design regarding solution principles and quality properties, or for change impact analysis. A design process that is fully traceable across model boundaries enables change propagation from requirements to realization artifacts.

The scope of the models used during design overlaps in a way that real world entities are represented by elements in more than one model. For example, an actor can be model in a URN goal model as well as in a UML use case diagram. For the utilization of these relations the various related model elements have to be interconnected across model boundaries. This can be achieved with traceability links. Two types of interconnections can be determined:

- links between corresponding elements, which represent the same entity of the real world, and
- links between related elements, which represent related entities of the real world.

The model elements to be linked are of a broad variety. All models from the goal-oriented architectural design method contribute to the set of model elements, ranging from goals over influence factors to issue cards and architectural solution elements, such as components and interfaces.

The relations to be considered for the traceability concept originate from different sources:

1. All types of relations that are represented by the models their selves,
2. Dependency relationships that have to be represented explicitly because they are not—or not completely—expressed by some models,
3. Dependency relationships that have been tracked by the developer during design activities and that have to be explored to understand the problem-solution mapping, and
4. Implicit relations that are caused by overlapping representations of real world entities in different models, as e.g., actors in UML and URN models.

In this regard traceability links actually are a subset of all existing dependency relationships. They represent the way a developer went (no. 3). However, these links technically are a good means to express also further dependencies of interest, which do not represent the way the developer went, because they can provide auxiliary information for further analyses. In the following all relationships, which are made explicit, are called traceability links or just links.

7.2 Overview and Classification of the Concept

The general idea of the traceability concept is to integrate and link all artifacts of the goal-oriented architectural design in a model repository as depicted in Figure 7.1. The scope of the concept is on *post-requirements specification traceability* (or design traceability) as it starts with dependencies from requirements or goals to design artifacts. A special contribution of this concept is the *comprehensive treatment of artifacts* and the special consideration of architectural analysis artifacts (factor tables and issue cards), which could not be found in existing approaches. This is an important means for tracing the transition between model elements from the problem space to model elements of the solution space.

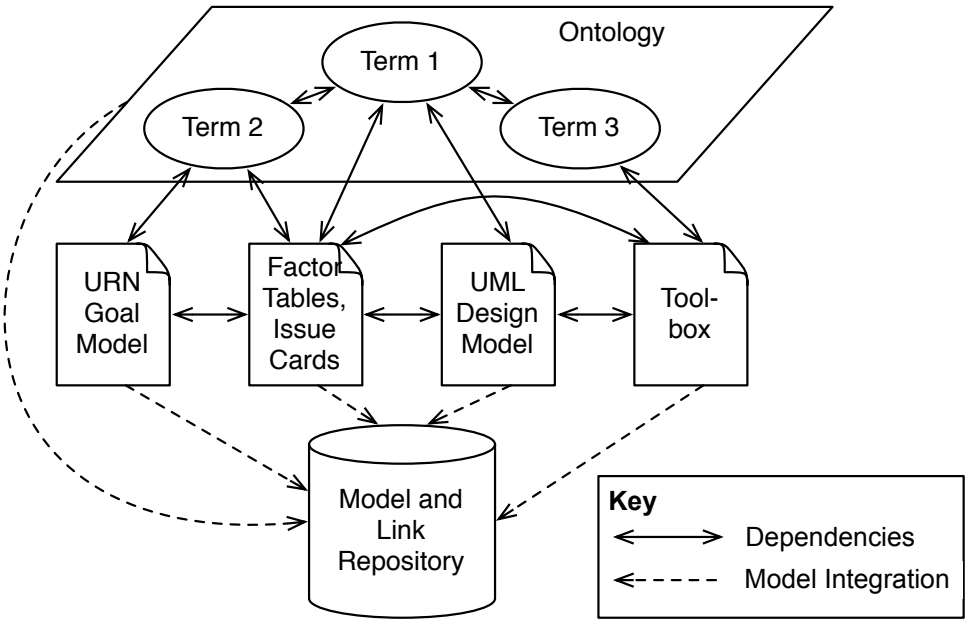


Figure 7.1: Overview of the repository-based traceability concept

The various artifacts of the design method are expressed as *heterogeneous models*. All these models are integrated into a *common repository* to enable *external link establishment*. The models should be expressed as far as possible with *standard modeling languages*, such as URN, UML, and OWL. In this way standard CASE tools can be used for managing the heterogenous models, and the models are not polluted with traceability link information. Moreover, the models and *links can be versioned* with the repository. All dependencies between elements of these models can be represented as traceability links inside the repository. Dependencies can be:

- *inter model dependencies*, e.g., between URN goal models and factor tables, or between issue cards and UML diagrams; as well as

- *intra model dependencies*, e.g., between quality goals, or between UML components and interfaces.

Additionally, an ontology is used to represent the meaning of terms and the relations to natural language expressions, for example, in the factor tables. The *ontology structures relations between terms* as a kind of glossary and thesaurus. It is used in this limited fashion for aspects that cannot be expressed by existing models. The ontology is not used to represent the complete analysis and design knowledge with all entities and relations because this would have the drawback of limited rigor in comparison to more specific models.

For managing the traceability links and their associated information inside the model repository a separate traceability metamodel is required. For representing links between textual descriptions a *hypertext concept* is used. Details on the definition of the traceability metamodel and the hypertext concept are described in Section 7.3.

Dependency identification and traceability link recording is realized with a *rule-based approach* based on existing works of Spanoudakis and Zisman et al. [SdGZ03, SZPMK04, JZ09]. This has the benefit of links with higher quality because of better precision and recall in comparison to information retrieval based approaches. Moreover, the rule-based approach is *complemented with an information retrieval technique* to support the detection of dependencies based on string comparison of names and attributes of model elements.

Besides, a rule-based approach can be used to enhance the traceability links with *semantic information in form of specific link types and attributes* that improve the utilization of the traceability links in later analyses. Types of traceability links are discussed in Section 7.4.

Furthermore, the rule-based approach is chosen to provide *(semi-) automated link establishment*. The approach is (semi-) automated because rules have to be defined manually but the links are established automatically by applying the rules. Further, the linkage of textual descriptions and specific design decisions demands for a manual establishment. By applying rules as near as possible after the development activities the quality of the resulting links is better than with retrieval afterwards.

Details of the rule concept are subject of Section 7.5.

7.3 Traceability Metamodel and Hypertext Concept

Following the general ideas of the traceability concept the definition of an explicit traceability metamodel is required for the exact definition of traceability links and their auxiliary information, such as a link type, as well as for storing and versioning the links in a repository. This metamodel is discussed in Section 7.3.1. Moreover, to enable the linkage of terms in natural language descriptions they need to be represented as model elements. This issue is addressed with the hypertext concept in Section 7.3.2.

7.3.1 The Traceability Link Metamodel

Since there is not yet a standardized metamodel for traceability links, for this thesis a metamodel was defined based on existing approaches [LG05, EAG06, MPR07, DKPF08], as for example the traceabiliy meta-type (cf. Section 2.3.3). The metamodel is shown in Figure 7.2.

The metamodel comprises a `Trace` class, which represents the way a developer went as a chain of traceability links. Correspondingly, it can consist of an ordered set of `TraceLink`s, the source and target elements of which fit together. This allows to express transitive relations between model elements, which cannot be expressed by single traceability links. `Trace`s can be created and extended by aggregating several existing `TraceLink`s that build a chain, which usually would be done during forward engineering. For reengineering purposes on the other hand, one could also start with a general `Trace`, for example, from a requirement to an implementation class, and refine it with `TraceLink`s step by step as more information is gained during software comprehension. `Trace`s support the utilization of traceability links. `TraceLink` and `Trace` share an abstract base class `TraceElement` with common attributes. A `TraceLink` comprises the attributes:

- `Name` mainly for visual display,
- `Description`, for example, for explaining the reason for a manual link,
- `Status` concerning, for example, the validity or correctness of a link,

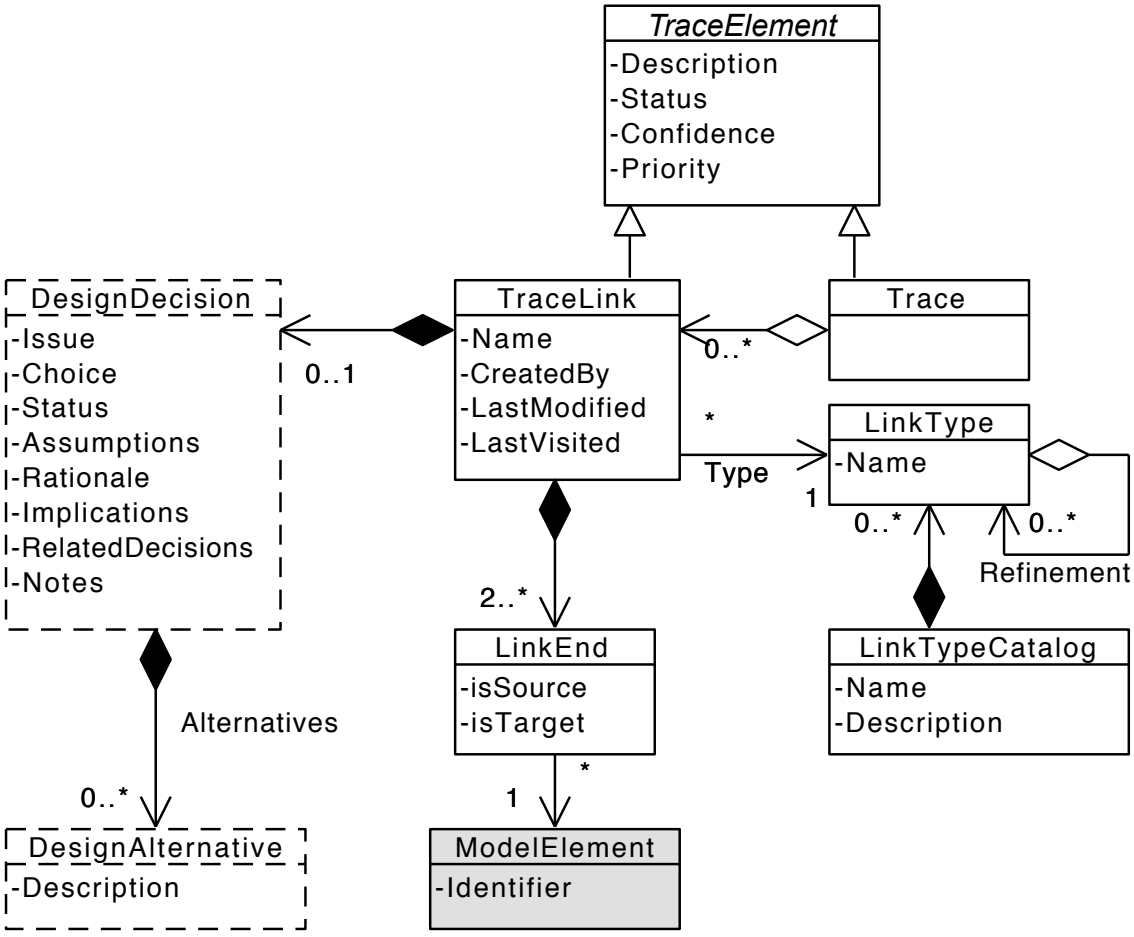


Figure 7.2: The defined traceability metamodel

- `Confidence` expressing the certainty of correctness, e.g., the accuracy of rule-matching,
- `Priority` for the utilization of the link, for example, expressing its relevance for software comprehension,
- `CreatedBy` a rule or developer responsible for the creation,
- `LastModified` and `LastVisited` as dates of creating and editing for further analyses. This can be used, for example, to implement a function of oblivion for determining the importance of a link.

Furthermore, a `TraceLink` has at least two `LinkEnd`s that connect the source and target `ModelElement`s. This potentially enables n-ary links and the flexibility to add further attributes to the connected elements. The `ModelElement` class has to

be a common base class of all artifacts to be traceable and has to provide a unique identifier, which is a prerequisite to enable the tracing.

Moreover, a `TraceLink` has a certain `LinkType` that characterizes the dependency relationship represented by the traceability link. This is important for the traceability links to hold a certain semantics. `LinkType`s have a refinement association for hierarchical clustering and can be grouped into `LinkTypeCatalog`s for better manageability in the repository. Details on the specific link types are explained in Section 7.4.

Besides, a `TraceLink` optionally can be connected with a `DesignDecision` that explains the choice of a certain `DesignAlternative`. The `DesignDecision` class reflects the attributes of the architectural decision template of [TA05] mentioned in Section 6.2.2.1 on page 113.

7.3.2 The Hypertext Concept

Some architectural design artifacts, as for example the factor tables and issue cards of Global Analysis, make heavy usage of natural language descriptions. To enable not only tracing of the factor tables, influence factors, or issue cards on the whole but on a more fine-granular level, the textual descriptions have to be split into model elements with a unique identifier. For this purpose, a hypertext concept is used to represent single relevant terms of the textual descriptions as a model element that can be source or target of traceability links (a `TraceLink` instance). The metamodel of the hypertext concept for textual descriptions is depicted in Figure 7.3.

The idea of the hypertext concept is the following. Instead of using a string for each larger textual description that might contain several traceability-relevant terms, it is represented as a `Hypertext` instance, which in turn is a `ModelElement` with a unique identifier. This `Hypertext` consists of `TextElement`s that represent its content—the actual textual description. A `TextElement` can either be a `Term` or a `Link`. A `Term` is just a replacement of a string and the default content of a `Hypertext` element. The content of a `Hypertext`, in the simplest case a single `Term`, can be split and partly replaced, or even fully replaced with a `Link`. A `Link` has another `ModelElement` as its target.

In this way each textual description represented by a `Hypertext` can contain links to other elements of any model, as for example terms of an Ontology modeled with OWL. However, these `Link` elements are rather intended for the visual representation

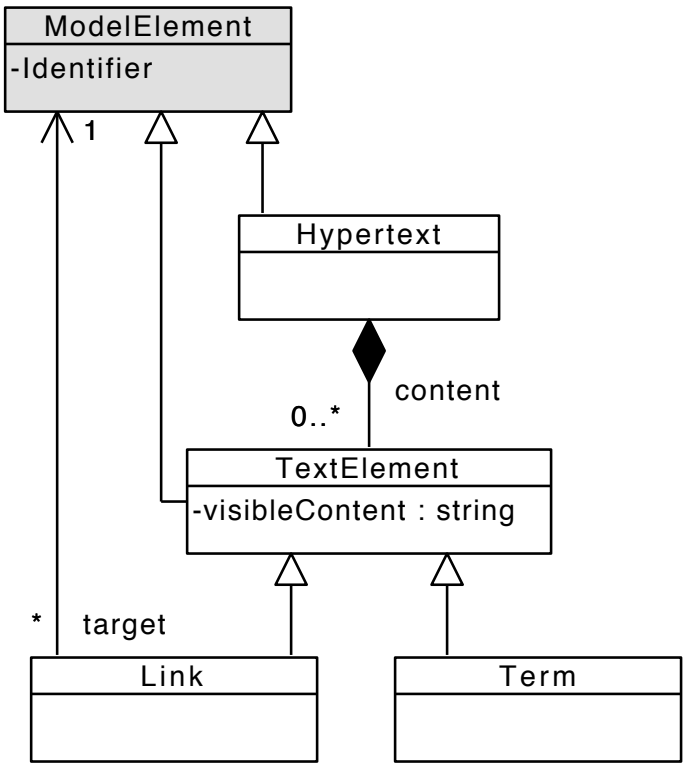


Figure 7.3: The metamodel for the hypertext linking concept

of a traceability link in a text field of a graphical user interface. Therefore, these traceability links should also be represented as a `TraceLink` of the traceability link metamodel in Figure 7.2. In this case one `LinkEnd`—the source—of a `TraceLink` refers to the `Link` element of a `Hypertext`. The other `LinkEnd`—the target—refers to the `ModelElement` referenced by the `target` attribute of the `Link` element.

As an example take the description of the product factor *P1.2 Serialization* from Table 6.4, which is: *"Data (objects) have to be (de-)serialized with a unified strategy, in different data format types (XML, binary, plain text) and to different targets (file, console, TCP, STL streams)"*. For this description Figure 7.4 shows the linkage of one term *"XML"* (represented as a `Link`) to an `OWLClass` instance, which represents the term *"XML"* in an ontology model. This can be useful because the ontology might also contain further information about XML, such as the explanation of this abbreviation. Furthermore, Figure 7.4 illustrates the split of a `Hypertext` content from one `Term` into one `Link` and two `Term`s with an object diagram, which shows the state before and after the linkage. The figure also depicts a possible graphical representation of the `Hypertext` content.

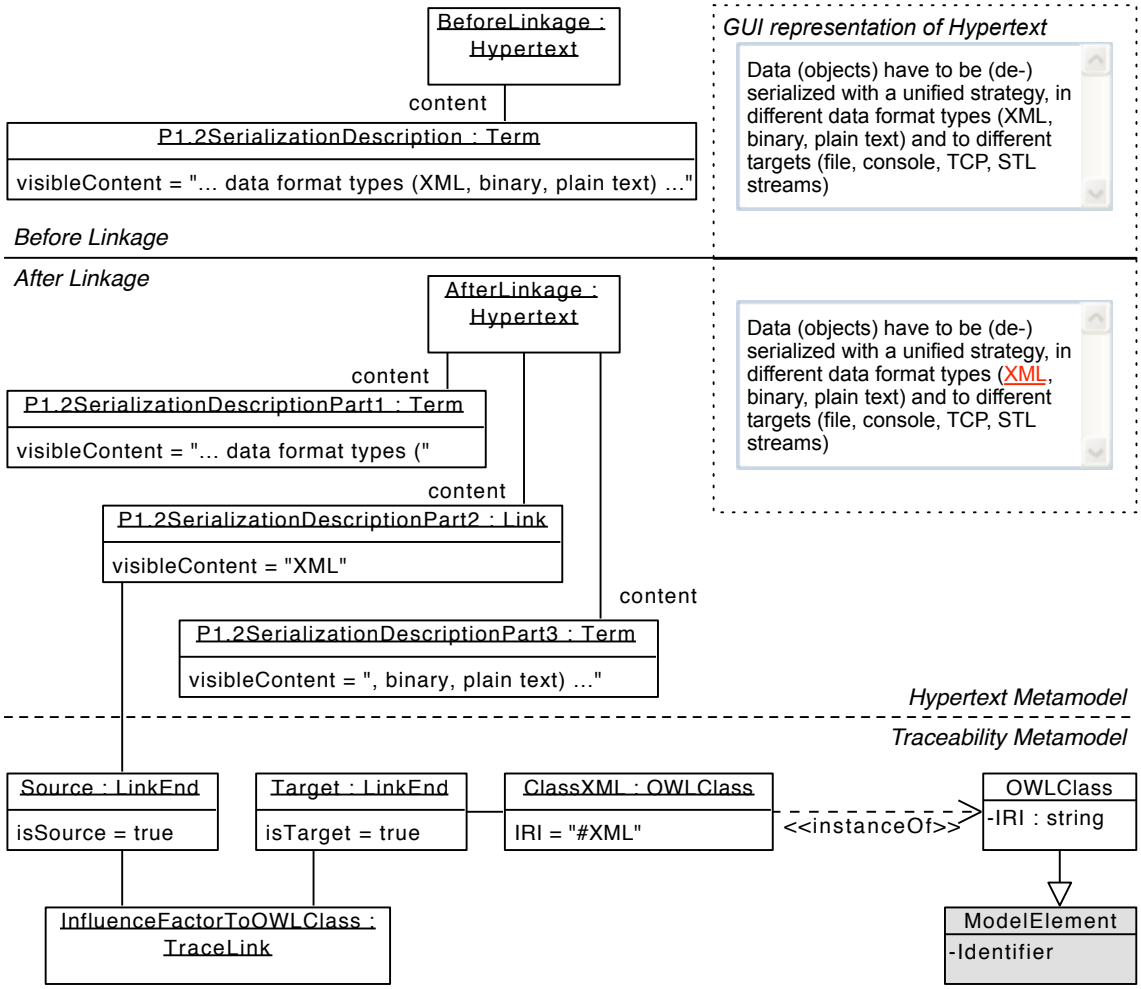


Figure 7.4: A linkage example using the hypertext concept

7.4 Traceability Link Types

As already mentioned, semantic information attached to traceability links, such as design decisions or link types (cf. Section 7.3.1), can enhance the analyses performed on the traceability data. For example, a *realization* dependency between model elements (e.g., component and interface) might result in a higher impact of a change on one of the model elements than this might be the case for a simple *overlap* dependency. This could be due to the fact that the overlap dependency is only based on the similarity of the model elements' identifiers, which carries some uncertainty.

At first, all kinds of types of relations represented by the variety of models have to be considered for the set of link types. However, only dependency relationships are of special interest because they are important, for example, for change impact analysis.

Therefore, such dependency relations should be expressed explicitly. Since not all of the dependencies are covered by the models themselves through their modeling languages, especially not for cross-model dependencies, these dependencies should be represented with traceability links. To further characterize the kind of dependencies certain link types are used.

Unfortunately, there is not a standardized set of link types, which could be used. However, there are several works that propose and specify certain types of traceability links, for example [Poh96, RJ01, Let02, FZS03, SZPMK04, SZ05, MPR07, JZ09]. Especially Spanoudakis and Zisman [SZ05] propose a consolidated view on link types that are grouped into clusters. Clusters help to ease managing the various, partly synonymous types of links by abstraction.

7.4.1 The Link Type Clusters

Based on the existing works, several link types and clusters were identified for the traceability approach of this thesis to trace dependencies between the various model elements of the goal-oriented architectural design method. They are depicted in Table 7.1. In the following some examples for the link type clusters are given. Each cluster comprises some subtypes or synonymous terms for the same kind of relationship as well as terms for the opposite direction.

The *refinement* cluster comprises two major kinds of dependencies between elements of different levels of detail, either a *decomposition* (*part-of*) or a *specialization* (*is-a*). For example, a quality goal in a goal graph can be *refined by* a *sub*goal, components can be *aggregated* to a system, or a (*super*)class can be *specialized by* a *sub*class.

The *realization* cluster describes dependencies that represent a step forward from the problem to the solution. For example, a use case can be *implemented by* a component, an object is an *instance of* a class, or a product factor is *realized by* a certain component.

The *satisfiability* cluster comprises dependencies between elements that specify certain conditions or properties and elements that meet these properties or conditions to a certain extent. For example, a quality goal can positively or negatively *contribute to* or *influence* other goals in a goal graph or even be in *conflict* with them. Thus some goals might be *satisfied by* others or not. Moreover, interfaces or glossary terms can be *defined by* a specification. Each time a term is used it can

possess a link of the type *defined by* to the glossary.

The *verification* dependencies are used to relate elements that proof the correctness or accuracy of a solution element. For example, a class is *verified by* a unit test, which makes use of *assert* statements, or a use case is *tested by* a test case.

The *use* link types express the utilization of other model elements. For example, a component *requires* or *provides* an interface, a class is *called by* a UML lifeline in a sequence diagram, or a package *imports* another one.

The *causation* cluster subsumes dependencies that represent causal relations that are related to a sequence in time. For example, in a UML sequence diagram one lifeline can *activate* or *deactivate* another lifeline. This activation or deactivation *causes* the creation or deletion of the targeted lifeline.

The *overlap* cluster refers to relationships that express common features, similarity, or equivalence often between elements of different types of models. *Overlap* relations can be identified, for example, between actors in a URN goal model and actors in a UML use case diagram because they both represent the same entity of the real world. Another example are elements that are somehow related because of *similar* terms, such as a UML comment and an influence factor. Furthermore, the OWL SubClassOf dependency is *equivalent* to a UML generalization. In this way even dependencies on dependencies can be expressed with traceability links.

The *evolution* relationships can express a temporal sequence of the development of elements. In this regard a model element *evolves to* another model element during the development, maintenance, or evolution of a system. These types of relations, for example, can express that a requirement has changed and was *replaced by* a new one without having clear causality between them.

Table 7.1: The traceability link type clusters

Link Type Cluster	Description	Subtypes and synonyms	Opposite	Appearance in literature
Refinement, refine	The relationship between elements of different levels of detail, regarding either its constituents or its properties and behavior.	Part-Of Decomposition Sub Is-A Specialization Extension Inheritance	Consist-Of Composition Aggregation Containment Super Generalization	[JZ09], [SZ05], [MPR07], [RJ01], [Let02], [Poh96], [vKP02], [PG96]
Realization, realize	The relationship that represents a step towards the solution.	Implementation Instance-Of	Type-Of	[MPR07], [JZ09]
Satisfiability, satisfy	The relationship between a specification of properties, conditions, expectations, needs, or desires and the element relying on it or contributing to its satisfaction.	Influence Contribution Conflict Inconsistency Condition Constraint Compliance Compatibility Contract Assert Definition		[SZ05], [JZ09], [RJ01], [Poh96], [MPR07]

Table 7.1: The traceability link type clusters (cont.)

Link Type Cluster	Description	Subtypes and synonyms	Opposite	Appearance in literature
Verification, verify	The relationship between a solution and something that demonstrates the truth, accuracy, or validity of its behavior or properties.	Test Validation Simulation Interpretation		[MPR07], [Let02]
Use	The relationship that expresses the utilization of something.	Invoke Require Call Import	Provide	[SZPMK04], [SZ05], [RJ01], [Let02]
Causation, caused by	The relationship between cause and effect regarding a partial ordering in time.	Activate Trigger	Deactivate	[MMMN03]
Overlap	The relationship that expresses the commonality between related elements.	Similarity Equivalence	Difference	[FZS03], [JZ09], [SZPMK04], [SZ05]
Evolution, evolve to	The relationship that signifies the temporal development of elements.	Replacement Based-On		[JZ09], [SZ05], [RJ01], [Poh96], [PG96]

7.4.2 Application and Utilization of the Link Types

The different link types explained above are related to the several design activities. According to certain design tasks different link types are relevant. During goal modeling between goals *decomposition* relationships, *contribution* relationships, or *conflicts* are expressed. The first transition of the Goal Solution Scheme is a *refinement* but also can be seen as a step towards the solution (*realization*). In the other transitions of the GSS *contribution* links model the impact of different solution principles or instruments and express alternatives.

However, it must be noted that the types of relationships of the clusters are not orthogonal. Thus two elements can be related by more than one type of relationship. Therefore, if a certain solution is chosen during architectural design, the respective model elements can not only be related with a *contribution* but also with a *realization* link. Typically this is the case for crossing model boundaries. For example, goals from the goal model can be linked with the type realization to product factors in a factor table and further to issue cards and UML elements. *Satisfiability* links can be used to evaluate certain solution concepts, which again can be *verified by* simulations. In this way links with types of the *satisfiability* or *realization* cluster enable to trace the way a developer went from the problem to the solution space.

As already explained, traceability links are established for intra and inter model dependencies. Intra model dependencies often have the type *use* and its derivates. They can be important for the analysis of changes. However, in a chain of links these types might have to be combined with other types to cross model boundaries. For inter model dependencies often the *overlap* type can be considered. For example, this is applicable between goal models and UML models or to an OWL ontology, which contains terms that are used in other models as indicated in Figure 7.1.

The *evolution* type is strongly related to versioning of model elements. The information about *replaced* model elements during development with links of this cluster are important, for example, for retrieving a change history. Summing up, it can be said that the knowledge about the specific semantics of a type attached to a link is important for different kinds of analyses. Therefore, the link types have to be added to the links during their establishment. With rules for traceability link establishment this is more easy than with information retrieval techniques. How the link types are combined with the rules is explained in the next section.

7.5 Traceability Rules

In the traceability concept of this thesis rules are primarily used to identify and record traceability links between the various model elements of the goal-oriented architectural design method. Nevertheless, the rules can be enhanced and increased to also maintain traceability links. The concept of rule-based link establishment is used because it provides higher values of precision and recall than pure information retrieval approaches as evaluated in Chapter 2.

Generally, the relationships between model elements are a product of the performed development activities. Accordingly, rules can be defined that express under which conditions and in which manner model elements should be linked as a result of the development activities. As described in Section 7.1 there are different sources of dependencies between model elements that have to be expressed by the rules. Therefore, the rules must cover the identification of links between corresponding model elements representing the same entities of the real world (e.g., URN actor and UML actor) and between other related model elements that represent related entities (e.g., two components with a common interface). Moreover, the rules must support intra model dependencies and inter model dependencies.

The identification of these dependencies to a large extent relies on the comparison of the model elements' attributes. Technically this results in a string-based matching of identifiers. A string comparison for equivalence often is two restrictive to find related elements. In this regard information retrieval techniques as n-gram matching can help as a less restrictive similarity measure. Therefore, the rule-based approach of this thesis is combined with n-gram matching. This increases the recall of finding links without loosing two much precision as with pure information retrieval approaches. Since the rules exactly specify the model elements, it is further possible to specify the type of the relationship that is recorded as a link.

Based on the evaluation of the state-of-the-art, the rule-based approach of this thesis follows the general concept for the design of rules proposed by Spanoudakis and Zisman et al. [SZPMK04, SZ05, ZEF00]. Accordingly, the rules consist of three parts as introduced in Chapter 2. The *head* or *element definition* part of the rule defines the relevant model elements that are effected by the rule together with an informal description. The *query* part specifies the logical conditions that express how the model elements, which are specified in the head, are related with each other. This

can include a simple string comparison, for example, of the name attributes of two model elements, or even structural information of the queried model together with a comparison of certain attributes for similarity based on an n-gram algorithm. The third part is the *result definition* or *action* part. It formulates the result processing. In the simplest case this is the establishment of a traceability link according to the traceability metamodel from Section 7.3. However, also further actions are imaginable as, for example, a triggering of validation rules or other analyses steps.

The structure of the traceability rules is defined with an XML Schema Definition (XSD), which can be found in Appendix B.1. Here for easier comprehensibility the parts of the rule definition are explained with example rules. Listing 7.1 shows an exemplary traceability rule for the linkage between UML components and OWL classes. The head specifies the `RuleID` and the `Description` as well as the `Elements`' definitions with `Type` and `Alias`. The query part defines certain `Conditions` such as `BaseConditions` using different comparison operators in their `Type` attribute as well as `LogicConditions` with the logic operators `And`, `Or`, `Not`, and `Xor`, which can be combined and nested. Finally, the `Actions` part specifies the `ActionType` *CreateLink* and the `LinkType` *overlap* for the two model elements to be connected. Potentially all link types of Section 7.4 can be used here. The scope operator "`::`" is used to access the attributes of the model elements.

Listing 7.1: A rule example for the establishment of traceability links between UML components and OWL classes

```
1  <Rule RuleID="TraceRule61" Description="Find␣similarities␣between␣
       UML-Components␣and␣OWL-Classes">
2      <Elements Type="Class" Alias="e1"/>
3      <Elements Type="Component" Alias="e2"/>
4      <Conditions Type="And">
5          <BaseConditions Type="NotNull" Source="e2::umlID"/>
6          <LogicConditions Type="Or">
7              <BaseConditions Type="Contains" Source="e1::IRI"
                   Target="e2::name"/>
8              <BaseConditions Type="Contains"
                   Source="e1::abbreviatedIRI" Target="e2::name"/>
9          </LogicConditions>
10     </Conditions>
11     <Actions ActionType="CreateLink" LinkType="Overlap"
           LinkSource="e1" LinkTarget="e2"/>
12 </Rule>
```

Listing 7.2 shows another example rule. This rule is used to create traceability links based on the links according to the hypertext concept as illustrated in Figure 7.4. Therefore, the rule searches for elements of the type `Link` from the metamodel of the hypertext concept and for any other model element represented by an asterisk (*). If the identifier of the `Link`'s target model element matches with any other model element's identifier, a corresponding `TraceLink` element will be created by the *"CreateLink"* `Action`.

Listing 7.2: A rule example for the establishment of a traceability link based on a hypertext link

```
1 <Rule RuleID="TraceRule69" Description="Connect␣an␣EMFfit␣'Link'␣
    element␣with␣its␣target␣ModelElement">
2     <Elements Type="Link" Alias="e1"/>
3     <Elements Type="*" Alias="e2"/>
4     <Conditions Type="And">
5         <BaseConditions Type="Equals" Source="e1::target::identifier"
              Target="e2::identifier"/>
6     </Conditions>
7     <Actions ActionType="CreateLink" LinkType="Overlap"
          LinkSource="e1" LinkTarget="e2"/>
8 </Rule>
```

As already mentioned, the query part of a rule with the `BaseConditions` can contain different comparison operators for evaluating the model elements' attributes. These operators are (cf. [Leh10]):

Equals checks if the value of a model element's attribute matches a required value

Contains checks if an attribute's value contains a certain string value

IsParent checks if an element has a parent relationship (refinement) with another element in the same model

SimilarTo checks if an attribute's value resembles a certain value

LesserThan checks if an attribute's value is lesser than a required value

GreaterThan checks if an attribute's value is greater than a required value

NotNull checks if an attribute exists in a model

These operators can be combined with logic operators to formulate complex conditional expressions. A special characteristic of the *SimilarTo* operator is that it uses the n-gram matching algorithm to compare attribute values. In this way the rule-based approach is combined with the information retrieval technique. In the future the operators could be supplemented with additional ones. Moreover, further information retrieval techniques can be integrated, such as word stemming, abbreviation expansion, stop-word elimination, or the more complex vector-based distance measures.

Currently, there are 76 rules specified for the identification and recording of traceability links with different types. A complete list of the rules can be found in Appendix B.1. A rule engine is responsible for the interpretation of the rules and to trigger the link creation. Details on this are explained in Chapter 8. The rules can operate on all model elements. Consequently, if change operations, as e.g., the modular operators (cf. Section 6.2.2.1, page 116), are expressed as instances of an explicit metamodel, they can be input for the traceability rules as well. In this way it is be possible to realize also the maintainability of traceability links as proposed by Mäder [Mäd09]. Furthermore, consistency checks would be possible with rules for the validation of models or traceability links.

7.6 Ontology Definition

As already explained in Section 7.2 the traceability concept includes the linkage of ontology terms. Ontologies are semantic networks of related terms and can be utilized to connect terms of different artifacts of the development process. In this way an ontology can be utilized to provide the ability for modeling additional dependencies between different models of the goal-oriented design method. The ontology is only used to facilitate the explicit expression of dependencies and not for the complete representation of other design models. For example, an ontology can comprise terms, such as known quality goals, architectural principles, or solution instruments, and their dependencies. In the traceability concept of this thesis an ontology especially is used to interrelate terms of the textual descriptions, for example, of factor tables and issue cards, with each other and with further model elements. This of course demands for the existence of an appropriate ontology. For this reason next to the activities of the goal-oriented design method an additional activity to model

an ontology was performed for the case study from Chapter 6.

For the establishment of an ontology several methods can be found in the literature [UK95, GF95, BLC96, SPKR97, LGPSS99, SSSS01, NM01]. However, because ontology engineering is out of scope of this thesis a pragmatic approach to establish an ontology was followed for the case study. The ontology was established in parallel to the other design activities. As a first step, the specifications of the different artifacts (factor tables, issue cards, as well as the goal model) were manually scanned for terms that seemed to be of special importance for the case study system. Then, as a second step, these terms were modeled in an ontology with the Web Ontology Language (OWL) using the tool Protégé[1]. The knowledge integrated in the ontology was structured akin to the architect's toolbox. The ontology of the case study is depicted in Figure 7.5. It contains important architectural concerns, solution templates, such as patterns, as well as tools, quality goals, and terms of basic hardware and software technology. The structuring according to the architect's toolbox enables the reuse and extension of the ontology for other software projects.

The benefits of the ontology integration in the traceability concept are: *a*) the facilitation of additional traceability links especially in combination with the hypertext concept for textual descriptions and *b*) the possible treatment of synonyms or homonyms. For example, synonyms can be expressed as equivalent classes in the class hierarchy of terms in the ontology. The ontology also contains general design knowledge in form of relationships. For example, the influence of the patterns on certain quality goals can be expressed by object properties of the OWL classes. In this way relationship types as *partOf* or *influencedBy* can be modeled inside the ontology. The ontology of the case study can be found in OWL/XML syntax in Appendix B.2.

[1] `http://protege.stanford.edu/`

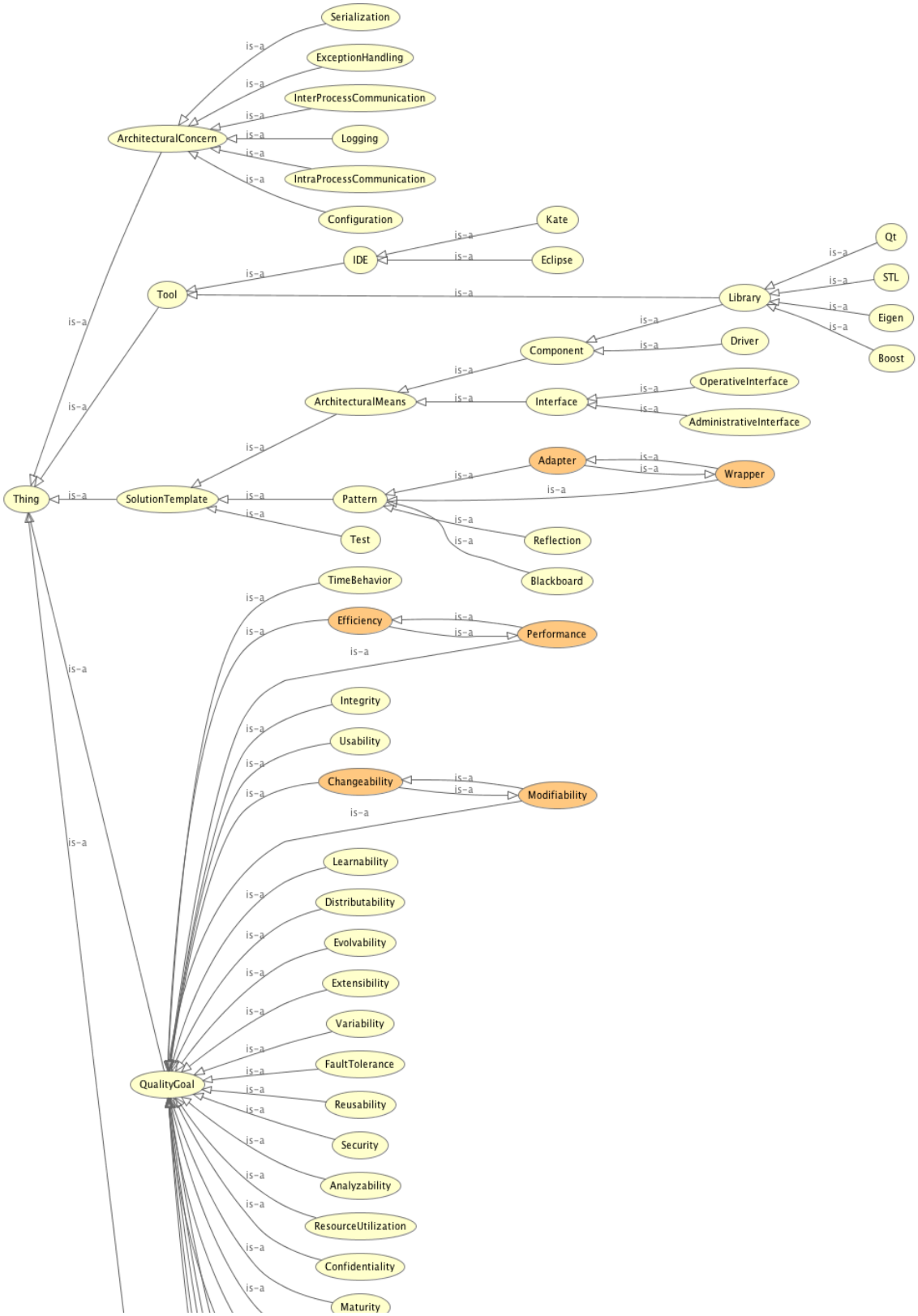


Figure 7.5: Cutout of the ontology used in the case study.

Chapter 8

Tool Support by EMFTrace

This chapter describes a prototype tool called EMFTrace[1], which is based on Eclipse Modeling Framework (EMF) technology. EMFTrace implements and supports the traceability concept introduced in Chapter 7. It provides capabilities to integrate the models of the goal-oriented architectural design method from Chapter 6 into a centralized repository for identifying dependencies between these models and recording them as traceability links. EMFTrace is accompanied by a tool called EMFfit, which allows to model factor tables and issue cards of the Global Analysis approach.

Section 8.1 describes the requirements for EMFTrace and its core concept. Section 8.2 provides information about the architecture and its components and how EMFTrace together with EMFfit realizes the traceability concept. Section 8.3 goes more into detail regarding the technological aspects of the model integration. Section 8.4 describes a typical usage scenario for EMFTrace. Parts of the ideas of this chapter were already published in [BLR11].

8.1 Requirements and Core Concept

The goal of the tool support by EMFTrace is to implement this thesis' concept for rule-based traceability in combination with information retrieval techniques and a centralized repository. EMFTrace shall provide comprehensive traceability for the artifacts of the goal-oriented design method with high values for precision and recall. It is conceived as an extensible platform enabling dependency analysis and traceability link establishment across model boundaries as well as goal-oriented decision

[1] See also the project's website `http://proinf.de/EMFTrace`.

support as a basis for further analyses, such as change impact analysis, or validation and consistency checks. According to this some important functional requirements for EMFTrace were:

1. Integrate metamodels complying to URN, UML, OWL as well as for Global Analysis into the repository.
2. Enable the import of corresponding models from CASE tools into the repository via adapters to enable dependency analysis.
3. Provide export capabilities of the models out of the repository to enable their editing with the original CASE tools.
4. Enable to edit, store, and maintain traceability rules organized in catalogs in the repository with a separate metamodel.
5. Store and maintain traceability links in the repository with a separate metamodel.
6. Support the validation of traceability rules and their import into the repository.
7. Store and maintain traceability link types organized in a catalog in the repository.
8. Support (semi-) automatic traceability link establishment through processing traceability rules enhanced with information retrieval techniques.
9. Support the analyses of traceability links regarding transitivity to combine chains of links as traces.

Some quality requirements were to provide a flexible and extensible platform to support further research activities. Therefore, Eclipse was chosen as the basic technology because its plug-in concept enables the easy exchange of certain components.

The core concept of EMFTrace is illustrated in Figure 8.1. As the basic repository for EMFTrace, EMFStore [KH10] was chosen because it was evaluated the most suitable, available model repository regarding criteria as maturity, supported features, usability, and documentation [BLR11, Leh10]. EMFStore is based on the Eclipse Modeling Framework (EMF) and provides capabilities for managing and versioning EMF-based models in a client-server fashion. By using this technology

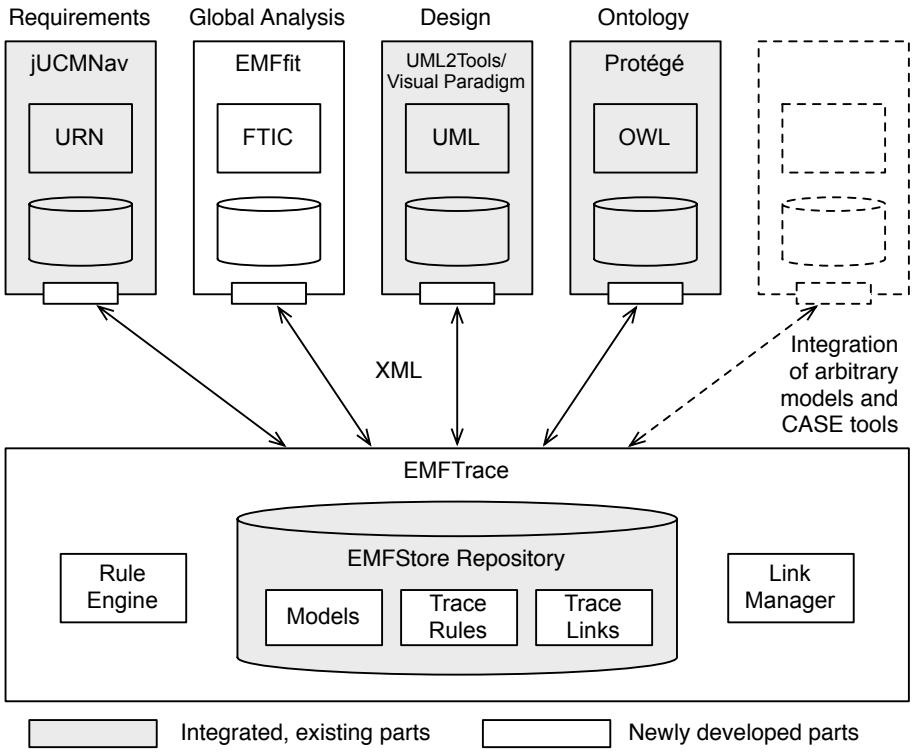


Figure 8.1: Overview of the core concept of EMFTrace

every kind of model can be treated in a unified manner if an EMF-based Ecore metamodel is available for it. Therefore, the metamodels for the traceability links and rules as well as the models for the various development artifacts were created using EMF.

The idea for the comprehensive traceability support was to rely on standardized modeling languages, such as URN, UML, and OWL, as far as possible, instead of creating a customized metamodel for all artifacts of the goal-oriented design method. Standardized modeling languages have the advantage to be relatively stable and seldom subject of changes. Moreover, in this way ordinary and established CASE tools can be used for modeling. Consequently, the CASE tools jUCMNav[2] for URN models, UML2Tools[3] and Visual Paradigm[4] for UML models, as well as Protégé[5] for OWL ontologies were integrated. Additionally, a customized tool called EMFfit was developed for Global Analysis artifacts, because no standardized model or any tool

[2] `http://jucmnav.softwareengineering.ca/ucm/bin/view/ProjetSEG/WebHome`
[3] `http://www.eclipse.org/modeling/mdt/?project=uml2tools`
[4] `http://www.visual-paradigm.com/`
[5] `http://protege.stanford.edu/`

does exist. These tools can provide their models in an XML representation. Besides, the EMFStore repository is based on XML. Hence, XML is the basic integration technology. This core concept is easily extendable for further modeling languages, such as BPMN, and other tools, as long as they provide access to their models via XML. More details on the model integration are discussed in Section 8.3. But first the architecture of EMFTrace and its components such as the rule engine and the link manager are described.

8.2 Architecture

As already mentioned above, EMFTrace is based on Eclipse technology. For this reason the various components are realized as Eclipse plug-ins, which provide the core functionality, a user interface, and the metamodels for the development artifacts, traceability rules, and links. By using the plug-in mechanism, new concepts and features can easily be added. EMFStore provides a server, the actual repository, and a client to operate on the repository. The features of EMFTrace are integrated in the client to provide a unified view and usability. Figure 8.2 shows the components of EMFTrace together with those for EMFfit, which were all implemented and tested with Java [Leh10, Wag10].

8.2.1 EMFStore

The `EMFStore` component represents the EMFStore server. The EMFStore server provides the basic functionality for managing projects as well as for storing and versioning models in projects. To work together with the specific CASE tools, it requires metamodels of the various design artifacts. Furthermore, the traceability link metamodel and a traceability rule metamodel are required. As a result, the following Eclipse plug-ins were developed for URN, UML, OWL, traceability links, and traceability rules:

- `EMFTrace_URN`,
- `EMFTrace_URN.edit`,
- `EMFTrace_UML`,
- `EMFTrace_UML.edit`,
- `EMFTrace_OWL`,
- `EMFTrace_OWL.edit`,
- `EMFTrace_Link`,
- `EMFTrace_Link.edit`,

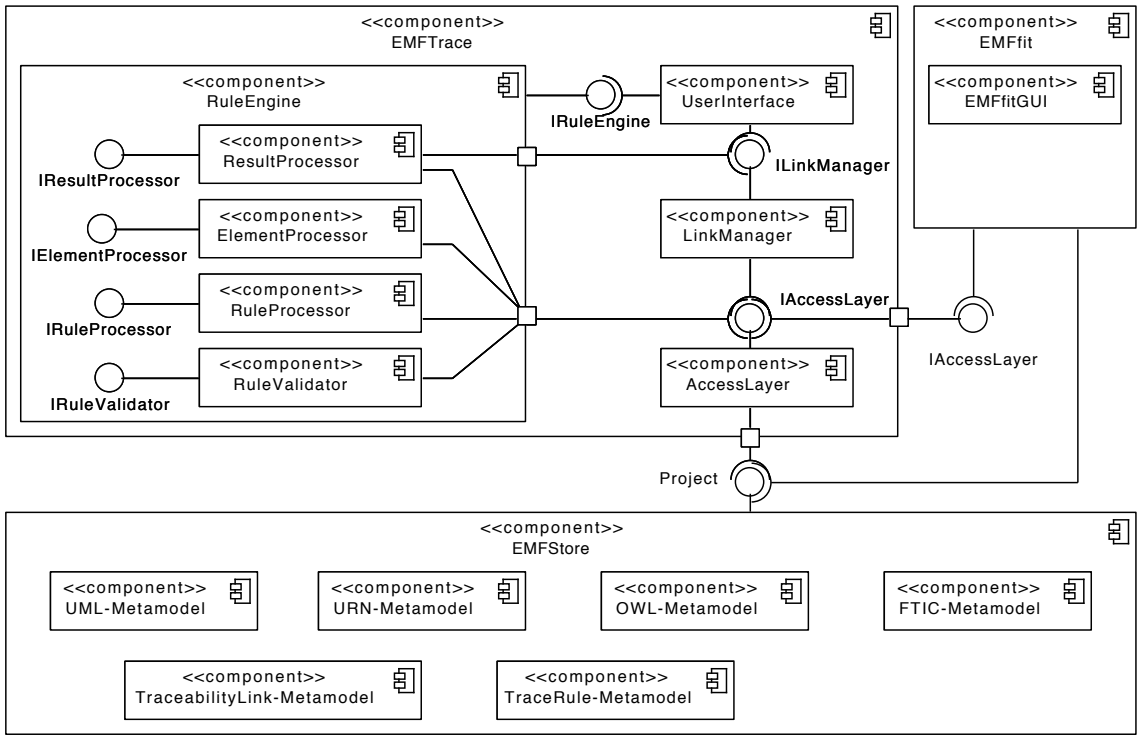


Figure 8.2: Architectural components of EMFTrace and EMFfit

- `EMFTrace_Rules`, and
- `EMFTrace_Rules.edit`.

The `FTIC-Metamodel` for factor tables and issue cards (FTIC) handled by EMFfit is discussed separately in Section 8.2.3. The integration of the metamodels with EMF and XML is discussed in detail in Section 8.3.

8.2.2 EMFTrace

The `EMFTrace` component represents an EMFStore client. It provides all functionality for importing and exporting models into the repository, which can be triggered from the `UserInterface`. Furthermore, `EMFTrace` is responsible for managing rules and link types, for dependency identification and traceability link management. The `RuleEngine` comprises components for the validation (`RuleValidator`), and for the processing of the tripartite rules (`ElementProcessor`, `RuleProcessor`, `ResultProcessor`). The `LinkManager` component is responsible for traceability link management and used by the rule engine to process the action of link creation. Moreover, the `AccessLayer` component enhances the access to the models managed by the EMFStore repository. The corresponding Eclipse plug-ins are `EMFTrace_GUI`

and `EMFTrace_Core`.

Access Layer The `AccessLayer` component eases the access to the models and their elements stored in the EMFStore repository. It provides additional access operations, such as `getAttribute`, `getAttributeValue`, and `getAllChildren`, which go beyond the default operations of EMFStore. Moreover, the `AccessLayer` is used to control when model changes are transferred to the repository server using the "commit" action. Typically the transfer of model changes to the repository is an expensive operation because it needs client-server communication via TCP/IP. For this reason, the `AccessLayer` manages a project cache to enable bundled transfer from the EMFTrace client to the EMFStore repository server and to accelerate the retrieval of model elements during rule processing.

Link Manager The `LinkManager` component is responsible for recording and maintaining traceability links. It uses the `AccessLayer` component to communicate with the repository. The `LinkManager` provides operations to create and delete links, which were identified via rule interpretation. Furthermore, it can analyze traceability links regarding transitive relations and create traces representing chains of links (cf. Section 7.3.1). Since model elements are subject to changes, the traceability links can become inconsistent or invalid. Therefore, the `LinkManager` also provides functionality to validate links and traces, and it can delete them if they are inconsistent. Traces can also be split if the chain of links is broken. Figure 8.3 illustrates a situation when model elements *M1* to *M7* are connected with traceability links *Link1* to *Link6*, which represent a chain of links. Therefore, these transitive relation is represented by trace *Trace1*. If for example *Link3* is deleted, the trace is split into *Trace1* and *Trace2*.

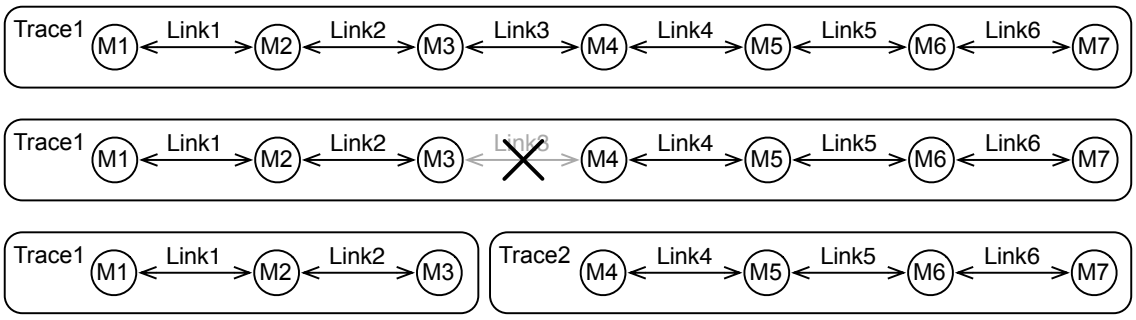


Figure 8.3: Split of a trace because of a broken chain of links

Rule Engine The `RuleEngine` component is the interpreter for the traceability rules described in Section 7.5. It loads the rules from a catalog and processes them with its four subcomponents.

Rule Validator The `RuleValidator` component checks the rules for their syntactical correctness according to the defined structure. The validation of the rules runs in four steps. First, all element definitions are checked. Second, the validator checks the `BaseConditions` with the comparison operators (*equals*, *contains*, *isParent*, *similarTo*, *lesserThan*, *greaterThan*, *notNull*), and third the `LogicConditions` (*AND*, *OR*, *NOT*, *XOR*) are checked. Finally, the fourth step is to validate the action part. If any problems are found, the `RuleValidator` produces warnings and hints in the log for their correction.

Element Processor The `ElementProcessor` component queries the defined model elements from the repository using the `AccessLayer` component. Then, it maps the elements to their alias names. As a result, for each element defined in the rule a list of queried elements is created. These queried elements are candidates to be linked if the conditions are met after processing the rule.

Rule Processor The `RuleProcessor` component evaluates the conditions defined in the query part of a rule. It gets a list of query elements from the component `ElementProcessor` and removes all elements that do not match the conditions. To evaluate the conditions, the logical operators and the query operators have to be executed. The resulting list of elements that match the conditions is given to the `ResultProcessor` component.

For the query operator *similarTo* an analysis of the similarity of two attribute values has to be performed. For this purpose the n-gram algorithm [CT94] as an information retrieval technique is applied. This is done in two steps:

1. For each string representation of the two attribute values create the n-grams.

2. Compare the strings using the n-grams.

An n-gram here is a slice of a longer string spanning n contiguous characters. Furthermore, stopping symbols (§) are added to the beginning and the end of a string. Accordingly, for example, the set of bi-grams for the word *"term"* would

be $\{\S t, te, er, rm, m\S\}$. The comparison step is performed with the Dice coefficient. Equation 8.1 shows the Dice coefficient for two terms a and b with $T(x)$ as the set of n-grams of the term x.

$$d(a,b) = \frac{2\,|T(a) \cap T(b)|}{|T(a)| + |T(b)|} \tag{8.1}$$

The resulting value is between 0 and 1. It can be set as a threshold for the *similarTo* operator. Thus, together with the n there are two parameters for rule processing that can influence the precision and recall of the established traceability links. Next to the n-gram algorithm further information retrieval techniques could be included to increase precision and recall, for example, stop-word elimination and word-stemming for pre-processing the terms, or other similarity measures.

Result Processor The `ResultProcessor` component is responsible for the completion of the rule processing. Therefore, the defined action of the third part of the rule is executed. Currently, the only supported action is to create a traceability link for elements that match the conditions as evaluated by the `RuleProcessor`. The links are created by calling the `LinkManager` component with the source and target elements, the link type defined in the rule, and additional information, such as the identifier of the rule.

EMFTrace GUI The `UserInterface` component of EMFTrace provides easy access to the import and export functionality for model integration as well as to the traceability analysis. For a seamless integration of EMFTrace and EMFStore, new menus and dialogs were embedded in the available EMFStore client using its plug-in extension points. Figure 8.4 shows a screenshot of the user interface with a context menu that acts as entry point for the EMFTrace functionality.

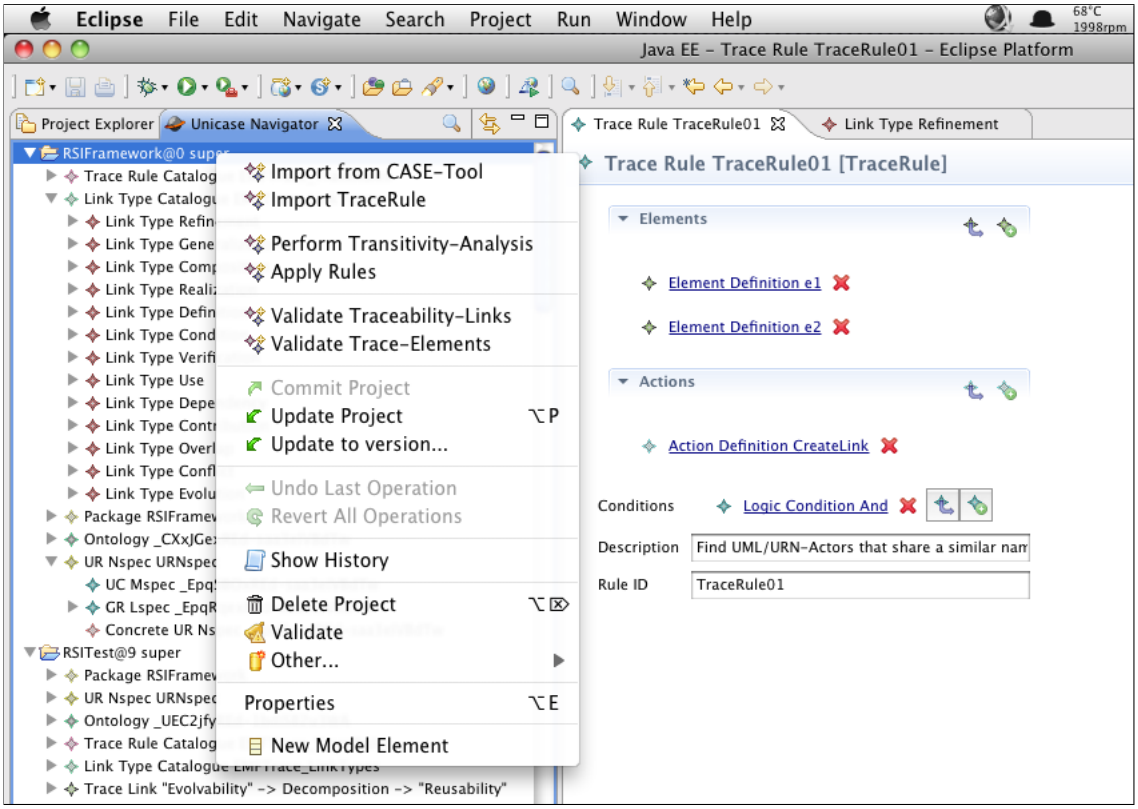


Figure 8.4: A screenshot of EMFTrace showing the context menu that allows to trigger its supported operations

8.2.3 EMFfit

EMFfit is a customized tool, which was developed for this thesis to support the Global Analysis activities [Wag10]. It is an important piece of the traceability puzzle because it enables improved traceability between requirements and design artifacts. EMFfit is realized as an EMFStore client, which communicates with the EMFStore server just like EMFTrace. Consequently, EMFfit needs EMFStore as a basic platform for project management and cannot run as a stand-alone tool. EMFfit provides a metamodel for influence factors and issue cards, which incorporates the hypertext metamodel from Section 7.3.2. EMFfit comprises the plug-ins `EMFfit` and `EMFfit.edit` for the metamodel, further `EMFfit_GUI` for the user interface. The GUI contains specific editors for factor tables and issue cards, as well as for the linkage to other artifacts according to the hypertext concept.

Figure 8.5 depicts the metamodel of EMFfit. The highlighted model elements stem from the metamodel of the hypertext concept (cf. Figure 7.3). The classes

`FactorTable`, `FTEntry`, `FactorCategory`, and `Factor` together with the enumeration `CategoryType` on the right-hand side of the figure realize the factor tables. The classes `IssueCard`, `InfluencingFactor`, `Strategy`, and `RelatedIssue` on the left-hand side, as well as `Item` realize the issue cards. The classes `FTICBase` and `ModelElement` are related to the integration into the EMFStore repository.

A speciality of this metamodel is the replacement of string attributes that contain informal textual descriptions with hypertext elements to enable the linking of terms. Examples are the attributes `description`, `flexibility`, `changeability`, and `influence` of a `Factor`. Novelties in comparison to the original Global Analysis are the enhancements mentioned in Section 6.2.1.4. For this reason, there are separate attributes for the flexibility and changeability of a factor as well as the additional priority attribute. Moreover, `Factor`s and `FactorCategorie`s are allowed to have `Factor`s or `FactorCategorie`s as child elements for a more flexible structuring of the tables. This is realized with the `children` relationship of `FTEntry`. The issue card is particularly interesting for traceability link establishment because it is related to other elements, which can be influence factors, factor categories, other issue cards, or strategies.

Another special feature of EMFfit next to the hypertext linkage is that it can adopt goals of URN models as factor categories of the product factor table. To realize this feature, all goals and softgoals that are modeled in a URN model in the same project as the factor tables are retrieved from EMFStore. Then, according to these goals, factor categories are created in the table for functional or quality product factors. Consequently, a special traceability rule, creates traceability links between the URN goals and the factor categories. A complementary feature, which is provided for filling a factor table, is to load default categories from a file.

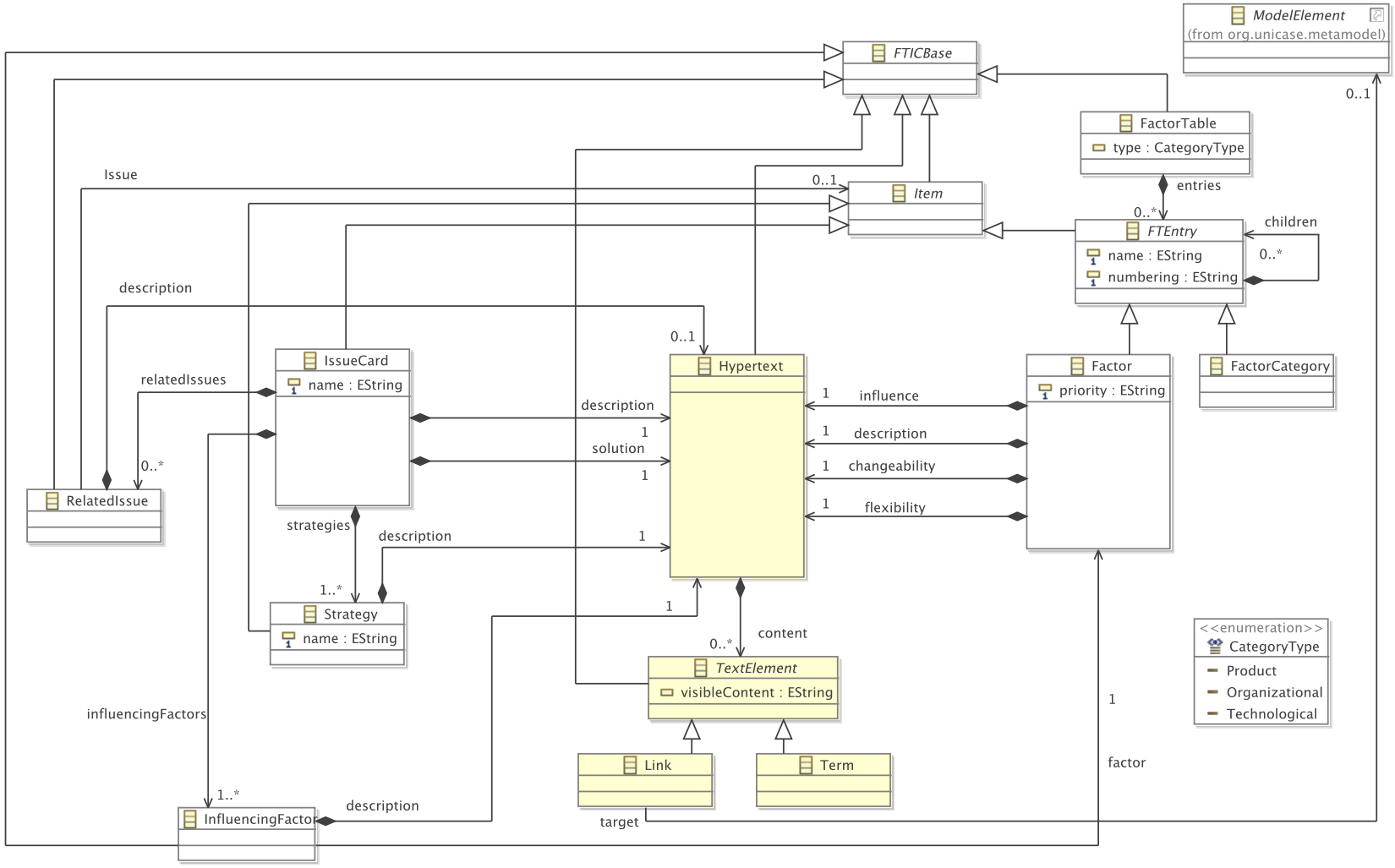


Figure 8.5: The metamodel of EMFfit for factor tables and issue cards as an Ecore model

8.3 Model Integration

As already mentioned above, the EMFStore repository can store and version EMF-based models. EMF provides the advantage of Java code generation. In this way code for model manipulation can directly be generated from metamodels. In combination with EMFStore this code is used for managing the models in the repository and for simple editing capabilities. Consequently, all types of artifacts from the goal-oriented design method had to be represented as an Ecore metamodel for an integration into EMFStore. Besides, the traceability link metamodel (cf. Figure 7.2) as well as a rule metamodel were created with EMF.

Integration of custom metamodels Custom metamodels to be integrated into the repository for the approach of this thesis are those for traceability links, EMFfit, and for traceability rules. An extra metamodel for traceability rules was necessary because in contrast to other approaches (e.g., [JZ09]) the rule-based traceability approach of this thesis operates directly on the conceptual level of models inside the repository and not on models materialized as XML files. Hence, the own XML representation of rules as introduced in Section 7.5 had to be represented as an Ecore model for the inclusion of the rules in the repository.

Figure 8.6 shows the metamodel designed as an Ecore model for the traceability rules. For rules, just as for the link types, a catalog class was modeled to enable grouping of rules. To enable integration into EMFStore all classes got a base class `RuleBase`. `RuleBase` in turn is a subclass of the root class `ModelElement` from EMFStore's metamodel. So every model element is easily enabled to work with EMFStore. This was modeled in the same way for EMFfit (cf. Figure 8.5) and for the traceability link metamodel.

The rule metamodel was used to generate an EMF-based editor for the traceability rules, which can be run inside EMFTrace as shown on the right-hand side of Figure 8.4. However, the rules can also be created with an external XML editor and imported afterwards. The external XML representation of the rules and the internal XML format of EMFStore slightly differ. This is due to the root class `ModelElement`, which has additional attributes. Therefore, an XSL transformation is applied for the import of the rules in the same way as described for the standard modeling languages in the next paragraph.

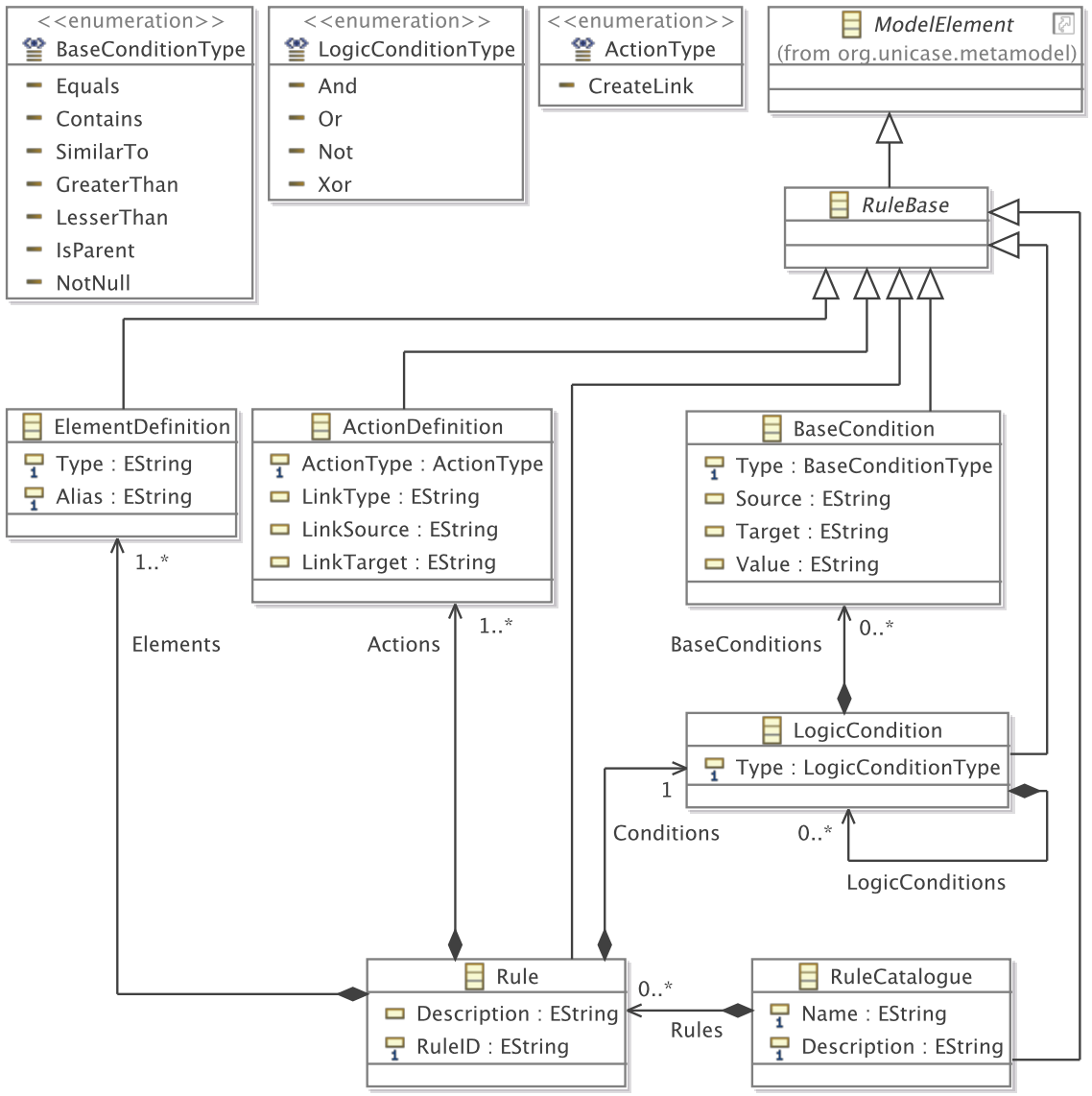


Figure 8.6: The traceability rule metamodel as an Ecore model

Integration of metamodels from standard modeling languages For the standard modeling languages URN, UML, and OWL, Ecore models had to be created. EMF allows to generate Ecore models from an XML Schema Definition (XSD). Hence, for example, for URN an Ecore model could be generated because a standardized XSD file is available. For UML an Ecore model implementation could be used from the UML2Tools project.

Once the Ecore models were available, they had to be adapted to the metamodel of EMFStore. Thus, the metamodels of URN, UML, and OWL were enhanced by the root class `ModelElement` as explained in the last paragraph. Furthermore, the

identifiers of the model elements of the used modeling languages had to be considered separately from the internal identifiers of EMFStore because their format is different. For example, in OWL an Internationalized Resource Identifier (IRI) is used, however, in EMFStore a string with a Universally Unique Identifier (UUID) is generated with EMF. After this adaptation, the Java model code and edit code could be generated with EMF and the metamodel could be used in EMFStore.

Since the adaptation to EMFStore's metamodel introduces new attributes to the metamodels, such as the internal identifier, a creator, and a timestamp, an adaptation of the metamodel instances is required as well. To add these attributes during the import of the models from the CASE tools, this adaptation was performed using Extensible Stylesheet Language Transformations (XSLT). Consequently, the import of models from the other CASE tools is done in the following steps:

1. The models are stored or exported as an XML file with the CASE tool.
2. XSLT templates are applied on the models.
3. The adapted models (XML files) are loaded with EMFStore.

For the export these steps are performed in reverse order with opposite operations and templates. The second and third step of the import and export procedure are encapsulated in components of EMFTrace.

Appropriate XSLT template were developed for URN, UML, and OWL for the CASE tools jUCMNav, UML2Tools, Visual Paradigm, and Protégé. Since especially UML tools use different formats for their XML representation, adapted XSLT templates are required for each tool. Moreover, all three steps of the import and export procedure can be automated with appropriate adapters to the CASE tools, as for example possible with the plug-in mechanism of Visual Paradigm.

8.4 Usage Scenario

A typical usage scenario for EMFTrace would be the following.

1. The developer starts according to the goal-oriented design method to model a goal graph with the CASE tool jUCMNav and saves the model as a URN file.
2. To import this model into EMFStore, the developer starts the EMFStore server and the EMFTrace client. Then he creates a project and uses the *"Import from*

CASE-Tool" feature as shown in the context menu in Figure 8.4. Afterwards the model is stored in the repository.

3. To proceed with the Global Analysis activities, the developer starts the EMFfit client and creates the factor tables and issue cards in the same project. By using EMFfit's mapping feature for URN goals and product factor categories, the establishment of the product factor table is facilitated. By executing the feature *"Get Product Factors from URN"* as illustrated in Figure 8.7 the imported URN model is searched for goals that can be adopted as factor categories.

4. In parallel the developer can create an ontology with Protégé or reuse an existing one, save it as an OWL/XML file, and import it with EMFTrace the same way as he did for the URN model (cf. Figure 7.5 for a visual representation of an OWL ontology in Protégé). Afterwards, the ontology is stored in the repository as well.

5. In EMFfit the developer can use the hypertext linkage feature to link model elements with terms of the ontology or other model elements, which are present in the EMFStore repository (cf. the example in Figure 7.4).

6. For architectural design, the developer proceeds to model UML diagrams, for example, with Visual Paradigm (cf. Figure 6.9 for a UML structure diagram of the case study). He uses the XML export feature to save a UML file, which he again imports into the repository with EMFTrace.

7. To run a dependency analysis and to create traceability links the developer performs several steps.

   (a) He loads the link type catalog, which is available in EMFStore's internal XML format into the project.

   (b) Then he imports an appropriate traceability rule catalog with EMFTrace's *"Import Trace-Rule"* feature.

   (c) The developer applies certain traceability rules with the help of a wizard that is available via the *"Apply Rules"* feature from the projects context menu. Figure 8.8 shows the wizard. This wizard allows to select rule catalogs from the project, traceability rules to be applied, and models that should be analyzed. Moreover, the parameters for the n-gram algorithm

can be specified. As a result of the analysis the traceability links are established.

(d) The developer can also perform a transitivity analysis on the traceability links with the feature *"Perform Transitivity-Analysis"*. Afterwards, traces for chains of traceability links are created.

(e) If elements of the design models or traceability links are changed or deleted, the developer can also run the features *"Validate Traceability-Links"* and *"Validate Trace-Elements"* to maintain the links and traces.

(f) Since all models are versioned in the repository, they can also be exported with EMFTrace's export feature for CASE tools as indicated in Figure 8.9.

This scenario shows the capabilities of EMFTrace and how they are intended to be used. Of course the mentioned steps do not necessarily have to be performed in this sequence. Furthermore, step 7c, for example, could be performed automatically after each import of a model from an external CASE tool. Also important to know is that the hypertext links created in EMFfit can be recorded as "real" traceability links (`TraceLinks`) with specific rules (cf. Section 7.3.2).

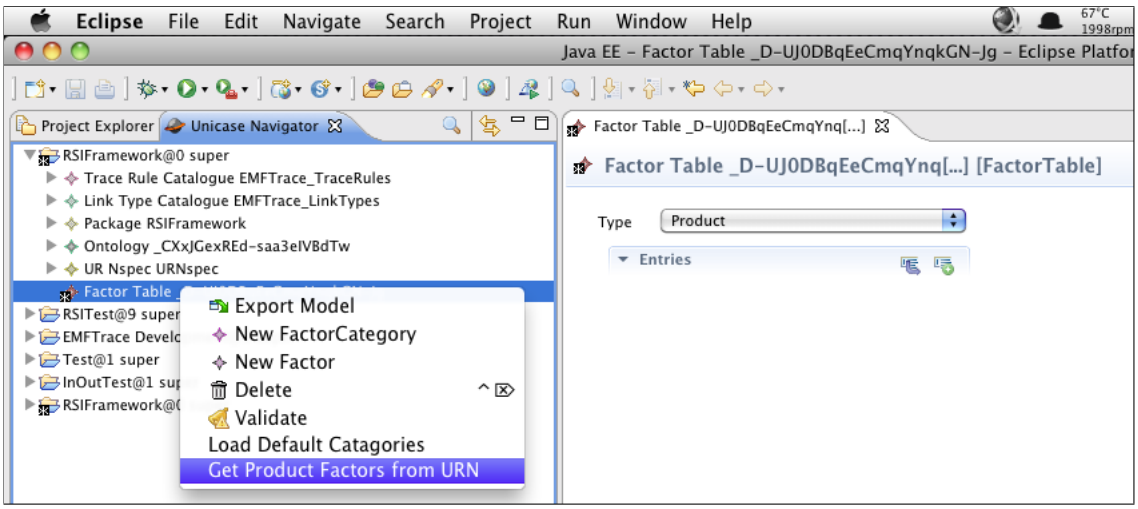


Figure 8.7: EMFfit with the feature for mapping URN goals to factor categories

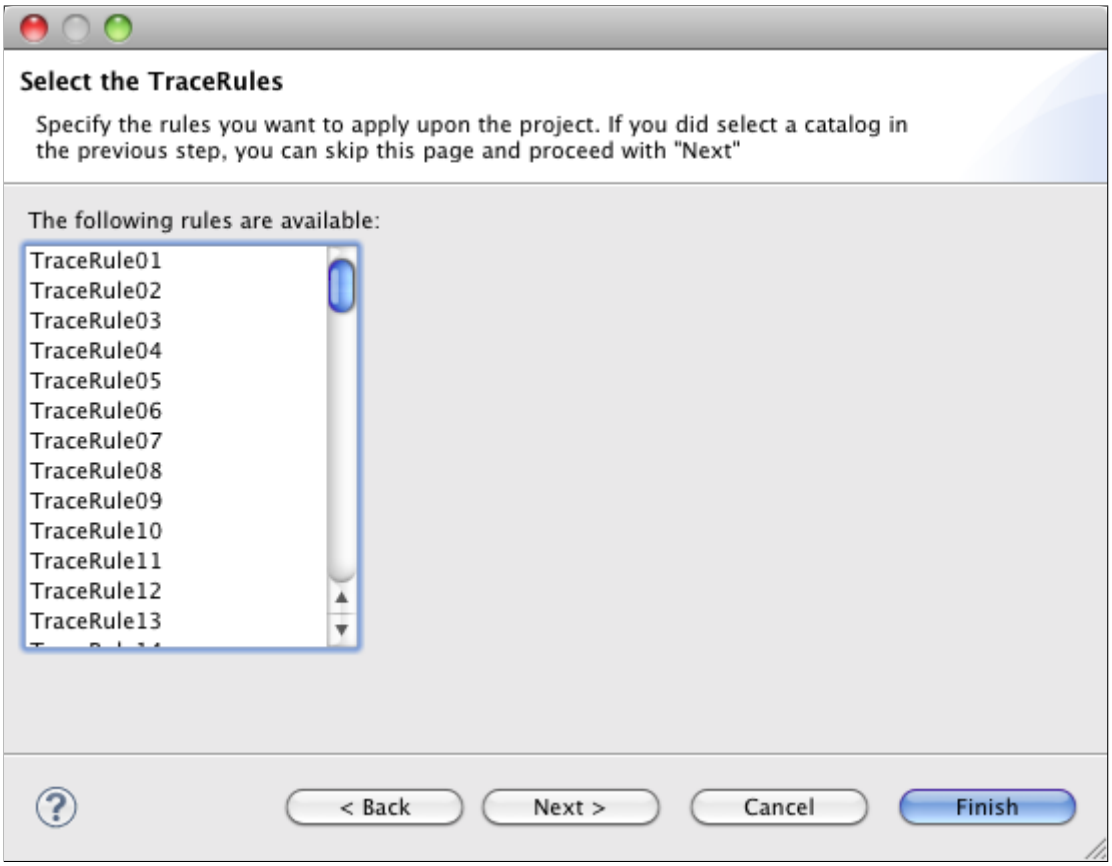


Figure 8.8: EMFTrace's rule wizard

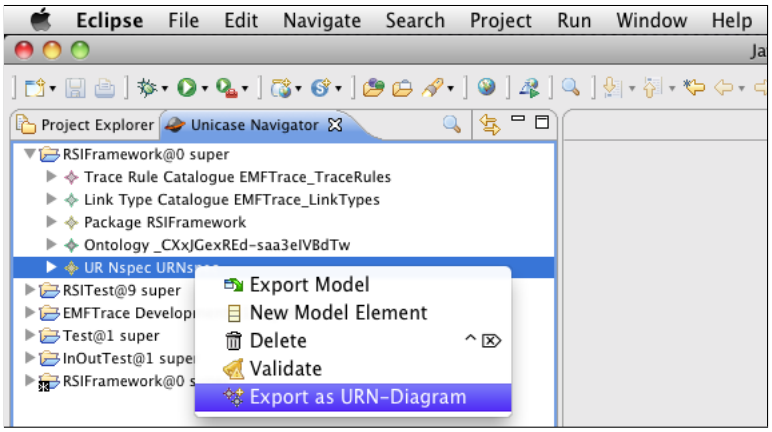


Figure 8.9: The export feature of EMFTrace

Chapter 9

Evaluation of the Approach

This chapter presents an evaluation of the approach of this thesis. It compares the results from Chapter 5, 6 and 7 against the refined goals of this thesis from Chapter 4.

Section 9.1 explains how the evaluation is performed. Then, Section 9.2 presents the evaluation of the Goal Solution Scheme and the goal-oriented architectural design method. In Section 9.3 the traceability concept is evaluated. For this reason the precision and recall of the link establishment is determined. Finally, in Section 9.4 the limitations of the approach are discussed.

9.1 Way of Evaluation

In Chapter 4 five refined goals **RG1** to **RG5** for the proposed approach were determined. For the evaluation of the approach regarding the goals, a case study was performed. This case study constitutes the basis for the discussion regarding the goal fulfillment.

The goals **RG1** to **RG3** are evaluated with an argumentative discussion based on the case study because another kind of evaluation for the GOAD method is hardly possible. For example, it is very unlikely to design a software system twice using different methods and to compare the results.

Concerning goal **RG4** the quantitative rating of solution instruments regarding their influence on evolvability has to be evaluated. For this purpose, a ranking of evaluated patterns is determined by using the values from Chapter 5. Then this ranking is compared to the patterns that were actually selected in the case study.

For the traceability concept, which fulfills goal **RG5**, a quantitative evaluation of the precision and recall of the link establishment is performed based on the models from the case study. This is done by comparing manually established links with those recorded by EMFTrace.

Characteristics of the Case Study The case study that is used for the evaluation of the approach of this thesis is the same as already used for illustration purposes in Chapter 6. The software system of the case study is a software platform for mobile interaction-robots. This software system was developed over several years in a joint venture of the department of Neuroinformatics and Cognitive Robotics of the Ilmenau University of Technology and a local company. The system was chosen for the case study, since it suits the scope of the approach of this thesis due to several reasons:

- Because the system has been growing over several years by many different developers including professional software developers, research staff, and students, it suffers from typical problems of architectural erosion. Such problems are lacking program comprehension, inconsistency, or hugh maintenance effort, which are risks for the research strategy of the department and for the business success of the company.

- The software system is complex by nature because it has to communicate with several different hardware components and has to run different domain-specific applications, such as navigation, person tracking, or speech recognition. This results in different quality goals to be addressed by the software architecture.

- Since the robot system is a research platform it is continuously evolving. Demands for changes frequently arise, for example, because of new research results, or changing technology. This makes evolvability and its subcharacteristics prominent goals of the software system.

- The system has an adequate size and is not a toy example. For the case study even the scope had to be limited. Out of the complete system, which is subject of a reengineering, the communication framework was selected. This part perfectly suits the needs of this thesis because it was decided to re-develop the communication framework in a rather forward engineering perspective.

9.2 Evaluation of the Goal Solution Scheme and the Goal-Oriented Architectural Design Method

This section discusses the Goal Solution Scheme and the goal-oriented architectural design method as results of this thesis regarding the goals **RG1** to **RG4**.

RG1 ***Combine existing approaches for a goal-oriented design regarding evolvability*** The Goal Solution Scheme from Chapter 5 and the goal-oriented architectural design method from Chapter 6 represent a comprehensive design approach for the treatment of quality goals and in particular evolvability. In the approach the modeling of the quality goals is combined with the activities of Global Analysis to provide an advanced architectural analysis phase. Further, the ADD method for synthesis is enhanced to deal with the additional input from the previous activities and to reflect the Goal Solution Scheme. This is proceeded with activities of the Quasar method for a more detailed design of components and interfaces. The design method can be completed with evaluation methods such as ATAM. Since the approach uses already established methods and concepts as far as possible, these parts need not to be evaluated on their own but rather their interplay.

The integration of the goal modeling with architectural analysis proved well for the case study to bridge from quality requirements to architectural design. The modeling of the quality goals as well as their refinement and prioritization according to the first and second layer of the GSS as described in Section 6.2.1 helped to raise the developers' awareness of the important quality aspects of the software system. The goal model was valuable input for the Global Analysis activities and helped to easily identify the important product factors.

Furthermore, the influence factors and issue cards together with the explicit consideration of solution principles and instruments as well as the discussion of scenarios helped with the architectural synthesis and evaluation. The developers were thankful for the additional input for the architectural design gained through applying the approach of this thesis. This underlines the feasibility of the approach. Moreover, the explicit documentation with the different models will help the developers with the future evolution of the software system.

Another experience to note is the initial reluctance of the developers against the additional effort for the explicit modeling, which was demanded according to

the approach of this thesis. However, over time the explicit documentation was appreciated, because now the developers, for example, can access their previous design decisions. Unfortunately, the integration of the Quasar method for detailed design could not be evaluated in the case study, since the developers had a preference to rely on their intuitive and experiential approach for design and implementation. However, the benefits of using Quasar were already discussed with a case study in earlier works [BR09, Bod08].

RG2 ***Provide a consolidated view on evolvability and a strategy for its realization*** Based on the evaluation of state-of-the-art works on evolvability, in this thesis an evolvability model as an instance of a Goal Solution Scheme was developed (cf. Section 5.3.1). The refinement of evolvability into subgoals and the mapping to solution principles and concepts was started with hypotheses and revised in a separate case study. As a result the evolvability model represents a consolidated view on this quality goal, which can be tailored for project-specific needs.

Furthermore, the goal-oriented architectural design method supports the realization of evolvability by several aspects: It utilizes the Goal Solution Scheme for selecting solution instruments during architectural synthesis. It supports explicit models and dependencies, e.g., between the influence factors and the quality goals of the case study. Moreover, the design method supports traceability and provides an iterative concept that not only supports top-down development but also bottom-up experimentation.

For the robot software system the evolvability model represented a valuable input. By knowing the subgoals of evolvability a suitable goal model, which also considers further top-level quality goals, could be developed for the case study, and the goals could be prioritized with the developers. The explicit consideration of evolvability as a quality goal during design should lead to a better evolvability of the newly developed software system in comparison to the old one. At least the developers now have better consciousness and confidence regarding the crucial quality aspects for the future evolution of the system.

RG3 ***Establish a concept for a systematic treatment of quality goals during design*** The Goal Solution Scheme concept was developed in this thesis for a systematic treatment of quality goals, which bridges from requirements analysis to

architectural design. Akin to the goal-oriented requirements engineering approaches it supports the modeling of goals and subgoals as well as the mapping to solution instruments as operationalizations.

However, the GSS goes beyond this. It also includes architectural solution principles, which are important for architectural design. It even considers technical constraints through the integration of the scheme in a design method that includes the activities of Global Analysis. Moreover, it supports conflict resolution by explicitly expressing the interdependencies. Further means that are targeting at conflict resolution, such as refinement, are the ones discussed in Section 6.2.2.4.

The beneficial treatment of quality goals in the case study was already mentioned with the discussion for **RG1**. The feasibility regarding the selection of solution instruments is discussed with **RG4**.

RG4 ***Provide a catalog of solution instruments with a quantitative rating regarding the impact on evolvability*** In connection with the development of the evolvability model also established solution instruments in form of architectural styles and patterns were evaluated. The result is a quantitative rating of the patterns regarding their influence on evolvability, which represents an extensible catalog of solution instruments for evolvability.

To evaluate these impact values, they are taken as input for a ranking of solution instruments using the prioritized quality goals from the case study. Table 9.1 shows the starting point for the ranking procedure. The solution instruments are already restricted to a set of five typical domain and technology-independent styles, which were also discussed regarding a selection in [GEM10]. The table shows the priority values for the quality goals from the case study (cf. Table 6.2) together with impact values that express the influence of the styles on the goals. The impact values for testability, changeability, and reusability stem from the rating determined in this thesis (cf. Table 5.7). The additional values for efficiency and reliability were taken from [GEM10] and represent an assessment based on several references. Three quality goals of the case study are not considered in the ranking: (*i*) security because it did not get a priority at all, (*ii*) usability because it is not that important for the communication framework of the case study and the styles would not considerably influence it anyway, as well as (*iii*) distributability because no impact values of the styles were available for it.

Table 9.1: Impact values and priorities for the ranking of the architectural styles

Quality Goal \ **Architect. Style**	Client-Server	Layers/Tiers	Blackboard	Pipes and Filters	Impl. Invocation	Goal Prioritization
Testability	1.00	1.29	−1.07	0.86	−0.79	0.0389
Changeability	1.39	1.56	0.02	1.56	1.11	0.2333
Reusability	1.50	1.75	0.25	1.50	1.38	0.1056
Efficiency	0	0	0	−1	1	0.2667
Reliability	0	0	1	0	0	0.1333
Distributability						0.0667
Security						0.0000
Usability						0.1556

For the ranking of the styles, the impact values in each row of the table are multiplied with the priority values of the goal in the rightmost column. Then, the ranking values for the different quality goals are summed up. Table 9.2 shows the results. In the bottom the ranking is shown first for the highly-prioritized goals changeability, efficiency, and reliability as well as second for all quality goals. The resulting ranked list of styles in the first case is: 1.) *implicit invocation*, 2.) *layers/tiers*, and 3.) *client-server*, followed by 4.) *blackboard*, and 5.) *pipes and filters*. In the second case it is: 1.) *implicit invocation*, 2.) *layers/tiers*, and 3.) *client-server*, followed by 4.) *pipes and filters*, and 5.) *blackboard*.

For comparison, in the case study the style *implicit invocation* was chosen based on expert opinion. A *layered* style would not have been appropriate because there is no hierarchical structure of communication in the robot software system. The *client-server* pattern was used for remote communication. The *blackboard* style of the old system was dismissed. In [GEM10] the recommended style for a mobile robot system by expert opinion was also *implicit invocation* (or interacting process as they called it), and the dismissed style was *layers/tiers*. In this regard the above calculated ranking matched the case study's conditions quite well. Only *layers/tiers* seems to be ranked too high.

Consequently, the impact values of the catalog that are provided in this thesis

proved quite well. However, they could be evaluated only partly. On the one hand not all subgoals of evolvability could be considered. On the other hand the impact values had to be supplemented by values for efficiency and reliability from other sources. Beyond, the impact values are subjective by nature and should not be taken as the ultimate truth. Nevertheless, the procedure of calculating a ranking of solution instruments can be a helpful hint for the software architect in complex situations regardless of the specific values provided in this thesis.

Table 9.2: The ranking values of the architectural styles for the case study example

Quality Goal ╲ Architect. Style	Client-Server	Layers/Tiers	Blackboard	Pipes and Filters	Impl. Invocation
Testability	0.04	0.05	−0.04	0.03	−0.03
Changeability	0.32	0.36	0.00	0.36	0.26
Reusability	0.16	0.18	0.03	0.16	0.15
Efficiency	0.00	0.00	0.00	−0.27	0.27
Reliability	0.00	0.00	0.13	0.00	0.00
Changeability+Efficiency+Reliability	0.32	0.36	0.14	0.10	0.53
All	0.52	0.60	0.12	0.29	0.64

9.3 Evaluation of the Traceability Concept

Concerning the goal **RG5** of this thesis there were several aspects that had to be covered by the traceability concept:

I. (semi-) automatic traceability link

   a) identification,
   b) recording, and
   c) maintenance

   of

   d) intra and
   e) inter model dependencies, and

II. a defined semantics for links based on specific link types.

The traceability concept from Chapter 7 defines a metamodel for traceability links. Following this metamodel, traceability links are enriched with link types and other attributes for a defined semantics according to II. For a consolidated view on possible traceability link types, different clusters were defined based on other works. Furthermore, the traceability concept considers all models and dependencies according to d) and e) by integrating all artifacts of the architectural design method with their model elements in a joint model repository.

For the link establishment, traceability rules were defined. These rules have to be defined manually and then are interpreted automatically by the developed prototype tool EMFTrace (cf. Chapter 8). In this way model dependencies are identified and recorded as traceability links according to a) and b). Maintenance of traceability links and traces according to c) is achieved by some validation capabilities of EMFTrace. If links get inconsistent or chains of links are broken because of model changes, the links and traces will be updated. However, for a comprehensive maintenance of links according to change activities that are performed on the models, additional rules have to be defined and the rule engine of EMFTrace has to be enhanced.

Research Question and Measures For the evaluation of the traceability concept an important research question was:

- *How is the quality of the link establishment in terms of completeness and correctness?*

The quality can be evaluated by determining the values for the common metrics precision and recall. For the calculation of the metrics three kinds of traceability links have to be distinguished:

c the number of links established correctly by the approach,

i the number of incorrectly established links, and

m the number of correct but missing links.

Precision refers to the correctness of the established links. It is the ratio of the correct established links to all links established with the approach. Precision is calculated with:

$$P = \frac{c}{c+i}$$

Recall refers to the completeness of the established links. It is the ratio of the correct established links to all correct links. Accordingly, the recall is calculated with:

$$R = \frac{c}{c + m}$$

Precision and recall can be combined to the so-called F-score. This measure is a weighted average of precision and recall. It can be used for an easier comparison of the precision and recall values of different approaches. The F_1-score, which weights precision and recall the same is calculated with:

$$F_1 = 2\,\frac{P \cdot R}{P + R}$$

Subject of Measurement To determine the evaluation factors precision and recall, data from two development projects were considered. One project was the development of EMFTrace itself. The other one was the robot software system from the case study. The input data for the evaluation from the EMFTrace project were UML models, which were created with the UML2Tools and imported into EMFTrace. The data from the case study system comprise a URN goal model created with jUCMNav, factor tables and issue cards created with EMFfit, a UML design model created with VisualParadigm, as well as an OWL ontology created with Protégé. Table 9.3 shows the number of model elements for each project.

Table 9.3: Amount of model elements from the evaluation projects

Model	EMFTrace	Robot Case Study
URN	–	178
FTIC	–	569
UML	869	217
OWL	–	752
Total	869	1716

Way of Measurement The determination of the evaluation factors was accomplished in three steps. First, a manual link establishment was performed to determine the number of correct links, which is needed for the calculation. The manual link establishment for the EMFTrace project was performed by a developer of the tool (not the author) (see [Leh10]). For the case study it was accomplished by the

author and partly by a developer of EMFTrace. The resulting traceability links of the manual step were considered to be the correct set of links. As the second step, the models were automatically analyzed for traceability links by EMFTrace. This was performed two times with different parameters for the n-gram matching algorithm. Once the parameters were n=3 and the similarity factor 75%. The second time the similarity factor was changed to 90%. Finally, the values for precision and recall were calculated based on the results from both the manual and the automated step.

Measurement Results and Discussion For the EMFTrace project the automated link establishment of EMFTrace even found some correct links that were not found manually by the developer. Consequently, these links were added to the set of correct links of the developer. Table 9.4 shows all results of the measurement. The resulting precision achieved with the traceability approach of this thesis is 86.4% on average. The recall factor is 84.6% on average.

Table 9.4: Results of the quality measurement

	Number of Links										
	Dev.	Tool $(c+i)$		D. $\cap$ T. (c)		**Precision** (P) [%]			**Recall** (R) [%]		
Project	$(c+m)$	75%	90%	75%	90%	75%	90%	avg.	75%	90%	avg.
EMFTrace	46	64	40	43	34	67.2	85.0	**86.4**	93.5	73.9	**84.6**
Robot Case Study	336	321	273	309	265	96.3	97.1		92.0	78.9	

Table 9.5: Comparison of precision and recall of different traceability approaches

Approach	**Technique**	**Precision**	**Recall**	**F_1-score**
[ACC+02]	Information Retrieval	50%	48%	0.49
[MM03]	Information Retrieval	27.4%	84.2%	0.41
[LFOT07]	Information Retrieval	46%	70%	0.56
[JZ09]	Rule-based	85.3%	83.3%	0.84
EMFTrace	Rule-based & Information Retrieval	86.4%	84.6%	0.85

Table 9.5 shows average values for precision and recall as well as the resulting F_1-score of some prominent traceability approaches. In comparison to pure information retrieval based approaches the rule-based concept provides at least a better

precision and also a good recall. Compared to other traceability approaches the achieved results with the approach of this thesis are quite competitive. It achieves an F_1-score of 0.85. The quality of the links for the models of this evaluation even is slightly better than that of the approach of Jirapanthong and Zisman [JZ09]. However, it must be admitted, that due to the limited number of studied systems the values for EMFTrace might be biased. Moreover, it is hard to compare the values of the different approaches since they deal with different kinds of artifacts.

Nevertheless, the precision and recall depends on the defined rules and not only on the rule-based concept itself. The correctness and completeness of the link establishment can be improved further by a revision of the existing rules and by a definition of new ones.

As a further advantage, the rule-based approach provides semantic information with the links in form of link types and further attributes. During the application of the approach it turned out that the available set of link types was appropriate for the definition of the necessary rules. But the link types might be revised in the future as well.

The establishment of rules of course is manual effort in the first place. But once they are established, the rules can be used for automated link establishment. Although not measured precisely, the effort in terms of time for the link establishment can be reduced significantly. The manuel link establishment took approximately at least an hour, whereas the tool needs approximately ten minutes without performance optimization.

9.4 Limitations of the Approach of this Thesis

Of course the approach proposed in this thesis has some limitations, which shall not be concealed. First, the goal-oriented design method relies on a model-based development. If the artifacts are not expressed explicitly in well-defined models, it is hard to make dependencies in between explicit. Moreover, an extension of the approach for domains such as embedded systems design is still work to do. But the basic concept of the Goal Solution Scheme is general enough to cover further domains.

Moreover, there is hardly any pure forward engineering project. Most recent industrial projects need reengineering activities. For this reason, on the one hand

the goal-oriented design method has to be enriched with architectural reengineering means to choose from. On the other hand the activities have to be integrated in a reengineering process. However, there will always be a realization of new features, for which the proposed architectural analysis and structuring activities can be performed.

Further effort also has to be invested in the architectural evaluation regarding evolvability because quantitative values for the architectural properties are necessary to control the quality of the architecture. Therefore, specific measurable metrics should be established especially for evolvability, which then can also recommend certain reengineering means.

The rule-based traceability approach is independent of a forward or reengineering perspective. It can be applied as long as appropriate rules for the considered artifacts are available. However, it relies on an appropriate integration of models from external CASE tools in a repository. In the prototype tool this currently is implemented using EMF technology, which might be inappropriate with very large models due to performance reasons. However, to implement the rule-based concept, also another technology can be chosen.

The use of the ontology as an artifact with different terms spanning the whole development process turned out to be a good means for connecting different modeling languages. While the linkage between goal models of requirements engineering and the analysis artifacts of Global Analysis was easy, it was more diffcult between the analysis models and the synthesis models. In this regard the ontology and the hypertext approach are helpful, although the hypertext links need manual effort. Further research effort might be useful to be invested in this issue.

Although promising results for precision and recall were measured, their general validity is not assured. More case studies or controlled experiments might be necessary for a statistical relevance and an improved validity of the measures. For example not all rules did match in the case study example, since certain model elements involved in rules did not exist in the case study artifacts. Moreover, the rule definition still needs a lot of manual effort.

Another limitation of the traceability concept is that automated model transformations are not yet considered for link establishment. Besides, already existing external traceability information should be considered and integrated in the repository.

Chapter 10

Conclusions and Outlook

This chapter presents conclusions from this thesis. In Section 10.1 a summary of the contributions from the preceding chapters is provided. Section 10.2 gives an outlook on further research issues based on the results of this thesis.

10.1 Contributions

This thesis presents a comprehensive approach for goal-oriented architectural design and for traceability to enable evolvability of software systems during design. The approach is targeted especially on a more advanced treatment of quality goals, such as security, usability, or modifiability, during architectural design and on an enhanced support for the evolution of software systems. Wherever possible the approach takes the best of established methods and concepts and extends them. It particularly addresses the qualitative aspects of the software, because they are often neglected by existing approaches in comparison to functional aspects. Furthermore, by its iterative nature and by considering especially evolvability it enables to cope with frequent demands for changes, which is a major challenge in the current software engineering practice. Moreover, the approach bridges between requirements engineering and architectural design by providing an integrated approach with concepts from both research communities, which are still separated from each other. In this regard a major contribution of this thesis is the combination and integration of different approaches, concepts, and ideas for the treatment of quality goals, for architectural design with analysis and synthesis activities, as well as for the traceability of the whole approach.

The Goal Solution Scheme presented in Chapter 5 is the most significant novelty and contribution of this thesis. It is based on the principles of model-based design and combines concepts and ideas from goal-oriented requirements engineering with architectural design principles and activities. The scheme has several benefits: It supports the transition from the problem space of a design task to the solution space by an explicit modeling of dependencies and by a mapping from goals to solution principles and further to solution instruments. In this way the Goal Solution Scheme facilitates the identification of goal conflicts and their resolution by prioritization, refinement, as well as the mapping to the solution principles and instruments. It enables a quantitative evaluation of the solution instruments regarding their influence on quality goals and guides the architect with making decisions and selecting appropriate solutions during design. Moreover, the scheme facilitates the establishment of traceability links between goals and design artifacts. The scheme was developed and applied in different case studies with different quality goals and described in detail for evolvability. Other quality goals can be treated in the same way.

The evolvability model is an instance of the Goal Solution Scheme for evolvability (cf. Section 5.3) and follows the discussion of the term evolvability and its definition derived from state-of-the-art works (Chapter 3). It is a contribution insofar as it presents a consolidated view of evolvability and its subcharacteristics, and further maps evolvability to design principles and solution instruments. Thus evolvability becomes more tangible, and it is easier to achieve this goal during design. With this model or scheme the influence of fundamental design principles and properties of good design on evolvability and its subcharacteristics is determined. Moreover, a set of patterns as solution instruments is evaluated regarding their influence on evolvability. In this way it guides the selection of appropriate solutions and facilitates the architects decision-making.

The goal-oriented architectural design (GOAD) method presented in Chapter 6 overcomes the limitations of state-of-the-art methods and concepts, which have been evaluated in Chapter 2. As a contribution it is a combination of the best parts of existing works and integrates the Goal Solution Scheme for a systematic treatment of quality goals during architectural design. GOAD starts with goal modeling

adopted from requirements engineering and uses goal prioritization for focussing the design. For the architectural analysis phase, Global Analysis as the most suitable method is integrated and enhanced. For architectural structuring during the synthesis phase, activities adapted from the ADD method are described to show the utilization of the Goal Solution Scheme.

GOAD and the GSS provide an easier transition from the problem to the solution space (transition from layer II to III and IV of the GSS). They provide guidance for a selection procedure regarding design solutions. This procedure explicitly considers principles and constraints, and it supports decision-making from a stock of solution instruments, which are evaluated regarding evolvability. For the integration of selected solution instruments into the whole architecture the transformations of the QASAR method are applied. Moreover, for detailing the conceptual architecture, Quasar with its software-categories is applied for an improved separation of concerns and explicit dependencies. Furthermore, bottom-up vs. top-down structuring of the architecture is discussed in regard to evolvability support of the resulting software. Beyond, general means for conflict resolution are adopted from requirements engineering to architectural design. The feasibility of the goal-oriented design approach is shown with a case study project.

The traceability concept from Chapter 7 contributes to the cluttered field of traceability with its various different approaches from requirements engineering and model-based development. It provides a comprehensive approach spanning all design artifacts of the GOAD method and especially considers quality goals. The goal of the concept is to trace the design steps that lead to the design artifacts to enable impact analysis of future evolutionary changes. Therefore, all models are integrated into a repository and as a novelty an ontology is included to enable more explicit dependencies between the models.

Moreover, the traceability concept combines different ideas from existing traceability approaches. A traceability metamodel is defined as a basis for tool support. It is accompanied by a metamodel for the hypertext concept, which was developed to be used especially for the linkage of informal textual descriptions, as for example prevailing in artifacts of the Global Analysis. Furthermore, a consolidated set of traceability link types is presented, which is organized in clusters. These link types are used for providing a significant semantic meaning of traceability links.

Moreover, as a novelty a rule-based approach for the (semi-) automated identification and recording of traceability links is combined with information retrieval techniques. This on the one hand is limited to a justifiable effort for rule definition and on the other hand results in a high precision and recall. The rules are defined using XML Schema Definition (XSD), and a metamodel is created for their integration into the repository. During rule processing n-gram based string matching as an information retrieval technique is utilized for the calculation of similarities of the model elements' identifiers.

Tool support with EMFTrace is implemented as a contribution to show and support the applicability of the thesis' concepts. EMFTrace is developed as an extensible platform based on Eclipse technology (cf. Chapter 8). It realizes the traceability concept and connects existing CASE tools, such as jUCMNav, VisualParadigm, and Protégé, for the modeling of the design artifacts from the GOAD method. For the integration of all artifacts, EMFTrace uses the EMFStore model repository and provides EMF-based metamodels for each artifact. As a benefit in comparison to other approaches it relies on standard modeling languages, such as URN and UML, as far as possible, since they are quite stable and seldom subject of changes. Moreover, EMFTrace manages traceability links and rules also using EMF-based metamodels. It comprises a rule engine for rule processing, a link manager for traceability link establishment, and it provides capabilities to manage traces as chains of traceability links as well as validation checks regarding consistency to maintain the links. EMFTrace is accompanied by a custom tool called EMFfit. This tool has been implemented to support the Global Analysis activities because tool support was not existing before. EMFfit also realizes the hypertext concept for informal textual descriptions.

10.2 Future Work

Regarding the Goal Solution Scheme different aspects of improvement can be discussed for future work. The GSS already was discussed and applied for different quality goals, such as evolvability, security, or usability. Additionally, prediction approaches for quality properties of an architecture could be integrated, such as Palladio [BKR07] for performance prediction. Furthermore, other domains can be

analyzed if an adaptation of the scheme is necessary, for example, the design of embedded systems, distributed systems, highly reliable systems, or a design specifically aimed at the goal efficiency via multi-core usage. This work has already started [RPB11, Neu11]. Moreover, the stock of solution instruments on the lower layer of the GSS, the so-called architect's toolbox, should continuously be extended and enhanced.

For software evolution the mainly forward-looking engineering perspective of this thesis' design method should be examined regarding a combination with a reengineering approach. One work with a reengineering perspective that utilizes a metrics-based approach for evaluating and improving the evolvability of existing systems is the one of Brcina [Brc11]. The support of established methods for architectural evaluation especially regarding evolvability is questionable. Existing works on evaluation should be analyzed and improved if necessary. Besides, the applicability of the GSS with aspect-oriented approaches that target on cross-cutting concerns and with transformation-based, model-driven design approaches would be interesting.

The traceability concept, which was implemented in this thesis with the tool EMFTrace, could be integrated further with requirements traceability approaches. This would result in the integration of further artifacts into the repository. Moreover, rules for maintaining traceability links as proposed by the event-based approach of Mäder [Mäd09] should be integrated with the approach of this thesis, which requires an extension of the rule engine. Finally, further effort should be invested in the utilization of traceability links, for example, regarding change impact analysis, consistency checks, or checks for the coverage of requirements by design and implementation artifacts.

Appendix A

Case Study Artifacts

A.1 Factor Tables

This section lists the factor tables of the robot case study.

Table A.1: Organizational factors of the case study

	Organizational Factor	***Flexibility & Changeability***	***Impact on Architecture***
O1	**Management**		
O1.1	*Build vs. buy* There is a preference to use mature free and open source software or to build.	If justified, buying would be an option.	There is small impact on meeting the schedule.
O1.2	*Schedule vs. functionality* There is a preference for schedule over new features. Basic functionality should be revised first.	Old components could be reused with new components.	Support for adapters between new and old components has to be ensured.
O1.3	*Environment* Next to the university, the company ML is involved.	Decisions are made in cooperation, but each partner has its own preferences.	Both partners have to decide on the architecture in common.

Table A.1: Organizational factors of the case study (cont.)

	Organizational Factor	***Flexibility & Changeability***	***Impact on Architecture***
O1.4	*Business goals*		
	The flexibility and evolvability of the whole system shall be recovered for future demands. Robots and their software shall be used in real world scenarios as for example as a service robot in a do-it-yourself store.	The old system has to continue operating; there cannot be a down time.	A migration strategy with adapters between old and new components has to be established.
O2	**Staff skills, interests, strengths, weaknesses**		
O2.1	*Application domain*		
	Knowledge about necessary libraries and frameworks is available in the team.	Team members can be trained.	There is a small impact on usability of the design.
O2.2	*Software design & architectural design*		
	Basic design knowledge is present in the team.	Advanced architectural design knowledge can be trained by consulting experts from the university.	There is a moderate impact on design quality.
O2.3	*Specialized implementation techniques*		
	All team members have the necessary implementation skills.	New skills can be trained if necessary.	There is small impact on usability of the code and on the code reviews.
O2.4	*Specialized analysis techniques such as architectural assessments*		
	Basic analysis knowledge is present in the team.	Advanced knowledge can be trained by consulting experts available at the university.	There is a moderate impact on design quality.

Table A.1: Organizational factors of the case study (cont.)

	Organizational Factor	***Flexibility & Changeability***	***Impact on Architecture***
O3	**Process and development environment**		
O3.1	*Development platform*		
	The main programming language is C++, some parts are developed in AngelScript.	There is no flexibility in the programming language. For scripting maybe Python will be used.	The system has to be implementable in C++, compatible libraries can be used. Maybe for the GUI some parts can be developed with C#.
O3.2	*Development process and tools*		
	Used IDEs are for example Eclipse and Kate or partly Visual Studio Express for C#.	The choice for tools of the development environment is flexible within the constraints of supported languages and operating systems.	
O3.3	*Configuration management and production process and tools*		
	Configuration management has to deal with different OS versions and library versions on several computers. There are two productive environments at ML and at the university. CMake is used as the build tool. Automatic nightly builds are planned.	There might be a strategy to align the versions of the used libraries, but there will never be the same environment on all computers.	The architecture has to consider especially which versions of libraries have to be adapted for new features. A concept for structuring releases and current development forks, which maps to trunk, branches, and tags in the versioning system has to be established.
O3.4	*Testing process and tools*		
	Used tools are CTest and CDash. Automized unit tests are introduced, reviews follow.	Additional tests and reviews can be introduced if necessary.	There is an impact on test coverage.

Table A.1: Organizational factors of the case study (cont.)

	Organizational Factor	***Flexibility & Changeability***	***Impact on Architecture***
O3.5	*Release process and tools* Releases have to be well documented, reviewed and tested. Robots have to run through a test track.	A release can be shifted if not sufficiently documented, reviewed, or tested.	There is a moderate impact on the schedule.
O4	**Development schedule**		
O4.1	*Time-to-market* The rework of 80% of the features should be complete at the end of 2010 for use with completely new software components.	Some of the features should be revised immediately, others are flexible.	There is an impact on the prioritization of the features for design. Old system parts should be reusable with adapters.
O4.2	*Delivery of features* The features to revise are prioritized.	The features are negotiable.	There is a small impact on meeting the schedule.
O4.3	*Release schedule* There are regular releases at ML approximately three to four times a year.	The feature release cycle is flexible.	The reengineering should not disturb maintenance releases at ML.
O5	**Development budget**		
O5.1	*Head count* There are about 10 developers at the university, 3 at ML.	The number of developers active in reengineering in parallel varies around 3-5. Some students might be available.	There is a moderate impact on the schedule.
O5.2	*Cost of development tools* Development tools, as IDEs and analysis tools, are available for free.	There might be a small budget for an academic license fee.	Analyzability of the old system for dependencies depends slightly on an appropriate tool.

Table A.2: Technological factors of the case study

	Technological Factor	***Flexibility & Changeability***	***Impact on Architecture***
T1	**General-purpose hardware**		
T1.1	*Processor*		
	The processor is currently an Intel DualCore with 2GHz. It has to be a low voltage variant.	Increases in processor speed are frequent but a processor changes approximately every 1 to 2 years usually with a completely new robot.	A change of the processor should be transparent for the system but should increase speed. The design has to ensure little overhead.
T1.2	*Network*		
	Network hardware supporting the Wi-Fi standards 802.11b/g/n are used.	The hardware changes usually only with a completely new robot approximately every 1 to 2 years.	Only robots using 802.11n allow higher data rates for example to distribute image processing.
T1.3	*Memory*		
	Between 1 and 2 GByte RAM are used.	The hardware changes usually only with a completely new robot approximately every 1 to 2 years.	The memory won't be a limiting factor.
T2	**Domain-specific hardware**		
T2.1	*Laser*		
	For example a SICK S 300 is used.	The hardware changes usually only with a completely new robot approximately every 1 to 2 years.	Impacts the robot configuration because of specific driver, might be OS specific. The laser has a safeguard that has to be dealt with during navigation.

Table A.2: Technological factors of the case study (cont.)

	Technological Factor	***Flexibility & Changeability***	***Impact on Architecture***
T 2.2	*Camera* There are different cameras that are for example connected via Firewire, USB or GigaBit-Ethernet.	The hardware changes usually only with a completely new robot approximately every 1 to 2 years.	A change of the camera should be transparent to the system if an appropriate driver is available. A new camera might increase the system load due to increased image data.
T3	**Software technology**		
T 3.1	*Operating system* The OS on the robot is usually linux but might be windows as well, and x64 support is necessary for the developer environment. There might be additional computers with linux or windows cooperating with the robot.	Changes to the OS occur frequently due to new release of linux distributions.	Used libraries have to be compatible with the OS version. Differences between linux and windows in handling threads have to be considered in the design maybe using a wrapper.
T 3.2	*User interface* The user interface is built with the Qt library to be platform independent. Small parts are developed with C#.	The usual library version updates have to be expected.	There might be interdependencies between Qt, C# and other libraries to be considered in the design.
T 3.3	*Software components* Different libraries, which are available for free, are used, such as Boost, STL, Qt, or Eigen.	Changes to the libraries are frequent. The libraries may be updated if essential new features are included.	Version updates of libraries should be transparent for the system.

Table A.2: Technological factors of the case study (cont.)

	Technological Factor	***Flexibility & Changeability***	***Impact on Architecture***
T3.4	*Implementation language*		
	The implementation language is C++. Some UI parts may be developed with C#.	There is no flexibility. Maybe C++0x will be used.	The compilation of libraries has to be assured to work.
T4	**Architecture technology**		
T4.1	*Architecture styles*		
	The old architectural style is a blackboard architecture.	The blackboard concept is likely to be changed, because of known problems with the architecture.	The new design should retain the flexibility and overcome the known deficiencies not necessarily sticking to the blackboard concept.
T4.2	*Domain-specific or reference architectures*		
	ROS is a known open source framework for a robot operating system.	ROS is open-source and changes frequently but won't be used directly.	Some concepts of ROS can inspire the design of the new architecture of the system. Maybe adapters to parts of ROS will be developed.
T4.3	*Architecture description languages*		
	No specific ADL will be used but UML could serve as a modeling language.	For the mature UML standard no major revisions are expected. Knowledge can be trained.	The design can be expressed using UML diagrams.
T4.4	*Product-line technologies*		
	There is a need for different configurations for various robots but product-line development has no priority.	Knowledge about feature modeling for product-line variability can be trained by experts at the university.	The design could include variability. The robot-specific configuration is handled with configuration files.

Table A.2: Technological factors of the case study (cont.)

	Technological Factor	***Flexibility & Changeability***	***Impact on Architecture***
T5	**Standards**		
T5.1	*Operating system interface*		
	The operating system interface is Linux/UNIX by default. Windows might be used to.	The standard is stable, but there might be changes in accompanied tools in different Linux distributions and versions.	There is an impact components for hardware drivers.
T5.2	*Database*		
	A SQL database might be used in the future for storing large data structures and for cooperative changes of this data.	Database integration currently has no priority.	Running a database impacts the hardware requirements. An integration in the system should be easy with adapters to the serialization component.
T5.3	*Data formats*		
	Data should be serialized with XML. Binary formats are necessary for video, audio or images. The video format should be compatible to MPEG or AVI.	The XML standard is stable.	There is a large impact on the component for serialization. Especially the binary formats need to be portable.
T5.4	*Communication*		
	For the communication between distributed components TCP/IP or UDP are used.	The standards are stable.	There is moderate impact on the components for serialization and inter process communication.

Table A.2: Technological factors of the case study (cont.)

	Technological Factor	***Flexibility & Changeability***	***Impact on Architecture***
T 5.5	*Algorithms and techniques* There are domain specific algorithms and techniques for robotics and image processing.	Some parts of the field are quite mature, but new research results frequently arise.	The is a moderate influence on components involved in image processing, navigation, pose recognition, person tracking.
T 5.6	*Coding conventions* A generally accepted coding standard does not exist.	Project specific coding conventions are defined, which should remain stable.	Adherence to the coding conventions has a major impact on understandability and code quality in all components.

Table A.3: Product factors of the case study

	Product Factor	***Flexibility & Changeability***	***Impact on Architecture***
P1	**Functional features**		
P1.1	*Data communication*		
P1.1.1	*Non-blocking communication*		
	The communication infrastructure should ensure that components do not block each other because of using the same data connections.	This feature is more important for inter process communication than for intra process communication. The data flow may vary from few and frequent to large and slow data exchange.	The design has to ensure an appropriate non-blocking data flow between the components regardless of the exchanged data and little delay.
P1.1.2	*Configuration of communication*		
	The initialization of interacting components has to be enabled by configuration files.	For some parts of the system a dynamic lookup strategy for suitable components in a processing chain may be used.	Configuration files have to be covered by the serialization components.
P1.2	*Serialization*		
	Data (objects) have to be (de-)serialized with a unified strategy, in different data format types (XML, binary, plain text) and to different targets (file, console, TCP, STL streams).	New data formats can be introduced every few months, but all are covered by the specified types.	This feature affects all components that deal with saving and loading data as well as transfering data via network connections. Serialization is influenced by portability.

Table A.3: Product factors of the case study (continued)

	Product Factor	***Flexibility & Changeability***	***Impact on Architecture***
P1.3	*Logging*		
	There should be a unified logging strategy with different log-levels, filter capabilities and targets as 'cout' or files. Logs should be transferable with a data channel.	The factor is stable, changes to the serialization should not influence logging.	There is a moderate impact on the components for serialization and communication and a minor influence on all components that use the logging because of its cross-cutting nature.
P1.4	*Object (person) tracking*		
	Modular design of detection and tracking components is an important demand.	Due to changing scenarios there is a demand on recombining detectors for tracking.	There is a minor influence on the serialization component due to special data types used in this domain.
P2	**Quality features**		
P2.1	*Evolvability*		
P2.1.1	*Testability*		
	New system components, on which other components rely on, have to be well tested before productive usage.	This constraint is flexible for experimental components.	This feature affects the test plan, the release cycle, and the quality assurance for the developers.
P2.1.2	*Modifiability*		
	Hardware components (e.g., laser, camera) should be replaceable by new ones without changing anything but the driver components.	There are many platforms with different combinations of hardware. New data types may emerge.	There is a major influence on the communication and serialization interfaces and components.

Table A.3: Product factors of the case study (continued)

	Product Factor	***Flexibility & Changeability***	***Impact on Architecture***
P 2.1.2.1	*Extensibility* Integrating a new component (e.g., a hardware driver) should only need few overhead effort in the configuration with less than 50LoC.	New components regularly appear due to new hardware or due to experimental software components that are target of research.	There is a major influence on the configuration and communication components.
P 2.1.2.2	*Portability* The system has to manage hardware drivers running on different operating systems.	A driver for a hardware component changing every few months might at first only be available for Windows.	The design of the communication system has to support interoperability between different OS. There is also an influence on the serialization component.
P 2.1.2.3	*Variability* A variable recombination of existing components (e.g., detectors) to new ones (e.g., for a new person tracker) is desirable.	This demand does matter in different domain-specific applications of the robot scenario.	This factor has a moderate influence on basic and domain-specific components and interfaces.
P 2.1.3	*Reusability* There should be a way to continue working with old component (blackboard clients) in the new system.	With a migration strategy the old and new system have to work together until the old system is phased out.	There is a moderate influence on the communication infrastructure.

Table A.3: Product factors of the case study (continued)

	Product Factor	***Flexibility & Changeability***	***Impact on Architecture***
P 2.2	*Efficiency*		
P 2.2.1	*Time behavior*		
	For a high-performance data exchange different types and amounts of data have to be considered: numerous but lesser sensor data (<KB), middle size but infrequent map data (MB), continuous large image or audio data (MB).	The amount of data at least increases with a new camera or due to more demanding navigation scenarios with larger map data.	There is a major influence on the processor load, on components that are responsible for data communication, and on distributability for a small delay in transferring the data.
P 2.2.2	*Resource utilization*		
	After removing (deleting) a component from a communication chain of the running system ideally all resources should be freed.	If resource consumption becomes a limiting factor, this requirement will become a must, otherwise freeing resources with a restart might be sufficient.	The design should ensure monitoring of loaded components in the running system and monitoring of threads.
P 2.3	*Reliability*		
	For fault handling human-readable exception messages and monitoring of the system should be provided.	A strategy for error messages has to be designed, but the feature is stable.	This feature affects the interfaces and usability of all modules (clients).
P 2.3.1	*Maturity*		
	Only mature and well tested components should be included in official releases.	The maturity may vary for experimental and productive components.	This feature affects all components that should make it into an official release.

Table A.3: Product factors of the case study (continued)

	Product Factor	***Flexibility & Changeability***	***Impact on Architecture***
P 2.3.2	*Fault tolerance*		
P 2.3.2.1	*Error handling*		
	Every module should handle errors in its responsibility to avoid ripple effects because of errors. An exception should not crash the whole system.	The fulfillment of this feature may vary for experimental and productive components.	This feature affects all components that should make it into an official release. All occurring exceptions have to be caught somewhere.
P 2.3.2.2	*Crash avoidance*		
	Critical components can reside in an own process if appropriate, to separate them from other components and reduce the negative effects of a possible crash.	The criticality of the modules has to be decided and can change over time.	The design has to enable intra and inter process communication. Modules with extensive data communication should share one process if possible.
P 2.3.3	*Recoverability*		
	A component should send a heartbeat signal to enable to trigger appropriate means for recovery, e.g., restart of a component or the whole system.	Recovery strategies can be established by components for themselves as well.	All components have to implement an administrative interface for a heartbeat signal and and have to deal with error handling.
P 2.4	*Distributability*		
	Components should work together also in a distributed environment with different robots and operating systems.	Distribution of components is target to future work.	There is a major influence on the communication components since they have to support data exchange between remote systems. For the GUI a C# interface on a remote machine might be used.

Table A.3: Product factors of the case study (continued)

	Product Factor	***Flexibility & Changeability***	***Impact on Architecture***
P2.5	*Security* In a distributed setting the communication between components should be secure.	Security features are target of future work.	There is the risk that many components of the system are affected by introducing security concepts.
P2.6	*Usability* The configuration for different components to be combined has to be so easy that it takes an expert only half an our to get them to work.	The feature is flexible in the time constraints, but nevertheless configuration should be manageable also for the inexperienced user.	This factor affects the structure and management of configuration files and the design of data communication interfaces.
P3	**Service**		
P3.1	*Maintenance of software* Service patches regularly have to be applied in a productive setting.	Either ML or other experts from university apply the patches, not the customers.	A life update feature on the running system is imaginable if effort is reduced.
P4	**Product cost**		
P4.1	*Hardware budget* The budget for hardware is limited, which restricts the memory capacity and processor power of the robots.	There might be a budget for new hardware in the future to improve computing power.	The design of the system has to meet current hardware constraints, no hardware can be build to assure a working design.
P4.2	*Software budget* There is a limited budget for licensing off-the-shelf software. Usually only free open source software is used.	If inevitable, necessary software will be bought.	This factor affects the design of the system and maybe the schedule since free open source software is not always available.

A.2 IssueCards

This section illustrates some exemplary issue cards of the robot case study.

Table A.4: Issue card Skills of the case study

Skills
The developers are good programmers but there might be some deficiencies with certain technologies and knowledge for architectural design. **Influencing Factors:** **O2.2** There are only few developers with experience in software design and architectural design. **O2.4** There are only few developers with experience in specialized analysis techniques such as architectural assessments.
Solution: There are two strategies to perform proper architectural design. *Strategy: Team members could be trained.* Training for specific design aspects could be introduced for example for architectural patterns. *Strategy: Consultancy.* Work together with specialists from the software systems department at the university. Invite them as consultants for input and feedback for the redesign of the system.
Related Issues:

Table A.5: Issue card Serialization of the case study

Serialization
In the old system there are several different inconsistent ways to save and load data: plain text files for parameters or for configuration or XML files. Serialization to files and data streams or to the console is necessary. **Influencing Factors:** **T3.4** With C++ it is difficult to save types because there is no built-in reflection mechanism. **T5.3** XML is used as a data format for configuration files. **T5.4** Communication via TCP needs serialization. **P1.1.2** Data communication needs serialization for configuration files. **P1.2** The serialization feature includes textural, binary, or XML files as well as output to the network via TCP, to the console or into STL data streams. **P1.3** "Log files" for data should be stored using the serialization component. **P2.2.1** Performance is important for binary data that are transmitted via inter process communication. **P2.1.2.2** Portability: Serialization has to deal with components that are working together on different operation systems.
Solution: There has to be a common and unified serialization component that is accessible and used in every part of the system where loading, saving, or transferring data in a serialized form is necessary. *Strategy: Use existing serialization library.* There are free and open source serialization libraries such as "libs11n" and "boost serialization" that could be used. They support object serialization and can serialize to XML files. It has to be considered if they are flexible enough for all demands.

Table A.5: Issue card Serialization of the case study (cont.)

Strategy: Build an own serialization component. If the available libraries are not flexible enough, an own serialization component could be built that can serialize to files, streams, the console in textual, binary or XML format. For XML serialization the "libXML2" could be used. The reflection pattern could be used for a non-intrusive approach for object serialization.
Related Issues: **O1.1** Build vs. buy: There is a preference to use mature free and open source software or to build.

Table A.6: Issue card Hardware Changes of the case study

Changes in general-purpose and domain-specific hardware
Changes in both general-purpose and domain-specific changes are expected to occur regularly when purchasing a new robot. The effort and time for adapting to the new hardware should be reduced to a minimum. **Influencing Factors:** **T1.1 Processors, T1.2 Network, T1.3 Memory** Changes frequently occur as technology improves. Advantage should be taken from increased processor speed, memory or disk capacity if new hardware is available. **T2.1 Laser, T2.2 Camera** Changes to the domain-specific hardware can be important for certain research approaches. A camera is replaced typically with every new robot. With more image data the data rate and memory requirements are affected.
Solution: The software that directly operates with the domain-specific hardware should be separated. Regarding improved general-purpose hardware algorithms should scale and the design has to consider multi core technology. *Strategy: Encapsulate domain-specific hardware.* The drivers for the domain-specific hardware such as cameras should be encapsulated in a software component. Other software components should not depend directly on the drivers and have to communicate with the driver modules in a way that allows an easy exchange. The drivers might be available only for certain operating systems as well, so portability in the communication between software components on different operating systems has to be ensured. *Strategy: Use multi threading and parallel data processing.* The algorithms and software components should be designed to take advantage from multi core processors by using threads and a design that allows parallel data processing by different components.

Table A.6: Issue card Hardware Changes of the case study (cont.)

Related Issues:
P2.1.2 Modifiability/Portability: New drivers for domain-specific hardware may only be available for certain operating systems. Their exchange should not produce ripple effects in changing software components.
P4.1 The hardware budget is limited.

Table A.7: Issue card Software Changes of the case study

Changes in software technology
Changes in the used Linux distributions and software libraries such as Boost or Qt frequently occur. Often different versions are in use in parallel. The effort for updating to the latest versions should be minimal. **Influencing Factors:** **T3.1 Operating system, T3.2 User interface, T3.3 Software components, T3.4 Implementation language** The different libraries that are used, are updated frequently. If new features are introduced in newer versions, advantage should be easily taken of it with low effort.
Solution: There has to be an easy way to configure the necessary libraries to be able to update their versions. *Strategy: Encapsulate software libraries.* Software libraries could be encapsulated in special components to reduce ripple effects because of changes in the external interfaces. *Strategy: Use standards.* Maybe the early introduction of the C++0x standard can provide helpful features for development and reduce the effort in contrast to a workaround an a later migration. *Strategy: Dynamic loading of libraries.* Integrate a mechanism that enables dynamic loading of libraries without directly specifying a library version in a configuration file.
Related Issues: **T5.1** Compatibility with different Linux distributions is an issue that often leads to the necessity to build libraries oneself from source code.

A.3 UML Diagrams

This section depicts some UML diagrams of the architectural view documentation of the robot case study.

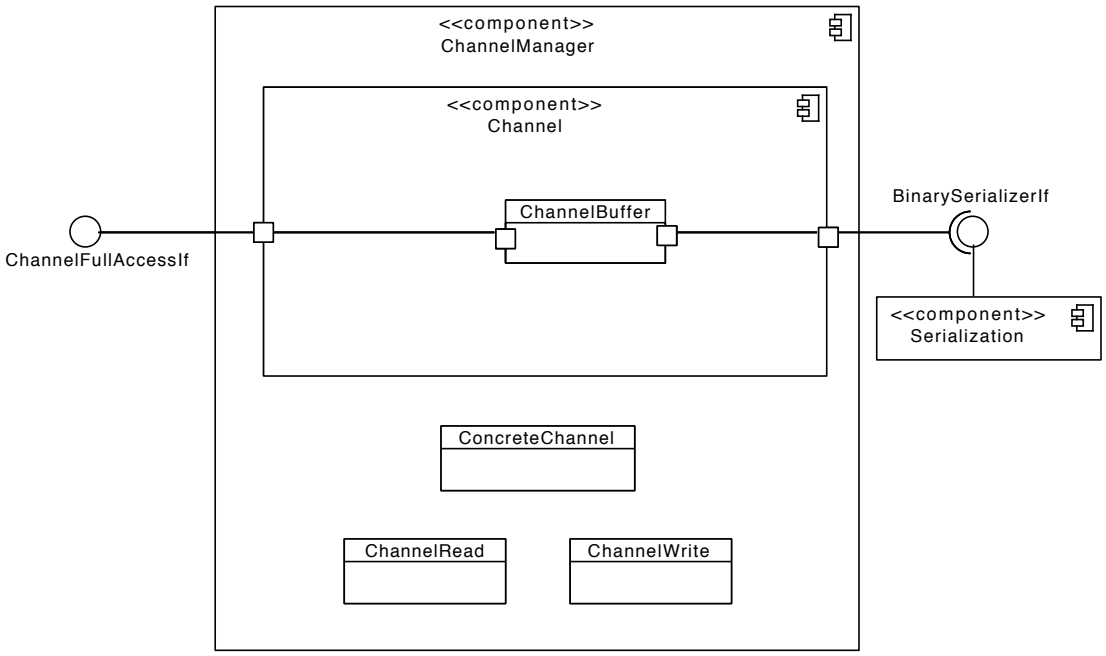


Figure A.1: Structural view of the `ChannelManager`

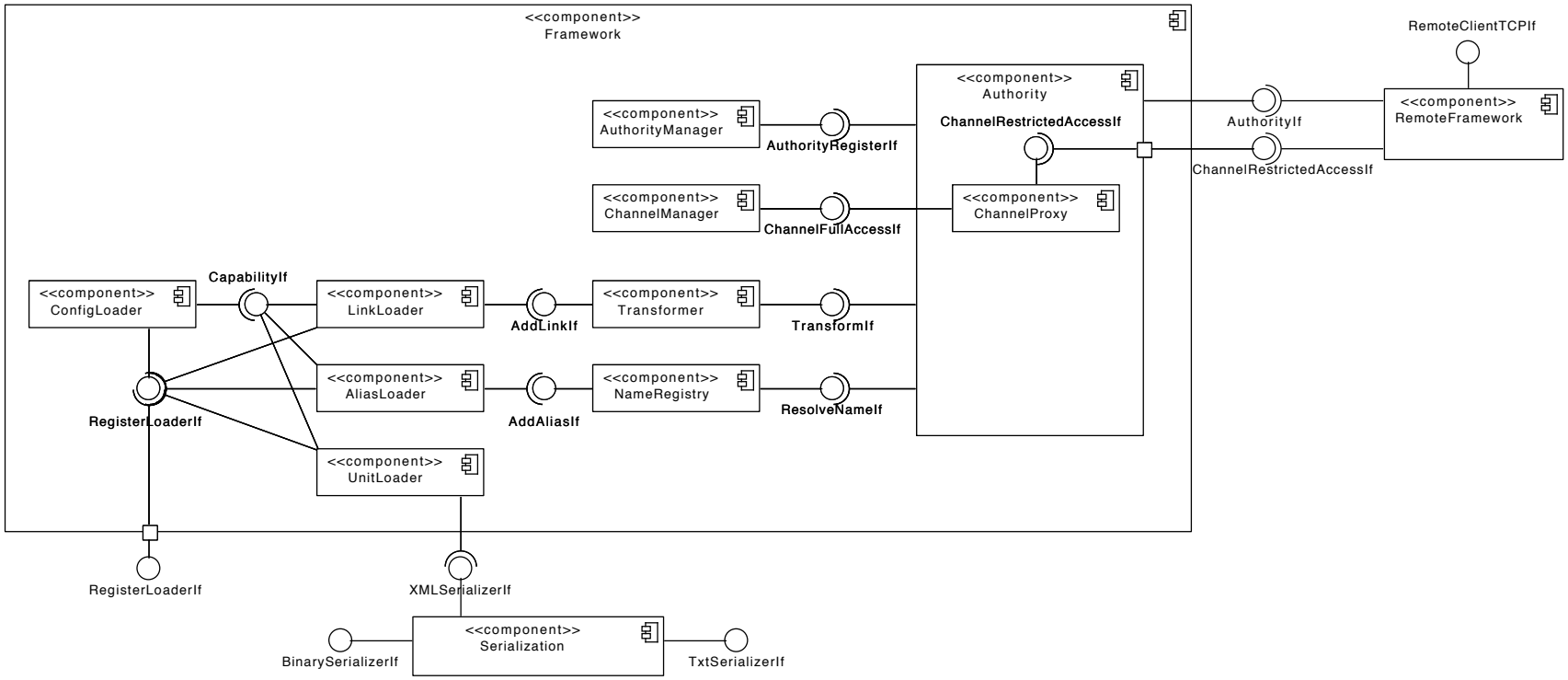


Figure A.2: Structural view of the communication framework

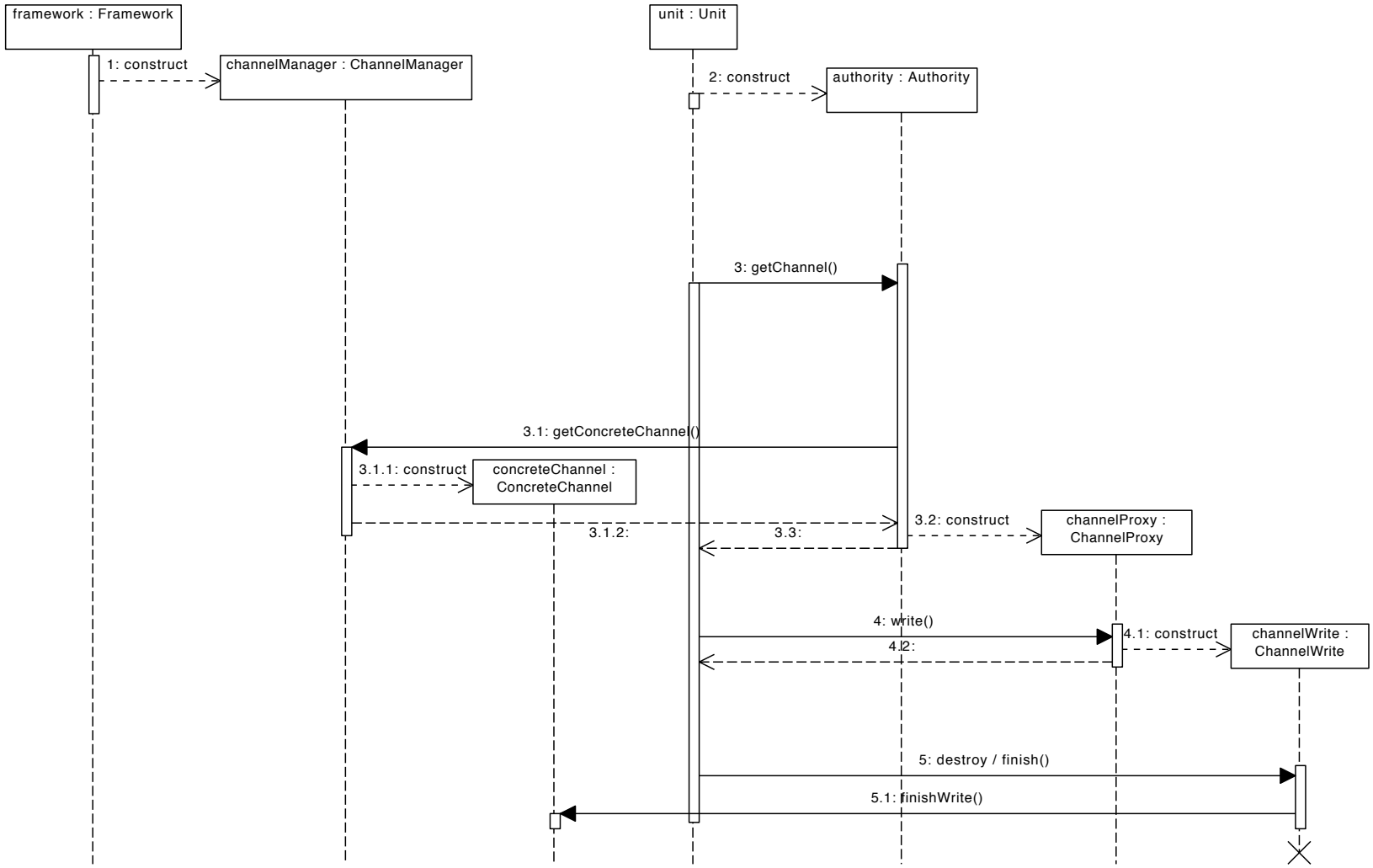


Figure A.3: Behavioral view of the procedure to write into a `Channel`

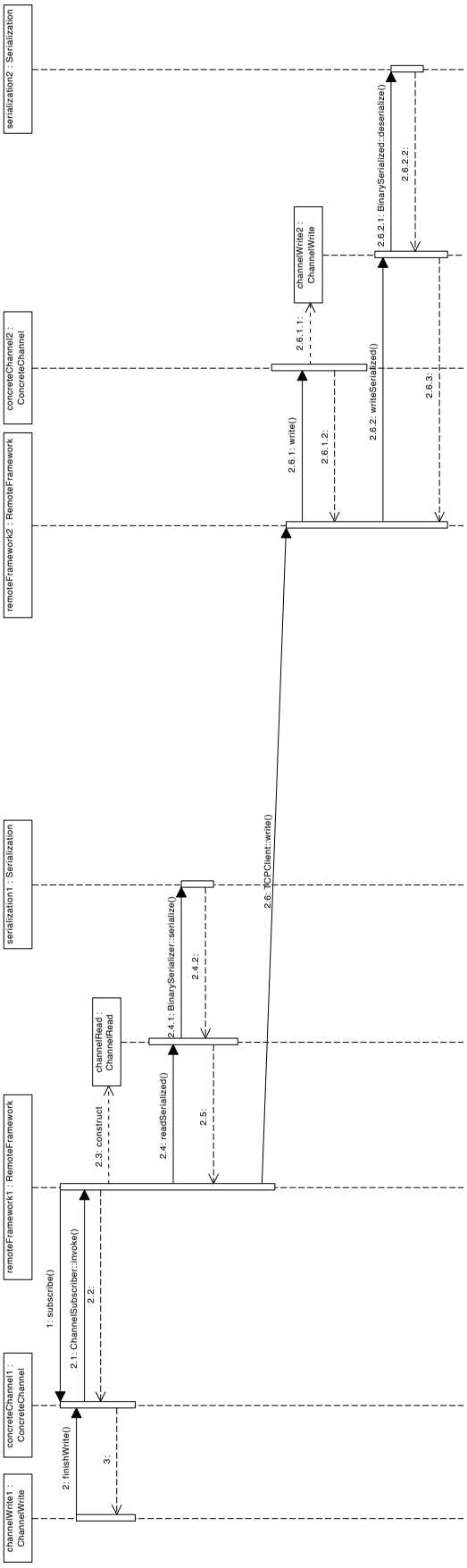


Figure A.4: Behavioral view of the communication between remote frameworks

Appendix B

Traceability Artifacts

B.1 Traceability Rule Catalog

This section contains the XML schema definition for the traceability rules and the list of traceability rules that were specified with XML. Listing B.1 presents the XSD for the traceability rules. Table B.1 lists the rules in a simplified representation style, and Listing B.2 shows a cutout of the rule catalog as XML representation.

Listing B.1: The XML Schema Definition for the traceability rules

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3    <xs:element name="RuleCatalogue">
4      <xs:complexType>
5        <xs:sequence>
6          <xs:element maxOccurs="unbounded" ref="Rule"/>
7        </xs:sequence>
8        <xs:attribute name="Description" use="required"
             type="xs:string"/>
9        <xs:attribute name="Name" use="required" type="xs:NCName"/>
10     </xs:complexType>
11   </xs:element>
12   <xs:element name="Rule">
13     <xs:complexType>
14       <xs:sequence>
15         <xs:element maxOccurs="unbounded" ref="Elements"/>
16         <xs:element ref="Conditions"/>
17         <xs:element maxOccurs="unbounded" ref="Actions"/>
18       </xs:sequence>
```

```
19        <xs:attribute name="Description" use="required"
              type="xs:string"/>
20        <xs:attribute name="RuleID" use="required" type="xs:NCName"/>
21      </xs:complexType>
22    </xs:element>
23    <xs:element name="Elements">
24      <xs:complexType>
25        <xs:attribute name="Alias" use="required" type="xs:NCName"/>
26        <xs:attribute name="Type" use="required" type="ElementType"/>
27      </xs:complexType>
28    </xs:element>
29    <xs:element name="Conditions">
30      <xs:complexType>
31        <xs:choice maxOccurs="unbounded">
32          <xs:element ref="BaseConditions"/>
33          <xs:element ref="LogicConditions"/>
34        </xs:choice>
35        <xs:attribute name="Type" use="required"
              type="LogicConditionType"/>
36      </xs:complexType>
37    </xs:element>
38    <xs:element name="Actions">
39      <xs:complexType>
40        <xs:attribute name="ActionType" use="required"
              type="ActionType"/>
41        <xs:attribute name="LinkSource" use="required" type="xs:NCName"/>
42        <xs:attribute name="LinkTarget" use="required" type="xs:NCName"/>
43        <xs:attribute name="LinkType" use="required" type="xs:NCName"/>
44      </xs:complexType>
45    </xs:element>
46    <xs:element name="LogicConditions">
47      <xs:complexType>
48        <xs:choice minOccurs="0" maxOccurs="unbounded">
49          <xs:element ref="BaseConditions"/>
50          <xs:element ref="LogicConditions"/>
51        </xs:choice>
52        <xs:attribute name="Type" use="required"
              type="LogicConditionType"/>
53      </xs:complexType>
54    </xs:element>
55    <xs:element name="BaseConditions">
```

```
56      <xs:complexType>
57        <xs:attribute name="Source" use="required" type="ScopedElement"/>
58        <xs:attribute name="Target" type="ScopedElement"/>
59        <xs:attribute name="Type" use="required"
              type="BaseConditionType"/>
60        <xs:attribute name="Value"/>
61      </xs:complexType>
62    </xs:element>
63    <xs:simpleType name="LogicConditionType">
64      <xs:restriction base="xs:string">
65        <xs:enumeration value="And"/>
66        <xs:enumeration value="Or"/>
67        <xs:enumeration value="Not"/>
68        <xs:enumeration value="Xor"/>
69      </xs:restriction>
70    </xs:simpleType>
71    <xs:simpleType name="BaseConditionType">
72      <xs:restriction base="xs:string">
73        <xs:enumeration value="Equals"/>
74        <xs:enumeration value="Contains"/>
75        <xs:enumeration value="SimilarTo"/>
76        <xs:enumeration value="GreaterThan"/>
77        <xs:enumeration value="LesserThan"/>
78        <xs:enumeration value="IsParent"/>
79        <xs:enumeration value="NotNull"/>
80      </xs:restriction>
81    </xs:simpleType>
82    <xs:simpleType name="ActionType">
83      <xs:restriction base="xs:string">
84        <xs:enumeration value="CreateLink"/>
85      </xs:restriction>
86    </xs:simpleType>
87    <xs:simpleType name="ElementType">
88        <xs:restriction base="xs:string">
89          <!-- NCName(|NCName)* or '*' -->
90          <xs:pattern
                value="([\i-[:]][\c-[:]]*(\|([\i-[:]][\c-[:]]*))*)|\*"/>
91        </xs:restriction>
92    </xs:simpleType>
93    <xs:simpleType name="ScopedElement">
94      <xs:restriction base="xs:string">
```

```
95        <!-- Element::Attribut -> NCName(::NCName)* -->
96        <xs:pattern value="[\i-[:]][\c-[:]]*(::([\i-[:]][\c-[:]]*))*"/>
97      </xs:restriction>
98    </xs:simpleType>
99  </xs:schema>
```

Table B.1: Rules for traceability link establishment

ID	**Description** (adopted from Source)
Rule01	**Find UML/URN-Actors that share a similar name** ([FZS03])
Elements	`Actor e1, Actor e2`
Conditions	`And(NotNull("e1::id"), NotNull("e2::umlID"), SimilarTo("e1::name", "e2::name"))`
LinkType	`Overlap(e1 -> e2)`
Rule02	**Find URN-Actors and UML-Classes that share a similar name** ([FZS03])
Elements	`Actor e1, Class e2`
Conditions	`And(NotNull("e1::id"), NotNull("e2::umlID"), SimilarTo("e1::name", "e2::name"))`
LinkType	`Overlap(e1 -> e2)`
Rule03	**Find URN-Resources and UML-Classes that share a similar name** ([FZS03])
Elements	`IntentionalElement e1, Class e2`
Conditions	`And(NotNull("e2::umlID"), Equals("e1::type", "Resource"), SimilarTo("e1::name", "e2::name"))`
LinkType	`Overlap(e1 -> e2)`
Rule04	**Find URN-Goals/Softgoals/Tasks and UML-Operations that share a similar name** ([FZS03])
Elements	`IntentionalElement e1, Operation e2`
Conditions	`And(Or(Equals("e1::type", "Task"), Equals("e1::type", "Softgoal"), Equals("e1::type", "Goal")), SimilarTo("e1::name", "e2::name"))`
LinkType	`Overlap(e1 -> e2)`

Table B.1: Rules for traceability link establishment (continued)

ID	**Description** (adopted from Source)
Rule05	**Find URN-Goals/Softgoals/Tasks and UML-UseCases that share a similar name** ([FZS03])
Elements	`IntentionalElement e1, UseCase e2`
Conditions	`And(Or(Equals("e1::type", "Task"), Equals("e1::type", "Softgoal"), Equals("e1::type", "Goal")), SimilarTo("e1::name", "e2::name"))`
LinkType	`Overlap(e1 -> e2)`
Rule06	**Find UML-StateMachines and UML-UseCases that share a similar name** ([JZ09])
Elements	`StateMachine e1, UseCase e2`
Conditions	`And(SimilarTo("e1::name", "e2::name"))`
LinkType	`Overlap(e1 -> e2)`
Rule07	**Find an UML-StateMachine that seems to be an evolution of another UML-StateMachine** ([JZ09])
Elements	`StateMachine e1, StateMachine e2`
Conditions	`And(Contains("e1::name", "e2::name"))`
LinkType	`Evolution(e1 -> e2)`
Rule08	**Find UML-Interactions that implement UML-UseCases** ([JZ09])
Elements	`Interaction e1, UseCase e2`
Conditions	`And(Equals("e1::name", "e2::name"))`
LinkType	`Implementation(e1 -> e2)`
Rule09	**Find UML-Interactions that refine UML-UseCases** ([JZ09])
Elements	`Interaction e1, UseCase e2`
Conditions	`And(Contains("e1::name", "e2::name"))`
LinkType	`Refinement(e1 -> e2)`
Rule10	**Find UML-Interactions that satisfy UML-UseCases** ([JZ09])
Elements	`Interaction e1, UseCase e2`
Conditions	`And(SimilarTo("e1::name", "e2::name"))`
LinkType	`Satisfiability(e1 -> e2)`
Rule11	**Find UML-Classes that implement UML-UseCases** ([JZ09])
Elements	`Class e1, UseCase e2`
Conditions	`And(NotNull("e1::umlID"), Equals("e1::name", "e2::name"))`
LinkType	`Implementation(e1 -> e2)`

Table B.1: Rules for traceability link establishment (continued)

ID	**Description** (adopted from Source)
Rule12	**Find UML-Interactions that refine UML-Classes** ([JZ09])
Elements	`Interaction e1, Class e2`
Conditions	`And(NotNull("e2::umlID"), Contains("e1::name", "e2::name"))`
LinkType	`Refinement(e1 -> e2)`
Rule13	**Find UML-Interactions that use UML-Classes** ([JZ09])
Elements	`Interaction e1, Message e2, Class e3, Operation e4`
Conditions	`And(NotNull("e3::umlID"), IsParent("e1", "e2"), Or(Equals("e2::name", "e3::name"), Contains("e2::name", "e3::name"), And(IsParent("e3", "e4"), Or(Equals("e2::name", "e4::name"), Contains("e2::name", "e4::name")))))`
LinkType	`Use(e1 -> e3)`
Rule14	**Find UML-StateMachines that refine UML-Interactions** ([JZ09])
Elements	`StateMachine e1, Interaction e2`
Conditions	`And(Contains("e1::name", "e2::name"))`
LinkType	`Refinement(e1 -> e2)`
Rule15	**Find UML-StateMachines that contain UML-Classes** ([JZ09])
Elements	`StateMachine e1, State e2, Class e3, Operation e4`
Conditions	`And(NotNull("e3::umlID"), IsParent("e1", "e2"), IsParent("e3", "e4"), Contains("e2::name", "e4::name"))`
LinkType	`Containment(e1 -> e3)`
Rule16	**Find UML-StateMachines and UML-Classes that share a similar name** ([JZ09])
Elements	`StateMachine e1, State e2, Class e3, Operation e4`
Conditions	`And(NotNull("e3::umlID"), IsParent("e1", "e2"), IsParent("e3", "e4"), SimilarTo("e2::name", "e4::name"))`
LinkType	`Overlap(e3 -> e1)`
Rule17	**Find UML-StateMachines that satisfy UML-UseCases** ([JZ09])
Elements	`StateMachine e1, Transition e2, UseCase e3`
Conditions	`And(IsParent("e1", "e2"), SimilarTo("e2::name", "e3::name"))`
LinkType	`Satisfiability(e1 -> e3)`
Rule18	**Find UML-Classes that satisfy UML-UseCases** ([JZ09])
Elements	`Class e1, Operation e2, UseCase e3`
Conditions	`And(NotNull("e1::umlID"), IsParent("e1", "e2"), SimilarTo("e2::name", "e3::name"))`
LinkType	`Satisfiability(e1 -> e3)`

Table B.1: Rules for traceability link establishment (continued)

ID	**Description** (adopted from Source)
Rule19	**Find UML-UseCases that represent evolutionary steps of other UML-UseCases** ([JZ09])
Elements	`UseCase e1, UseCase e2, Component e3, Component e4`
Conditions	`And(NotNull("e3::umlID"), NotNull("e4::umlID"), IsParent("e3", "e1"), IsParent("e4", "e2"), Equals("e3::name", "e4::name"), Contains("e1::name", "e2::name"))`
LinkType	`Evolution(e1 -> e2)`
Rule20	**Find UML-Classes that represent evolutionary steps of other UML-Classes** ([JZ09])
Elements	`Class e1, Class e2`
Conditions	`And(NotNull("e1::umlID"), NotNull("e2::umlID"), SimilarTo("e1::name", "e2::name"))`
LinkType	`Evolution(e1 -> e2)`
Rule21	**Find UML classes that refine UML-UseCases** ([JZ09])
Elements	`Class e1, Operation e2, UseCase e3`
Conditions	`And(NotNull("e1::umlID"), IsParent("e1", "e2"), Contains("e3::name", "e1::name"), Contains("e3::name", "e2::name"))`
LinkType	`Refinement(e1 -> e3)`
Rule22	**Find UML-StateMachines that implement UML-UseCases** ([JZ09])
Elements	`StateMachine e1, UseCase e2, Transition e3`
Conditions	`And(IsParent("e1", "e3"), Equals("e2::name", "e3::name"))`
LinkType	`Implementation(e1 -> e2)`
Rule23	**Find UML-Interactions that realize UML-UseCases** ([JZ09])
Elements	`Interaction e1, UseCase e2, Class e3, Message e4, Operation e5`
Conditions	`And(NotNull("e3::umlID"), IsParent("e1", "e4"), IsParent("e3", "e5"), Equals("e5::name", "e4::name"), SimilarTo("e2::name", "e3::name"))`
LinkType	`Realization(e1 -> e2)`
Rule24	**Find UML-Activities that realize UML-UseCases** ([JRH+03])
Elements	`Activity e1, UseCase e2`
Conditions	`And(Equals("e1::name", "e2::name"))`
LinkType	`Realization(e1 -> e2)`

Table B.1: Rules for traceability link establishment (continued)

ID	**Description** (adopted from Source)
Rule25	**Find UML-Activities that implement UML-Operations** ([JRH+03])
Elements	`Activity e1, Operation e2`
Conditions	`And(Equals("e1::name", "e2::name"))`
LinkType	`Implementation(e1 -> e2)`
Rule26	**Find Classes that refine Classes in Packages**
Elements	`Package e1, Class e2, Class e3`
Conditions	`And(NotNull("e2::umlID"), NotNull("e3::umlID"), IsParent("e1", "e2"), Not(Equals("e2::umlID", "e3::umlID")), Equals("e2::name", "e3::name"))`
LinkType	`Refinement(e3 -> e1)`
Rule27	**Find UML-Collaborations that realize UML-UseCases** ([JRH+03])
Elements	`Collaboration e1, UseCase e2`
Conditions	`And(Equals("e1::name", "e2::name"))`
LinkType	`Realization(e1 -> e2)`
Rule28	**Find similarities between UML-Classes and OWL-Classes**
Elements	`Class e1, Class e2`
Conditions	`And(Not(NotNull("e1::umlID")), NotNull("e2::umlID"), Or(Contains("e1::IRI", "e2::name"), Contains("e1::abbreviatedIRI", "e2::name")))`
LinkType	`Overlap(e1 -> e2)`
Rule29	**Find UML-Activities that refine UML-UseCases**
Elements	`Activity e1, UseCase e2`
Conditions	`And(Not(Equals("e1::name", "e2::name")), Contains("e1::name", "e2::name"))`
LinkType	`Refinement(e1 -> e2)`
Rule30	**Find UML-Classes that are part of an UML-Collaboration**
Elements	`Class e1, Collaboration e2, CollaborationUse e3`
Conditions	`And(NotNull("e1::umlID"), IsParent("e2", "e3"), Equals("e1::name", "e3::name"))`
LinkType	`Part-Of(e1 -> e2)`

Table B.1: Rules for traceability link establishment (continued)

ID	**Description** (adopted from Source)
Rule31	**Find UML-Classes that implement UML-Interfaces**
Elements	`Class e1, Interface e2`
Conditions	`And(NotNull("e1::umlID"), Or(Contains("e1::name", "e2::name"), Contains("e2::name", "e1::name"), SimilarTo("e1::name", "e2::name")))`
LinkType	`Implementation(e1 -> e2)`
Rule32	**Find UML-Classes that implement UML-Actors** ([JRH+03])
Elements	`Class e1, Actor e2`
Conditions	`And(NotNull("e1::umlID"), NotNull("e2::umlID"), Equals("e1::name", "e2::name"))`
LinkType	`Implementation(e1 -> e2)`
Rule33	**Find UML-Operations that refine UML-Interactions** ([JRH+03])
Elements	`Operation e1, Interaction e2`
Conditions	`And(Equals("e1::name", "e2::name"))`
LinkType	`Refinement(e1 -> e2)`
Rule34	**Find UML-Interactions that refine UML-Classes** ([JRH+03])
Elements	`Message e1, Property e2, Class e3, Interaction e4`
Conditions	`And(NotNull("e3::umlID"), IsParent("e3", "e2"), IsParent("e4", "e1"), Equals("e1::name", "e2::name"))`
LinkType	`Refinement(e4 -> e3)`
Rule35	**Find similar comments in UML and URN**
Elements	`Comment e1, Comment e2`
Conditions	`And(NotNull("e1::umlID"), Not(NotNull("e2::umlID")), SimilarTo("e1::body", "e2::description"))`
LinkType	`Overlap(e1 -> e2)`
Rule36	**Find similar comments in UML and OWL**
Elements	`Comment e1, Literal e2`
Conditions	`And(NotNull("e1::umlID"), SimilarTo("e1::body", "e2::Value"))`
LinkType	`Overlap(e1 -> e2)`
Rule37	**Find similar comments in URN and OWL**
Elements	`Comment e1, Literal e2`
Conditions	`And(Not(NotNull("e1::umlID")), SimilarTo("e1::description", "e2::Value"))`
LinkType	`Overlap(e1 -> e2)`

Table B.1: Rules for traceability link establishment (continued)

ID	**Description** (adopted from Source)
Rule38	**Find equivalent relations between URN and UML**
Elements	`Association e1, ElementLink e2, Class e3, Class e4, IntentionalElement e5, IntentionalElement e6`
Conditions	`And(NotNull("e3::umlID"), NotNull("e4::umlID"), Contains("e1::memberEnd", "e3::umlID"), Contains("e1::memberEnd", "e4::umlID"), Equals("e2::src", "e5::id"), Equals("e2::dest", "e6::id"), Equals("e3::name", "e5::name"), Equals("e4::name", "e6::name"))`
LinkType	`Equivalence(e1 -> e2)`
Rule39	**Find equivalent relations between URN and UML**
Elements	`Association e1, ElementLink e2, Interface e3, Interface e4, IntentionalElement e5, IntentionalElement e6`
Conditions	`And(Contains("e1::memberEnd", "e3::umlID"), Contains("e1::memberEnd", "e4::umlID"), Equals("e2::src", "e5::id"), Equals("e2::dest", "e6::id"), Equals("e3::name", "e5::name"), Equals("e4::name", "e6::name"))`
LinkType	`Equivalence(e1 -> e2)`
Rule40	**Find equivalent relations between URN and UML**
Elements	`Association e1, ElementLink e2, Interface e3, Class e4, IntentionalElement e5, IntentionalElement e6`
Conditions	`And(Contains("e1::memberEnd", "e3::umlID"), Contains("e1::memberEnd", "e4::umlID"), Equals("e2::src", "e5::id"), Equals("e2::dest", "e6::id"), Or(And(Equals("e3::name", "e5::name"), Equals("e4::name", "e6::name")), And(Equals("e3::name", "e6::name"), Equals("e4::name", "e5::name"))))`
LinkType	`Equivalence(e1 -> e2)`
Rule41	**Find equivalent subclass/superclass-relations in UML and OWL**
Elements	`SubClassOf e1, Generalization e2, Class e3, Class e4`
Conditions	`And(NotNull("e3::umlID"), NotNull("e4::umlID"), SimilarTo("e1::subClass", "e3::name"), SimilarTo("e1::superClass", "e4::name"), IsParent("e4", "e3"), Equals("e2::general", "e4::umlID"))`
LinkType	`Equivalence(e1 -> e2)`

Table B.1: Rules for traceability link establishment (continued)

ID	**Description** (adopted from Source)
Rule42	**Find equivalent subclass/superclass-relations in UML and OWL**
Elements	`SubClassOf e1, Generalization e2, Interface e3, Interface e4`
Conditions	`And(SimilarTo("e1::subClass", "e3::name"), SimilarTo("e1::superClass", "e4::name"), IsParent("e4", "e3"), Equals("e2::general", "e4::umlID"))`
LinkType	`Equivalence(e1 -> e2)`
Rule43	**Find equivalent subclass/superclass-relations in UML and OWL**
Elements	`SubClassOf e1, Generalization e2, Class e3, Interface e4`
Conditions	`And(NotNull("e3::umlID"), Or(And(SimilarTo("e1::subClass", "e4::name"), SimilarTo("e1::superClass", "e3::name"), IsParent("e3", "e4"), Equals("e2::general", "e3::umlID")), And(SimilarTo("e1::subClass", "e3::name"), SimilarTo("e1::superClass", "e4::name"), IsParent("e4", "e3"), Equals("e2::general", "e4::umlID"))))`
LinkType	`Equivalence(e1 -> e2)`
Rule44	**Find equivalent properties in UML and OWL**
Elements	`ObjectProperty e1, Property e2`
Conditions	`Or(Contains("e1::IRI", "e2::name"), Contains("e1::AbbreviatedIRI", "e2::name"), Contains("e1::FullIRI", "e2::name"))`
LinkType	`Equivalence(e1 -> e2)`
Rule45	**Find equivalent properties in UML and OWL**
Elements	`DataProperty e1, Property e2`
Conditions	`Or(Contains("e1::IRI", "e2::name"), Contains("e1::AbbreviactedIRI", "e2::name"), Contains("e1::FullIRI", "e2::name"))`
LinkType	`Equivalence(e1 -> e2)`
Rule46	**Find equivalent datatypes in UML and OWL**
Elements	`Datatype e1, DataType e2`
Conditions	`And(NotNull("e2::umlID"), Or(Contains("e1::IRI", "e2::name"), Contains("e1::AbbreviatedIRI", "e2::name"), Contains("e1::FullIRI", "e2::name")))`
LinkType	`Equivalence(e1 -> e2)`

Table B.1: Rules for traceability link establishment (continued)

ID	**Description** (adopted from Source)
Rule47	**Find equivalent datatypes in UML and OWL**
Elements	`Datatype e1, Class e2`
Conditions	`And(NotNull("e2::umlID"), Or(Contains("e1::IRI", "e2::name"), Contains("e1::AbbreviatedIRI", "e2::name"), Contains("e1::FullIRI", "e2::name")))`
LinkType	`Equivalence(e1 -> e2)`
Rule48	**Find contribution-relationships between URN goals/softgoals**
Elements	`IntentionalElement e1, IntentionalElement e2, Contribution e3`
Conditions	`And(Or(Equals("e1::type", "Goal"), Equals("e1::type", "Softgoal")), Or(Equals("e2::type", "Goal"), Equals("e2::type", "Softgoal")), Equals("e3::dest", "e1::id"), Equals("e3::src", "e2::id"))`
LinkType	`Contribution(e1 -> e2)`
Rule49	**Find decomposition-relationships between URN goals/softgoals**
Elements	`IntentionalElement e1, IntentionalElement e2, Decomposition e3`
Conditions	`And(Or(Equals("e1::type", "Goal"), Equals("e1::type", "Softgoal")), Or(Equals("e2::type", "Goal"), Equals("e2::type", "Softgoal")), Equals("e3::dest", "e1::id"), Equals("e3::src", "e2::id"))`
LinkType	`Decomposition(e2 -> e1)`
Rule50	**Find similarities between UML-Aggregation and OWL-ObjectIntersection**
Elements	`ObjectIntersectionOf e1, Property e2, Class e3, Class e4`
Conditions	`And(NotNull("e3::umlID"), NotNull("e4::umlID"), Contains("e1::class", "e4::name"), Equals("e2::aggregation", "composite"), IsParent("e3", "e2"), Equals("e2::type", "e4::umlID"))`
LinkType	`Overlap(e1 -> e3)`
Rule51	**Find similarities between UML-Aggregation and OWL-ObjectUnion**
Elements	`ObjectUnionOf e1, Property e2, Class e3, Class e4`
Conditions	`And(NotNull("e3::umlID"), NotNull("e4::umlID"), Contains("e1::unionClasses", "e4::name"), Equals("e2::aggregation", "composite"), IsParent("e3", "e2"), Equals("e2::type", "e4::umlID"))`
LinkType	`Overlap(e1 -> e3)`

Table B.1: Rules for traceability link establishment (continued)

ID	**Description** (adopted from Source)
Rule52	**Find Classes that are a Part-Of a Component**
Elements	`Component e1, Class e2`
Conditions	`And(NotNull("e1::umlID"), NotNull("e2::umlID"), IsParent("e1", "e2"))`
LinkType	`Part-of(e2 -> e1)`
Rule53	**Fina a StateMachine that is a refinement of a UseCase** ([JZ09])
Elements	`StateMachine e1, State e2, UseCase e3`
Conditions	`And(IsParent("e1", "e2"), Equals("e2::name", "e3::name"))`
LinkType	`Refinement(e1 -> e3)`
Rule54	**Find Classes that are part of an Interaction**
Elements	`Interaction e1, Property e2, Lifeline e3, Class e4`
Conditions	`And(NotNull("e4::umlID"), IsParent("e1", "e3"), Or(Equals("e3::name", "e4::name"), SimilarTo("e3::name", "e4::name"), And(IsParent("e1", "e2"), Equals("e2::type", "e4::umlID"), Equals("e2::umlID", "e3::represents"))))`
LinkType	`Instance-Of(e3 -> e4)`
Rule55	**Find Interfaces that are part of an Interaction**
Elements	`Interaction e1, Property e2, Lifeline e3, Interface e4`
Conditions	`And(IsParent("e1", "e3"), Or(Equals("e3::name", "e4::name"), SimilarTo("e3::name", "e4::name"), And(NotNull("e4::umlID"), IsParent("e1", "e2"), Equals("e2::type", "e4::umlID"), Equals("e2::umlID", "e3::represents"))))`
LinkType	`Instance-Of(e3 -> e4)`
Rule56	**Find Components that are part of an Interaction**
Elements	`Interaction e1, Lifeline e2, Component e3`
Conditions	`And(IsParent("e1", "e2"), NotNull("e3::umlID"), Or(Equals("e2::name", "e3::name"), SimilarTo("e2::name", "e3::name")))`
LinkType	`Instance-Of(e2 -> e3)`

Table B.1: Rules for traceability link establishment (continued)

ID	**Description** (adopted from Source)
Rule57	**Find Interfaces that are part of an Interaction via used Operations**
Elements	`Interaction e1, Message e2, Interface e3, Operation e4`
Conditions	`And(IsParent("e1", "e2"), Or(Equals("e2::name", "e3::name"), SimilarTo("e2::name", "e3::name"), And(IsParent("e3", "e4"), Or(Equals("e2::name", "e4::name"), SimilarTo("e2::name", "e4::name")))))`
LinkType	`Use(e2 -> e3)`
Rule58	**Establish Part-Of-relations between components and their parts**
Elements	`Component e1, Component|Class e2`
Conditions	`And(NotNull("e1::umlID"), NotNull("e2::umlID"), IsParent("e1", "e2"))`
LinkType	`Part-Of(e2 -> e1)`
Rule59	**Find UML-Interfaces provided by UML-Components**
Elements	`Component e1, Interface e2, InterfaceRealization e3`
Conditions	`And(NotNull("e1::umlID"), IsParent("e1", "e3"), Equals("e2::umlID", "e3::supplier"))`
LinkType	`Provide(e1 -> e2)`
Rule60	**Find equivalents for URN-Goals/Softgoals in Ontologies**
Elements	`IntentionalElement e1, Class e2`
Conditions	`And(Or(Equals("e1::type", "Goal"), Equals("e1::type", "Softgoal")), Or(SimilarTo("e2::IRI", "e1::name"), SimilarTo("e2::abbreviatedIRI", "e1::name")))`
LinkType	`Equivalence(e1 -> e2)`
Rule61	**Find similarities between UML-Components and OWL-Classes**
Elements	`Class e1, Component e2`
Conditions	`And(NotNull("e2::umlID"), Or(Contains("e1::IRI", "e2::name"), Contains("e1::abbreviatedIRI", "e2::name")))`
LinkType	`Overlap(e1 -> e2)`
Rule62	**Find UML-Interfaces that satisfy UML-UseCases**
Elements	`Interface e1, Operation e2, UseCase e3`
Conditions	`And(IsParent("e1", "e2"), SimilarTo("e2::name", "e3::name"))`
LinkType	`Satisfiability(e1 -> e3)`

Table B.1: Rules for traceability link establishment (continued)

ID	**Description** (adopted from Source)
Rule63	**Find all OWL subclass relations**
Elements	`SubClassOf e1, Class e2, Class e3`
Conditions	`And(Or(Equals("e2::IRI", "e1::subClass"),` `Equals("e2::abbreviatedIRI", "e1::subClass")),` `Or(Equals("e3::IRI", "e1::superClass"),` `Equals("e3::abbreviatedIRI", "e1::superClass")))`
LinkType	`Is-A(e2 -> e3)`
Rule64	**Find UML-Interfaces required by UML-Components**
Elements	`Interface e1, Component e2, Usage e3`
Conditions	`And(Equals("e3::supplier", "e1::umlID"), Equals("e3::client",` `"e2::umlID"))`
LinkType	`Require(e2 -> e1)`
Rule65	**Find UML-Lifelines that construct/init other UML-Lifelines**
Elements	`Lifeline e1, Lifeline e2, Message e3`
Conditions	`And(Equals("e3::messageSort", "createMessage"),` `Contains("e1::coveredBy", "e3::sendEvent"),` `Contains("e2::coveredBy", "e3::receiveEvent"))`
LinkType	`Activate(e1 -> e2)`
Rule66	**Find UML-Lifelines that destroy/close/de-init other UML-Lifelines**
Elements	`Lifeline e1, Lifeline e2, Message e3`
Conditions	`And(Equals("e3::messageSort", "deleteMessage"),` `Contains("e1::coveredBy", "e3::sendEvent"),` `Contains("e2::coveredBy", "e3::receiveEvent"))`
LinkType	`Deactivate(e1 -> e2)`
Rule67	**Find UML-Lifelines that call other UML-Lifelines**
Elements	`Lifeline e1, Lifeline e2, Message e3`
Conditions	`And(Not(Equals("e3::messageSort", "createMessage")),` `Not(Equals("e3::messageSort", "deleteMessage")),` `Contains("e1::coveredBy", "e3::sendEvent"),` `Contains("e2::coveredBy", "e3::receiveEvent"))`
LinkType	`Use(e1 -> e2)`

Table B.1: Rules for traceability link establishment (continued)

ID	**Description** (adopted from Source)
Rule68	**Find similarities between UML-Interfaces and OWL-Classes**
Elements	`Interface e1, Class e2`
Conditions	`Or(Contains("e2::IRI", "#Interface"), Contains("e2::abbreviatedIRI", "#Interface"))`
LinkType	`Similarity(e1 -> e2)`
Rule69	**Connect an EMFfit 'Link' element with its target ModelElement**
Elements	`Link e1, * e2`
Conditions	`And(Equals("e1::target::identifier", "e2::identifier"))`
LinkType	`Overlap(e1 -> e2)`
Rule70	**Find factors and factor categories that are realizations of URN goals or softgoals**
Elements	`Factor|FactorCategory e1, IntentionalElement e2`
Conditions	`And(Or(Equals("e2::type", "Goal"), Equals("e2::type", "Softgoal")), Equals("e1::name", "e2::name"))`
LinkType	`Realization(e2 -> e1)`
Rule71	**Find issue cards that are realizations of influencing factors**
Elements	`IssueCard e1, Factor e2, InfluencingFactor e3`
Conditions	`And(IsParent("e1", "e3"), Equals("e3::factor::identifier", "e2::identifier"))`
LinkType	`Realization(e2 -> e1)`
Rule72	**Find issue cards that overlap with (are related to) other factors or factor categories**
Elements	`IssueCard e1, Factor|FactorCategory e2, RelatedIssue e3`
Conditions	`And(IsParent("e1", "e3"), Equals("e3::issue::identifier", "e2::identifier"))`
LinkType	`Overlap(e1 -> e2)`
Rule73	**Find issue cards that overlap with (are related to) other issue cards**
Elements	`IssueCard e1, IssueCard e2, RelatedIssue e3, Strategy e4`
Conditions	`Or(And(IsParent("e1", "e3"), Equals("e3::issue::identifier", "e2::identifier")), And(IsParent("e1", "e3"), Equals("e3::issue::identifier", "e4::identifier"), IsParent("e2", "e4")))`
LinkType	`Overlap(e1 -> e2)`

Table B.1: Rules for traceability link establishment (continued)

ID	**Description** (adopted from Source)
Rule74	**Find issue cards that overlap with (are related to) strategies of other issue cards**
Elements	`IssueCard e1, Strategy e2, RelatedIssue e3`
Conditions	`And(IsParent("e1", "e3"), Equals("e3::issue::identifier", "e2::identifier"))`
LinkType	`Overlap(e1 -> e2)`
Rule75	**Find factors and factor categories that overlap with OWL classes**
Elements	`Factor|FactorCategory e1, Class e2`
Conditions	`Or(SimilarTo("e1::name", "e2::IRI"), SimilarTo("e1::name", "e2::abbreviatedIRI"))`
LinkType	`Overlap(e1 -> e2)`
Rule76	**Find issue cards that overlap with OWL classes**
Elements	`IssueCard e1, Class e2`
Conditions	`Or(SimilarTo("e1::name", "e2::IRI"), SimilarTo("e1::name", "e2::abbreviatedIRI"))`
LinkType	`Overlap(e1 -> e2)`

Listing B.2: Cutout of the traceability rule catalog in XML representation

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <RuleCatalogue Name="EMFTrace_TraceRules" Description="This␣catalogue␣
       contains␣rules␣aimed␣upon␣finding␣traceability-links␣between␣
       models">
3
4      <Rule RuleID="TraceRule01" Description="Find␣UML/URN-Actors␣that␣
           share␣a␣similar␣name">
5          <Elements Type="Actor" Alias="e1"/>
6          <Elements Type="Actor" Alias="e2"/>
7          <Conditions Type="And">
8              <BaseConditions Type="NotNull" Source="e1::id"/>
9              <BaseConditions Type="NotNull" Source="e2::umlID"/>
10             <BaseConditions Type="SimilarTo" Source="e1::name"
                   Target="e2::name"/>
11         </Conditions>
12         <Actions ActionType="CreateLink" LinkType="Overlap"
               LinkSource="e1" LinkTarget="e2"/>
13     </Rule>
```

```
14
15      <Rule RuleID="TraceRule02" Description="Find␣URN-Actors␣and␣
            UML-Classes␣that␣share␣a␣similar␣name">
16          <Elements Type="Actor" Alias="e1"/>
17          <Elements Type="Class" Alias="e2"/>
18          <Conditions Type="And">
19              <BaseConditions Type="NotNull" Source="e1::id"/>
20              <BaseConditions Type="NotNull" Source="e2::umlID"/>
21              <BaseConditions Type="SimilarTo" Source="e1::name"
                    Target="e2::name"/>
22          </Conditions>
23          <Actions ActionType="CreateLink" LinkType="Overlap"
                LinkSource="e1" LinkTarget="e2"/>
24      </Rule>
25
26      <Rule RuleID="TraceRule03" Description="Find␣URN-Resources␣and␣
            UML-Classes␣that␣share␣a␣similar␣name">
27          <Elements Type="IntentionalElement" Alias="e1"/>
28          <Elements Type="Class" Alias="e2"/>
29          <Conditions Type="And">
30              <BaseConditions Type="NotNull" Source="e2::umlID"/>
31              <BaseConditions Type="Equals" Source="e1::type"
                    Value="Resource"/>
32              <BaseConditions Type="SimilarTo" Source="e1::name"
                    Target="e2::name"/>
33          </Conditions>
34          <Actions ActionType="CreateLink" LinkType="Overlap"
                LinkSource="e1" LinkTarget="e2"/>
35      </Rule>
```

B.2 Ontology for the Traceability Approach

This section shows the OWL/XML representation of the case study ontology. In the listing only a cutout of the ontology is presented.

Listing B.3: Cutout of the case study ontology in OWL/XML syntax

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <Ontology ontologyIRI="https://pix.theoinf.tu-ilmenau.de/EMFTrace/
       ontologies/RSIOntology.owl">
3      <Prefix name="rdf" IRI="http://www.w3.org/1999/02/22-rdf-syntax-ns#
           "/>
4      <Prefix name="rdfs" IRI="http://www.w3.org/2000/01/rdf-schema#"/>
5      <Prefix name="xsd" IRI="http://www.w3.org/2001/XMLSchema#"/>
6      <Prefix name="owl" IRI="http://www.w3.org/2002/07/owl#"/>
7      <Annotation>
8          <AnnotationProperty IRI="rdfs:comment"/>
9          <Literal xml:lang="">The ontology that describes important
               terms for the design of RSI.</Literal>
10     </Annotation>
11     <Declaration>
12         <Class IRI="#Adapter"/>
13     </Declaration>
14     <Declaration>
15         <Class IRI="#AdministrativeInterface"/>
16     </Declaration>
17     <Declaration>
18         <Class IRI="#Analyzability"/>
19     </Declaration>
20     <Declaration>
21         <Class IRI="#ArchitecturalConcern"/>
22     </Declaration>
23     <Declaration>
24         <Class IRI="#ArchitecturalMeans"/>
25     </Declaration>
26     <Declaration>
27         <Class IRI="#Attractiveness"/>
28     </Declaration>
29     <Declaration>
30         <Class IRI="#Audio"/>
31     </Declaration>
32     <Declaration>
```

```
33          <Class IRI="#Authenticity"/>
34      </Declaration>
35      <Declaration>
36          <Class IRI="#BasicTechnology"/>
37      </Declaration>
38      <Declaration>
39          <Class IRI="#Binary"/>
40      </Declaration>
41      <Declaration>
42          <Class IRI="#Blackboard"/>
43      </Declaration>
44      <Declaration>
45          <Class IRI="#Boost"/>
46      </Declaration>
47      <Declaration>
48          <Class IRI="#C++"/>
49      </Declaration>
50      <Declaration>
51          <Class IRI="#CSharp"/>
52      </Declaration>
53      <Declaration>
54          <Class IRI="#Camera"/>
55      </Declaration>
56      <Declaration>
57          <Class IRI="#Changeability"/>
58      </Declaration>
59      <Declaration>
60          <Class IRI="#Component"/>
61      </Declaration>
62      <Declaration>
63          <Class IRI="#Confidentiality"/>
64      </Declaration>
65      <Declaration>
66          <Class IRI="#Configuration"/>
67      </Declaration>
68      <Declaration>
69          <Class IRI="#DataFormat"/>
70      </Declaration>
```

List of Figures

2.1 The EMPRESS metamodel for NFR 20
2.2 Steps of ADD 25
2.3 The Siemens Four Views approach 27
2.4 The EMPRESS metamodel for design 28

3.1 The Staged Model of the software lifecycle 47

4.1 Overview of development phases and data flow 57

5.1 Structure of the Goal Solution Scheme 63
5.2 Overview of the integration interface 66
5.3 First transition of the Goal Solution Scheme for the case study 68
5.4 Second transition of the Goal Solution Scheme for the case study . . . 71
5.5 Third transition of the Goal Solution Scheme for the case study . . . 73
5.6 Graphical representation of the evolvability model 80
5.7 Pattern application in the case study example 85
5.8 Resulting impact of patterns on evolvability 88

6.1 Overview of a mobile interaction robot and its architecture 94
6.2 Backlog concept and iterations between the development phases . . . 95
6.3 Activities of the analysis phase 97
6.4 URN model for the quality goals of the case study with prioritization 101
6.5 Illustration of the rating procedure of the $100 approach 102
6.6 Overview of the activities of the synthesis phase 109
6.7 Activities of the architectural structuring 110
6.8 Cutout of GSS concerning usability and JGoodies 113
6.9 Structural view of the communication framework 115
6.10 The activities and steps of Quasar for detailing the architecture . . . 118

6.11 An exemplary category graph for the robot case study 119

7.1 Overview of the repository-based traceability concept 125
7.2 The defined traceability metamodel 128
7.3 The metamodel for the hypertext linking concept 130
7.4 A linkage example using the hypertext concept 131
7.5 Cutout of the ontology used in the case study. 142

8.1 Overview of the core concept of EMFTrace 145
8.2 Architectural components of EMFTrace and EMFfit 147
8.3 Split of a trace because of a broken chain of links 148
8.4 A screenshot of EMFTrace 151
8.5 The metamodel of EMFfit 153
8.6 The traceability rule metamodel as an Ecore model 155
8.7 EMFfit with the feature for mapping URN goals to factor categories . 158
8.8 EMFTrace's rule wizard 159
8.9 The export feature of EMFTrace 159

A.1 Structural view of the `ChannelManager` 200
A.2 Structural view of the communication framework 201
A.3 Behavioral view of the procedure to write into a `Channel` 202
A.4 Behavioral view of the communication between remote frameworks . . 203

List of Tables

2.1 The structure of an architectural scenario 19

4.1 Development phases with activities, artifacts, and modeling notations 58

5.1 Evolvability subcharacteristics 76
5.2 Architectural design principles 78
5.3 Mapping of subcharacteristics to principles 81
5.4 The set of architectural patterns for the impact evaluation 84
5.5 Rating scheme 84
5.6 Values for the patterns' impact on the properties 86
5.7 Impact values for subcharacteristics and evolvability 89

6.1 Quality goals of the case study 98
6.2 Results of the goal prioritization with the $100 approach 103
6.3 An example scenario for modifiability from the case study project . . 104
6.4 Product factors of the case study 106
6.5 Issue card example of the case study 107

7.1 The traceability link type clusters 134

9.1 Impact values and priorities for the ranking of the architectural styles 166
9.2 The ranking values of the architectural styles for the case study example 167
9.3 Amount of model elements from the evaluation projects 169
9.4 Results of the quality measurement 170
9.5 Comparison of precision and recall of different traceability approaches 170

A.1 Organizational factors of the case study 179
A.2 Technological factors of the case study 183
A.3 Product factors of the case study 188

A.4 Issue card Skills of the case study 194
A.5 Issue card Serialization of the case study 195
A.6 Issue card Hardware Changes of the case study 197
A.7 Issue card Software Changes of the case study 199

B.1 Rules for traceability link establishment 208

Listings

7.1 A rule example for the establishment of traceability links between UML components and OWL classes 138
7.2 A rule example for the establishment of a traceability link based on a hypertext link 139
B.1 The XML Schema Definition for the traceability rules 205
B.2 Cutout of the traceability rule catalog in XML representation 221
B.3 Cutout of the case study ontology in OWL/XML syntax 223

Bibliography

[ACC+02] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng.*, 28(10):970–983, Oct 2002.

[ACC+05] C. P. Ayala, C. Cares, J. P. Carvallo, G. Grau, M. Haya, G. Salazar, X. Franch, E. Mayol, and C. Quer. A comparative analysis of i*-based agent-oriented modeling languages. In *Proceedings of the International Workshop on Agent-Oriented Software Development Methodology (AOSDM'05) at the 7th International Conference on Software Engineering and Knowledge Engineering (SEKE'05)*, pages 43–50. KSI Press, 2005.

[AE03] D. Amyot and A. Eberlein. An evaluation of scenario notations and construction approaches for telecommunication systems development. *Telecommunication Systems*, 24(1):61–94, 2003.

[Amy99] D. Amyot. Use case maps quick tutorial. Technical report, SITE, University of Ottawa, September 1999. Version 1.0.

[Amy03] D. Amyot. Introduction to the user requirements notation: Learning by example. *Computer Networks*, 42(3):285–301, June 2003.

[And72] J. P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, U.S. Air Force, Electronic Systems Division, Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), L. G. Hanscom Field, Bedford, MA 01730 USA, Oct. 1972.

[And01] R. J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, 2001.

[ANGB+05] T. Al-Naeem, I. Gorton, M. Babar, F. Rabhi, and B. Benatallah. A quality-driven systematic approach for architecting distributed software applica-

tions. In *Proceedings 27th International Conference on Software Engineering, (ICSE 2005)*, pages 244–253. IEEE, May 2005.

[ANS] ANSI/ISA-95.00.05-2007 Enterprise-Control System Integration, Part 5: Business-to-Manufacturing Transactions.

[AR05] P. Arkley and S. Riddle. Overcoming the traceability benefit problem. In *Proceedings 13th IEEE International Conference on Requirements Engineering, 2005*, pages 385–389, 2005.

[Arn93] R. S. Arnold. *Software Reengineering.* IEEE Computer Society Press, Los Alamitos, CA, USA, 1993.

[ARNRSG06] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni. Model traceability. *IBM Systems Jounal*, 45(3):515–526, July 2006.

[AZ05] P. Avgeriou and U. Zdun. Architectural patterns revisited – a pattern language. In *Proceedings 10th European Conference on Pattern Languages of Programs (EuroPlop 2005), Irsee*, pages 1–39, 2005.

[BBK03] F. Bachmann, L. Bass, and M. Klein. Deriving architectural tactics: A step toward methodical architectural design. Technical Report CMU/SEI-2003-TR-004, CMU/SEI, Mar 2003.

[BBL76] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd International Conference on Software Engineering, (ICSE '76)*, pages 592–605. IEEE, 1976.

[BBN07] F. Bachmann, L. Bass, and R. Nord. Modifiability tactics. Technical Report CMU/SEI-2007-TR-002, CMU/SEI, September 2007.

[BBR09] R. Brcina, S. Bode, and M. Riebisch. Optimization process for maintaining evolvability during software evolution. In *Proceedings 16th International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2009)*, pages 196–205. IEEE, April 2009.

[BC00] C. Y. Baldwin and K. B. Clark. *Design Rules: The Power of Modularity.* MIT Press, Cambridge, MA, USA, 2000.

[BCE07] H. P. Breivold, I. Crnkovic, and P. Eriksson. Evaluating software evolvability. In T. Arts, editor, *Proceedings of the 7th Conference on Software*

Engineering Research and Practice in Sweden (SERPS'07), number 2007:02 in Software Engineering and Management, pages 96–103, Göteborg, Sweden, Oct 2007. IT University of Göteborg.

[BCE08] H. P. Breivold, I. Crnkovic, and P. Eriksson. Analyzing software evolvability. In *IEEE International Computer Software and Applications Conference (COMPSAC 2008)*, pages 327–330, Turku, Finland, July 2008. IEEE.

[BCK03] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Longman, 2 edition, 2003.

[BCLL08] H. P. Breivold, I. Crnkovic, R. Land, and S. Larsson. Using dependency model to support software architecture evolution. In *23rd IEEE/ACM International Conference on Automated Software Engineering - Workshops, 2008 (ASE Workshops 2008)*, pages 82–91. IEEE, Sept. 2008.

[BCR94] V. R. Basili, G. Caldiera, and H. D. Rombach. The goal question metric approach. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 528–532. John Wiley & Sons, 1994.

[BDP06] M. Broy, F. Deissenboeck, and M. Pizka. Demystifying maintainability. In *Proceedings of the 2006 International Workshop on Software Quality, WoSQ '06*, pages 21–26. ACM, 2006.

[BFKR09] S. Bode, A. Fischer, W. E. Kühnhauser, and M. Riebisch. Software architectural design meets security engineering. In *Proceedings 16th International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2009)*, pages 109–118. IEEE, April 2009.

[BKB02] L. J. Bass, M. Klein, and F. Bachmann. Quality attribute design primitives and the attribute driven design method. In *Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, volume 2290 of *LNCS*, pages 169–186. Springer, 2002.

[BKR07] S. Becker, H. Koziolek, and R. Reussner. Model-based performance prediction with the Palladio component models. In *Proceedings of the 6th International Workshop on Software and Performance (WOSP'07)*, pages 54–65. ACM, 2007.

[BLBvV04] P. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet. Architecture-level modifiability analysis (ALMA). *J. Syst. Softw.*, 69(1-2):129–147, 2004.

[BLC96] A. Bernaras, I. Laresgoiti, and J. Corera. Building and reusing ontologies for electrical network applications. In *Proceedings European Conference on Artificial Intelligence (ECAI'96)*, pages 298–302, Budapest, Hungary, 1996.

[BLR11] S. Bode, S. Lehnert, and M. Riebisch. Comprehensive model integration for dependency identification with EMFTrace. In *Joint Proceedings of the First International Workshop on Model-Driven Software Migration (MDSM 2011) and the Fifth International Workshop on Software Quality and Maintainability (SQM 2011), Workshops at the 15th European Conference on Software Maintenance and Reengineering (CSMR), Oldenburg, Germany, March 1, 2011*, pages 17–20. CEUR-WS.org, Mar 2011. online CEUR-WS.org/Vol-708/mdsm2011-bode-et-al-11-emftrace.pdf.

[BM07] A. W. Brown and J. A. McDermid. The art and science of software architecture. In F. Oquendo, editor, *Proceedings First European Conference on Software Architecture (ECSA 2007)*, volume 4758 of *LNCS*, pages 237–256. Springer, September 2007.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1 edition, July 1996.

[Bod08] S. Bode. Traceability und Entwurfsentscheidungen für Softwarearchitekturen mit der Quasar-Methode. Diploma thesis, Ilmenau University of Technology, Ilmenau, Germany, 2008.

[Bod09] S. Bode. On the role of evolvability for architectural design. In *Workshop Modellierung und Beherrschung der Komplexität, Informatik 2009*, LNI, pages 3256–3263. Koellen, 2009.

[Bon00] A. B. Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd International Workshop on Software and Performance (WOSP '00)*, pages 195–203. ACM, 2000.

[Boo93] G. Booch. *Object Oriented Analysis and Design. With Applications*. Addison-Wesley Longman, Amsterdam, October 1993.

[Bos00] J. Bosch. *Design and use of software architectures: Adopting and evolving a product-line approach*. ACM Press/Addison-Wesley, 2000.

[BR00] K. H. Bennett and V. T. Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering, ICSE '00*, pages 73–87. ACM, 2000.

[BR09] S. Bode and M. Riebisch. Tracing quality-related design decisions in a category-driven software architecture. In P. Liggesmeyer, G. Engels, J. Münch, J. Dörr, and N. Riegel, editors, *Software Engineering 2009*, volume P-143 of *LNI*, pages 87–98. Köllen, 2009.

[BR10] S. Bode and M. Riebisch. Impact evaluation for quality-oriented architectural decisions regarding evolvability. In M. Babar and I. Gorton, editors, *Proceedings 4th European Conference on Software Architecture, ECSA 2010*, volume 6285 of *LNCS*, pages 182–197. Springer, 2010.

[BR11] S. Bode and M. Riebisch. Tracing the implementation of non-functional requirements. In N. Milanovic, editor, *Non-Functional Properties in Service-Oriented Architecture: Requirements, Models and Methods*, chapter 1, pages 1–23. IGI Global, 2011.

[Brc11] R. Brcina. *Zielorientierte Erkennung und Behebung von Qualitätsdefiziten in Software-Systemen am Beispiel der Weiterentwicklungsfähigkeit.* PhD thesis, Ilmenau University of Technology, 2011. (to appear).

[Bro95] F. P. Brooks. *The Mythical Man-Month : Essays on Software Engineering.* Addison-Wesley, 1995.

[CdPL04] L. Cysneiros and J. do Prado Leite. Nonfunctional requirements: From elicitation to conceptual models. *IEEE Transactions on Software Engineering*, 30(5):328–350, May 2004.

[CdPL09] L. Chung and J. do Prado Leite. On Non-Functional Requirements in Software Engineering. In A. Borgida, V. Chaudhri, P. Giorgini, and E. Yu, editors, *Conceptual Modeling: Foundations and Applications*, volume 5600 of *Lecture Notes in Computer Science*, pages 363–379. Springer, 2009.

[CH06] J. Cleland-Huang. Just enough requirements traceability. In *30th Annual International Computer Software and Applications Conference, 2006 (COMPSAC '06)*, volume 1, pages 41–42, Sept 2006.

[Cha04] D. A. Chappell. *Enterprise Service Bus.* O'Reilly Media, USA, 1 edition, 2004.

[CHCC03] J. Cleland-Huang, C. Chang, and M. Christensen. Event-based traceability for managing evolutionary change. *IEEE Transactions on Software Engineering*, 29(9):796–810, Sept. 2003.

[CHSB+05] J. Cleland-Huang, R. Settimi, O. BenKhadra, E. Berezhanskaya, and S. Christina. Goal-centric traceability for managing non-functional requirements. In *Proceedings. 27th International Conference on Software Engineering, 2005 (ICSE '05)*, pages 362–371. IEEE, May 2005.

[CJH01] S. Cook, H. Ji, and R. Harrison. Dynamic and static views of software evolution. In *17th IEEE International Conference on Software Maintenance (ICSM'01)*, pages 592–601, Los Alamitos, CA, USA, Nov. 2001. IEEE.

[CNYM00] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-functional Requirements in Software Engineering*, volume 5 of *International Series in Software Engineering*. Kluwer, 2000.

[Coc00] A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley, Boston, MA, USA, 2000.

[CT94] W. B. Cavnar and J. M. Trenkle. N-gram-based text comparison. In *Proceedings of the 3rd Annual Symposium on Document Analysis and Information Retrieval (SDAIR-94)*, pages 161–175, 1994.

[CvdB06] S. Ciraci and P. van den Broek. Evolvability as a quality attribute of software architectures. In M. D. Laurence Duchien and K. Mens, editors, *Proceedings of the International ERCIM Workshop on Software Evolution 2006*, pages 29–31, LIFL et l'INRIA, Université des Sciences et Technologies de Lille, France, April 2006.

[Dep85] Department of Defense. Trusted Computer System Evaluation Criteria (TCSEC). Department of Defense Standard, DoD 5200.28-STD, Dec. 1985.

[DG02] L. Davis and R. F. Gamble. Identifying evolvability for integration. In *Proceedings of the First International Conference on COTS-Based Software Systems (ICCBSS '02)*, LNCS, pages 65–75. Springer, Jan. 2002.

[DKPF08] N. Drivalos, D. S. Kolovos, R. F. Paige, and K. J. Fernandes. Engineering a DSL for software traceability. In *First International Conference on Software Language Engineering (SLE 2008)*, volume 5452 of *LNCS*, pages 151–167. Springer, 2008.

[DMCS07] J.-C. Deprez, F. Monfils, M. Ciolkowski, and M. Soto. Defining software evolvability from a free/open-source software perspective. In *Proceedings of the Third International IEEE Workshop on Software Evolvability*, pages 29–35. IEEE, Oct. 2007.

[dPLHDK00] J. C. S. do Prado Leite, G. D. S. Hadad, J. H. Doorn, and G. N. Kaplan. A scenario construction process. *Requirements Engineering*, 5:38–61, 2000.

[DRW06] L. Duboc, D. S. Rosenblum, and T. Wicks. A framework for modelling and analysis of software systems scalability. In *Proceedings of the 28th International Conference on Software engineering (ICSE '06)*, pages 949–952. ACM, 2006.

[DRW07] L. Duboc, D. Rosenblum, and T. Wicks. A framework for characterization and analysis of software system scalability. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07)*, pages 375–384. ACM, 2007.

[dSdSdC03] G. M. C. de Sousa, I. G. L. da Silva, and J. B. de Castro. Adapting the NFR framework to aspect-oriented requirements engineering. In *XVII Brazilian Symposium on Software Engineering*, Manaus, Brazil, October 2003.

[DWP+07] F. Deissenboeck, S. Wagner, M. Pizka, S. Teuchert, and J.-F. Girard. An activity-based quality model for maintainability. In *Proceedings of the 23rd International Conference on Software Maintenance (ICSM 2007)*, pages 184–193. IEEE, 2007.

[EAG06] A. Espinoza, P. P. Alarcón, and J. Garbajosa. Analyzing and systematizing current traceability schemas. In *Proceedings of the 30th Annual IEEE/-NASA Software Engineering Workshop SEW-30 (SEW'06)*, pages 21–32. IEEE, April 2006.

[Egy01] A. Egyed. A scenario-driven approach to traceability. In *Proceedings 23rd International Conference on Software Engineering, (ICSE'01)*, pages 123–132. IEEE, May 2001.

[EM06] A. Eden and T. Mens. Measuring software flexibility. *IEE Proceedings - Software*, 153(3):113–125, June 2006.

[Erl07] T. Erl. *SOA: Principles of Service Design.* Prentice Hall Press, Upper Saddle River, NJ, USA, July 2007.

[Erl08] T. Erl. *SOA Design Patterns.* Prentice Hall International, 2008.

[Eur07] European Community for Software & Software Services (ECSS). 3S Green Paper on Software and Service Architecture, Infrastructures and Engineering – a working document for a future EU Action Paper on the area, Okt. 2007. Version 1.2.

[FK08] A. Fischer and W. E. Kühnhauser. Integration von Sicherheitsmodellen in Web Services. In *Proceedings D-A-CH Security*, 2008.

[FL93] C. J. Fidge and A. M. Lister. The challenges of non-functional computing requirements. In *Seventh Australian Software Engineering Conference (ASWEC'93)*, pages 77–84, Sydney, September 1993.

[Fow02] M. Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley Longman, 2002.

[FZS03] G. A. A. C. Filho, A. Zisman, and G. Spanoudakis. Traceability approach for i* and UML models. In *Proceedings of 2nd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS'03)*, 2003.

[Gas88] M. Gasser. *Building a secure computer system.* Van Nostrand Reinhold Co., New York, NY, USA, 1988.

[GEM06] M. Galster, A. Eberlein, and M. Moussavi. Transition from requirements to architecture: A review and future perspective. In *Seventh ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2006 (SNPD 2006)*, pages 9–16. IEEE, June 2006.

[GEM10] M. Galster, A. Eberlein, and M. Moussavi. Systematic selection of software architecture styles. *IET Software*, 4(5):349–360, Oct. 2010.

[GF94] O. C. Z. Gotel and A. C. W. Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of the First International Conference on Requirements Engineering, Colorado Springs, CO, USA*, pages 94–101. IEEE, April 1994.

[GF95] M. Grüninger and M. Fox. Methodology for the design and evaluation of ontologies. In *Proceedings Workshop on Basic Ontological Issues in Knowledge Sharing*, Montreal, 1995.

[GF07] G. Grau and X. Franch. A goal-oriented approach for the generation and evaluation of alternative architectures. In *Proceedings of the First European Conference on Software Architecture (ECSA'07)*, volume 4758 of *LNCS*, pages 139–155. Springer, 2007.

[GG07] I. Galvão and A. Goknil. Survey of traceability approaches in model-driven engineering. In *11th IEEE International Enterprise Distributed Object Computing Conference, 2007 (EDOC 2007)*, pages 313–313, Oct. 2007.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Softwaresystemen.* Addison-Wesley Professional, 1994.

[GHR07] S. Giesecke, W. Hasselbring, and M. Riebisch. Classifying architectural constraints as a basis for software quality assessment. *Advanced Engineering Informatics*, 21(2):169–179, 2007.

[GKW07] T. Graham, R. Kazman, and C. Walmsley. Agility and experimentation: Practical techniques for resolving architectural tradeoffs. In *29th International Conference on Software Engineering (ICSE 2007)*, pages 519–528. IEEE, May 2007.

[GM82] J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings IEEE Symposium on Security and Privacy*, pages 11–20, Apr. 1982.

[GRH08] S. Giesecke, M. Rohr, and W. Hasselbring. Architectural styles for early goal-driven middleware selection. In *Postproceedings of the 13th European Conference on Pattern Languages of Programs (EuroPLoP 08)*, Irsee, Germany, July 2008.

[GRL08] Goal-oriented Requirements Language (GRL). http://www.cs.toronto.edu/km/GRL/, 2008.

[GY01] D. Gross and E. S. K. Yu. From non-functional requirements to design through patterns. *Requirements Engineering*, 6(1):18–36, February 2001.

[HA07a] N. Harrison and P. Avgeriou. Pattern-driven architectural partitioning: Balancing functional and non-functional requirements. In *Second International Conference on Digital Telecommunications, 2007 (ICDT '07)*, pages 21–26. IEEE, July 2007.

[HA07b] N. B. Harrison and P. Avgeriou. Leveraging architecture patterns to satisfy quality attributes. In *Proceedings First European Conference on Software Architecture, ECSA 2007*, volume 4758/2007, pages 263–270. Springer, 2007.

[HDO03] J. H. Hayes, A. Dekhtyar, and J. Osborne. Improving requirements tracing via information retrieval. In *Proceedings of the 11th IEEE International Conference on Requirements Engineering (RE '03)*, pages 138–147. IEEE, 2003.

[Heu04] Heumesser, N. (Ed.). Framework for requirements. Technical report, EMPRESS Project at ITEA, April 2004.

[Hil90] M. D. Hill. What is scalability? *SIGARCH Computer Architecture News*, 18(4):18–21, 1990.

[HKN+07] C. Hofmeister, P. Kruchten, R. L. Nord, H. Obbink, A. Ran, and P. America. A general model of software architecture design derived from five industrial approaches. *Journal of Systems and Software*, 80(1):106–126, January 2007.

[HNS00] C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Addison-Wesley Longman, Boston, MA, USA, 2000.

[HNS05] C. Hofmeister, R. Nord, and D. Soni. Global analysis: moving from software requirements specification to structural views of the software architecture. *IEE Proceedings - Software*, 152(4):187–197, August 2005.

[Hor06] J. Horkoff. Using i* models for evaluation. Master's thesis, University of Toronto, 2006.

[IK06] I. Ivkovic and K. Kontogiannis. Towards automated establishment of model dependencies using formal concept analysis. *International Journal of Software Engineering and Knowledge Engineering*, 16(4):499–522, Aug 2006.

[Ins84] Institute of Electrical and Electronics Engineers. IEEE Guide to Software Requirements Specification, 1984.

[Ins90] Institute of Electrical and Electronics Engineers. IEEE Standard Glossary of Software Engineering Terminology, 1990.

[Ins98] Institute of Electrical and Electronics Engineers. IEEE Standard for Software Maintenance. IEEE Std 1219-1998, Oct 1998.

[Ins00] Institute of Electrical and Electronics Engineers. IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE Std 1471-2000 (also ISO/IEC 42010:2007), Sept 2000.

[Int01] International Standardization Organisation. ISO/IEC 9126-1 International Standard. Software Engineering – Product quality – Part 1: Quality models, June 2001.

[Int05] International Standardization Organisation. ISO/IEC 27001:2005 Information technology – Security techniques – Information security management systems – Requirements, June 2005.

[Int06] International Standardization Organisation. Software Engineering—Software Life Cycle Processes—Maintenance. ISO/IEC 14764:2006, IEEE Std 14764-2006, 2006.

[Int11] International Standardization Organisation. ISO/IEC 25010:2011 Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models, 2011.

[ITU08] ITU-T. Recommendation ITU-T Z.151 User requirements notation (URN) – Language definition, Nov 2008.

[JGo10] JGoodies. http://www.jgoodies.com/, 2010.

[JRH+03] M. Jeckle, C. Rupp, J. Hahn, B. Zengler, and S. Queins. *UML 2 glasklar*. Hanser Fachbuchverlag, 2003.

[JZ09] W. Jirapanthong and A. Zisman. Xtraque: traceability for product line systems. *Software and Systems Modeling*, 8(1):117–144, 2009.

[KB03] R. Kolb and J. Bayer. Pattern-based architecture analysis and design of embedded software product lines. Technical report, Fraunhofer IESE, December 2003.

[KC99] R. Kazman and S. J. Carrière. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engineering*, 6(2):107–138, 1999.

[KCD09] P. Kruchten, R. Capilla, and J. Dueas. The decision view's role in software architecture practice. *IEEE Software*, 26(2):36–42, 2009.

[KH10] M. Koegel and J. Helming. EMFStore: a model repository for EMF models. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering – Volume 2 (ICSE '10)*, pages 307–308. ACM, 2010.

[KKC00] R. Kazman, M. Klein, and P. Clements. ATAM: Method for Architecture Evaluation. Technical Report CMU/SEI-2000-TR-004, CMU/SEI, August 2000.

[KPP06] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. On-demand merging of traceability links with models. In *Proceedings ECMDA Traceability Workshop (ECMDA-TW) 2006*, pages 6–14. Sintef, Trondheim, 2006.

[Kru95] P. B. Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12(6):42–50, 1995.

[LABW92] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: theory and practice. *ACM Transactions on Computer Systems (TOCS)*, 10(4):265–310, 1992.

[Leh80] M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, Sept. 1980.

[Leh10] S. Lehnert. Softwarearchitektur-Entwurf und Realisierung eines Repositories für Modell-übergreifende Traceability. Diploma thesis, Ilmenau University of Technology, Ilmenau, Germany, November 2010.

[Let02] P. Letelier. A framework for requirements traceability in UML-based projects. In *Proceedings 1st Int. Workshop on Traceability in Emerging Forms of SE (TEFSE'02)*, pages 32–41, Edinburgh, UK, 2002.

[LFOT07] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Trans. Softw. Eng. Methodol.*, 16(4, Article 13), September 2007.

[LG05] A. E. Limón and J. Garbajosa. The need for a unifying traceability scheme. In *Proceedings ECMDA Traceability Workshop (ECMDA-WS) 2005*, pages 47–56, Sintef, Trondheim, 2005.

[LGPSS99] M. López, A. Gómez-Pérez, J. Sierra, and A. Sierra. Building a chemical ontology using methontology and the ontology design environment. *IEEE Intelligent Systems and their Applications*, 14(1):37–46, Jan/Feb 1999.

[LMvV09] P. Lago, H. Muccini, and H. van Vliet. A scoped approach to traceability management. *Journal of Systems and Software*, 82(1):168–182, 2009. Special Issue: Software Performance – Modeling and Analysis.

[LR06] M. Lehman and J. C. F. Ramil. Software evolution. In Madhavji et al. [MFRP06], chapter 1, pages 7–40.

[LS96] M. Lindvall and K. Sandahl. Practical implications of traceability. *Softw. Pract. Exper.*, 26:1161–1180, October 1996.

[LY01] L. Liu and E. Yu. From requirements to architectural design – using goals and scenarios. In *From Software Requirements to Architectures Workshop (STRAW 2001)*, Toronto, Canada, May 2001.

[Mäd09] P. Mäder. *Rule-Based Maintenance of Post-Requirements Traceability.* PhD thesis, TU Ilmenau, Ilmenau, Germany, 2009.

[McK05] E. McKean. *The New Oxford American Dictionary.* Oxford University Press, 2 edition, May 2005.

[MD08] T. Mens and S. Demeyer, editors. *Software Evolution.* Springer, 2008.

[MFRP06] N. H. Madhavji, J. Fernandez-Ramil, and D. E. Perry, editors. *Software Evolution and Feedback: Theory and Practice.* John Wiley & Sons, 2006.

[MJS+00] H. A. Müller, J. H. Jahnke, D. B. Smith, M.-A. D. Storey, S. R. Tilley, and K. Wong. Reverse engineering: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering, ICSE '00*, pages 47–60, June 2000.

[MM98] T. Mens and K. Mens. Assessing the evolvability of software architectures. In *Workshop on Object-Oriented Technology (ECOOP '98)*, volume 1543/1998 of *LNCS*, pages 54–55. Springer, 1998.

[MM03] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *International Conference on Software Engineering (ICSE'03)*, pages 125–135. IEEE, 2003.

[MMMN03] J. I. Maletic, E. V. Munson, A. Marcus, and T. N. Nguyen. Using a hypertext model for traceability link conformance analysis. In *Proceedings of the 2nd International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE2003)*, pages 47–54, 2003.

[MN03] M. Matinlassi and E. Niemelä. The impact of maintainability on component-based software systems. In *Proceedings 29th Euromicro Conference, 2003*, pages 25–32. IEEE, September 2003.

[Mor06] G. Morgan. Design for flexibility. online, Oct 2006. Last retrieved: 29 Oct 2010.

[MPR07] P. Mäder, I. Philippow, and M. Riebisch. Customizing traceability links for the unified process. In *Proceedings Third International Conference on the Quality of Software-Architectures (QOSA2007)*, volume 4880 of *LNCS*, Medford MA, USA, July 2007. Springer.

[MSW+05] C. Martin, A. Scheidig, T. Wilhelm, C. Schroeter, H.-J. Boehme, and H.-M. Gross. A new control architecture for mobile interaction robots. In *Proceedings of the 2nd European Conference on Mobile Robots, (ECMR 2005)*, pages 224–229. Stampalibri, 2005.

[MVN06] D. Manolescu, M. Voelter, and J. Noble. *Pattern Languages of Program Design 5*. Addison-Wesley Professional, May 2006.

[MWD+05] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. Challenges in software evolution. In *Eighth International Workshop on Principles of Software Evolution*, pages 13–22, September 2005.

[NE00] B. Nuseibeh and S. Easterbrook. Requirements engineering: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering, ICSE '00*, pages 35–46. ACM, 2000.

[Neu11] S. Neugebauer. Softwarearchitektur-Entwurf für Anwendungen auf Mehrkern-Prozessorarchitekturen. diploma thesis, Ilmenau University of Technology, 2011. (to appear).

[NM01] N. F. Noy and D. L. McGuinness. Ontology development 101: A guide to creating your first ontology. Technical Report KSL-01-05, Stanford Knowledge Systems Laboratory, Stanford University, Stanford, CA, 94305, 2001.

[NNG97] K. Nelson, H. Nelson, and M. Ghods. Technology flexibility: conceptualization, validation, and measurement. In *Proceedings of the Thirteeth Hawaii International Conference on System Sciences*, volume 3, pages 76–87. IEEE, January 1997.

[OG02] T. Olsson and J. Grundy. Supporting traceability and inconsistency management between software artifacts. In *Proceedings of the IASTED International Conference on Software Engineering and Applications*, 2002.

[Par72] D. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[Par94] D. Parnas. Software aging. In *Proceedings 16th International Conference on Software Engineering, (ICSE '94)*, pages 279–287. IEEE, May 1994.

[PB04] M. Pizka and A. Bauer. A brief top-down and bottom-up philosophy on software evolution. In *Proceedings 7th International Workshop on Principles of Software Evolution, (IWPSE'04)*, pages 131–136. IEEE, 2004.

[PBG04] T. Posch, K. Birken, and M. Gerdom. *Basiswissen Softwarearchitektur: Verstehen, entwerfen, wiederverwenden.* dpunkt.verlag, 1 edition, 2004.

[PG96] F. A. C. Pinheiro and J. Goguen. An object-oriented tool for tracing requirements. *IEEE Software*, 13(2):52–66, March 1996.

[Pin04] F. A. C. Pinheiro. Requirements traceability. In J. Leite and J. Doorn, editors, *Perspectives on Software Requirements*, chapter 5, pages 91–113. Kluwer, Norwell, MA, USA, 2004.

[Poh96] K. Pohl. PRO-ART: Enabling requirements Pre-traceability. In *Proceedings of the Second International Conference on Requirements Engineering, ICRE*, pages 76–84. IEEE, April 1996.

[PvKD+03] B. Paech, A. von Knethen, J. Doerr, J. Bayer, D. Kerkow, R. Kolb, A. Trendowicz, and T. Punter. An experience-based approach for integrating architecture and requirements engineering. In *Second International SofTware Requirements to Architecture Workshop, (STRAW '03)*, pages 142–149, 2003.

[RB00] V. Rajlich and K. Bennett. A staged model for the software life cycle. *IEEE Computer*, 33(7):66–71, July 2000.

[RB09] M. Riebisch and S. Bode. Software-evolvability. *Informatik-Spektrum*, 32(4):339–343, August 2009.

[RBFL11] M. Riebisch, S. Bode, Q.-U.-A. Farooq, and S. Lehnert. Towards comprehensive modelling by inter-model links using an integrating repository. In *Proceedings 8th IEEE Workshop on Model-Based Development for Computer-Based Systems – Covering Domain and Design Knowledge in Models*, 2011. (accepted for publication).

[RE93] B. Ramesh and M. Edwards. Issues in the development of a requirements traceability model. In *Proceedings of IEEE International Symposium on Requirements Engineering, 1993*, pages 256–259, January 1993.

[RJ01] B. Ramesh and M. Jarke. Toward reference models for requirements traceability. *IEEE Trans. Softw. Eng.*, 27(1):58–93, 2001.

[RLL98] D. Rowe, J. Leaney, and D. Lowe. Defining systems architecture evolvability - a taxonomy of change. In *Proceedings of the 11th International Conference on the Engineering of Computer Based Systems (ECBS'98)*, pages 45–52, Jerusalem, Israel, 1998. IEEE.

[RPB11] M. Riebisch, A. Pacholik, and S. Bode. Towards optimization of design decisions for embedded systems by exploiting dependency relationships. In *Proceedings Dagstuhl-Workshop Modellbasierte Entwicklung eingebetteter Systeme IV (MBEES)*, pages 11–20. fortiss GmbH, February 2011.

[RPV03] W. N. Robinson, S. D. Pawlowski, and V. Volkov. Requirements interaction management. *ACM Computing Surveys*, 35(2):132–190, June 2003.

[RR99] S. Robertson and J. Robertson. *Mastering the Requirements Process*. Addison-Wesley, 1999.

[RSN09] M. Riaz, M. Sulayman, and H. Naqvi. Architectural decay during continuous software evolution and impact of 'design for change' on software architectures. In D. Slezak, T.-h. Kim, A. Kiumi, T. Jiang, J. Verner, and S. Abrahao, editors, *Advances in Software Engineering*, volume 59 of *Communications in Computer and Information Science*, pages 119–126. Springer, 2009.

[Rum09] M. Rumpf. Evolutionäre Softwarearchitekturentwicklung mit der Methode Attribute Driven Design (ADD). Diploma thesis, Ilmenau University of Technology, Oct 2009.

[RW05] N. Rozanski and E. Woods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Professional, 2005.

[SB01] K. Schwaber and M. Beedle. *Agile Software Development with Scrum*. Prentice Hall, Upper Saddle River, NJ, USA, 1st edition, 2001.

[SC03] N. Subramanian and L. Chung. Process-oriented metrics for software architecture evolvability. In *Proceedings Sixth International Workshop on Principles of Software Evolution*, pages 65–70, September 2003.

[SdGZ03] G. Spanoudakis, A. d'Avila Garces, and A. Zisman. Revising rules to capture requirements traceability relations: A machine learning approach. In *Proceedings of the 15th International Conference in Software Engineering and Knowledge Engineering (SEKE 2003)*, pages 570–577. Knowledge Systems Institute, Skokie, 2003.

[SEW09] H. Schwarz, J. Ebert, and A. Winter. Graph-based traceability: a comprehensive approach. *Software and Systems Modeling*, 9(4):473–492, 2009.

[Sie04] J. Siedersleben. *Moderne Software-Architektur: Umsichtig planen, robust bauen mit Quasar*. dpunkt.verlag, Heidelberg, Germany, 2004.

[SM98] A. G. Sutcliffe and S. Minocha. Scenario-based analysis of non-functional requirements. In *REFSQ'98*, pages 219–234. Presses universitaeires de Namur, 1998.

[SPKR97] B. Swartout, R. Patil, K. Knight, and T. Russ. Toward distributed use of large-scale ontologies. In *AAAI Symposium on Ontological Engineering*, Stanford, 1997.

[SSSS01] S. Staab, R. Studer, H.-P. Schnurr, and Y. Sure. Knowledge processes and ontologies. *IEEE Intelligent Systems*, 16(1):26–34, Jan/Feb 2001.

[Sto10] R. Stollberg. Klassifikation von Architekturstilen und -mustern hinsichtlich qualitativer Ziele für den Softwarearchitekturentwurf. Bachelor thesis, Ilmenau University of Technology, Ilmenau, Germany, 2010.

[SvGB05] M. Svahnberg, J. van Gurp, and J. Bosch. A taxonomy of variability realization techniques. *Software: Practice and Experience*, 35(8):705–754, 2005.

[SZ05] G. Spanoudakis and A. Zisman. Software traceability: A roadmap. In C. S. K., editor, *Handbook of Software Engineering and Knowledge Engineering*, volume III, pages 395–428. World Scientific Publishing Co., River Edge, NJ, 2005.

[SZPMK04] G. Spanoudakis, A. Zisman, E. Perez-Minana, and P. Krause. Rule-based generation of requirements traceability relations. *Journal of Systems and Software*, 72(2):105–127, 2004.

[TA05] J. Tyree and A. Akerman. Architecture Decisions: Demystifying Architecture. *IEEE Software*, 22(2):19–27, 2005.

[TAJ+10] A. Tang, P. Avgeriou, A. Jansen, R. Capilla, and M. A. Babar. A comparative study of architecture knowledge management tools. *Journal of Systems and Software*, 83(3):352–370, 2010.

[TKM03] L. Tahvildari, K. Kontogiannis, and J. Mylopoulos. Quality-driven software re-engineering. *Journal of Systems and Software*, 66(3):225–239, June 2003.

[UK95] M. Uschold and M. King. Towards a methodology for building ontologies. In *Proceedings Workshop on Basic Ontological Issues in Knowledge Sharing (IJCAI95)*, Montreal, 1995.

[Ver07] Verein Deutscher Ingenieure. Fertigungsmanagementsysteme - Manufacturing Execution Systems (MES). VDI 5600, December 2007.

[vKP02] A. von Knethen and B. Paech. A survey on tracing approaches in practice and research. IESE-Report 095.01/E, Fraunhofer Institut Experimentelle Software Engineering, Kaiserslautern, Germany, 2002.

[vL03] A. van Lamsweerde. From system goals to software architectures. In B. M. and I. P., editors, *Formal Methods for Software Architectures*, volume 2804/2003 of *LNCS*, pages 25–43. Springer, 2003.

[vL09] A. van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, March 2009.

[Wag10] P. Wagner. Werkzeugunterstützung für die Analyse beim Softwarearchitekturentwurf. Bachelor thesis, Ilmenau University of Technology, Ilmenau, Germany, December 2010.

[WBB+06] R. Wojcik, F. Bachmann, L. Bass, P. Clements, P. Merson, R. Nord, and B. Wood. Attribute-Driven Design (ADD), Version 2.0. Technical Report CMU/SEI-2006-TR-023, CMU/SEI, November 2006.

[Woh08] S. Wohlfarth. *A Process of Rational Decision-Making for Architectural Decisions (in German: Entwicklung eines rationalen Entscheidungsprozesses für Architekturentscheidungen).* PhD thesis, TU Ilmenau, 2008.

[Woo07] W. G. Wood. A Practical Example of Applying Attribute-Driven Design (ADD), Version 2.0. Technical Report CMU/SEI-2007-TR-005, CMU/SEI, February 2007.

[WvP10] S. Winkler and J. von Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software and Systems Modeling*, 9(4):529–565, 2010.

[YB97] J. W. Yoder and J. Barcalow. Architectural patterns for enabling application security. In *Proceedings 4th Conf. on Patterns Languages of Programs (PLoP '97)*, 1997.

[YNGB+09] Y. Yu, N. Niu, B. González-Baixauli, J. Mylopoulos, S. Easterbrook, and J. C. S. do Prado Leite. Requirements engineering and aspects. In W. Aalst, J. Mylopoulos, N. M. Sadeh, M. J. Shaw, C. Szyperski, K. Lyytinen, P. Loucopoulos, J. Mylopoulos, and B. Robinson, editors, *Design Requirements Engineering: A Ten-Year Perspective*, volume 14 of *Lecture Notes in Business Information Processing*, pages 432–452. Springer, 2009.

[Yu95] E. S.-K. Yu. *Modelling Strategic Relationships for Process Reengineering.* PhD thesis, University of Toronto, Toronto, Ontario, Canada, 1995.

[ZEF00] A. Zisman, W. Emmerich, and A. Finkelstein. Using XML to build consistency rules for distributed specifications. In *Proceedings of the 10th International Workshop on Software Specification and Design, (IWSSD '00)*, pages 141–150. IEEE, 2000.

[ZZ02] D. Zeng and J. Zhao. Achieving software flexibility via intelligent workflow techniques. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences, (HICSS)*, pages 606–615. IEEE, Jan 2002.

Theses

- There is a gap between requirements engineering and architectural design approaches, which hampers the development of high quality software systems.
- The Goal Solution Scheme helps to overcome this gap by an explicit mapping of quality goals to architectural solution instruments.
- The goal-oriented design method of this thesis guides the developer in fulfilling the quality goals for a software system in an advanced and systematical way.
- The selection of known architectural solution instruments, such as patterns, should be supported with a quantitative evaluation regarding the influence on quality goals, which can be provided by applying the Goal Solution Scheme.
- An explicit expression of dependencies between all artifacts of the development process is essential for the support of software evolution and maintenance by enabling impact analysis of frequent software changes.
- Evolvability is a complex quality goal, which for its realization has to be refined into subgoals that all have to be treated appropriately—traceability is one of these subgoals.
- There is still no comprehensive approach that deals with all aspects of traceability for explicit dependency representation and spans the whole software development life-cycle.
- A rule-based traceability concept is a promising approach for the model-spanning establishment of traceability links with high quality and defined semantics.
- The rule-based traceability concept can and should be enhanced by information retrieval techniques to determine the similarity of model elements.
- An ontology can be a helpful means to establish traceability links for inter model dependencies.

Erklärung

Ich versichere, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Weitere Personen waren an der inhaltlich-materiellen Erstellung der vorliegenden Arbeit nicht beteiligt. Insbesondere habe ich hierfür nicht die entgeltliche Hilfe von Vermittlungs- bzw. Beratungsdiensten (Promotionsberater oder anderer Personen) in Anspruch genommen. Niemand hat von mir unmittelbar oder mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalte der vorgelegten Dissertation stehen.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer Prüfungsbehörde vorgelegt.

Ich bin darauf hingewiesen worden, dass die Unrichtigkeit der vorstehenden Erklärung als Täuschungsversuch angesehen wird und den erfolglosen Abbruch des Promotionsverfahrens zu Folge hat.

Ilmenau,