A Framework to Improve the Architecture Quality of Software-Intensive Systems

Steffen Thiel

A Framework to Improve the Architecture Quality of Software-Intensive Systems

Eine vom Fachbereich Wirtschaftswissenschaften
der Universität Duisburg-Essen
zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften (Dr. rer. nat.)

genehmigte Dissertation von

Dipl.-Inform. Steffen Thiel

aus Langen

Tag der mündlichen Prüfung:	28. Oktober 2005	
Promotionskommission:	Vorsitzender:	Prof. Dr. K. Echtle
	Erstgutachter:	Prof. Dr. K. Pohl
	Zweitgutachter:	Prof. Dr. M. Goedicke

Whoever thinks a faultless piece to see,
thinks what never was, nor is, nor ever shall be.

ALEXANDER POPE, AN ESSAY ON CRITICISM

Table of Contents

Abstract .... **xix**

Acknowledgements .... **xxi**

1 Introduction **1**

1.1 Motivation 1

1.2 Problem Context 3

1.3 Problems of Scope and Focal Research Fields 7

1.3.1 Research Gaps 9

1.3.2 Topics Out of Scope 11

1.4 Hypothesis 12

1.5 Approach 13

1.6 Research Contribution 13

1.7 Organization of the Thesis 15

2 Related Work in Architecture Development **17**

2.1 Software Architecture and Architecture Development 18

2.1.1 Basic Principles 18

2.1.2 Definitions of Software Architecture 19

2.1.3 Definitions of Architecture Development 20

2.2 Designing the Architecture 21

2.2.1 Inputs to Architecture Design 23

2.2.2 Outputs from Architecture Design 32

2.2.3 Design Support Concepts 38

2.2.4 Architecture Design Methods 42

2.2.5 Comparison of Design Methods 48

2.3 Evaluating the Architecture 52

2.3.1 Inputs to Architecture Evaluation 53

2.3.2 Outputs from Architecture Evaluation 53

2.3.3 Evaluation Support Concepts 55

2.3.4 Architecture Evaluation Methods 59

2.3.5 Comparison of Evaluation Methods 67

2.4 Summary 70

3 Bridging the Research Gaps with QUADRAD 71
3.1 Key Concepts to Address Research Gaps 72
3.1.1 Architectural Decisions Have Impact on the Overall System Design 73
3.1.2 Architectural Decisions are Based on Architectural Drivers 74
3.1.3 Architectural Strategies and Mechanisms Support Decision Making 74
3.1.4 Architectural Decisions Influence Each Other 75
3.2 Metamodel and Approach 77
3.3 Refining and Packaging the Approach 79
3.4 An Overview of the QUADRAD Framework 81
3.5 Summary 86
4 The QUADRAD Preparation Workflow 87
4.1 Workflow Overview 88
4.2 Identify Architectural Drivers 90
4.2.1 Calculate Priorities 91
4.2.2 Determine High-Priority Requirements 97
4.3 Determine Architectural View Types 101
4.4 Preparation Activities and Research Gaps 104
4.5 Summary 105
5 The QUADRAD Modeling Workflow 107
5.1 Workflow Overview 108
5.2 Define Architectural Strategies and Mechanisms 110
5.2.1 Determine Architectural Strategies 112
5.2.2 Determine Architectural Mechanisms 118
5.2.3 Determine Suitability 120
5.3 Model the Architectural Infrastructure 127
5.3.1 Derive Architectural Decisions 129
5.3.2 Apply Decisions and Adapt Architecture 135
5.4 Refine the Architecture 140
5.5 Modeling Activities and Research Gaps 143
5.6 Summary 144
6 The QUADRAD Evaluation Workflow 145
6.1 Workflow Overview 146
6.2 Define Evaluation Goals 149
6.3 Elicit and Prioritize Evaluation Scenarios 150
6.3.1 Brainstorm and Organize Evaluation Scenarios 152
6.3.2 Calculate Scenario Priorities 156
6.4 Map Scenarios and Identify Decisions 158

6.5 Analyze Decisions and Summarize Findings 163
6.5.1 Analyze Scenario Maps 165
6.5.2 Refine Scenario Analysis 169
6.5.3 Identify Solution Approaches 176
6.5.4 Cluster Findings 177
6.6 Evaluation Activities and Research Gaps 180
6.7 Summary 182

7 QUADRAD Tool Support 183
7.1 Architecture Exploration Tool 183
7.2 AET Requirements 184
7.3 AET Software Architecture and Data Model 186
7.4 Applying AET During Architecture Evaluation 189
7.4.1 Capturing Requirements and Quality Attributes 190
7.4.2 Scenarios and Prioritization 191
7.4.3 Scenario Analysis 192
7.4.4 Clustering Findings 193
7.4.5 Report Generation 194
7.5 Summary 195

8 Validation of the QUADRAD Framework 197
8.1 Validation Goals and Settings 197
8.2 Validation Framework 199
8.2.1 Probability, Impact, and Risk Ranking 202
8.2.2 Severity of Risks 204
8.2.3 Architectural Quality 204
8.2.4 Architectural Quality Improvement 205
8.3 Validation Approach 207
8.3.1 Step 1: Evaluating the Original Architectures 207
8.3.2 Step 2: Creating the Revised Architectures 207
8.3.3 Step 3: Evaluating the Revised Architectures 207
8.3.4 Step 4: Comparison 208
8.4 Validation in Project A 209
8.4.1 Step 1: Evaluating the Original Architecture A 209
8.4.2 Step 2: Creating a Revised Architecture A* 217
8.4.3 Step 3: Evaluating the Revised Architecture A* 220
8.4.4 Step 4: Comparison 223
8.5 Validation in Project B 224
8.5.1 Step 1: Evaluating the Original Architecture B 224
8.5.2 Step 2: Creating a Revised Architecture B* 229
8.5.3 Step 3: Evaluating the Revised Architecture B* 233

8.5.4 Step 4: Comparison ... 235
8.6 Validation in Project C ... 236
8.6.1 Step 1: Evaluating the Original Architecture C ... 236
8.6.2 Step 2: Creating a Revised Architecture C* ... 243
8.6.3 Step 3: Evaluating the Revised Architecture C* ... 250
8.6.4 Step 4: Comparison ... 252
8.7 Validation Results ... 254
8.8 Lessons Learned ... 258
8.9 Summary ... 260

9 Summary and Outlook ... 261
9.1 The Problem Revisited ... 261
9.2 The Research Contribution Revisited ... 262
9.3 The Validation Revisited ... 263
9.4 Outlook ... 263

10 Appendix: Architectural Patterns ... 267

Acronyms ... 275

Glossary ... 277

Bibliography ... 287

List of Figures

Figure 1-1: Automotive Electronics Systems 1
Figure 1-2: Increasing Importance of Software in Automobile Electronics [Mercer 2002] 2
Figure 1-3: Mismatch Between Planned and Implemented System 3
Figure 1-4: Performance of Software System Development Projects [Davis 1990] 4
Figure 1-5: Source of Problems in Software System Development (Summary) 6
Figure 1-6: Problems of Scope of this Work 8
Figure 1-7: Focal Research Fields of this Work 10
Figure 2-1: Overview of Architecture Design 23
Figure 2-2: System Requirements and Architectural Drivers 31
Figure 2-3: Elements of an Architecture Description 32
Figure 2-4: Module View of the MP3 Player Architecture 35
Figure 2-5: Deployment View of the MP3 Player Architecture 35
Figure 2-6: Architectural Decision and Architectural View 37
Figure 2-7: Broker Pattern [Buschmann 1996] 41
Figure 2-8: Architecture Evaluation 52
Figure 2-9: Elements of a Quality Attribute Scenario [Bass 2003] 56
Figure 2-10: Use Case Map [Buhr 1996] 57
Figure 3-1: Decision Scope and Impact [Malan 2002] 73
Figure 3-2: Applying a Virtual Machine Mechanism 75
Figure 3-3: Impact of Architectural Decisions on Multiple Quality Attributes 76
Figure 3-4: Key Concepts of the Approach 77
Figure 3-5: Fundamental Approach 78
Figure 3-6: QUADRAD Context Diagram 83
Figure 3-7: Core Workflows of QUADRAD (Level-1 Diagram) 85
Figure 4-1: The QUADRAD Preparation Workflow (Level-1 Diagram) 87
Figure 4-2: The QUADRAD Preparation Workflow (Level-2 Diagram) 88
Figure 4-3: Identifying Architectural Drivers (Level-2 Diagram) 90
Figure 4-4: Identifying Architectural Drivers (Level-3 Diagram) 91
Figure 4-5: Criteria for Identifying Architectural Drivers 92
Figure 4-6: Requirements Portfolio Diagram Template 98
Figure 4-7: Portfolio Diagram of the EVCS Requirements 100
Figure 4-8: Determining Architectural View Types (Level-2 Diagram) 101
Figure 5-1: The QUADRAD Modeling Workflow (Level-1 Diagram) 107
Figure 5-2: The QUADRAD Modeling Workflow (Level-2 Diagram) 109

Figure 5-3: Defining Architectural Strategies and Mechanisms (Level-2 Diagram)............110
Figure 5-4: Defining Architectural Strategies and Mechanisms (Level-3 Diagram)............111
Figure 5-5: Kiviat Diagram of an Architectural Mechanism............121
Figure 5-6: Quality Profiles for Slicing, Caching, and Layering............123
Figure 5-7: Modeling the Architectural Infrastructure (Level-2 Diagram)............128
Figure 5-8: Modeling the Architectural Infrastructure (Level-3 Diagram)............128
Figure 5-9: Traceability Chain From Architectural Driver to Architectural Views............136
Figure 5-10: Layered View of the EVCS Architecture............138
Figure 5-11: Deployment View of the EVCS Architecture............139
Figure 5-12: Refining the Architecture (Level-2 Diagram)............140
Figure 5-13: Layered View of the EVCS Architecture (Refined)............142
Figure 6-1: The QUADRAD Evaluation Workflow (Level-1 Diagram)............145
Figure 6-2: The QUADRAD Evaluation Workflow (Level-2 Diagram)............147
Figure 6-3: Defining Evaluation Goals (Level-2 Diagram)............149
Figure 6-4: Eliciting and Prioritizing Evaluation Scenarios (Level-2 Diagram)............151
Figure 6-5: Eliciting and Prioritizing Evaluation Scenarios (Level-3 Diagram)............151
Figure 6-6: Mapping Scenarios and Identifying Decisions (Level-2 Diagram)............158
Figure 6-7: Scenario Map for Scenario ES3 (Software View)............161
Figure 6-8: Scenario Map for Scenario ES3 (Physical View)............162
Figure 6-9: Analyzing Decisions and Summarizing Findings (Level-2 Diagram)............163
Figure 6-10: Analyzing Decisions and Summarizing Findings (Level-3 Diagram)............164
Figure 7-1: AET Software Architecture............187
Figure 7-2: Deployment View of the AET Architecture............188
Figure 7-3: AET Data Model............188
Figure 7-4: Quality Attribute Tree............191
Figure 7-5: Classification of Scenarios............192
Figure 7-6: Scenario Analysis............193
Figure 7-7: Tree View of Qualities, Scenarios, and Analysis Results............194
Figure 7-8: Architecture Evaluation Report............195
Figure 8-1: Validation Concept............200
Figure 8-2: Overview of the Validation Approach............208
Figure 8-3: Original Architecture of Project A............210
Figure 8-4: Scenario Map for ES6............212
Figure 8-5: Scenario Map for ES9............213
Figure 8-6: Risk Rankings of Project A (Original Architecture)............216
Figure 8-7: Revised Architecture of Project A............217
Figure 8-8: Introduction of User Privileges............218
Figure 8-9: Supervision of Safety-Critical Components............219
Figure 8-10: Integration of a Software Update Capability............220
Figure 8-11: Risk Rankings of Project A (Revised Architecture)............222
Figure 8-12: Original Architecture of Project B............225

Figure 8-13: Risk Rankings of Project B (Original Architecture) 227
Figure 8-14: Revised Architecture of Project B 229
Figure 8-15: Improved Environment Description for High-Quality Sensor Data 230
Figure 8-16: State Transitions for Power-Controlled Applications 230
Figure 8-17: Multi Sensor Type Support 231
Figure 8-18: Software Update Capability for Sensor Devices 232
Figure 8-19: Risk Rankings of Project B (Revised Architecture) 234
Figure 8-20: Original Architecture of Project C 237
Figure 8-21: Data Management in Project C 238
Figure 8-22: Risk Rankings of Project C (Original Architecture) 241
Figure 8-23: Revised Architecture of Project C 243
Figure 8-24: Abstract Factory and Builder Mechanisms for Report Generation 244
Figure 8-25: Report Generator Code Samples 245
Figure 8-26: Report Generation 245
Figure 8-27: Architecture Report (Sample Page) 246
Figure 8-28: Revised Data Model 248
Figure 8-29: Database Query for Mapping Risks to Business Requirements 248
Figure 8-30: Database Recovery 250
Figure 8-31: Risk Rankings of Project C (Revised Architecture) 252
Figure 8-32: Baseline vs. Revised Architectures A, B, and C 255
Figure 8-33: Risk Reduction and Quality Improvement Achieved with QUADRAD 256

List of Tables

Table 1-1: Source of Problems in Different Application Domains [Neumann 1995] 6
Table 2-1: Quality Attributes 26
Table 2-2: Typical Relationships Between Quality Attributes [Thiel 2001c] 30
Table 2-3: Architectural View Types [Clements 2003] 34
Table 2-4: Architectural Strategies Addressing Availability [Bass 2003] 39
Table 2-5: Capability Summary of Architecture Design Methods 49
Table 2-6: Probability and Impact 55
Table 2-7: Quality Attribute Scenario 57
Table 2-8: AQA Scoring Schema [MITRE 1996] 62
Table 2-9: Capability Summary of Architecture Evaluation Methods 68
Table 3-1: Requirements Derived From Research Gaps 79
Table 3-2: DFD Modeling Notation [Gane 1979] 82
Table 4-1: Template for Requirements Prioritization 95
Table 4-2: Prioritization of Requirements 96
Table 4-3: EVCS Requirements 99
Table 4-4: Quality Attribute Clusters for Architectural Drivers of EVCS 102
Table 4-5: Architectural View Types and Quality Attributes 103
Table 4-6: Architectural View Types to be Considered for EVCS 103
Table 4-7: Mapping Preparation Activities to Research Gaps and Requirements 104
Table 5-1: Template for Capturing Design Problems of Architectural Drivers 113
Table 5-2: Design Problems for Architectural Driver REQ1 114
Table 5-3: Template for Clustering Design Problems 114
Table 5-4: Clustered Design Problems of the EVCS 115
Table 5-5: Template for Linking Architectural Strategies to Design Problem Clusters 115
Table 5-6: Template for Correlating Architectural Drivers with Strategies 116
Table 5-7: Correlation of Architectural Drivers and Strategies 117
Table 5-8: Template for Creating a Strategy-Mechanism Matrix 119
Table 5-9: Architectural Strategies and Supporting Mechanisms 120
Table 5-10: Template for Creating a Driver-Mechanism Matrix 124
Table 5-11: Driver-Mechanism Matrix of the EVCS 125
Table 5-12: Template for Documenting the Suitability of Mechanisms 126
Table 5-13: Suitability of Mechanisms for the EVCS 127
Table 5-14: Template for Documenting Architectural Decisions 130
Table 5-15: Template for Documenting Design Issues 131

Table 5-16: Architectural Decision AD001 for the EVCS 132
Table 5-17: Architectural Decision AD002 for the EVCS 133
Table 5-18: Design Issues of the EVCS 134
Table 5-19: Template for Documenting Architectural Decision and Affected View Types . 135
Table 5-20: View Types Affected By Architectural Decisions AD001 and AD002 135
Table 5-21: Template for Documenting Design Elements 137
Table 5-22: Documentation of EVCS Design Elements 139
Table 5-23: Refinement of EVCS Design Elements 142
Table 5-24: Mapping Modeling Activities to Research Gaps and Requirements 143
Table 6-1: Template for Capturing Evaluation Goals 150
Table 6-2: Evaluation Goals for the EVCS architecture 150
Table 6-3: Template for Describing Evaluation Scenarios 153
Table 6-4: Template for Checking Quality Coverage of Evaluation Scenarios 153
Table 6-5: Examples of Evaluation Scenarios 155
Table 6-6: Quality Coverage of Evaluation Scenarios 156
Table 6-7: Template for Scenario Prioritization 157
Table 6-8: High Priority Evaluation Scenarios 157
Table 6-9: Template for Documenting the Achievement of an Evaluation Scenario 160
Table 6-10: Design Element Responsibilities and Decisions for Scenario ES3 162
Table 6-11: General Analysis Questions 166
Table 6-12: Questions for Analyzing Security Aspects 166
Table 6-13: Questions for Analyzing Performance Aspects 167
Table 6-14: Questions for Analyzing Modifiability Aspects 167
Table 6-15: Questions for Analyzing Safety and Reliability Aspects 167
Table 6-16: Template for Documenting Findings and Open Issues 168
Table 6-17: Documentation of Findings Identified during Architectural Analysis 169
Table 6-18: Environmental Data for Estimating Execution Times 171
Table 6-19: Template for Describing the Resource Usage of an Evaluation Scenario 172
Table 6-20: Environmental Data for the VNS Example 174
Table 6-21: Resource Usage of the Scenario ES3 175
Table 6-22: Execution Time of the Scenario ES3 175
Table 6-23: Template for Documenting Solution Approaches 177
Table 6-24: Solution Approaches for Risk 2 of the VNS Example 177
Table 6-25: Template for Clustering Risks 178
Table 6-26: List of Findings for the VNS Evaluation 179
Table 6-27: Risk Clusters 179
Table 6-28: Mapping Evaluation Activities to Research Gaps and Requirements 180
Table 7-1: QUADRAD Activities Supported by AET 183
Table 7-2: Functional Requirements of AET 185
Table 7-3: Quality Requirements of AET 186
Table 7-4: AET Functionality 189

Table 7-5: Typical Activities of an Architecture Evaluation Workshop 190
Table 8-1: Pilot Projects of the Validation 198
Table 8-2: GQM Model for Validation 201
Table 8-3: Probability of a Risk 202
Table 8-4: Impact of a Risk 202
Table 8-5: Risk Ranking 203
Table 8-6: Risk Severity and Architectural Quality 204
Table 8-7: Evaluation Results for the Original Architecture A 209
Table 8-8: Evaluation Scenarios of Project A 211
Table 8-9: Probability and Impact of the Risks in Project A 214
Table 8-10: Architectural Risks of Project A (Extract) 215
Table 8-11: Risks of Project A (Original vs. Revised Architecture) 221
Table 8-12: New Risks of Project A (Revised Architecture) 222
Table 8-13: Validation Results of Project A 223
Table 8-14: Evaluation Results for the Original Architecture B 224
Table 8-15: Evaluation Scenarios of Project B 226
Table 8-16: Probability and Impact of the Risks in Project B 227
Table 8-17: Architectural Risks of Project B (Extract) 228
Table 8-18: Risks of Project B (Original vs. Revised Architecture) 233
Table 8-19: Validation Results of Project B 235
Table 8-20: Evaluation Results for the Original Architecture C 236
Table 8-21: Evaluation Scenarios of Project C 239
Table 8-22: Probability and Impact of the Risks in Project C 241
Table 8-23: Architectural Risks of Project C (Extract) 242
Table 8-24: Risks of Project C (Original vs. Revised Architecture) 251
Table 8-25: New Risks of Project C (Revised Architecture) 251
Table 8-26: Validation Results of Project C 253
Table 8-27: Validation Results 254

Abstract

Over the past decade, the amount and complexity of software for almost any business sector has increased substantially. Unfortunately, the increased complexity of software in the systems to be built has often lead to a significant mismatch between the planned and the implemented products. One common problem is that system-wide quality attributes such as safety, reliability, performance, and modifiability are not sufficiently considered in software architecture design. Typically, they are addressed in an ad-hoc and unstructured fashion. Since rationales for architectural decisions are frequently missing, risks associated with those decisions can be neither identified, nor mitigated in a systematic way. Consequently, there is a high probability that the resulting software architecture fails to meet business goals and does not allow the building of an adequate system.

This work presents QUADRAD, a framework for Quality-Driven Architecture Development. QUADRAD is capable of improving architecture quality for software-intensive systems in a systematic way. It supports the development of architectures that are optimized according to their essential quality requirements. Such architectures permit the building of systems that are better aligned to the principal market needs and business goals. QUADRAD is complemented by the Architecture Exploration Tool (AET), which supports architecture evaluations and helps in documenting the fundamental design decisions of an architecture.

QUADRAD has been validated in three industrial projects. For each of these projects the architecture quality could be significantly increased. The results confirm the hypothesis of this work and demonstrate how critical problems in the transition from requirements to architecture design can be mitigated.

Keywords: Software architecture, architecture development, quality attributes, architecturally driving requirements, architecture design, architecture evaluation, architectural strategy, architectural mechanism, architectural pattern, architectural decision, architectural risk, tool support.

Acknowledgements

I would like to thank the following persons:

- *Prof. Dr. K. Pohl* and *Prof. Dr. M. Goedicke* for giving me the opportunity to carry out this work.
- *M. Auerswald, Dr. S. Ferber, Dr. T. Fischer, A. Hein* and *Prof. Dr. S. Kowaleweski* for their contributions in the European research projects ESAPS and CAFÉ. These projects have encouraged me to prepare this work.
- *H. Engelhardt* and *A. Hein* for their support in developing the AET research tool.
- *Dr. L. Bass, F. Bachmann, Dr. G. Böckle, Dr. G. Chastek, P. Donohoe, P. Graubmann, G. Halmans, A. Hein, Dr. E. Kamsties, Prof. Dr. P. Knauber, F. Peruzzi, Dr. K. Schmid, Prof. Dr. K. Pohl, A. Reuys, Dr. A. Schreiber, Dr. M. Verlage,* and *Dr. J. Weingärtner* for many stimulating discussions about software architecture development and related topics.
- *M. Auerswald, C. Fritsch, Dr. R. Hahn, A. Hein, Dr. T. Strohm,* and *N. Thiel* for their comments on earlier versions of this work.
- *M. Andrew* for his support in proofreading this work.
- And last but not least my *wife* for her patience and mental support during the development of this work.

1 Introduction

This work demonstrates that the quality of architecture development for software-intensive systems can be improved based on a systematic derivation of design decisions for architecturally driving requirements and a methodical evaluation of the effects of those decisions in relevant architectural views. The thesis provides a framework with concrete activities to improve the transition from requirements to architecture.

1.1 Motivation

Over the past decade, the amount and complexity of software for almost any business sector has increased substantially. This is especially true for automotive electronics systems. Today's luxury cars, for example, include more than 50 electronic control units [Grimm 2003] that operate as partly networked systems to improve passenger comfort, safety, economy, and security (see Figure 1-1). Parking assistance and adaptive cruise control systems, for example, make it easier to operate the car in various driving situations, thus reducing the drivers' workload and increasing their comfort. Safety-related systems, such as automatic stability and airbag control systems help drivers to avoid or reduce the impact of accidents. Fuel economy systems lower emissions and increase fuel efficiency, while security systems protect the car from unauthorized manipulation.

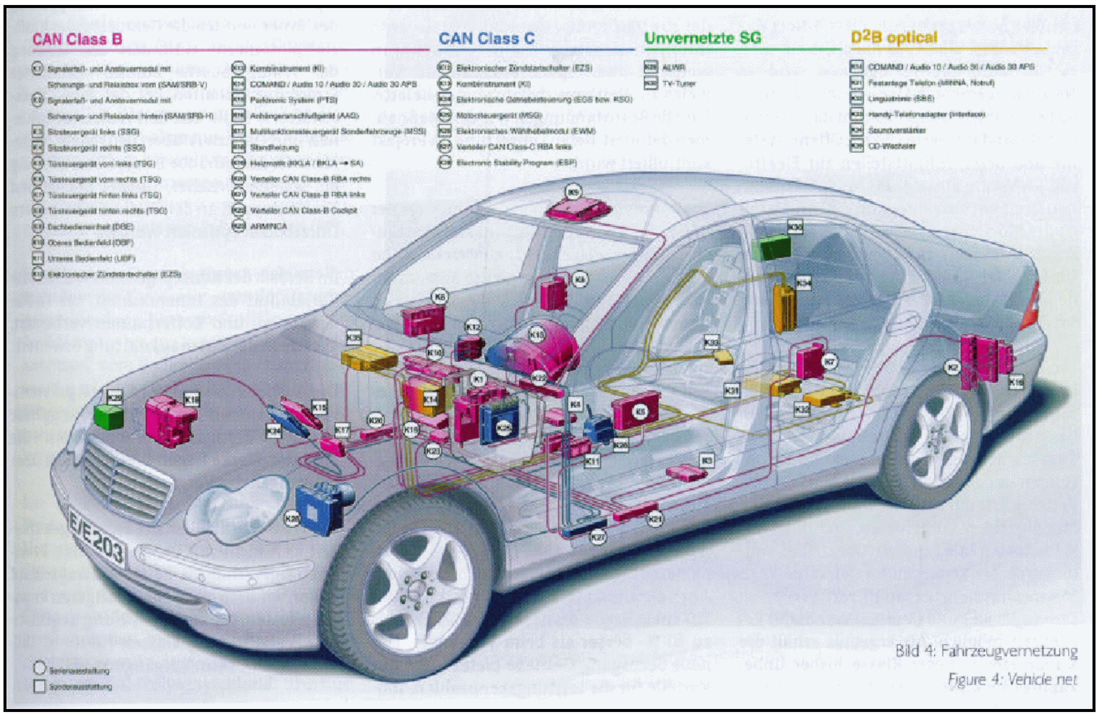


Figure 1-1: Automotive Electronics Systems

While the number of electronic control units in the car has increased, so the importance and complexity of software in these systems has significantly increased, too. According to a study about the future of electronics and software in cars conducted by Mercer Management Consulting [Mercer 2002], the amount of software in automobile electronics will increase by 8% per year until 2010. As shown in Figure 1-2, the software in cars will rise up from a €25 billion market in 2000 to a €100 billion market in 2010, reaching 13% of the total cost for a standard automobile [Mercer 2002].

At the same time the business case in the automotive domain changes (e.g., [Weber 1999]). While innovation has been a traditional factor for business success, cost leadership becomes more and more important. Innovative features of today's systems become standard (or "commodity") features of tomorrow – the innovation time cycles constantly shrink. The competitive pressure rises accordingly and hence those commodity features must be implemented in a cost effective way. A well-grounded knowledge of the market needs becomes essential in order to remain competitive and to defend market positions.

Therefore, organizations are forced to adapt their engineering to accommodate business changes as well as changes in software system development. Strong technology orientation, which is appropriate for innovative development, is going to be complemented by engineering approaches that are driven by business goals and market needs. This is one reason why product line development has gained much attention over the past couple of years (e.g., [Böckle 2004], [Pohl 2005]).

In this context, the establishment of an overall perspective in software systems engineering takes on a pivotal role. Sustainable solutions that meet the organization's business goals can only be developed by adopting an appropriate system abstraction and by following a systematic approach. Many authors have proposed software architecture as the appropriate abstraction and architecture development as the key factor in system success or failure (e.g., [Perry 1992], [Garlan 1993], [Booch 1999], [Jazayeri 2000], [Bass 2003]). This thesis is strongly connected to the opinion of these authors. It discusses fundamental problems in architecture development practices and demonstrates how they can be improved accordingly.

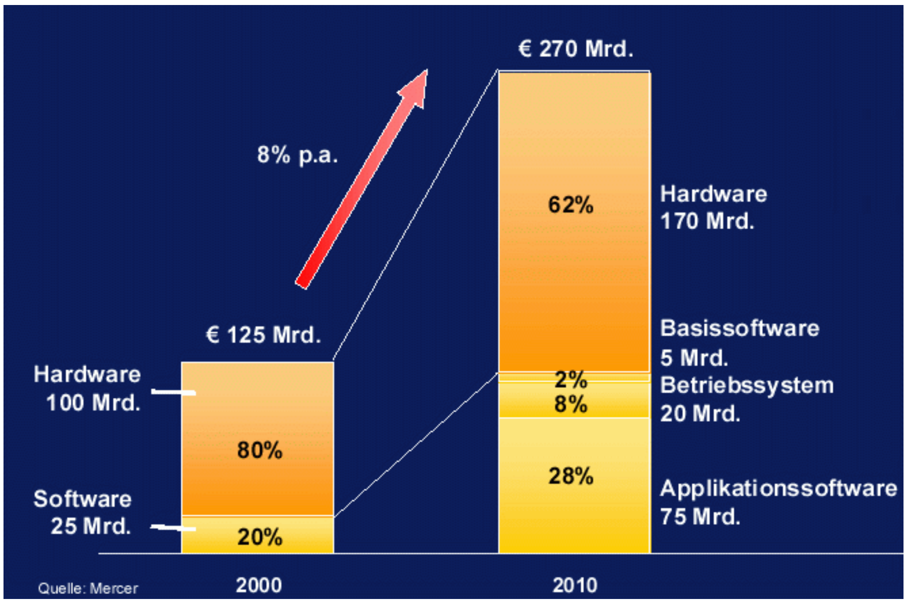


Figure 1-2: Increasing Importance of Software in Automobile Electronics [Mercer 2002]

1.2 Problem Context

Experience reports have shown that creating the “right” system for a set of given requirements is still a general problem in software system development (e.g., [Davis 1990], [Neumann 1995], [Bosch 2000], [CHAOS 2001], [Bass 2003]). In many cases, the system that results from an organization’s development effort does not adequately match the organization’s intended business goals, market needs or customer expectations although requirements analysis and system design activities might have been performed as part of the organization’s prescribed development process (see Figure 1-3).

This mismatch may have – and usually has – serious consequences for the organization. For example, if essential quality requirements are not met because they were not considered or underestimated during system development, the system may fail to fulfill key customer requirements or the expected market standard. Consequently, the system will not achieve the targeted market share. The revenue then does not compensate the investment, which results in financial loss of the organization.

The same may happen if the system implements the essential customer features but fails to meet the cost targets. Then the system cannot be offered at competitive prices or it does not generate profits for the organization. Redesigning the system to optimize cost aspects may lengthen the times to market. The danger thereby is that competitors will have already established their systems in the market and that the remaining demand is too small to be profitable. Note that this smaller demand probably also forces a reduced selling price and that the resulting margin may be the same as if the cost optimization measures had never been attempted.

In this context, Davis [Davis 1990] has published an interesting study about the project success rates of several U.S. software development organizations (see Figure 1-4). The study indicates that a lot of the financial investment spent by the organizations for system development is being wasted because of serious mismatches between planned and implemented system. At the same time, it indicates that many of these organizations have fundamental problems with the approaches they have adopted for developing software-based systems.

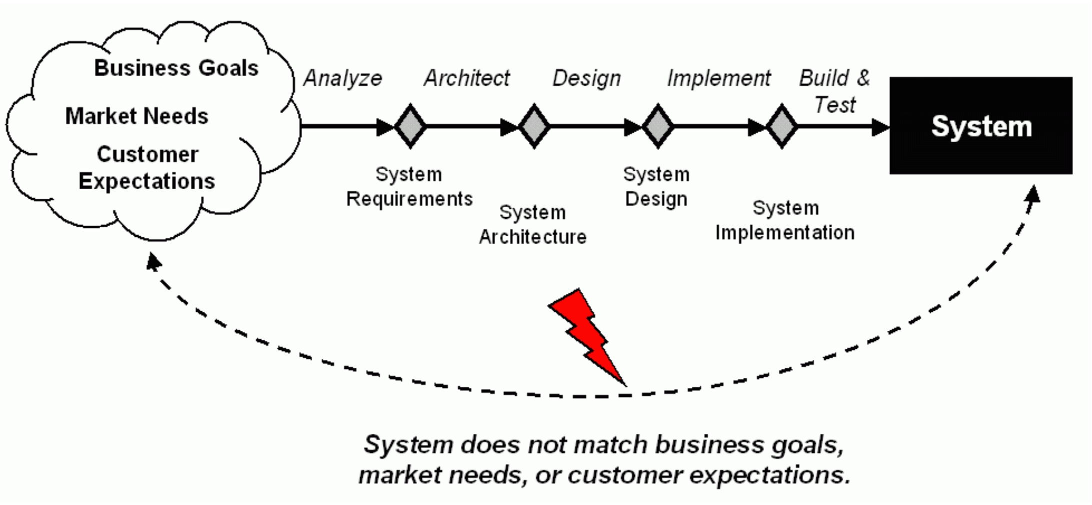


Figure 1-3: Mismatch Between Planned and Implemented System

As shown in Figure 1-4, 47% of the money was spent for systems that were never used since they did not implement the required features. An additional 29% of the money was spent for systems that were never even delivered. Another 19% resulted in software that was either extensively reworked after delivery or later abandoned. Three percent of the systems could be used after changes. Only two percent of the systems could achieve the customer requirements and have been delivered on time.

Another, more recent study, the Chaos Report 2000, conducted by The Standish Group [CHAOS 2001] shows a similar although slightly better result for the software industry. In this study, the project success rates of more than 280,000 application projects in large, medium, and small cross-industry U.S. companies have been analyzed between 1998 and 2000. Many of these projects are related to the development of software-intensive systems. The results: 65,000 (23%) of the projects were cancelled before completion or were never implemented. 137,000 (49%) projects have challenged (i.e., the projects were completed but over-budget, over the time estimate, and with fewer features than initially specified). Nevertheless, 78,000 (28%) of the projects analyzed by The Standish Group could be completed on time and on budget, with all features originally specified. Although these numbers have improved over the results from 1994, where 36% of the projects had failed, 53% had challenged, and only 16% had succeeded, they are far from being satisfactory for the companies and their customers.

The interesting question for the purpose of this work now is: What are the main reasons for the low project success rates reported above and which phases of software system development are most error-prone and need fundamental improvement?

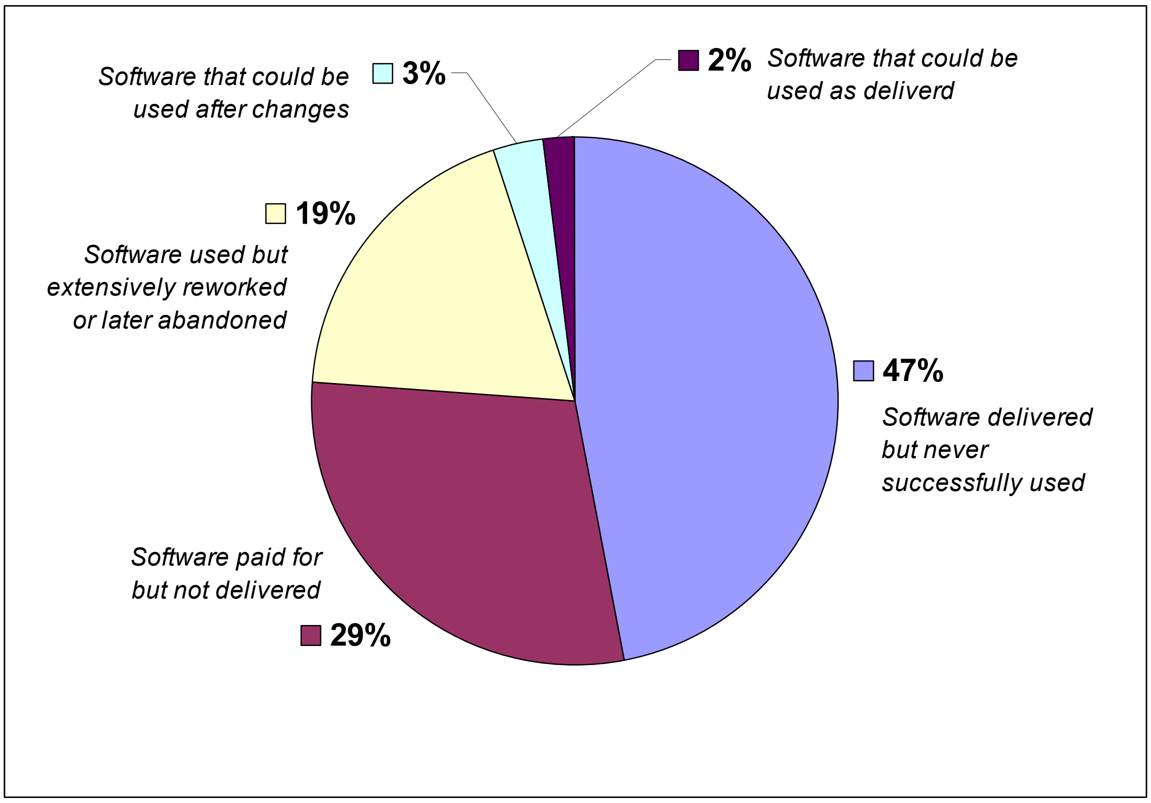


Figure 1-4: Performance of Software System Development Projects [Davis 1990]

A starting point to answering this question is given by Neumann [Neumann 1995]. In his book "Computer-Related Risks" Neumann provides information about why many system development efforts have failed (see [Neumann 1995], pp. 13-95). The study is based on the analysis of a large collection of systems from different application domains such as communication, space, civil aviation, railroad, ships, control, robotics, medical health, and electrical power. Most of the misconceptions and flaws identified in these systems have caused serious safety, reliability, and robustness problems during operation. Some of them have resulted in human injuries, others even in loss of life.

Neumann [Neumann 1995] has analyzed the causative factors that contributed to the problems occurring in these systems and has classified the factors according to different development phases and typical problem types (see Table 1-1):

- *Requirements engineering:* Erroneous requirements definition, incomplete requirements (e.g., system use and operation not specified), inconsistent requirements
- *Architecture design:* Fundamental system design misconceptions, malfunctions of system elements not considered in system design (e.g., communication errors)
- *Hardware implementation:* Errors in chip fabrication, wiring errors
- *Software implementation:* Programming bugs, compiler bugs, malicious code
- *Evolution and maintenance:* Faulty upgrades, sloppy redevelopment or maintenance, decommission

As shown in Table 1-1, the reasons for the problems of 34% of the systems involved result from the introduction of bugs during evolution and maintenance. For 40% of the systems the root cause of the problems stems from implementation errors in either software or hardware. The study also clearly shows that the most problems have their origin in requirements engineering and, especially, architecture design with more than 58% and almost 75%, respectively. Figure 1-5 gives a graphical summary of the ratios.

Similar tendencies for sources of problems have been reported by other authors (e.g., [Henderson 1990], [Brooks 1995], [Maier 2000], [Bass 2003]). The authors conclude that since the complexity of software-intensive systems has significantly increased during the last decade and will increase by orders of magnitude in future, the need for sound architecture practices based on the set of significant requirements is one of the major challenges for tomorrow's software system engineering.

Focusing on architectural practices such as architecture design and evaluation also supports the establishment of a holistic view of software system development. The change is especially noticeable in the automotive industry but can also be perceived in other industrial sectors (e.g., telecommunications, automation technology, and household applications).

Table 1-1: Source of Problems in Different Application Domains [Neumann 1995]

Application Domain	Source of Problems					Analysis Data	
	Requirements Engineering	Architecture Design	Hardware Implementation	Software Implementation	Evolution and Maintenance	# Investigated Systems	# Problems Analyzed
Communication Systems	4	11	4	4	6	11	29
Space Systems	10	19	5	16	8	32	58
Defense Systems	8	4	2	5	2	10	21
Civil Aviation	12	8	7	6	3	15	36
Railroad	13	19	9	5	8	22	54
Ships	5	5	3	4	2	6	19
Control Systems	5	9	6	2	3	11	25
Robotics Systems	3	7	4	5	0	10	19
Medical Health Systems	7	7	3	3	2	7	22
Electrical Power Systems	19	20	16	9	16	23	80
Total	**86**	**109**	**59**	**59**	**50**	**147**	**363**
Ratio of Problems	**58,5%**	**74,1%**	**40,1%**	**40,1%**	**34,0%**		

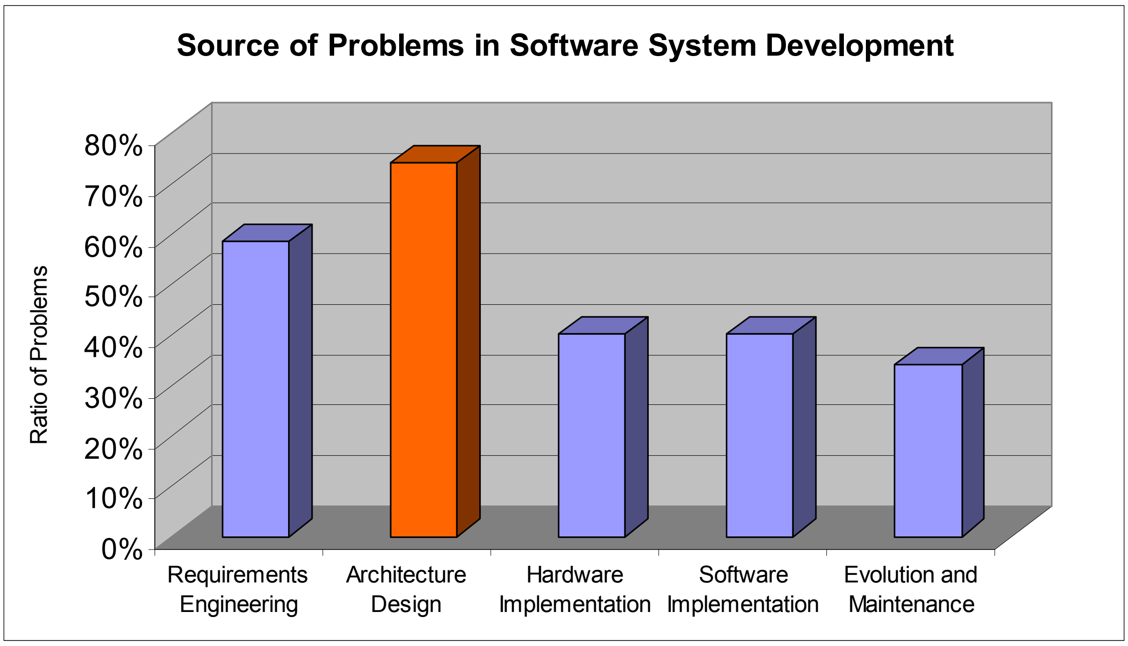


Figure 1-5: Source of Problems in Software System Development (Summary)

1.3 Problems of Scope and Focal Research Fields

As shown in the Neumann study presented in the previous section, three out of four problems stem from an inappropriate architecture design. This means that the architecture is in many cases not suitable for building the adequate system. There are two fundamental reasons why this can happen: (1) The architecture is created or evolved based on wrong requirements or (2) the architecture does not adequately implement the requirements imposed on it.

Concerning the first reason, Neumann [Neumann 1995] noted on page 234 that *"a priori requirements definition is vital within a total system context [...], but is nevertheless often given inadequate attention. Requirements should include aspects of security, [...] ease of use, generality, flexibility, efficiency, portability, maintainability, evolvability, and any other issues that might come home to roost if not specified adequately."* Using these results from Neumann's analysis, we can further refine the problem as follows:

Problem 1: Software system architectures are primarily created or evolved based on wrong requirements.

The term "wrong requirements" in this context refers to insufficient, less important, or insignificant requirements. These requirements are not suitable for achieving the major system concerns. Often, the architecture is driven by functionality. System-wide quality attributes such as safety, reliability, performance, or modifiability are not sufficiently considered or they are treated in an ad-hoc and unstructured fashion during architecture design. Therefore, the architecture fails to meet important market requirements and customer expectations. Business goals become impossible to achieve.

Concerning the second reason, Neumann [Neumann 1995] wrote on page 227: *"Seldom do we find a system development whose design decisions were well documented, individually justified, and subsequently evaluated – explaining how the decisions were arrived at and how effective they were. Clearly, however painful it may be to carry out such an effort, the rewards could be significant."* On page 235 he added that the *"evaluation of the consistency of a design with respect to its design criteria and requirements should be done with considerable care prior to any implementation, determining the extent to which the design specifications are consistent with the properties defined by the requirements. Such an evaluation was clearly lacking on many of the cases discussed here."* These statements lead to the formulation of the second key development problem:

Problem 2: Making appropriate design decisions during architecture development is not or only vaguely understood. This results in serious development risks.

Architectural decisions are made on instinct. Rationales for decisions are missing. The interactions among architectural decisions are uncontrolled and remain vague. Risks associated with architectural decisions are neither identified, nor addressed in a systematic fashion. Since the system behavior is only vaguely understood, system-wide optimizations are not achieved. Single components of the system are modified in an uncontrolled fashion. These local optimizations fail to promote an optimal solution for the overall system. Whether the given requirements are achieved cannot be determined before the system has been implemented.

Figure 1-6 summarizes the two problem areas, the results for system development, as well as the consequence for the organization's business. The problems represent critical risks for an organization. Failing to meet the business goals, market needs, or customer expectations with the target system will inevitably diminish the market share and system quality. Fixing design flaws or implementing requirements to meet missing quality attributes in a later development phase usually requires high efforts and thus high extra cost. Sometimes fixing errors is simply not possible because it would require a new system design (i.e., restructured architecture). In both cases, this would lead to late market introduction and smaller revenues for an organization since competitor systems may have been already established. Therefore, improving architecture practices and bridging the gap between requirements specification and architecture development in order to meet the business goals more accurately are problems of scope for system development. This work deals with these problems.

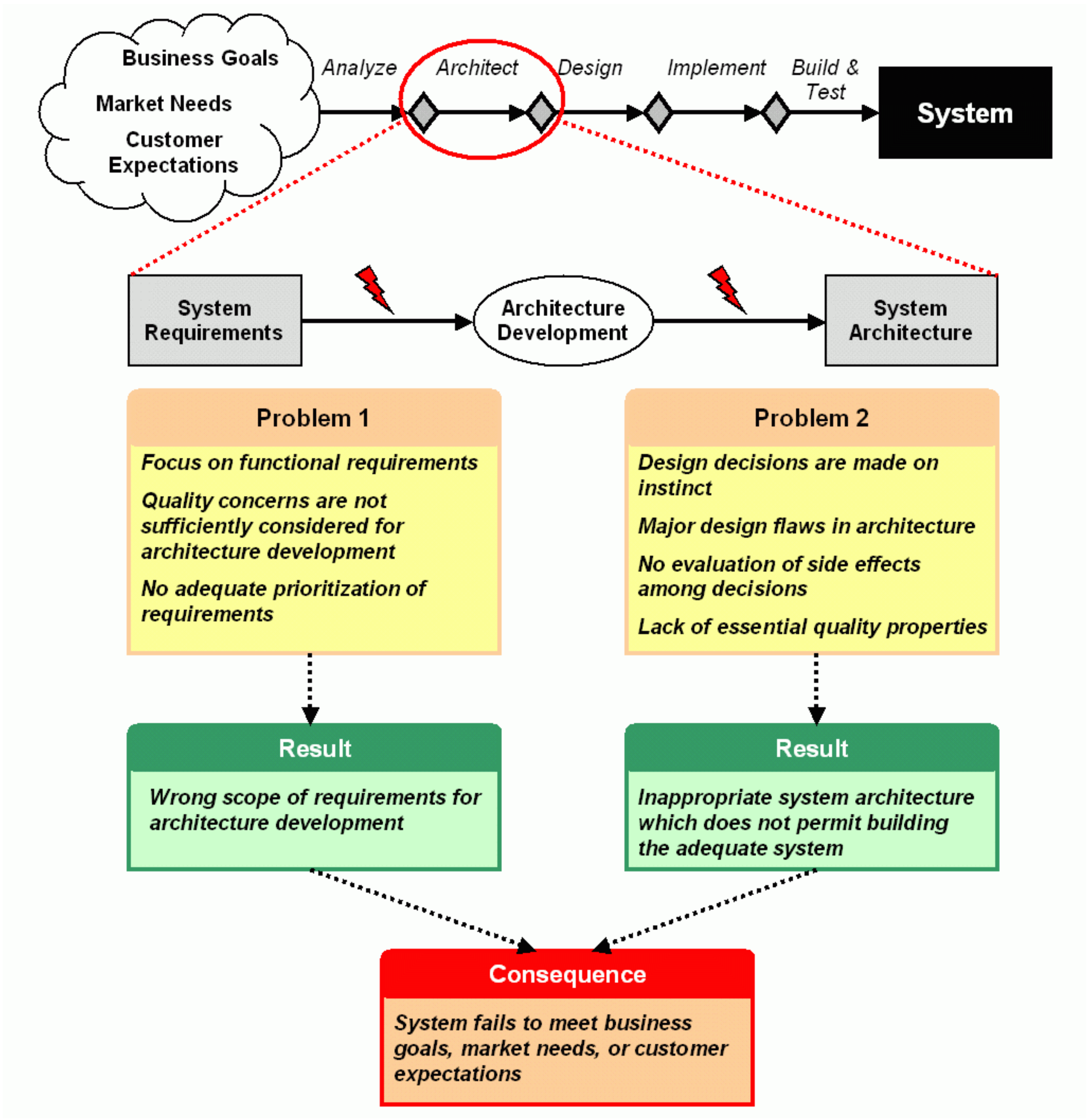

Business Goals
Market Needs
Customer Expectations
Analyze
Architect
Design
Implement
Build & Test
System
System Requirements
Architecture Development
System Architecture
Problem 1
Focus on functional requirements
Quality concerns are not sufficiently considered for architecture development
No adequate prioritization of requirements
Problem 2
Design decisions are made on instinct
Major design flaws in architecture
No evaluation of side effects among decisions
Lack of essential quality properties
Result
Wrong scope of requirements for architecture development
Result
Inappropriate system architecture which does not permit building the adequate system
Consequence
System fails to meet business goals, market needs, or customer expectations

Figure 1-6: Problems of Scope of this Work

1.3.1 Research Gaps

Architecture development continues to be an area of growing importance (e.g., [Hofmeister 2000], [Bass 2003], [Garland 2003]). Indeed, it is a widely agreed view in the systems development community that errors generated during the requirements engineering phase and implemented in the software system architecture are the most expensive to fix (e.g., [Pohl 1996], [Bass 2003]). Various studies have shown that fixing an error after the system has been delivered leads to costs of orders of magnitude more than those that would be incurred by fixing it in the architecture phase (e.g., [Neumann 1995], [Brooks 1998], [Maier 2000]). These symptoms reflect, by a large margin, the lack of adequate architecture development approaches. In this work, the following three gaps in current research have been identified as major reasons why the problems mentioned have not yet been addressed sufficiently[1]:

Research Gap G1: Lack of guidance for identifying requirements that are essential for architecture development

It is widely agreed that especially the transition from requirements specification to architecture design is one of the most critical and error-prone stages in system development (e.g., [Hofmeister 2000], [Kruchten 2000], [Bass 2003]). System architects are often overwhelmed by the myriad of requirements collected in different documents and in various forms. There is a considerable lack of guidance in current research to support the architect in deciding which of the requirements are essential for the architecture design and therefore should be considered first (e.g., [Neumann 1995], [Bass 2003], [Garland 2003]). Although different approaches for requirements prioritization have been proposed (e.g., [Karlsson 1997]), none of these approaches can adequately support the specific needs of the architect – this is, identifying those requirements that are essential for creating a sustainable architecture that permit achieving the business goals (cf. Section 2.2.5). The loose connection between the requirements engineering and architecture development community might be one reason why this problem has not been explored in sufficient detail, yet.

Research Gap G2: No systematic support for making adequate design decisions in order to implement architecturally relevant requirements

A fundamental task of a software or system architect is to make design decisions in order to evolve and shape the architecture's structures. Although many researchers have proposed architecture design methods (e.g., [Bosch 2000], [Kruchten 2000], [Hofmeister 2000], [Bass 2003]), a systematic support for making adequate decisions for architecturally relevant requirements is missing. Using architecture styles and design patterns (e.g., [Shaw 1996], [Kircher 2004]) during architecture design has slightly improved the decision making process but the loose coupling between patterns and architecture design approaches as well as the insufficient connection between patterns and system quality attributes are seen as major weak points by many researchers (e.g., [Bass 2003]). Moreover, styles and patterns often need considerable modifications that require additional design decisions in order to be applicable in a specific system context (e.g., [Bosch 2000]). The support for making adequate decisions and for documenting the rationales for those decisions therefore remains a largely unexplored field in current research.

[1] In Chapter 2 the research gaps are approved by an evaluation of related work.

Research Gap G3: Insufficient support for the identification and mitigation of unwanted effects of design decisions in architecture development

Architecture design is a decision making process in which the architect is faced with the resolution of many design problems associated to (architecturally relevant) requirements. Some of the problems may be obvious beforehand; others may show up after several iterations or releases such that the decisions must be incrementally made during development. This is natural in practice (e.g., [Malan 2002], [Bass 2003]). As a consequence, the decision making process to implementing requirements adequately cannot be controlled completely and may introduce additional side effects into the architecture (e.g., [Clements 2002]). The repeated application and composition of architectural styles and design patterns, for example, may also lead to side effects since multiple requirements are affected in a largely uncontrolled fashion. Current research in architecture development lacks overall approaches for making and evaluating design decisions (cf. Sections 2.2.5 and 2.3.5). Many approaches focus only on the design tasks of the architecture but do not include activities for evaluating if the intended qualities or business goals are met (cf. Sections 2.2.5). Other approaches aim primarily on the evaluation task. They focus on the analysis of particular architectural quality properties but say nothing about how to improve the architecture according to the risks identified (cf. Sections 2.3.5). Combinations of the two approaches are lacking but aree crucial in order to systematically plan the quality improvement of an architecture in order to mitigate unexpected behavior (e.g., [Neumann 1995]).

This work aims at bridging these gaps and at mitigating the system development problems reported above such that the architecture quality is improved. More precisely, the work focuses on the development of architectures that are better aligned to architecturally significant requirements and that include less design risks. Consequently, this would increase the probability that the architectures permit the building of systems that match the business goals more accurately. This would also mitigate the two development problems shown in Figure 1-6.

Figure 1-7 shows how the research gaps can be mapped to important research fields of requirements engineering and architecture development.

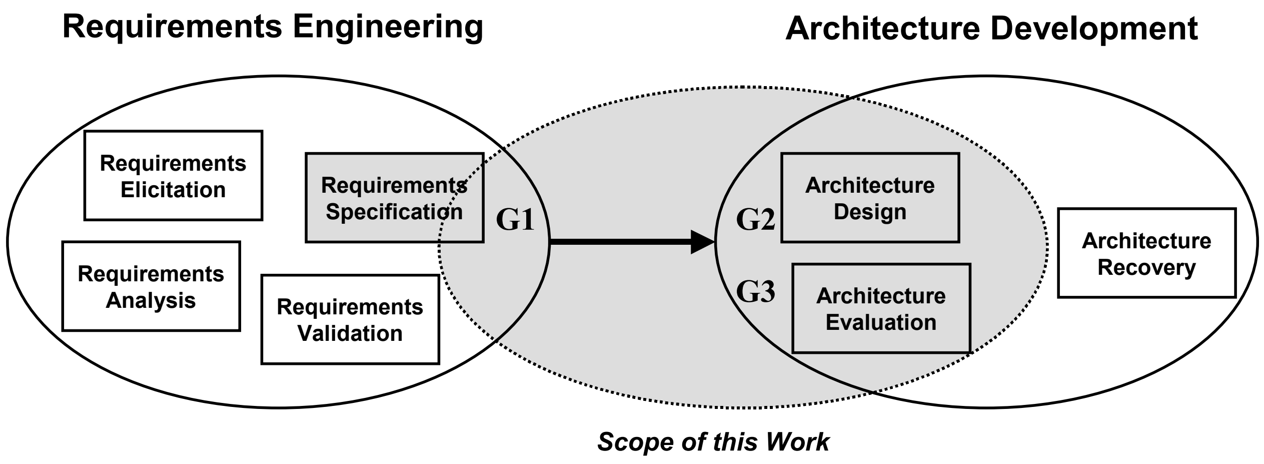


Figure 1-7: Focal Research Fields of this Work

G1 is related to preparing requirements for architecture design and thus associated with requirements engineering research. Requirements engineering is concerned with elicitation, analysis/negotiation, specification, and validation of requirements of a software-based system (e.g., [Pohl 1997]). Activities for *requirements specification* are of special concern for addressing G1 because they must be complemented to provide a description of requirements that support a clearly focused architecture development.

G2 deals with providing an appropriate architecture for achieving architecturally relevant requirements, which affects *architecture design*. Architecture design and architecture evaluation are important research areas of architecture development (cf. Chapter 3). During architecture design, decisions for specifying the design elements of the architecture are fixed. Addressing G2 would complement existing approaches of architecture design.

G3 is concerned with analyzing the architecture with respect to its impact on the overall system properties. This gap is thus related to research in *architecture evaluation*.

In summary, this work aims at providing solutions to the research gaps G1-G3 and thus contributes to the research fields requirements specification, architecture design, and architecture evaluation, respectively.

1.3.2 Topics Out of Scope

As mentioned above, this work addresses particular research areas, especially in the field of architecture development. It explicitly does *not* deal with the following topics:

- *Requirements elicitation, analysis/negotiation, and validation.* This work does not cover requirements elicitation, analysis, and validation in detail. However, it builds on results of those activities as far as they effect requirements specification to support architectural design and validation. Concerning requirements specification, the work does only consider the identification of relevant requirements for architecture design.
- *Organizational and business aspects of architecture development.* This work does not cover organizational and business sources of failure for architecture development. However, setting up the right organization and promoting the business goals are important to build successful systems.
- *COTS selection for architecture development.* This work does not consider the COTS (Components-off-the-Shelf) selection process during architecture design.
- *Formalization of architecture documentation.* This work does not deal with the formalization of architecture documentation by means of architecture description languages (ADLs).
- *Component specification, implementation, and validation.* This work does not discuss component development in detail. However, this work does prepare certain essential results for component specification (e.g., architecture description). For example, the component interface descriptions must conform to the responsibilities of the design elements documented in the architecture description.
- *Architecture recovery:* This work does not discuss issues of recovering an architecture description from source code, although architecture reconstruction can support architecture development (e.g., validation).

1.4 Hypothesis

A conclusion from the previous discussion is that an architecture manifests decisions that either support or preclude the achievement of essential quality requirements. If these requirements are not satisfied, the resulting system, which is based on that architecture, will probably fail to meet the intended business goals, market needs, or customer expectations.

In order to mitigate the risks, a considerable effort in improving the effectiveness of the architecture development process in order to produce high quality architectures is required. Improving the effectiveness means that an organization’s architecture development must focus on solving the hardest design problems first – i.e., those problems that shape the fundamental structures of the architecture and that therefore influence the overall quality properties of the system. Narrowing the scope of architecture design to architecturally relevant problems is particularly beneficial in creating a strong vision of the system concept. Having such a strong vision supports developing clearly scoped architectures that permit the implementation of those qualities that are most essential for the organization’s business success.

This leads to the following hypothesis:

Hypothesis:

The architecture of software-intensive systems can be improved with respect to *quality* based on a systematic derivation of design decisions for architecturally driving requirements and a methodical evaluation and revision of the effects of those decisions in the architecture.

The hypothesis comprises five assumptions that are important for the research of this work:

- **A1:** It is possible to identify architecturally driving requirements that are the primary source for the most fundamental decisions in an architecture.
- **A2:** It is possible to derive appropriate design decisions systematically from architectural strategies and mechanisms in order to address architecturally driving requirements.
- **A3:** It is possible to document these decisions in a way that their origin and rationale can be made explicit.
- **A4:** It is possible to evaluate the effects of decisions in the architecture methodically, which may result in design risks.
- **A5:** It is possible to address the design risks of the evaluation and to revise the architecture accordingly.

The hypothesis includes the idea of systematically identifying the set of architectural drivers for which strategic design decisions must be made in the architecture, and to evaluate the adequacy of the architecture that results from those decisions. This is non-obvious since many existing approaches try to optimize architectures with respect to single quality attributes (cf. Section 2.2.5). However, except for very specific classes of systems, an architecture generally depends on the balanced achievement of multiple, often competing, quality requirements.

1.5 Approach

Over a decade ago, Garlan and Shaw [Garlan 1993] noted that "as the size and complexity of software systems increases, the design problem goes beyond the algorithms and data structures of the computation: designing and specifying the overall system structure emerges as a new kind of problem. […] This is the software architecture level of design."

Clearly then, architecture deals with decisions which are derived from a broad scope or system perspective. These decisions are *architectural decisions*. Architectural decisions represent fundamental or *strategic* decisions made during architecture design. They have implications on the overall system design structure and therefore have a high impact on the achievement of business goals. In general, design decisions are made to achieve particular requirements but only a portion of the requirements of a development effort is important from an architectural perspective. These requirements are called *architectural drivers* (e.g., [Bass 2003]).

Experienced architects apply specific *architectural strategies* and *mechanisms* to make decisions for solving design problems. Architectural strategies describe general principles to address particular classes of design problems. Architectural mechanisms provide concrete design solutions for these problems (e.g., [Buschmann 1996], [Booch 1996], [Schmidt 2000], [Bass 2003], [Kircher 2004]).

Most decisions involve tradeoffs. This means that an architectural decision usually has an impact on more than one system quality (e.g., [Clements 2002], [Bass 2003]). Because many requirements are competing, introducing a design solution to improve one quality may diminish other qualities (e.g., [Chung 1995], [Kruchten 2000], [Bass 2003]). Thus, the tradeoffs of each decision must be carefully considered. However, because of its complexity, the decision making process cannot be controlled completely. Side effects may be introduced into the architecture. This may result in unexpected behavior of the architecture that must be identified and resolved during architecture development.

These observations must be taken into account in order to provide an approach that supports addressing the research gaps. The approach will be deduced in Chapter 3 and further elaborated in Chapter 4, Chapter 5, and Chapter 6.

1.6 Research Contribution

This work has implications on architecture development for software-intensive systems. In particular, it provides the following research contributions:

- A **framework** – QUADRAD (Quality-Driven Architecture Development) – that complements existing architecture development approaches by addressing the previously discussed problems and research gaps systematically in the transition process from requirements to architecture. The framework includes activities, which support the development of architectures for software-intensive systems that are optimized to their essential quality requirements. The benefit of such an architecture is that it better permits building systems that meet the intended business goals and market needs of the development organization. This typically leads to higher quality systems, higher market shares, a better reputation, and a healthier financial situation. In particular, the framework comprises

  - *Activities for identifying and specifying requirements, strategies, and mechanisms that drive the architecture development.* This includes the selection and clarification of the most important quality attribute requirements for the system, the systematic identification of architectural strategies and mechanisms that focus on essential design problems, and the support for traceability between requirements and architecture solutions (QUADRAD Preparation Workflow, cf. Chapter 4).
  - *Activities for making appropriate decisions that implement the essential requirements in the architecture by applying the strategies and mechanisms.* This includes the modeling of the most significant architectural views and the capturing of relevant decisions that lead to those views. In particular, the activities support the definition of an architectural infrastructure, which comprises those design elements that are essential for achieving the most important quality requirements (QUADRAD Modeling Workflow, cf. Chapter 5).
  - *Activities for evaluating the consequences of architectural decisions with respect to the achievement of essential quality requirements and in light of an overall system perspective.* This includes the goal-oriented alignment of the architecture evaluation and the elicitation of related evaluation scenarios and decisions. It further comprises techniques and activities for making those parts of the architecture explicit that implement the driving requirements. In addition, steps for analyzing the appropriateness of the architectural decisions made by the architect as well as for organizing the findings and planning risk mitigation strategies are included (QUADRAD Evaluation Workflow, cf. Chapter 6).

- A **research tool** – AET (Architecture Exploration Tool) – that supports and automates essential activities of QUADRAD. The tool supports the systematic documentation of architecture design results and managing information of an architecture evaluation. AET helps in capturing requirements and refining them with the help of scenarios. It further supports the prioritization of scenarios with regard to their significance for development and allows for a description of the major design decisions made to implement each scenario. AET also supports the documentation of trace information. In particular, it establishes traces between quality requirements, refined scenarios, design decisions that pertain to a particular scenario, and risks associated to particular design decisions recorded during an architecture evaluation (QUADRAD Tool Support, cf. Chapter 7).
- A set of **metrics** to evaluate architecture quality and quality improvement. The definition of the metrics follows the Goal Question Metric (GQM) approach [Basili 1992]. Essentially, the metrics rely on the assessment of the probability and impact of architectural risks. They are based on the fact that the higher the probability and impact of a risk the more critical it is for the development and the more likely the architecture will fail to achieve the intended requirements. In particular, metrics to measure the severity of architectural risks, to estimate the associated quality of the architecture, and to compare the risk severity and architectural quality of two architectures are provided (Validation of the QUADRAD Framework, cf. Chapter 8).

1.7 Organization of the Thesis

This thesis is organized into three parts:

Part I – State-of-the-Art and Research Approach

Part I introduces fundamental concepts and existing work in architecture development. It relates these results to the research gaps and deduces an approach to address the gaps.

Chapter 2: Related Work in Architecture Development. This chapter discusses related work in software architecture and architecture development. It describes essential activities and artifacts of architecture design and evaluation and discusses key concepts that support designing and evaluating architectures. Furthermore, it provides a comprehensive overview of existing design and evaluation methods and compares them with respect to the research gaps.

Chapter 3: Bridging the Research Gaps with QUADRAD. This chapter elaborates key concepts to address the research gaps in this work. It particularly shows how an architecture can be better aligned to architecturally significant requirements in order to support building systems that meet the intended business goals more accurately. It summarizes the main concepts in a metamodel and uses this model to derive the fundamental approach of this work. Moreover, it elaborates a set of key requirements to guide the refinement and packaging of the approach in a framework. The chapter then briefly introduces the framework (QUADRAD, Quality-Driven Architecture Development) and describes the notation used to decompose it into concrete activities.

Part II – Research Contribution

Part II elaborates the approach to address the research gaps and provides the major research contribution of this work.

Chapter 4: The QUADRAD Preparation Workflow. This chapter describes the Preparation workflow of QUADRAD. The purpose of this workflow is to support early architectural considerations. The chapter describes the activities for selecting the set of requirements that are most essential for developing an architecture that permits achieving the business goals. Furthermore, it provides a means for defining architectural view types that address the appropriate views affected by those drivers.

Chapter 5: The QUADRAD Modeling Workflow. This chapter describes the Modeling workflow of QUADRAD. The purpose of this workflow is to support the modeling of architectural views and to record the decisions that lead to those views. The chapter provides essential steps to make this process more systematic and repeatable. In particular, it provides activities for determining strategies and mechanisms for architectural drivers, creating an appropriate architectural infrastructure, and refining the architecture description.

Chapter 6: The QUADRAD Evaluation Workflow. This chapter describes the Evaluation workflow of QUADRAD. The purpose of this workflow is to understand the consequences of architectural decisions made by the architect and to analyze them with respect to the achievement of essential quality requirements. In particular, the chapter describes activities for defining the goals of the architecture evaluation and for eliciting related evaluation scenarios and decisions. It further discusses activities for making those parts of the architecture explicit that implement the driving requirements. In this context, the chapter presents steps for analyzing the appropriateness of the architectural decisions made by the architect, for providing solution approaches as well as for organizing the findings.

Chapter 7: QUADRAD Tool Support. This chapter gives an overview of the Architecture Exploration Tool (AET). AET is a research tool that supports and automates essential activities of QUADRAD. In particular, it supports an architect and review team in documenting the architecture and in managing results of an architecture evaluation. The chapter briefly introduces AET, presents the underlying requirements, and discusses the software architecture and data model of AET. It further illustrates how AET can be applied to support a QUADRAD architecture evaluation workshop.

Part III – Validation, Lessons Learned, and Outlook

Part III covers the validation of the approach. It provides the validation results, lessons learned, a summary, as well as a look forwards to opportunities for further research.

Chapter 8: Validation of the QUADRAD Framework. This chapter presents the results of three industrial case studies that have been performed to validate the hypothesis and research contribution of this work. In particular, it describes the validation goals and introduces the pilot projects used to investigate the hypothesis. It presents a GQM (Goal Question Metric) framework for setting up the validation and the respective metrics. The chapter briefly describes the main steps of the validation approach. It gives a comprehensive overview of the validation results obtained in the three case studies and the lessons learned. In addition, it provides a discussion, interpretation, and conclusion of the results.

Chapter 9: Summary and Outlook. This chapter provides a summary of the research problem investigated in this work, the major research contributions, and the achieved validation results. In addition, it gives an outlook on promising topics for further research.

Appendix

Appendix: Architectural Patterns. The appendix provides a brief description of architectural patterns referenced in literature. The patterns can be examined for mechanisms which can be used during architecture design.

Acronyms. The acronyms section describes the abbreviations used in this work.

Glossary. The glossary section defines important terms introduced in this document.

2 Related Work in Architecture Development

This chapter discusses related work in architecture development. Architecture development is the phase of a software or systems engineering process in which the software and system architecture is created and evolved. The architecture of a system is the first artifact, which defines *how* the requirements of that system shall be achieved. It manifests important decisions from the development effort and provides the fundamental basis for detailed design and implementation. Important quality attributes such as performance, modifiability, safety, and reliability are largely promoted or hindered by the architecture. A good architecture development is thus a prerequisite for creating systems that meet business goals and customer expectations.

Throughout this document, we will typically talk about software architecture, but this clearly includes a consideration of non-software aspects such as hardware devices, bus systems, and even mechanical components. Since this thesis is focused on software-intensive systems however, the software architecture is the primary concern.

The chapter is organized as follows: Section 2.1 introduces basic principles of software architecture and defines important terms of architecture development. In Section 2.2 architecture design is described in more detail. In particular, functional and quality requirements which serve as an input to architecture design as well as tradeoffs among requirements are discussed. Additionally, architectural views and decisions which are part of the architecture description that results from design are introduced. The section further covers a description of architectural strategies and mechanisms which are frequently applied to drive the design. Moreover, it provides a comprehensive overview of existing architecture design methods and compares them with respect to the research gaps. Section 2.3 introduces architecture evaluation. It discusses architectural risks and describes scenarios, use case maps, and questioning techniques which can be used to support an evaluation. The section also deals with a comprehensive overview of existing architecture evaluation methods. The methods are analyzed concerning the research gaps of this thesis in order to check if they can be approved. Finally, Section 2.4 summarizes the key issues of this chapter.

2.1 Software Architecture and Architecture Development

The study of *software* architectures has been a subject of increasing industrial and academic attention over the past decade. Its roots reach back more than 30 years to the work of Parnas, Brooks, and Dijkstra, but the intense attention of the last couple of years is new. Why is this? Quite simply, this field is of interest because it fills a gap that existed in development practices for large software-intensive systems. The gap existed between requirements engineering and the detailed design stage of development.

Once the requirements for a project are understood well enough to continue with design, there is a temptation to proceed directly to detailed design and coding. In small to medium sized projects, this technique can work well, particularly if the staff is familiar with the application domain. For many years this was all that was needed for most system development. In large development efforts or in unprecedented development, however, this technique does not work. The kind of human ingenuity and attention to detail that enables us to create reasonable small- and medium-sized systems does not scale up to large systems. What is needed is an intermediate level of design, and this is the level of architecture design. In projects where architecture design was not explicitly identified as an area of study and practice, the software architecture evolves in an *ad hoc* manner, resulting in systems that nobody could understand anymore and that fail to achieve stability and other important quality attributes.

2.1.1 Basic Principles

Today’s knowledge about software architecture is a result of the continuous evolution of software development into a more mature engineering discipline. This evolution took place during the last couple of decades. During these decades, two key principles that largely influence our current understanding of software architecture have been identified and investigated. These principles are abstraction and structure.

Abstraction is necessary in order to improve the understanding of larger and more complex software systems. This is one reason why higher-level programming languages and the concept of abstract data types have been developed in the 1950s [Shaw 1996]. Dahl and Nygaard [Dahl 1966] extended these ideas and proposed the integration and encapsulation of state information for conceptual objects along with operations on these objects. This work was the basis for the development of object-oriented programming languages (e.g., Smalltalk, Eiffel, and C++) and other object-oriented technologies during the 1970s and 1980s [Berard 2001], as well as for component-based software development that emerged during the 1990s [Szyperski 1998].

Structure matters and one can gain various benefits if a software system is structured in the right way. Based on this observation, approaches for structured programming were developed in the 1960s. In the 1970s the concept of information hiding was introduced [Parnas 1971] and criteria for decomposing software systems into modules were proposed and analyzed (e.g., [Parnas 1972] and [Parnas 1974]). In 1976, DeRemer and Kron claimed that “structuring a large collection of modules to form a system is an essentially distinct and different intellectual activity from that of constructing the individual modules [DeRemer 1976].” Their work was the basis for the development of a new class of languages, the module interconnection languages (MILs). MILs specify what the modules of software systems are and how they can be combined [Prieto-Diaz 1986]. Thus, they build the basis for today’s thinking about software architecture.

The need for a discipline of software architecture emerged in the late 1980s and the early 1990s (e.g., [Shaw 1989] and [Perry 1992]). Active research in software architecture as the next step along these lines became very prominent during the 1990s. However, software architecture and architecture development is a growing but still young discipline [Bass 2003].

2.1.2 Definitions of Software Architecture

What exactly is software architecture? Unfortunately, there is no universal definition of this term, although countless authors defined the term "software architecture" during the last decade to emphasize specific characteristics or to build a foundation of their work (e.g., [Perry 1992], [Garlan 1993], [Garlan 1995], [Gacek 1995], [Booch 1999], [IEEE 1999], [Bosch 2000], [Jazayeri 2000], [Bass 2003]). The Software Engineering Institute [SEI 2004] provides a comprehensive overview of definitions proposed in textbooks and research papers. However, there are similarities among many of these definitions.

For example, in [Perry 1992] software architecture is defined as "*elements, form, and rationale. That is, a software architecture is a set of architectural [...] elements that have a particular form. We distinguish three different classes of architectural elements: processing elements, data elements, and connecting elements. [...]*"

In [Garlan 1993] software architecture is characterized as "*beyond the algorithms and data structures of the computation*" and later it is defined in [Garlan 1995] as "*the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.*"

Booch, Rumbaugh, and Jacobson [Booch 1999] defined the term software architecture as "*the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements [...]*".

The IEEE Recommended Practice for Architectural Description [IEEE 1999] defines software architecture as "*the fundamental organization of a system embodied in its components, their relationships to one another, and to the environment, and the principles guiding its design and evolution.*"

These and other similar definitions take a largely *structural* perspective on software architecture. They hold that software architecture is composed of elements, connections among them, and, usually, some other aspects such as constraints or design rationales. These perspectives do not preclude one another; nor do they represent a fundamental conflict. Instead, they represent a spectrum in the software architecture community about the emphasis that should be placed on architecture. This emphasis is also clearly expressed in a definitive book about software architecture [Bass 2003]. In this book the software architecture of a program or computing system is defined as "*the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.*" This definition has several interesting implications that can help us to understand the concept of software architecture in more detail.

First, *architecture defines software elements*. The architecture embodies information about how the elements relate to one another. This means that it specifically omits certain information about elements that does not pertain to their interaction. Thus, an architecture is foremost an *abstraction* of a system that suppresses details of elements that do not affect how they use, are used by, relate to, or interact with other elements. In nearly every modern system, elements interact with each other by means of *interfaces* that partition details about an element into public and private parts. Architecture is concerned with the public side of this division; private details – those having to do solely with internal implementation – are not architectural.

Second, the definition makes clear that systems can and do comprise *more than one structure* and that no one structure can irrefutably claim to be *the* architecture. For example, all nontrivial projects are partitioned into implementation units; these units are given specific responsibilities and are frequently the basis of work assignments for programming teams. This type of element comprises programs and data that software in other implementation units can call or access, and programs and data that are private. In large projects, these elements are almost certainly subdivided for assignment to subordinate teams. This is one kind of structure often used to describe a system. It is very static in that it focuses on the way the system's functionality is divided up and assigned to implementation teams. Other structures are much more focused on the way the elements interact with one another at runtime to carry out the system's function. Suppose the system is to be built as a set of parallel processes. The processes that will exist at runtime, the programs in the various implementation units described previously that are strung together sequentially to form each process, and the synchronization relations among the processes form another kind of structure often used to describe a system.

Third, the definition implies that *every computing system with software has a software architecture* because every system can be shown to comprise elements and the relations among them. This does not mean that the architecture is known to anyone. Perhaps all the people who designed the system are long gone, the documentation has vanished, the source code has been lost, and all we have is the executable binary code. The system still has its software architecture. This reveals the difference between the architecture of a system and the representation of that architecture. Unfortunately, an architecture can exist independently of its description, which raises the importance of architecture documentation.

Finally, the *behavior of each element is part of the architecture* insofar as that behavior can be observed or discerned from the point of view of another element. Such behavior is what allows elements to interact with each other.

In summary, the definition in [Bass 2003] not only clarifies that an architecture is composed of elements and connections (as advocated in the former definitions) but also that these elements and connections can appear in a variety of forms with different semantics, thus creating multiple structures that show different aspects of the architecture.

2.1.3 Definitions of Architecture Development

Generally speaking, *architecture development* is the task of a software or system development process in which the (software and/or system) architecture of the system is created and evolved. Unfortunately, there is no uniform definition of this phase in literature. However, many authors advocate that there is a development phase in which activities such as architecture definition, architecture description/documentation, and architecture quality attribute analysis for large and complex systems are performed (e.g., [Garlan 1993], [Booch 1999], [Bosch 2000] and [Bass 2003]). Some of those authors emphasize the existence of such a phase by speaking of an architecture-centric development process (e.g., [Booch 1999] and [Bosch 2000]).

The Rational Unified Process (RUP) [Booch 1999] defines two primary artifacts in the RUP Analysis and Design workflow to emphasize the importance of architectural considerations: (1) the architecture description, which captures the architectural structures relevant for the project and (2) the architectural prototype, which serves to validate the architecture and serves as a baseline for the rest of the development. The RUP also includes activities for identifying design elements, for describing their distribution, and for reviewing the architecture [Booch 1999].

Other authors discuss the reconstruction of the architectural description from an already implemented system as a practice that can complement architecture development (e.g., [Harris 1995], [Kazman 1998], [Bowman 1999], and [Bass 2003]). Their opinion is that since the architecture description is often missing in real projects there is a strong need for recovering it from code structures of the system in order to allow maintenance, analysis or mining for existing software assets.

As these references show, a dedicated process for developing architectures of large and complex software-intensive systems is identified by many researchers and practitioners and it is seen as essential. Looking at the proposals the authors made about specific activities of this process we can identify practices

- for the *design* and documentation of the architecture and
- for *evaluating* the architecture with respect to particular properties (e.g., quality attributes).

During architecture design a candidate architecture is created based on essential requirements. This architecture is analyzed or reviewed in architecture evaluation and the results of this task are used for revising and improving the architecture during successive iterations of architecture design. The practices will be further elaborated in the remainder of this chapter.

2.2 Designing the Architecture

In the preceding sections we have reflected the definitions of architecture and architecture development. In this section we will provide an introduction to designing the architecture of a software-intensive system.

The goal of *architecture design* is to create an architecture (description) that satisfies the requirements of the system. Architecture design is commonly seen as the most critical design activity in system development since it is concerned with fundamental decisions that affect the system’s major quality attributes (e.g., [Shaw 1996], [Booch 1999], [Hofmeister 2000], [Bass 2003]). Quality attributes such as performance, safety, and reliability are often impossible to “correct” or build in during later development phases (e.g., implementation). The decisions made during architecture design define the constraints on detailed design and implementation. An implementation, for example, exhibits an architecture if it conforms to the structural design decisions described by the architecture. This means that the implementation must be divided into the prescribed elements, the elements must interact with one another in the prescribed fashion, and each element must fulfill its responsibility to the others as dictated by the architecture.

Many authors distinguish architecture design from detailed design by putting emphasis on the fact that the former is related to design decisions that affect the “overall system structure” or “architectural elements” (e.g., [Kruchten 2000], [Hofmeister 2000], [Malan 2002], [Bass

2003]). Bass et al. [Bass 2003], for example, note that "architecture *is* design, but not all design is architecture. That means, many design decision are left unbound by the architecture and are postponed to detailed design and implementation. The architecture establishes constraints on downstream activities, and these activities must produce artifacts – finer-grained designs and code – that are compliant with the architecture."

The definition of software architecture given in Section 2.1.2 says that an architecture comprises "software elements, the externally visible properties of those elements, and the relationships among them." We can deduce from this definition that if a design decision affects a software design element that is not visible to other design elements, then that element is not an "architectural element" – i.e., it is not important for architecture design. The selection of data structures, along with the algorithms to manage and access that data structure, is a typical example of the internal design of an element not visible to other elements. It is thus not of architectural importance.

These statements clearly reflect the fact that architecture design is a design activity performed during an early stage of development, but it does not define the implementation of the system. Instead, architecture design is concerned with building the infrastructure of the system [Clements 2002]. The infrastructure deals with aspects such as how design elements communicate, how they pass or share data, how they initialize, shut down, self-test, report faults, and so on. During early architecture design the architect makes design decisions in order to constitute (major parts of) the system's infrastructure. These decisions document the architecture's approach to achieving the most important quality attributes of the system. In later design the infrastructure is refined and application functionality is added. If the early design decisions were "appropriate" this should be done without changing the general system structure. If the infrastructure cannot be preserved then the architecture needs to be restructured which typically results in a cost- and labor-intensive extra development effort. This is why many authors argue that architecture design is one of the most critical development activities (e.g., [Shaw 1996], [Booch 1999], [Bosch 2000], [Hofmeister 2000], [Bass 2003]).

Figure 2-1 illustrates the major input and output data flow of architecture design. In the following subsection, we will particularly explore the input artifacts to architecture design, the output artifacts from architecture design, design support concepts, as well as architecture design methods in more detail.

2.2.1 Inputs to Architecture Design

The primary input of architecture design is the *requirements specification*. The requirements specification documents the set of requirements that the system must achieve. According to [IEEE 1998b] a *requirement* is:

(1) A condition or capability needed by a user to solve a problem or achieve an objective.

(2) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification or other formally imposed documents.

(3) A documented representation of a condition or capability as in (1) or (2).

The definition includes the aspect that a requirement defines a desired system characteristic important to the user (or customer). This involves the fact that the user may not accept a system if some of the characteristics are missing. Consequently, the user would not pay for the system if the requirements were not met.

It is widely agreed that the requirements specification should state *what* a system should do and not *how* it should do it (e.g., [Davis 1990], [Loucopoulos 1995], [Sommerville 1995], [Pohl 1996], [Wiegers 2003]). If the specification describes both hardware and software, it is called a system requirements specification; if it describes only software, it is called a software requirements specification (cf. [IEEE 1998b]). Based on the nature and complexity of the system under development, a system requirements specification may contain mechanical requirements in addition to electrical and physical requirements.

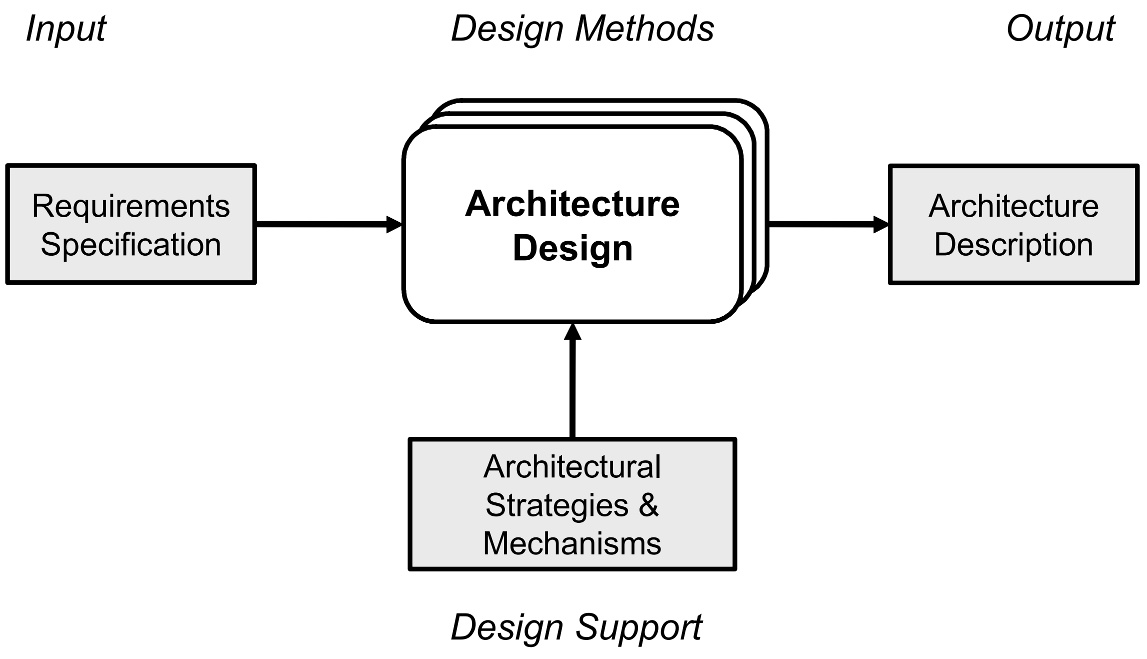


Figure 2-1: Overview of Architecture Design

The requirements specification should state both the functional and non-functional requirements of the system (e.g., [Davis 1990], [Sommerville 1995], [Kruchten 2000]). Non-functional requirements are often called quality attribute requirements (e.g., [Sommerville 1995], [Bass 2003]). In the remainder of this work, we will use the term quality attribute (requirement) to refer to non-functional requirements that address quality characteristics of a system such as modifiability, safety, reliability, and performance.

Next, we explore different types of requirements in more detail. We start the discussion by considering the difference between functional and quality issues of a system.

2.2.1.1 Functional and Quality Requirements

As introduced above a requirement is a desired system characteristic the user or customer cares for. When a user receives a new system, he or she is primarily interested in two things:

(1) Does the system solve the intended problem?

(2) How well does it solve the problem?

The first question is related to the *functionality*, the second to the *quality* of the system. For example, from a word-processing system the user expects that it would support him or her in writing a text, formatting it, storing it, and printing the text on demand. However, the user might also expect that the system has a graphical user interface that is easy to learn, that it fits on a single 1.44 MB diskette, and that it can be used both on his home computer running a Windows® Operating System and his PDA (Personal Digital Assistant) running the Palm® Operating System. Each of these expected "system features" might have been requirements of the user. The former describe typical *functions* of a word-processing system (typing a text, formatting it, storing and printing it), the latter describe special *quality attributes* the user expects from the system, namely usability, resource efficiency, and portability.

The [IEEE 1990] defines the term quality as follows:

(1) The degree to which a system, component, or process meets specified requirements.

(2) The degree to which a system, component, or process meets customer or user needs or expectations.

Taking this definition into consideration, the quality of a software system can loosely be characterized as "fitness for its intended use." It is widely agreed that quality attributes such as modifiability, portability, performance, safety, security, and usability largely determine a system's fitness for its use (e.g., [Bosch 2000], [Hofmeister 2000], [Kruchten 2000], [Clements 2002], [Bass 2003]).

Many authors treat functionality and quality attributes as largely orthogonal (e.g., [Hofmeister 2000], [Kruchten 2000], [Clements 2002], [Bass 2003]). In [Bass 2003], for example, functionality is defined as "ability of the system to do the work for which it was intended. A task requires that many or most of the system's elements work in a coordinated manner to complete the job, just as framers, electricians, plumbers, drywall hangers, painters, and finish carpenters all come together to build cooperatively a house. Therefore, if the elements have not been assigned the correct responsibilities or have not been endowed with the correct facilities for coordinating with other elements […], the system will be unable to offer the required functionality. […] In fact, if functionality were the only requirement, the system could exist as a single monolithic module with no internal structure at all. Instead, it is decomposed in modules to make it understandable and to support a variety of other purposes."

This statement clearly drives the common understanding that there *is* a difference between achieving functional and quality requirements. Meeting quality attributes such as modifiability, performance, and safety requires a comprehensive system design expertise [Maier 2000]. Quality attributes requirements are among the hardest to satisfy during architecture design and subsequent development (e.g., [Hofmeister 2000], [Bass 2003]).

According to the discussion above, we can now give the following characterization for functional and quality requirements:

- **Functional requirements:** Functional requirements describe *what* the system must do, without considering physical constraints. They express typical actions the system must perform and hence characterize the principal behavior of the system. The behavior that is specified by functional requirements manifests the functionality of the system.
- **Quality requirements:** Quality (attribute) requirements specify requirements that the system must fulfill to achieve particular quality attributes such as performance, safety, security, or modifiability. In other words, quality requirements describe *how well* the functional capabilities of the system are satisfied, where "how well" is judged by some externally observable/measurable property of the system behavior, not its internal implementation. Ideally, the "how well" is justified in terms of some characteristics that users of the system can value or are concerned about. For example, the quality attribute performance refers to the responsiveness of the system. The achievement of performance requirements may be perceived by the user either by the time required to respond to specific events or the number of events processed in a given interval. Quality requirements can be identified during requirements elicitation by stating questions about how the system shall perform the required functions: *How fast* must a function be performed? *How reliable* must a function be? *How modifiable* must a function be implemented?

In the next section, we will discuss different quality attributes which an architect has to cope with during architecture design. This will introduce us to requirements covering quality aspects and help our understanding of their implications for architecture design.

2.2.1.2 Quality Attributes

A first step commonly used to characterize system quality more precisely is to break it down into a set of more concrete quality attributes (e.g., [ISO 1992]). Quality attributes have been of interest to the software and systems engineering community since the early seventies. Different quality models have been proposed to link quality attributes to the measurable and observable properties of a system, for example [McCall 1977], [Boehm 1978], [ISO 1992], [IEEE 1998], and [Bass 2003]. There are also a variety of published taxonomies and definitions, and many of them have their own research and practitioner community (e.g., performance [Smith 2002], availability and reliability [Laprie 1992], and safety [Randell 1995]). Each quality attribute addresses a particular quality characteristic of the system. Some quality attributes are observable via execution (e.g., performance, security, availability, and usability) [Bass 2003]. The achievement of others (e.g., modifiability) is typically addressed in reviews (e.g., [Kazman 1996], [Clements 2002]).

Table 2-1 summarizes typical quality attributes and sub-attributes that can be found in literature (e.g., [ISO 1992], [IEEE 1998], [Bass 2003]). The quality attributes are defined as follows:

Table 2-1: Quality Attributes

Quality Attributes	**Sub-Attributes**
Modifiability	Extensibility
	Adaptability
	Integrability
	Configurability
	Maintainability
Portability	--
Interoperability	--
Security	Confidentiality
	Integrity
	Authentication
Dependability	Availability
	Reliability
Safety	--
Performance	Execution Efficiency
	Storage Efficiency
Usability	Learnability
	Memorability
	User Error Avoidance
	User Error Handling
	Interaction Efficiency
	Satisfaction

- **Modifiability.** Modifiability considers how the system can accommodate anticipated and unanticipated changes and is largely a measure of how changes can be made locally, with little ripple effect on the system at large (e.g., [Buschmann 1996], [Bosch 2000], [Bass 2003]). Modifiability requirements usually include attributes such as extensibility, adaptability, integrability, configurability, and maintainability. Modifiability requirements can be classified according to their origin as internal and external modifications. Internal modifications are driven from within the organization and include enhancement of capabilities, extension of capabilities, and deletion of unwanted capabilities [Bass 2003]. External modifications are driven by independent external organizations and comprise the introduction of new or improved standards, new COTS versions, and changes of the environmental interfaces [Bass 2003].
- **Portability.** Portability is the ability of a system to run under different computing environments [Bass 2003]. These environments can be hardware, software, or a combination of the two. A system is portable to the extent that all of the assumptions about any particular computing environment are confined to a small number of dedicated components [Bass 1998]. Portability makes a system more flexible in how it can be fielded, appealing to a broader customer base, and is thus a special case of modifiability. Typical portability cases are the adaptation of a system to a new hardware and/or software environment [Bass 1998].
- **Interoperability.** Interoperability measures the ability of a system, or of a group of parts constituting a system, to work with another system [Barbacci 2000]. This includes the exchange of information and the interpretation and usage of the information that has been exchanged. Interoperability sometimes exploits compatibility, which is defined as the ability of two or more systems or components to perform the required functions while sharing the same hardware or software environment [C4ISR 1998]. Interoperability affects system procedures for data exchange (e.g., security policy) and the infrastructure of the exchange (e.g., communication, system services, and hardware).
- **Security.** Security is the ability to resist unauthorized attempts at usage and denial of service while still providing its services to legitimate users (e.g., [Howard 2001], [Ramachandran 2002], [Bass 2003]). This includes the protection of system data against disclosure, modification, or destruction as well as the protection of computer systems themselves and the property that a particular security policy is enforced. Security includes the following sub-attributes: confidentiality, integrity, and authentication [Bass 2003]. Confidentiality is concerned with keeping information hidden from all but the intended viewers. Integrity deals with keeping information intact or at least being able to detect whether information has been altered. Authentication relates to identifying the origins of information.
- **Dependability.** Dependability is that property of a computer system such that reliance can justifiably be placed on the service it delivers (e.g., [Laprie 1992], [Randell 1995]). Dependability includes sub-attributes such as availability and reliability. Availability measures the proportion of time the system is up and running [Laprie 1992]. It is thus concerned with the system's readiness for usage. Reliability is the ability of a system to keep operating over time [Laprie 1992].
- **Safety.** Safety is defined to be the absence of catastrophic consequences on the environment [Randell 1995]. In [Leveson 1995] it is defined as freedom from accidents and loss. This leads to a binary measure of safety: a system is either safe or it is not safe. Safety is a property of a computer system such that reliance can justifiably be placed in the

absence of accidents. Whereas dependability is concerned with the occurrence of failures, defined in terms of internal consequences (services are not provided), safety is concerned with the occurrence of accidents or mishaps, defined in terms of external consequences (accidents happen). The difference of intents gives rise to the following paradox: If the services are specified incorrectly, a system can be dependable but unsafe; conversely, it is possible for a system to be safe but undependable. A system might be dependable but unsafe — for example, an avionics system that continues to operate under adverse conditions yet directs the aircraft into a collision course. A system might be safe but undependable — for example, a railroad signaling system that always fail-stops. Both, the definitions of [Randell 1995] and [Leveson 1995] treat safety as freedom from damage (e.g., death or injuries). However, this is not in accordance with the safety notion of the traditional safety engineering community (e.g., [Spellman 2004]) for which safety is freedom from danger, which is defined as a situation in which the risk is higher than the highest risk that is still justifiable.

- **Performance.** Performance is the degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage [Barbacci 2000]. Thus, performance refers to the response time or throughput as seen by the users [Smith 2002]. Responsiveness limits the amount of work processed, so it determines a system’s effectiveness and productivity of its users. Responsiveness is not limited to normal operation but may include maintenance tasks like failure recovery and so on. While responsiveness is primarily concerned with the speed (execution efficiency) of a system it should be noted that performance is often influenced by the amount of memory (e.g., RAM or hard disk space) that is consumed (storage efficiency). For example, memory limitations may influence responsiveness of multitasking operating systems, especially if memory data must be physically stored on a significantly slower hard drive (e.g., in case of a process context switch). Thus, memory required for an executing software system might also be considered as a performance requirement, as it relates to system’s run-time behavior and as it also affects the speed of system operations.
- **Usability.** Usability is the ease with which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component [Bass 2003]. It is a measure of how well users can take advantage of some system functionality. Usability concerns the following sub-attributes (e.g., [Bass 1998], [Isensee 2000]): Learnability, memorability, user error avoidance, user error handling, interaction efficiency, and satisfaction. Learnability is concerned with the time and level of difficulty for a user to learn to use the system’s user interface and functionality. Memorability deals with the time and ability to remember how to do system operations between uses of the system. User error avoidance is the system’s anticipation and prevention of common user errors. User error handling copes with the ability of the system to help the user recovering from errors. Interaction efficiency is the ability of the system to respond with appropriate speed to a user’s requests. Finally, satisfaction is concerned with the ability of the system to make the user’s job “easy”. Making a system’s user interface clear and easy to use is primary a matter of getting the details of a user’s interaction correct. This is supported by matching the interface to the user’s mental model of the system, using familiar metaphors, standards, and interface conventions, providing consistency in the user interface, dealing with aesthetics, providing adequate performance, and allowing the user to feel that they are in control. Note that user error handling and interaction efficiency have safety and performance implications, respectively.

2.2.1.3 Quality Attribute Tradeoffs

When adopting a system perspective one can observe that many of the quality attributes presented in the previous section trade off against one another; that is, if requirements pertaining to one quality attribute are changed, other quality attributes (and associated requirements) will also be affected by this change. In particular, it is not possible to maximize all quality requirements at the same time. In an early paper on software quality, Boehm and his colleagues have identified this tradeoff as an important problem [Boehm 1978]:

"The major problem is that many of the individual characteristics of quality are in conflict; added efficiency is often purchased at the price of portability, accuracy, understandability, and maintainability."

The reason for this problem is that the decisions a system architect makes during system design to meet particular requirements often influence more than one quality attribute (e.g., [Clements 2002]). If the system architect changes one quality requirement by modifying the corresponding design decisions, other quality attributes might also be affected. Increasing the performance of a system by bridging abstraction layers, for example, will decrease the modifiability of the system.

Tradeoffs among quality attributes are the reason for quality requirement conflicts. A quality requirement conflict occurs if it is difficult or impossible to satisfy two or more quality requirements at the same time. One major problem in this context is that requirement conflicts usually are not obvious, especially when they are related to complex design dependencies.

Different authors discuss competing quality attributes (e.g., [McCall 1977], [Boehm 1978], [McCall 1994], [Chung 1995], [Kitchenham 1996], [Bass 1998], [Barbacci 2000], [Thiel 2001c]). Table 2-2 shows typical relationships among quality attributes identified during a study conducted in [Thiel 2001c]. A cross (✕) represents a conflict between two qualities. Qualities that usually support one another are marked by a dot (●). A blank means that there is no dependency between two qualities or that the dependency is up to a specific application context. By giving counter examples the authors emphasize that the relationships are not strict measures that apply in every case. They also point out that conflicting requirements should be treated carefully since it is very likely that they will require tradeoff decisions during architecture design.

Table 2-2: Typical Relationships Between Quality Attributes [Thiel 2001c]

Quality Relationships	Modifiability	Portability	Interoperability	Security	Safety	Dependability	Performance	Usability
Modifiability								
Portability	●							
Interoperability	●							
Security	✕		✕					
Safety				●				
Dependability				●	●			
Performance	✕	✕	✕	✕		✕		
Usability				✕	●		●	

Unfortunately, relatively little attention has been paid to the process of systematically dealing with conflicts in quality requirements with the result that this research field is still immature. However, goal-oriented approaches such as the NFR-Framework [Chung 1995] that treat quality requirements as potentially conflicting or synergistic goals to achieve, while considering development alternatives, which could meet the stated quality goals seem to be promising. Besides, there are also research tools that support the identification and documentation of quality requirement conflicts during requirements engineering (e.g., [Boehm 1996], [Chung 1999]).

2.2.1.4 Architectural Drivers

Architectural drivers are those requirements of a requirements specification that are most relevant for architecture design (e.g., [Jazayeri 2000], [Bass 2003]). The existence of architectural drivers is based on the fact that some requirements are more important for the design of an architecture than others and that it is good practice to start designing the system by considering those requirements first.

Architectural drivers are important for achieving the business goals of a system development effort [Bass 2003]. Typically, the drivers are a subset of the quality requirements but they can also cover functional aspects which provide essential services of the system [Bass 2003]. Figure 2-2 illustrates the candidate set of architectural drivers within a system's requirements specification.

Other authors such as Jazayeri et al. [Jazayeri 2000] define architectural drivers as requirements which address infrastructure aspects of a system. They propose to seek for requirements that address aspects such as system configuration, software upgrade/update, and failure management since these requirements usually make up a system's infrastructure [Jazayeri 2000].

Karlsson and Ryan [Karlsson 1997] focus on the cost and value of requirements in order to identify those requirements that should be considered for the system's architecture. They interpret quality in relation to a candidate requirement's potential contribution to customer satisfaction with the resulting system. Their cost-value approach ranks candidate requirements in two dimensions: according to their value to customer and users, and according to their estimated cost of implementation (cf. [Karlsson 1997]). Karlsson and Ryan [Karlsson 1997] propose to apply the Analytic Hierarchy Process (AHP) [Saaty 1994] for ranking requirements. AHP is a decision support methodology which is based on pair-wise comparisons (e.g., [Saaty 1994]).

As these statements demonstrate there is no common definition of the term architectural driver. However, the authors agree on the fact that architectural drivers are requirements that are important for architecture design since they shape the architectural infrastructure (e.g., [Karlsson 1997], [Jazayeri 2000], [Bass 2003]). Bass et al. [Bass 2003] add that architectural drivers are also essential for achieving the business goals.

An interesting observation is that, there is currently no design method which is concerned with systematically judging requirements according to their relevance for architecture design (cf. Section 2.2.4). This issue is related to research gap 1 of this thesis (cf. Section 1.3.1) and will be further discussed in Section 2.2.5.

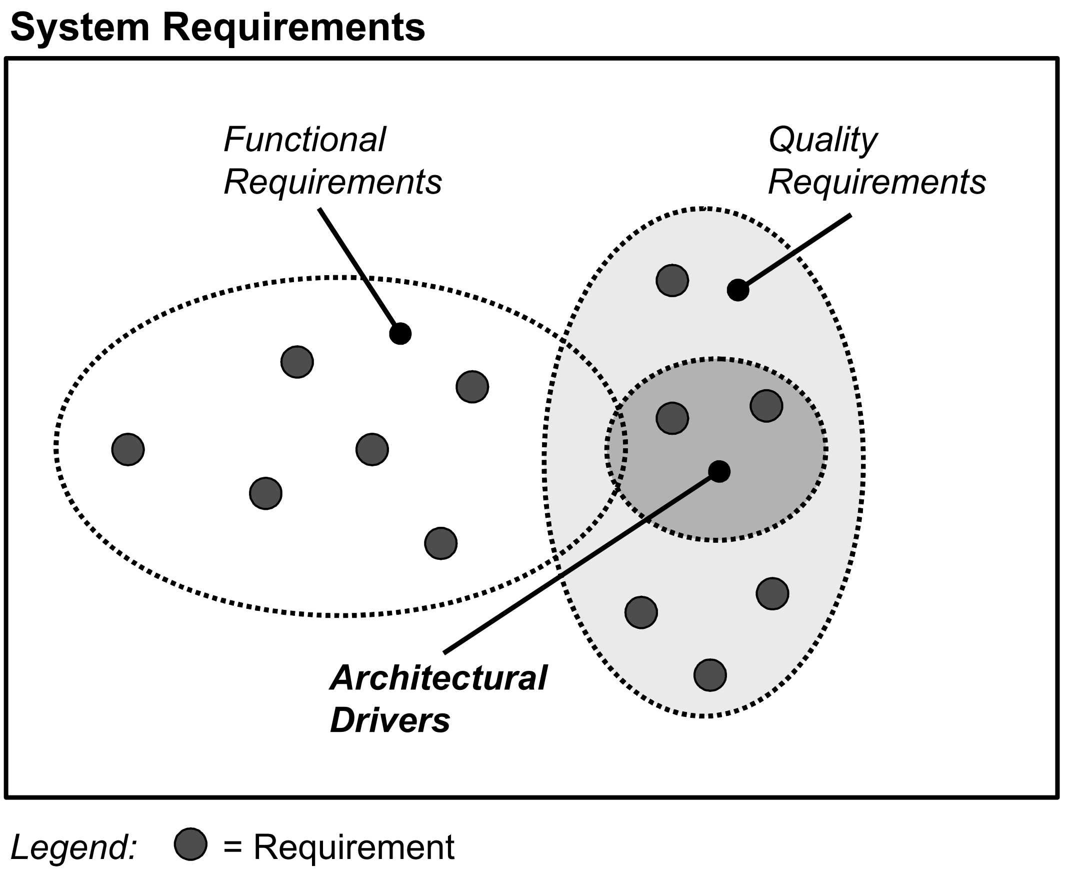


Figure 2-2: System Requirements and Architectural Drivers

2.2.2 Outputs from Architecture Design

The primary product of architecture design is the architecture documentation or *architecture description* (e.g., [Clements 2003]). The architecture description documents the (software and/or system) architecture and shows how the architect has implemented the requirements.

There are two fundamental approaches to capture an architecture's description: a formal and an informal approach (e.g., [Shaw 1996], [Clements 2003]). In the *formal* approach, the architecture description is documented in a formal Architecture Description Language (ADL), usually to increase the reusability of architectural designs, and to support analysis and simulation (e.g., [Garlan 1997]). Different ADLs have been proposed in literature (e.g., Wright [Allen 1994], UniCon [Shaw 1995], Rapide [Luckham 1995], Darwin [Magee 1996], ACME [Garlan 1997], and AADL [Feiler 2004]). A comprehensive comparison of ADLs is provided in [Medvidovic 2000]. In the *informal* approach the constituting elements and relationships of an architecture are recorded in a semi-formal way – typically by using a particular design notation with textual annotations (e.g., UML [Jacobson 1999]). We focus on the informal way of documenting architectures in the remainder of this work. ADLs are beyond the scope of this thesis (cf. Section 1.3.2).

Although an IEEE working group has created a recommended practice for architecture description [IEEE 1999], there is no established standard for documenting an architecture. However, many authors propose different *views* to document an architecture (e.g., [Zachmann 1987], [Kruchten 1995], [Booch 1999], [IEEE 1999], [Hofmeister 2000], [Malan 2002], [Clements 2003]). The use of different views naturally reflects the fact that an architecture consists of multiple structures, not only a single structure (cf. Section 2.1.2).

Besides the documentation of *what* the architecture is (architectural views), Clements et al. [Clements 2003] also propose to include information about *why* the architecture is the way it is. In this context, they advocate to document the reasoning behind *architectural decisions*, especially if decisions apply to more than one view. This is beneficial for a better understanding of the architecture design. Figure 2-3 illustrates the two concepts of architecture description in UML notation [Jacobson 1999]. We will explore architectural views and architectural decisions in the following subsections.

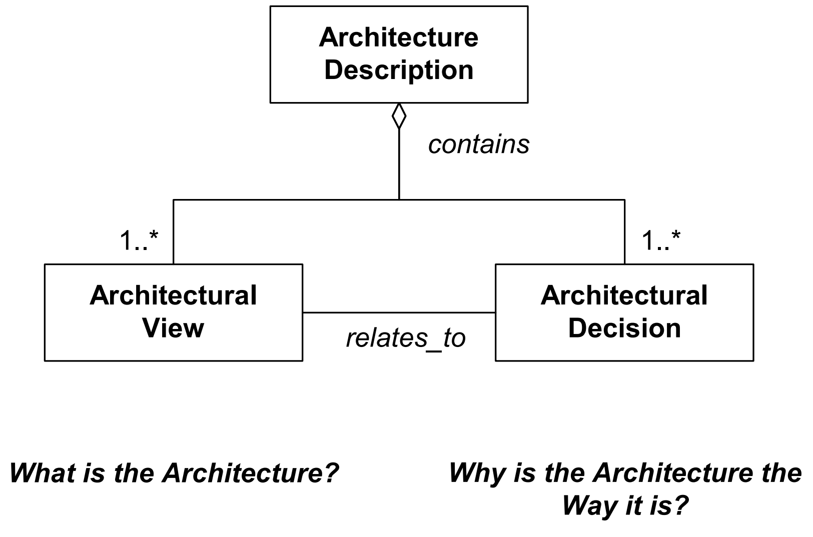


Figure 2-3: Elements of an Architecture Description

2.2.2.1 Architectural Views and View Types

Many stakeholders are interested in an architecture description. Kruchten [Kruchten 2000], for example, has conducted the following list of persons that have a stake in the architecture:

- The architect, who uses it to reason about evolution or reuse
- Users or customers, who use it to visualize at a high level what they are buying
- The software project manager, who uses it to organize teams and plan the development
- The designers, who use it to understand the underlying principles and locate the boundaries of their own design
- Subcontractors, who use it to understand the boundaries of their chunk of development
- The system specialist, who uses it to understand the technological constraints and risks
- Other development organizations, which use it to understand how to interface with it

To allow the various stakeholders communicating and reasoning about the architecture, the architecture must be represented in a way that is understandable for them and that reflects their concerns. For example, for a building, different types of blueprints are used to represent different aspects of the building's architecture: floor plans, elevation, electrical cabling, water pipes, central heating, and ventilation. Over decades, blueprints have evolved into standard forms such that each person involved in the design and construction of the building understands how to read and use them. Each of these blueprints tries to convey one aspect of the building's architecture for one category of stakeholders. The blueprints are not independent of each other but must carefully be coordinated (e.g., [Hofmeister 2000]).

Similarly, for the architecture of a software-intensive system, one can envisage various blueprints for different purposes. Examples from [Kruchten 2000] include blueprints

- To organize the functionality of the system
- To address concurrency aspects
- To describe the physical distribution of the software on the underlying platform

These blueprints are what the software architecture community calls *architectural views* (e.g., [Kruchten 1995], [IEEE 1999], [Hofmeister 2000], [Bass 2003].) An architectural view is a simplified description (or abstraction) of a system (architecture) from a particular perspective or vantage point, covering particular concerns and omitting concerns that are not relevant to this perspective [Kruchten 2000]. Views help reducing the perceived complexity through a separation of concerns. A view illustrates a particular structure of the architecture, and multiple views describe *the* architecture (as reflected in the definition of software architecture given in Section 2.1.2). Views are not "orthogonal" as concerns are normally overlapping [IEEE 1999]. But each view usually contains new information about the system. The dependency among views itself can be represented by dedicated views (e.g., [Clements 2003]).

Different authors have proposed a variety of architectural views to be used for architectural description (e.g., [Zachmann 1987], [Kruchten 1995], [Soni 1995], [Issarny 1998], [Booch 1999], [IEEE 1999], [Hofmeister 2000], [Clements 2003]). They recommend to use different modeling elements and relations for constructing a particular view as well as specific system concerns the view addresses. There are three key system aspects almost every author considers:

- *Structural* or static aspects, showing how the system elements are decomposed and relate to each other (e.g., logical [Kruchten 1995], implementation [Hofmeister 2000]), and module view [Kruchten 1995])
- *Behavioral* or dynamic aspects, showing the interactions among system elements (e.g., process [Kruchten 1995] and execution view [Hofmeister 2000])
- *Distribution* or allocation aspects, showing how the system elements are mapped to each other or to the environment (e.g., deployment [Kruchten 1995] and code view [Hofmeister 2000])

Structural, behavioral, and distribution views are called basic architectural structures or *view types* (e.g., [Bass 2003], [Clements 2003]). From these view types, a variety of architectural view styles that represent different concerns of a system can be derived. Clements et al. [Clements 2003], for example, introduce their own view type terminology. They introduce three view types, which they call module, component-and-connector, and allocation view type for representing structural, behavioral, and distribution aspects of an architecture. For each view type, they define, in turn, a set of sub-types (styles) which can be used to represent different concerns of the system. Table 2-3 shows a summary of their view type terminology.

Table 2-3: Architectural View Types [Clements 2003]

View Type	Style	Concerns
Module	Decomposition	Resource allocation; encapsulation; configuration control
	Uses	Dependencies among components; separation in processes/tasks
	Layered	Encapsulation; virtual machines
	Class	Instances of components; shared methods
Component-and-Connector	Client-Server	Distributed operation; separation of concerns; performance analysis; load balancing
	Concurrency	Thread analysis; resource contention analysis
	Shared Data	Data producer/consumer analysis
Allocation	Deployment	Allocation of software to hardware nodes; safety/security analysis
	Implementation	Configuration control; integration; test activities
	Work Assignment	Assignment of software components to development team; best use of expertise; management of commonality

Figure 2-4 shows the module view of a MP3 Player software architecture with subsystems and dependencies among them. The deployment view of the MP3 Player architecture is depicted in Figure 2-5. It shows how many software processes exist in the system and how these processes are allocated to hardware elements.

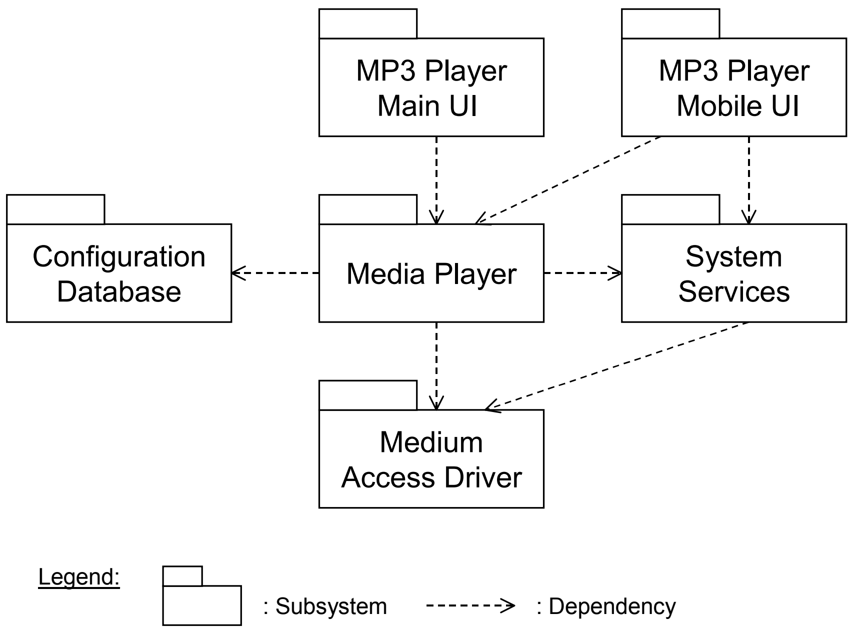


Figure 2-4: Module View of the MP3 Player Architecture

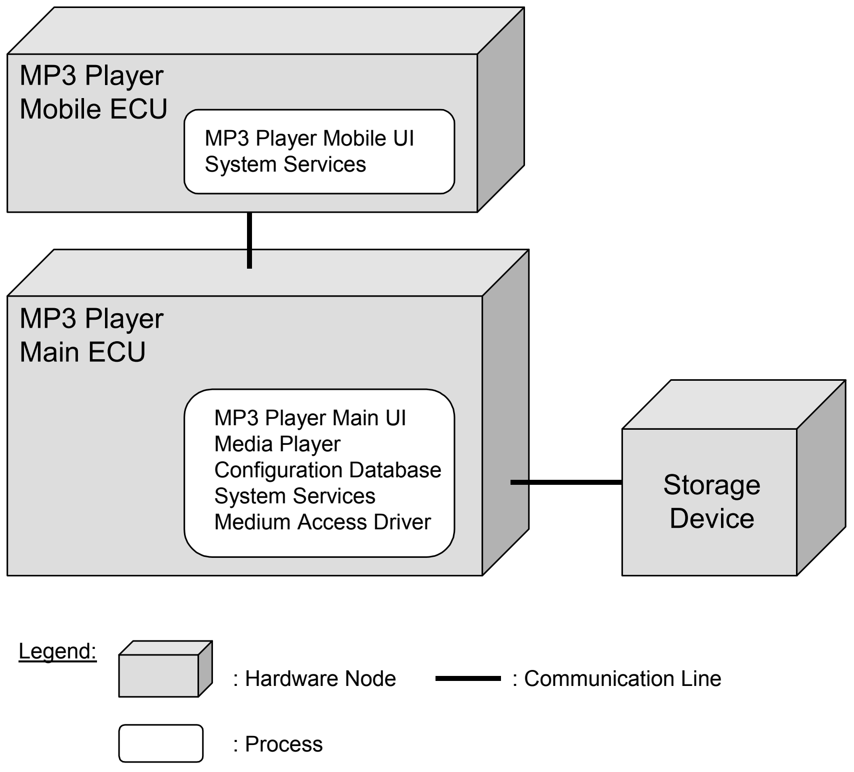


Figure 2-5: Deployment View of the MP3 Player Architecture

Different architectural views also expose different quality attributes (cf. Section 2.2.1.2). For example, a concurrency view [Clements 2003] enables to reason about the performance of the system. That is, the responsiveness of the system as well as the amount of memory that is consumed during runtime. Therefore, the quality attributes that are of concern to the stakeholders in the system's development will affect the choice of views to document. In other words, the set of relevant architecture views must be determined during architecture development. As we will see in Section 2.2.5, there is currently no architecture method that supports a flexible view construction/selection. Additionally, the application domain or complexity of the system can constrain the use of views (e.g., [IEEE 1999]). Therefore, usually not all possible architectural views must be described and views for a particular view types can be omitted [Clements 2003].

Furthermore, each architectural view embodies information about the name and responsibilities of the design elements and about how the elements relate to one another [Clements 2003]. This is also clearly reflected in the definition of software architecture (cf. Section 2.1.2).

An architecture design element's *name* is the primary means to refer to it (e.g., [Bachmann 2000]). It often suggests something about its role in the system: an element called "Engine Management," for instance, probably has little to do with numeric simulations of chemical reactions. In addition, an element's name may reflect its position in some decomposition hierarchy.

The *responsibility* for a design element is a way to identify its role in the overall system, and establishes an identity for it beyond the name (e.g., [Clements 2003]). Whereas an element's name may suggest its role, a statement of responsibility establishes a description of the services the element provides (e.g., [Hofmeister 2000]). Clements et al. [Clements 2003] recommend that the description of responsibilities should not overlap among different design elements since this is an indicator of misconception. A typical representation of responsibilities and collaborations of design elements is the Class-Responsibility-Collaborator (CRC) card [Beck 1989].

Design elements interact with one another by means of *interfaces* that partition details about an element into public and private parts [Clements 2003]. Architecture is concerned with the public information among elements [Bass 2003]. An interface of a design element describes which services can be accessed by other elements of the architecture (e.g., [Szyperski 1998]). An element may have zero, one, or multiple interface(s) [Szyperski 1998]. As mentioned above, some of the interfaces exist for internal purposes only. That is, they are used only by the subordinate design elements (e.g., software module) within the enclosing parent element (e.g., subsystem). They are not visible outside that context and therefore they are not related to the parent interfaces [Bass 2003].

2.2.2.2 Architectural Decisions

Architecture development is a decision making process. A software/system architect makes design decisions in order to satisfy the system's requirements (e.g., [Malan 2002]). The decisions the architect makes shape the architecture and finally result in different architectural views. For example, assume an architect has to satisfy the following portability requirement:

"The system's software portion shall be independent from the underlying hardware."

A decision of the architect could be to introduce a hardware abstraction layer into the software architecture. This may affect particular architectural views.
Figure 2-6 depicts the influence of the decision in the module view [Kruchten 1995]. As shown the structure of the system is changed by the decision. Note that the introduction of the *Hardware Abstraction* subsystem requires changes in the *Application* and *Communication* subsystems in order to allow a proper communication.

In practice, the decision making process during architecture design is highly complex. The architect has usually to cope with many competing decisions during architectural design (e.g., [Malan 2002]). Each decision leads to progress in design and thus in extensions and adaptation of architectural views. A big problem is that after some design progress is made usually many of the key decisions cannot be understood anymore by simply studying the created architectural views. In other words, a lot of the information why the architecture (or portions of it) is the way it is gets lost and cannot be reconstructed from the views anymore. This is critical because making key decisions (and documenting the rationales behind the decisions) explicitly supports understanding the main approaches the architect adopts to satisfy the most important requirements.

Clements et al. [Clements 2003] discuss further circumstances in which it is beneficial to document the rationale of a design decision:

- The design team spent significant time evaluating options before making a decision.
- The decision seems not to make sense at first blush but becomes clear when more background is considered.
- The issue is confusing to (new) team members.
- The decision has a widespread effect that will be difficult to undo.
- The architect thinks it is cheaper to capture it now than capturing it later.

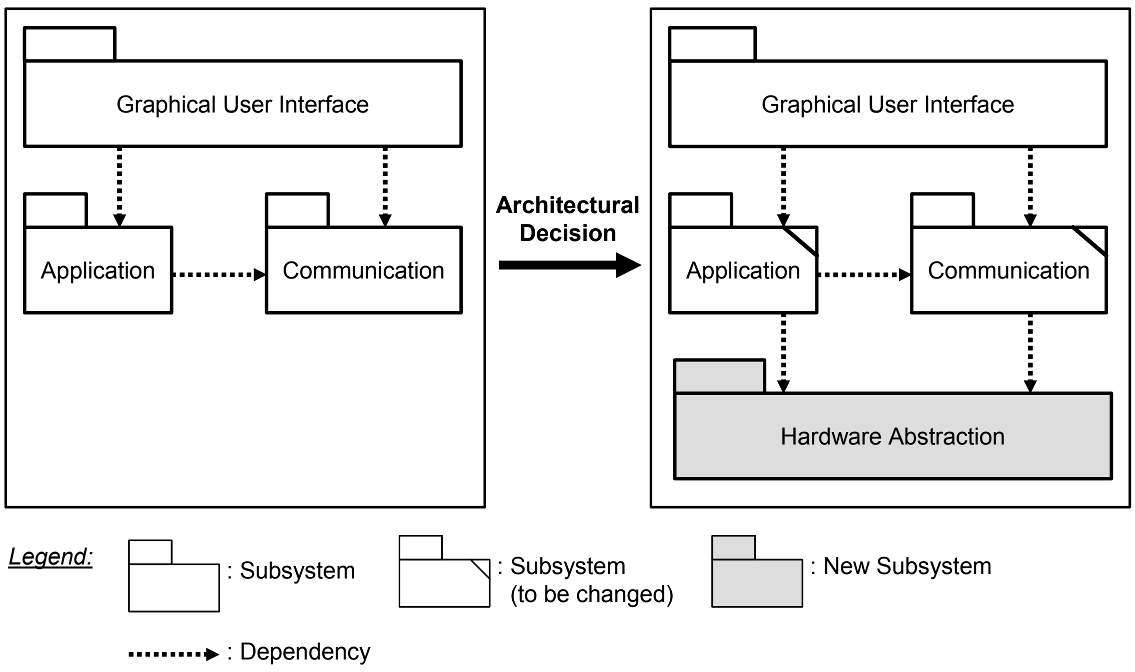


Figure 2-6: Architectural Decision and Architectural View

Thus, it is useful to complement the architecture description with a documentation of important design decisions (e.g., [Malan 2002]). Clements et al. [Clements 2003] propose to capture the following information for each decision:

- *Decision*: A summary of the design decision.
- *Constraints*: The key constraints that ruled out other possibilities. For example, a particular operating system or components from a legacy system must be used. Alternatively, a particular commercial component must be chosen because of the organization's relationship with its vendor.
- *Alternatives*: The options the architect considered and the reason for ruling them out. The reasons can be technical or non-technical, as noted in the constraints part of the description.
- *Effects*: The implications or ramifications of the decision. For example, what constraints does the decision impose to downstream developers, users, maintainers, testers, or builders of other interacting systems?
- *Evidence*: Any confirmation that the decision was a good one. For example, an explanation how the design choice meets the requirements (e.g., quality requirements) or satisfies constraints.

Like all documentation, the recording of design decisions and the reasoning behind these decisions is subject to a cost/benefit equation (e.g., [Hofmeister 2000], [Bass 2003], [Clements 2003]). Not every design decision comprising an architecture should be accompanied by rationales explaining them. The reason is that it is simply too time consuming and many design decisions may not be important enough to warrant documentation [Bass 2003]. The selection of the important decisions for documentation in the architecture description is lastly the responsibility of the architect. However, it is an essential activity of architecture design (e.g., [Hofmeister 2000], [Malan 2002]).

2.2.3 Design Support Concepts

In this section we discuss architectural strategies, patterns, and mechanisms which are important concepts to support the decision making process during architecture design.

2.2.3.1 Architectural Strategies

According to [Garland 2003], an *architectural strategy* is a collection of design decisions that support the achievement of a particular requirement. Architectural strategies represent design options for the architect. Bass et al. [Bass 2003] adopted the term tactics in order to describe design options that influence the control of quality attributes. Smith and Williams [Smith 2002] use the term design principles for describing options to address performance requirements of a system. However, we use the term architectural strategy throughout this work in order to denote an approach to solve a particular class of design problems that is related to a set of requirements.

Critical design problems are usually associated with quality attribute requirements (e.g., [Smith 1993], [Kruchten 2000], [Bass 2003]). Architectural strategies help the architect in getting started to addressing a design problem. For example, reducing the communication overhead between software components is an architectural strategy that an architect can use for addressing performance problems. Architectural strategies are different from architectural patterns (e.g., [Buschmann 1996]) as they do not capture a particular implementation of the solution concerning a design problem but rather provide an outline how to tackle the problem (e.g., [Ramachandran 2002], [Smith 2002], [Bass 2003). Architectural patterns thus make use of architectural strategies in order to approach a particular solution to a design problem [Bass 2003]. Architectural patterns and mechanisms are further discussed in Section 2.2.3.2.

Various authors have documented architectural strategies for different quality attributes (e.g., availability [Jalote 1994], security [Ramachandran 2002], and performance [Smith 1994]). Bass et al. [Bass 2003] provide a comprehensive summary of strategies for achieving availability, modifiability, performance, security, testability, and usability. Table 2-4 shows strategies for addressing availability problems [Bass 2003]. Ping/echo, for example, is a strategy that can be used for detecting faults and, thus, improving availability. It includes the following solution approach to fault detection:

Ping/echo. One design element issues a ping to another element and expects to receive back an echo. If the echo is not received within a predefined time then the issuing element will report that the element under scrutiny is faulty.

In summary, architectural strategies help an architect to reason about design decisions to address particular quality requirements adequately. Architectural strategies are broader defined than patterns and do not provide a concrete implementation of the solution.

Table 2-4: Architectural Strategies Addressing Availability [Bass 2003]

Category	**Architectural Strategy**
Fault detection	Ping/echo
	Heartbeat
	Exceptions
Fault recovery	Voting
	Active redundancy
	Passive redundancy
	Spare
	Shadow operation
	State resynchronization
	Checkpoint/rollback
Fault prevention	Removal from service
	Transactions
	Process monitor
	Generalize the module
	Limit possible options
Prevention of ripple effects	Hide information
	Maintain existing interfaces
	Restrict communication paths
	Use an intermediary

2.2.3.2 Architectural Patterns and Mechanisms

In contrast to architectural strategies, architectural patterns capture existing solutions to recurring design problems in a comprehensive format (e.g., [Buschmann 1996]). During the last decade, a lot of patterns and pattern-based approaches have been proposed (e.g., [Gamma 1995], [Shaw 1996], [Buschmann 1996], [Fowler 1997], [Douglas 1999], [Schmidt 2000], [Kircher 2004]). These approaches originate in the work of the building architect Christopher Alexander, who developed a pattern language for designing buildings and cities [Alexander 1977].

Several different formats have been used for describing patterns (e.g., [Gamma 1995], [Buschmann 1996], [Fowler 1997]). Despite the use of these formats, it is generally agreed that a pattern should contain certain essential elements. Regardless of the particular format/headings used, the following essential elements are clearly recognizable in a pattern description (e.g., [Buschmann 1996]):

- *Name:* A meaningful name.
- *Problem:* A statement of the problem which describes its intent: the goals and objectives it wants to reach within the given context and forces.
- *Context:* The preconditions under which the problem and its solution seem to recur, and for which the solution is desirable.
- *Forces:* A description of the relevant forces and constraints and how they interact/conflict with one another.
- *Solution:* Static relationships and dynamic rules describing how to realize the desired outcome.
- *Examples:* One or more sample applications of the pattern.
- *Resulting Context:* The state or configuration of the system after the pattern has been applied, including the consequences (both good and bad) of applying the pattern, and other problems and patterns that may arise from the new context.
- *Rationale:* A justifying explanation of steps or rules in the pattern in terms of how and why it resolves its forces in a particular way.
- *Related Patterns:* The static and dynamic relationships among this pattern and other patterns.
- *Known Uses:* Known occurrences of the pattern and its application within existing systems.

Architectural patterns include mechanisms that describe how the patterns implement the solution [Buschmann 1996]. According to [Booch 1996], these mechanisms make up the soul of a pattern. Booch [Booch 1994] originally introduced the term mechanism as “a structure whereby objects collaborate to provide some behavior that satisfies a requirement of the problem.” Essentially, a mechanism describes the core solution approach and structure how patterns address the design problem. Each mechanism is built on architectural strategies [Booch 1994]. We will use the term *architectural mechanism* in the remainder of this work in order to refer to the design solutions captured in an architectural pattern.

The description of an architectural mechanism includes a number of pre-defined design decisions together with a concrete implementation of those decisions (e.g., [Gamma 1995]). The more design decisions are codified in a mechanism, the more precise the properties of the system part based on this mechanism can be predicted. However, the more design decisions are pre-defined, the more restricted is the problem domain to which the mechanism can be applied. Figure 2-7 shows an example of the design structure of the broker pattern provided in [Buschmann 1996]. The mechanism included in the pattern is as follows:

A broker is a messenger that is responsible for the transmission of requests from clients to servers, as well as the transmission of responses and exceptions back to the client. A broker must have some means of locating the receiver of a request based on its unique system identifier. A broker offers APIs (Application Programming Interfaces) to clients and servers that include operations for registering servers and for invoking server methods.

When a request arrives for a server that is maintained by the local broker, the broker passes the request directly to the server. If the server is currently inactive, the broker activates it. All responses and exceptions from a service execution are forwarded by the broker to the client that sent the request. If the specified server is hosted by another broker, the local broker finds a route to the remote broker and forwards the request using this route.

Client-side proxies represent a layer between clients and the broker. This additional layer provides transparency, in that a remote object appears to the client as a local one. In detail, the proxies allow the hiding of implementation details from the clients such as the inter-process communication mechanism used for message transfers between clients and brokers, the creation and deletion of memory blocks, and the marshaling of parameters and results.

Server-side proxies are similar to client-side proxies. They are responsible for receiving requests, unpacking incoming messages, unmarshaling the parameters, and calling the appropriate service. They are also used for marshaling results and exceptions before sending them to the client. When results or exceptions are returned from a server, the client-side proxy receives the incoming message from the broker, unmarshals the data and forwards it to the client.

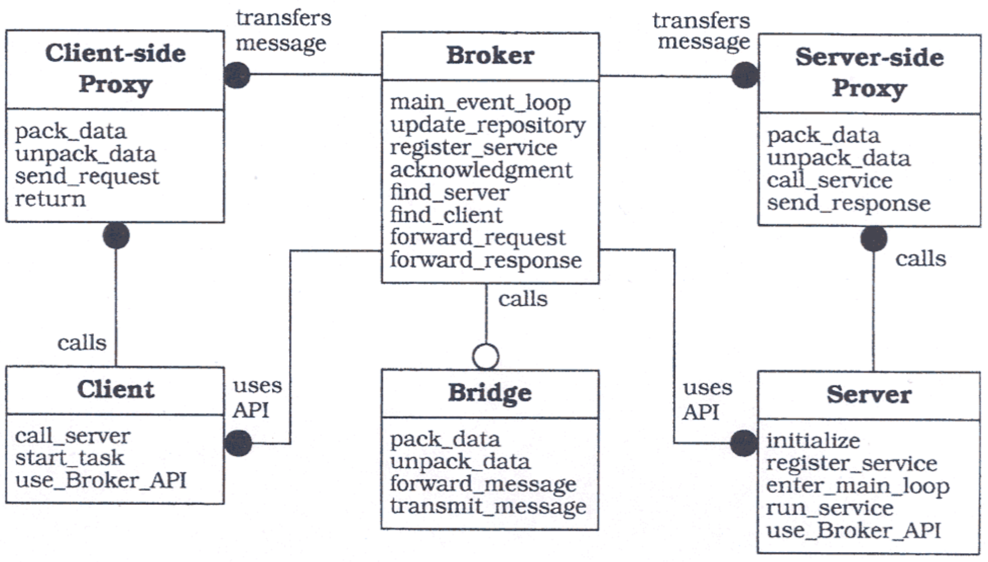


Figure 2-7: Broker Pattern [Buschmann 1996]

Bridges are optional components used for hiding implementation details when two brokers interoperate. Suppose a broker system runs on a heterogeneous network. If requests are transmitted over the network, different brokers have to communicate independently of the different network and operating systems in use. A bridge builds a layer that encapsulates all these system-specific details.

In summary, a mechanism within an architectural pattern refines architectural strategies and adds details to the solution. A mechanism *implements* architectural strategies. The implementation details are usually recorded in the pattern description. During design and implementation an architectural mechanism is further refined through corresponding design and implementation decisions. This refinement does not affect the design structure the pattern/mechanism has originally introduced into the architecture. Consequently, architectural strategies and mechanisms support architecture design activities (e.g., [Bass 2003]).

2.2.4 Architecture Design Methods

In this section, we provide an overview of methods for architecture design. Despite the importance of architecture design for the success of a system development project, there is no generally accepted standard process in literature for creating an architecture. Instead, a variety of approaches can be found (e.g., [Rumbaugh 1991], [Jacobson 1997], [Bosch 2000], [Bass 2003]). However, the essence of design is decision-making (cf. Section 2.2.2). In general, an architecture design method specifies: 1) what decisions a designer must make, 2) how those decisions should be made, and 3) in what order they should be made [Brooks 1995]. An architecture design method, then, provides the intellectual roadmap that enables a designer to refine the architecture successfully to achieve the given requirements. Architecture design methods aim at improving the ability of software designers to produce designs of reasonable quality on a repeatable basis (e.g., [Kruchten 2000]). Unfortunately, the studies presented in Chapter 1 reveal that current approaches do not always guarantee appropriate architectures.

Next, existing architecture design methods are discussed in more detail. In Section 2.2.5, a summary of the commonalities and differences of these methods is provided. The methods are then evaluated according to the research gaps of this thesis (cf. Section 1.3.1).

2.2.4.1 Structured Analysis and Design (SA/SD)

One of the first integrated approaches for software design was the combination of Structured Analysis with Structured Design (SA/SD) [Yourdon 1979]. The software modeling and design paradigms established have continued to the present as one of the fundamental approaches to software development. SA/SD models different system views (e.g., data flow, control flow, functional structure). It uses a variety of heuristics to form each view, and connects to the management view through measurable characteristics of the analysis and design models.

SA/SD prescribes development in two basic steps.

1. *Define the flow of data in the system:* Prepare a data flow decomposition of the system to be built. A data flow decomposition is a tree hierarchy of data flow diagrams, textual specifications for the leaf nodes of the hierarchy, and an associated data dictionary. The data flow decomposition reflects the flow of data between functional operations of the system. The functional operations are derived from major functional system requirements.

2. *Transform data flow to structural entities:* Transform the data flow decomposition into a function and module hierarchy that fully defines the structure of the software in subroutines and their interaction. Interactions can be specified by control flow models. State models are used to show sequential actions and combinations of actions and modes

The design hierarchy is then coded in the programming language of choice. The design hierarchy is converted to software code. The internals of each routine are coded from the included process specifications, though this requires human effort.

The SA/SD method provides heuristics for transformation of an analysis model into a structure chart, and for evaluation of alternative designs [Yourdon 1979]. The heuristics are strongly prescriptive in the sense that they are stated procedurally. The transformation is a type of refinement or reduction of abstraction. At the same time, heuristic guidelines like "strive for loose coupling" are given measurable form, as the design is refined into specific programming constructs.

2.2.4.2 Object-Oriented Design (OOD)

Since the hype of object-oriented (OO) software development in the 1980s and early 1990s, a lot of methods for object-oriented design (OOD) have been proposed (see [TOA 1995] for a comprehensive overview). Leading members of those methods are the Object Modeling Technique (OMT) [Rumbaugh 1991], Object-Oriented Software Engineering (OOSE) [Jacobson 1992], and the Booch method [Booch 1994] which later evolved into the Rational Unified Process [Jacobson 1999].

OOD methods derive from data-oriented and relational database software design practice. They package data and functional decomposition together. Where structured methods build a functional decomposition backbone on which they attempt to integrate a data decomposition, the OOD methods emphasize a data decomposition on which the functional decomposition is arranged. OOD methods consist of activities, which lead to different analysis and design models that correspond to different views of a system. The models are initially defined, then refined as the phases of the method progress. Three typical models of OOD are:

- The object model, which describes the static structure of the objects in a system and their relationships.
- The dynamic model, which describes the aspects of the system that changes over time. This model is used to specify and implement the control aspects of a system.
- The functional model, which describes the data value transformations within a system.

OOD methods are usually divided into different phases, which represent stages of the development process. The following three phases are typical in OOD:

1. *Analyze problem:* Based on a statement of the problem or user requirements, build a model of the real world situation. User requirements can be refined by use cases (e.g., [Jacobson 1992]). They typically reflect the functional requirements of the system.
2. *Design system:* Partition the target system into subsystems, based on a combination of knowledge of the problem domain and the proposed architecture of the target system (solution domain). This results in a system design document that includes the basic system architecture and high-level decisions.

3. *Specify design objects:* Construct a design, based on the analysis model enriched with implementation detail, including the computer domain infrastructure classes.

The logic of object-oriented methods is to decompose the system in a data-first fashion, with functions and data tightly bound together in classes. Instead of a functional decomposition hierarchy, a class hierarchy is created. Functional definition is deferred to the detailed definition of the classes. The object-oriented logic works well where data, and especially data relation complexity, dominates the system.

OOD methods combine data (relational), behavioral, and physical views. The physical view is well captured for software-only systems, but specific abstractions are not given for hardware components. While, in principle, OMT, OOSE and other object-oriented methods [TOA 1995] can be extended to mixed hardware/software systems, and even more general systems, there is a lack of real examples to demonstrate feasibility. Broad, real experience has been obtained only for predominantly software-based systems [TOA 1995].

2.2.4.3 Reuse-Based Software Engineering Business (RSEB)

The Reuse-Based Software Engineering Business (RSEB) [Jacobson 1997] consists of a process framework that supports systematic software reuse during system development. It is founded on a component-based approach [Szyperski 1998] to software development and fundamentally includes the following activities for architecture design:

1. *Model the domain:* For modeling the domain, a use case driven approach [Jacobson 1992] is followed. During this approach high-level use cases are identified and the purpose, flow of events, and variation points addressed are described. Next, objects and methods in the use case are identified and an analysis model that shows the variation between the objects is created. From this analysis model logical classes/components used to prepare the domain design are derived.
2. *Design the domain:* First, a layered functional (logical) architecture containing the main service blocks (logical building blocks) of the domain is created. Based on the functional architecture a so-called class architecture is derived, and the dependencies between classes and services are recorded. Next, the dynamic behavior of the object classes is modelled. At last, the class interfaces with the specified variation are developed.

2.2.4.4 User Centered Design (UCD)

User Centered Design (UCD) [Isensee 2000] is an approach that supports the entire development process with user-centred activities. The goal is to create applications, which are easy to understand and easy to use. An UCD approach focuses on optimizing the *usability* of software systems. Usability requirements can take the form of how accurately users complete their tasks, how long they take, and how satisfied they are (cf. Section 2.2.1.2).

UCD includes the following general steps for architecture design:

1. *Analyze:* Identify and prioritize which user issues will contribute to the success of the project. Describe who will use the system, and how it will be used. Define the main goals the users are to perform and define a comprehensive list of all tasks the users will perform. Prioritize tasks according to their importance.
2. *Design:* Start by designing flow structure and navigation to support main tasks. Produce prototypes (ranging from simple paper mock-ups to interactive computer-based

prototypes) to obtain user feedback on the extent to which proposed solutions meet user needs. Their use will make the potential outcome and interaction scenario more tangible to users.

3. *Evaluate:* Evaluate the design from a user perspective. This should be done early and continuously during the design process. Design solutions are improved until requirements are met. When a complete prototype is available, usability requirements for user performance and satisfaction can be tested.

2.2.4.5 Rational Unified Architecture Design Process (RUP-AD)

The Rational Unified Process (RUP) [Kruchten 2000] includes best practices for developing software systems. These practices are packaged in various workflows and can be applied to a wide range of organizations. RUP deals with activities of the entire software development life-cycle. It also includes guidelines for architecture design (RUP-AD). The primary result of RUP-AD is the design model, which strives to preserve a structure of the system as imposed by the requirements of the system. RUP-AD primarily considers the following quality attributes [Kruchten 2000]: *usability, reliability, performance, and maintainability*.

RUP-AD proposes the following activities for architecture design:

1. *Define a candidate architecture:* Create an initial sketch of the architecture of the system, by defining an initial set of architecturally significant elements to be used as the basis for analysis, an initial set of analysis mechanisms, the initial layering and organization of the system; and the use-case realizations to be addressed in the current iteration. Identify analysis classes from the architecturally significant use cases and update the use-case realizations with analysis class interactions.
2. *Refine the architecture:* Identify appropriate design elements from analysis elements and appropriate design mechanisms from related analysis mechanisms. Maintain the consistency and integrity of the architecture, ensuring that new design elements identified for the current iteration are integrated with pre-existing design elements and maximum reuse of available components is achieved as early as possible in the design effort. Describe the organization of the system's runtime and deployment architecture.
3. *Analyze behavior:* Transform the behavioral descriptions provided by the use cases into a set of elements on which the design can be based. Analyze behavior is concerned with how to deliver the needed application capabilities (functionality).
4. *Design components:* Refine the definitions of design elements by working out the details of how the design elements implement the behavior required of them. Refine and update the use-case realizations based on new design elements introduced, thus keeping the use-case realizations up to date. In addition, define the concurrent threads of control in the system and the protocols among them. Review the design as it evolves.
5. *Design the database:* This is an optional activity, invoked when the system involves a large amount of data in a database. Its purpose is to identify the persistent classes in the design and to design appropriate database structures to store the persistent classes.

All models created during the RUP architecture design process are expressed in terms of the Unified Modeling Language (UML) [Jacobson 1999].

2.2.4.6 Attribute Driven Design Method (ADD)

The Attribute Driven Design (ADD) method [Bass 2003] is an approach to defining a software architecture that bases its decomposition on quality attributes the software has to fulfill. ADD is positioned in the life cycle after requirements analysis and can start when the architecturally relevant requirements are known. The method produces the resulting architecture by recursively decomposing the system into a set of subsystems and then each subsystem into a set of components. For large systems, these components can be further decomposed if necessary. Systems, subsystems, and components are called design elements [Bass 2003]. Once a design element has been decomposed, it can be seen as an aggregation of its child design elements.

ADD involves the following seven steps:

1. *Choose design element:* The design element to be decomposed in the current iteration must be selected. To decompose a design element, its specification must be available. At the very beginning of the architectural design, only the design element that represents the entire system can be selected. Once this design element has been decomposed, any of its child design elements can be chosen to be decomposed next.
2. *Choose architectural drivers:* The architecturally driving requirements to be addressed in the decomposition are selected from the available requirements. This selection should be done based on the importance of the requirements.
3. *Choose attribute primitives:* Architectural solutions are chosen to address the quality drivers that have been selected in the previous step.
4. *Instantiate design elements:* The architectural solutions are instantiated to address the requirements of the system. Functional responsibilities are assigned to the resulting child design elements.
5. *Define interfaces of the child design elements:* The interfaces of the added child design elements are specified. The interfaces describe the services provided and required by the design elements and the data input and output realized by these services.
6. *Validate results:* The decomposition is validated to ensure that the requirements for the system can be satisfied.
7. *Determine requirements for the child design elements:* Finally, the child design elements are prepared for their own decomposition. The requirements and constraints are assigned to each child design element.

ADD has been applied to design a garage door opener within a home information system [Bass 2003].

2.2.4.7 Quality Attribute-Oriented Software Architecture Design Method (SADM)

Like ADD, the Quality Attribute-Oriented Software Architecture Design Method (SADM) [Bosch 2000] aims at guiding the design of a software architecture in order to satisfy the important quality requirements. SADM requires that both functional and quality requirements for the system have been specified before the architectural design starts. It provides a software architecture that addresses these requirements as output. This architecture is described in terms of components and the relationships among those components. The description also includes a documentation of the system context and the core abstractions incorporated in the architecture.

SADM is a scenario-based method and requires that the quality requirements are refined by quality scenarios. To document quality scenarios, the method introduces the concept of scenario profiles. A scenario profile refines a quality attribute into a number of categories. Each category is in turn refined into a set of quality scenarios. For example, there may be a change profile to characterize the modifiability requirements for the system, or there may be a security profile to characterize the security requirements.

SADM involves an iterative process with the following basic steps:

1. *Design the functional structure of the architecture:* The architecture is designed based on the functional system requirements. The system boundaries are defined explicitly by specifying the system's interfaces and its behavior that can be perceived via these interfaces. Then the functional requirements are assigned to the interfaces. In addition, the core architectural abstractions to be used in the architecture design are defined. In the context of SADM, these core abstractions are called architectural archetypes. The goal of defining the archetypes is to ensure conceptual integrity of the architectural design, so that similar architectural problems are addressed by a similar solution. To define the archetypes, the architect must understand and analyze the application domain and the overall functionality of the system to be defined. Once the archetypes have been identified, the relationships between the archetypes are determined and the system is decomposed into components. Once the components have been determined, the required relationships between the components are identified.
2. *Estimate quality attributes:* In this step it is analyzed if and to what extent the quality requirements can be satisfied based on the functional design of the architecture. This is done by using different analysis techniques: scenario-based analysis, simulation-based analysis, mathematical model-based analysis, and experience-based analysis. The goal is to identify those quality requirements that cannot be satisfied based on the existing functional structure. These quality requirements are next used as input for architecture transformations.
3. *Make architecture transformations:* The architecture is optimized for the quality requirements not yet satisfied by the functional structure of the system. This is done by transforming the architecture either by imposing an architectural style, by imposing an architectural pattern, or by converting the quality attributes to functionality. Architecture transformations change the existing structure of the architecture with the goal to improve certain quality attributes of the architecture without affecting the implemented functionality. The result of this step is an architecture that can be used as a basis to implement both the functional and the quality requirements of a software system.

2.2.4.8 Applied Software Architecture Approach (ASAA)

The goal of the Applied Software Architecture Approach (ASAA) proposed by Hofmeister et al. [Hofmeister 2000] is to provide "guidelines and techniques to produce good architecture designs more quickly." The approach is centered on four architectural views: the conceptual view, the module view, the execution view, and the code view (cf. Section 2.2.2.1).

ASAA proposes the following three steps for each view:

1. *Perform a global analysis:* Analyze the organizational, technological, and product factors that influence the architecture and develop strategies for accommodating these factors in the architecture design. This includes the following activities: Identify and describe the

factors, characterize the flexibility and changeability of the factors, and analyze the impact of the factors on the architecture.

2. *Make a central design:* Component and relationship types required by the architectural structures are instantiated, and the configuration of the component and relationship instances are determined. In particular, the following activities are performed: Define component and connector types, define how component and connector types interconnect, and map the system functionality to components and connectors.
3. *Finalize the design:* Finally, the decisions that have been made in the central design step are refined. Interfaces of components are determined and decisions regarding resources budgeting are made.

ASAA has been applied and validated in several industrial case studies including an image acquisition and processing system [Hofmeister 2000].

2.2.5 Comparison of Design Methods

Table 2-5 provides an overview of the design methods investigated in the previous section. In the following the commonalities and differences of the methods are discussed. The methods are then evaluated with respect to the research gaps of this thesis (cf. Section 1.3.1).

2.2.5.1 Commonalities and Differences

One major difference among existing architecture design methods is related to the question which kinds of requirements should be used for decision making in constructing the architecture. Basically, two distinct groups of design methods can be identified: functionality oriented and quality attribute oriented design methods. Functionality oriented methods primarily focus on the achievement of functional requirements during architecture design. Quality attribute oriented design methods deal with the achievement of a particular quality attribute or a set of quality attributes.

Traditional SA/SD and OOD methods are primarily concerned with the implementation of functional requirements at the architecture level of abstraction. Methods that are concerned with a single quality include RSEB and UCD. RSEB focuses on reusability, UCD considers usability requirements. Existing methods for multiple attribute oriented design include RUP-AD, ADD, SADM, and ASAA.

RUP-AD is mainly focused on the attributes usability, reliability, performance, and maintainability. ADD is based on architecturally driving quality requirements but does not say anything about how to identify those requirements. ASAA proposes to perform a global analysis to identify the most influencing requirements of the architecture. SADM differs from the other quality oriented methods since it proposes first to design an initial architecture based on functional requirements and then to transform this functional structure iteratively into a design that also permits the achievement of quality requirements. The assumption behind this approach is that an architecture design can be optimized for quality after an initial version of the design has been determined. This assumption is fundamentally different from those of RUP-AD, ADD, and ASAA.

Table 2-5: Capability Summary of Architecture Design Methods

Methods	**SA/SD**	**OOD**	**RSEB**	**UCD**	**RUP-AD**	**ADD**	**SADM**	**ASAA**
Classification of Design Approach	Functionality oriented	Functionality oriented	Quality oriented with respect to reusability	Quality oriented with respect to usability	Quality oriented with respect to multiple attributes	Quality oriented with respect to multiple attributes	Quality oriented with respect to multiple attributes	Quality oriented with respect to multiple attributes
Method's Goals	Architecture design with primary focus on achieving functional requirements	Architecture design with primary focus on achieving functional requirements	Architecture design with primary focus on achieving reusability requirements	Architecture design with primary focus on achieving usability requirements	Architecture design with primary focus on achieving usability, reliability, performance, and maintainability requirements	Design of architecture with primary focus on achieving quality requirements	Architecture design with focus on functionality first and then on achieving quality requirements	Architecture design with primary focus on achieving quality requirements
Functional / Quality Requirements	Functional requirements	Functional requirements	Primarily reusability requirements	Primarily usability requirements	Primarily usability, reliability, performance, maintainability	Multiple quality requirements	Multiple quality requirements	Multiple quality requirements
Architectural Views Considered	Functional/structural view, control flow view, data flow view	Functional/structural view, control flow view, data flow view	Logical view, dynamic view, module interface view	Control flow view, data flow view	Logical view, process view, implementation view, deployment view, use case view	Logical/conceptual view	Logical view	Conceptual view, module view, execution view, code view
Design Support Concepts	Design heuristics	Use cases, object-oriented principles	Use cases	Scenarios, prototyping	Scenarios, design patterns	Scenarios, design patterns	Scenarios, design patterns, simulation, mathematical modeling	Global requirements analysis
Application Domain	Any	Any	Primarily product line systems	Primarily user-centered systems	Any	Any	Any	Any
Method's Validation	Various systems in different domains	Various systems in different domains	No information available	No information available	Various systems in different domains	Home information systems	No information available	Various systems in medical imaging domain

Another important aspect of the investigated approaches is that they differ in the extent of modeling different views for describing the resulting architecture. In Section 2.2.2.1, an architectural view has been introduced as an abstraction of the architecture from a particular perspective, covering particular concerns and omitting concerns that are not relevant to this perspective. Generally speaking, the more views are covered in an approach the more precise the architecture can be described and designed.

Most approaches model static architectural structures such as a functional view (SA/SD and OOD), a logical/conceptual view (ROD, RUP-AD, ADD, SADM, and ASAA), or a module and code view (ASAA). ADD and SADM are restricted to static views. This may limit the accuracy of the architecture description and thus increases the risk that the architecture is not correctly implemented during detailed design and coding because of an architectural drift (e.g., [Clements 2003]). The concept of using different views in OOD is potentially very powerful, but can also be considered to be very complex. In many OOD approaches it remains unclear whether the views are to be developed independently of one another, or whether the knowledge of one model should be used to influence the construction of another model (e.g., [Rumbaugh 1991], [Jacobson 1992]).

Some approaches additionally consider modeling dynamic structures such as a control and data flow view (SA/SD, OOD, and UCD) or process and execution views that document the behavior of the system during run-time (ROD, RUP-AD, and ASAA). RUP-AD additionally proposes a use case view that shows the interaction among elements of a particular view. A distribution view that shows the allocation of run-time elements to hardware nodes is only covered by RUP-AD and ASAA. The other approaches do not even care for a hardware view, which is the basis to describe distribution aspects.

A major difference between UCD and other approaches is that UCD develops simple models, mock-ups or prototypes on parts or all of the designs (graphical designs, information architecture, interaction design, and information visualization). Prototypes are used as touch-points with users to keep checking that design concepts and solutions are on course from a user perspective. The risk of developing a solution that does not work is thus minimized. Usability effort focuses on providing feedback on the acceptability to users of design solutions while they are being developed.

Interestingly, none of the approaches suggests documenting architectural decisions that lead to the architectural views. RSEB provides only a very general process outline for architecture design. RUP-AD, ADD, SADM, and ASAA at least provide information about the design patterns/strategies that have been applied in the course of architecture design. SA/SD and OOD propose to apply design heuristics and object-oriented principles such as information hiding and loose coupling during modeling but do not enforce the documentation of these heuristics and principles in the design activities. Thus, much of the information about why the architecture is the way it is gets lost. This is critical because if the rationales behind the key decisions are missing, then a sufficient understanding of the architecture cannot be achieved (e.g., [Bass 2003]).

Most approaches make use of scenarios during design in order to refine requirements or to prove if a particular requirement is addressed by the architecture (i.e., evaluation with respect to a single requirement). UCD, for example, includes an evaluation activity in which the resulting design is analyzed from a user perspective with respect to usability. RUP-AD proposes to evaluate if the functional behavior of the system can be achieved with the design.

In ADD, the design decomposition is validated to ensure that the requirements for the system can be satisfied. None of the methods seriously provides steps for an overall evaluation, which would uncover side effects between the architecture decisions made during design. This means that potential risks that may be introduced in the architecture due to competing requirements cannot be identified at this level of abstraction. The overall evaluation is left to special architecture evaluation efforts. None of the methods offers such combined design and evaluation steps.

Further, most of the approaches can principally be applied to systems of any domain (SA/SD, OOD, RUP-AD, ADD, SADM, and ASAA). However, UCD is primarily focused on user-centered systems. For some of the approaches there are records about the application in a particular domain (SA/SD, OOD, RUP, ADD, and ASAA).

2.2.5.2 Evaluation Against Research Gaps

In this section, we provide an evaluation of the design methods against the research gaps of this thesis:

- *Gap 1: Lack of guidance for identifying requirements that are essential for architecture development.*

  Research gap 1 can be confirmed. None of the approaches provides adequate steps for prioritizing requirements that drive the architecture design. The approaches rely on the assumption that functionality, a particular quality attribute (e.g., reusability and usability), or a set of quality attributes drives the architecture. In the latter case the set of driving quality attributes is either required to be predefined by some other effort or all quality attributes are considered as driving.

- *Gap 2: No systematic support for making adequate design decisions in order to implement architecturally relevant requirements.*

  Research gap 2 can be confirmed. None of the methods does directly support making adequate decisions to implement the architecturally driving requirements. Moreover, none of the methods documents the architectural decisions that lead to the adequate documentation of architectural views. The number and type of supported views differs which limits the accuracy of the architecture description and thus increases the risk that the architecture is not correctly implemented during detailed design and coding.

- *Gap 3: Insufficient support for the identification and mitigation of unwanted effects of design decisions in architecture development.*

  Research gap 3 can be confirmed. None of the methods provides steps for an overall evaluation of the architecture. This means that potential risks that may be introduced in the architecture due to competing requirements cannot be identified at this level of abstraction. Thus, there is a high probability for unexpected behavior of the architecture (e.g., due to side effects among the decisions) that at least involves extra cost to resolve in later phases (if possible). The overall evaluation is generally left to special architecture evaluation efforts. There are no such combined design and evaluation steps included.

2.3 Evaluating the Architecture

This section provides an overview of architecture evaluation. The goal of an *architecture evaluation* is to verify the architecture against its requirements (e.g., [Clements 2002], [Smith 2002], [Bass 2003]). As mentioned in Section 2.2.1, quality attribute requirements and, particularly, architectural drivers are the most critical requirements that must be satisfied during architecture design. In the course of architecture evaluation, it is analyzed if the architectural decisions made during design to achieve those requirements are adequate. The result of an evaluation is typically a list of architectural risks. Risks may arise if important decisions to satisfy particular requirements are missing and if design decisions conflict with critical requirements such that the achievement of these requirements cannot be guaranteed. Generally speaking, an architecture implies risks if it is based on problematic design decisions [Clements 2002].

Organizations that practice architecture evaluations as a standard part of their development process have experienced improvements in the quality of the architectures that have been evaluated (e.g., [AT&T 1993]). Abowd et al. [Abowd 1997] reports an improvement of documenting design choices and their rationales since architecture evaluation focuses on analyzing the achievement of requirements and requires answers to specific questions about the design. Bass et al. [Bass 2003] share the experience that an architecture evaluation uncovers conflicts and tradeoffs among requirements and provides a forum for their negotiated resolution. These statements lead to the observation that an architecture evaluation tends to increase quality, improve documentation, and reduce budget risks within a system development effort (e.g., [Clements 2002]).

Figure 2-8 illustrates the major input and output data flow of architecture evaluation. In the following subsection, we will explore the input artifacts to and the output artifacts from architecture evaluation. In addition, we will introduce evaluation support concepts and discuss methods for evaluating an architecture in more detail.

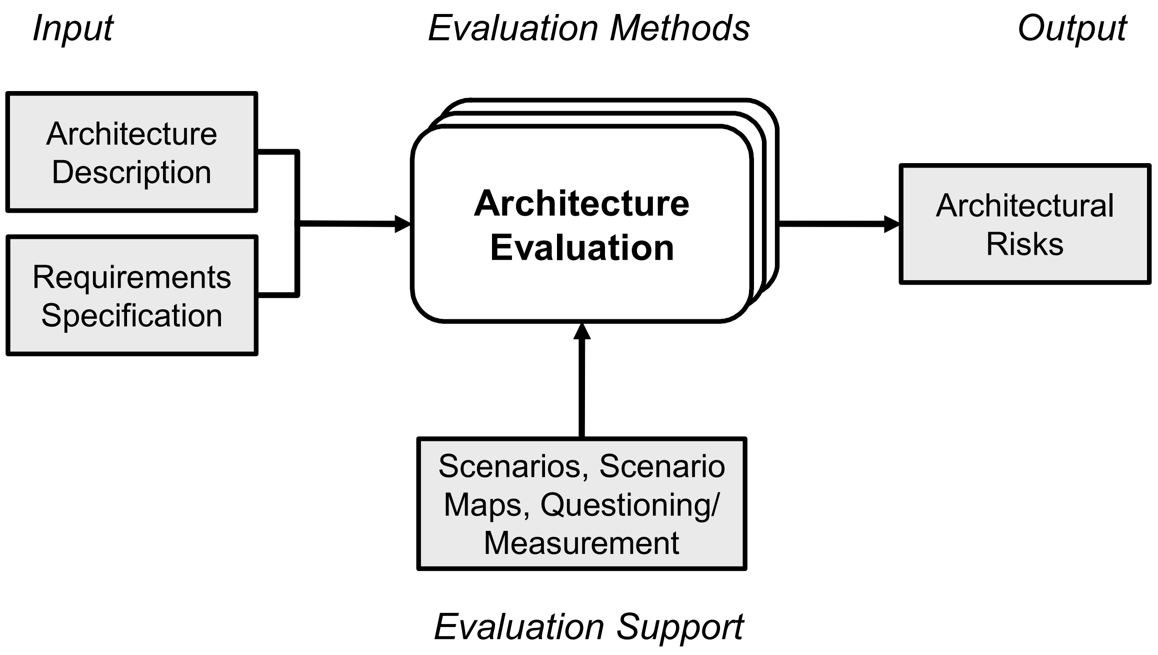


Figure 2-8: Architecture Evaluation

2.3.1 Inputs to Architecture Evaluation

As mentioned above, the objective of an architecture evaluation is to analyze, whether the architecture meets the given set of requirements. Consequently, the following two artifacts should be considered as inputs for the evaluation:

1. The requirements specification
2. The architecture description

The requirements specification should state the functional and quality requirements of the system and how these requirements relate to the business goals of the system. A description of how the requirements specification should look like was provided in Section 2.2.1.

The architecture description should document the architectural views and major design decisions made to create the architecture. Details about the architecture description were discussed in Section 2.2.2.

When architecture evaluation is applied to a group of competing architectures (e.g., [Kazman 1996], [Bass 2003]), then the requirements specification and architecture description of each architecture considered is required as an input for the analysis.

2.3.2 Outputs from Architecture Evaluation

The primary output of architecture evaluation is a (prioritized) list of architectural risks. We will explore architectural risks and their prioritization in the following subsections.

2.3.2.1 Architectural Risks

The primary result of an architecture evaluation is a list of risks (e.g., [Kazman 1999], [Clements 2003]). Risks arise if the design decisions made in the architecture are not technically sound or if they do not support the business goals related with the requirements specification. According to Webster's dictionary [Webster 1999] a *risk* is the "possibility of suffering loss." In the context of architecture evaluation, the loss may take the form of diminished quality of the end product, increased cost, delayed completion, or failure.

Risks captured during an architectural evaluation include aspects such as the following (e.g., [Kazman 1996], [MITRE 1996], [Clements 2002], [Smith 2002], [Bass 2003]):

- The architecture does not support software updates by end-users.
- The architecture does not provide a proper protection against unauthorized access.
- The architecture does not recover from system failures.
- The application components of the architecture are coupled to the underlying hardware.
- The architecture does not allow the replacement of a component without changing other components.
- The architecture does not guarantee proper management of time-critical events.
- The architecture has a limited performance for network communication.

The risks uncovered during an architecture evaluation have implications on the question whether the architecture is suitable for the system for which it was designed. According to [Clements 2002], suitability is only relevant in the context of specific goals for the architecture and the system it spawns. Clements et al. [Clements 2002] point out that "an architecture designed with high-speed performance as the primary design goal might lead to a system that runs like the wind but requires hordes of programmers working for months to make any kind of modification to it. If modifiability were more important than performance for that system, then that architecture would be unsuitable for that system."

Another implication of evaluating and architecture for suitability is that the answer that comes out of the evaluation is not going to be the sort of scalar results (e.g., [AT&T 1993], [Kazman 1996], [MITRE 1996], [Bengtsson 1999], [Clements 2002]). Since the purpose is rather learning where a quality attribute of interest is affected by architectural design decisions, an evaluation will tell that the architecture has been found suitable with respect to one set of goals and problematic with respect to another set of goals. Sometimes the goals will be in conflict with each other, or at least, some goals will be more important than others. And so the manager of the project will have a decision to make if the architecture evaluates well in some areas and not so well in others. Can the manager live with the areas of weakness? Can the architecture be strengthened in those areas? Or is it time for a wholesale restart?

According to [Kazman 2001], an architecture evaluation will help reveal where an architecture is weak, but weighing the cost against benefit to the project of strengthening the architecture is solely a function of project context and is in the realm of management. However, improving the architecture requires detailed insights into the risks.

2.3.2.2 Prioritization of Risks

Risk prioritization helps a project focusing on its most severe risks by assessing the risk exposure (e.g., [Jones 1994]). Risk prioritization is essential for risk control, the process of managing risks to achieve the desired outcomes. It further supports planning and developing appropriate risk avoidance and mitigation strategies [Jones 1994].

Architecture evaluation is an early risk identification activity [Clements 2002]. It checks if the architecture permits the achievement of the requirements. It can identify critical architectural design decisions that make it impossible or difficult to meet the requirements – independent from how well the subsequent process phases will be performed. According to [Reynolds 1996], risk prioritization would be an additional activity of an overall risk assessment effort

An equation for risk assessment commonly used in literature (e.g., [Jones 1994], [Reynolds 1996], [Broekman 2003]) is:

$$Risk = probability \times impact$$

The *probability* estimates the occurrence of a particular risk. If the risk associates a system failure then the probability is related to aspects including frequency of use and the chance of a fault being present in the system. With a component that is used many times a day by many people, the chance of fault showing itself is high. Schäfer [Schäfer 1996] provides a list of locations where faults tend to cluster.

The *impact* of the risks estimates the damage for the organization. The impact can take the form of cost for repair, loss of market share caused by negative press, legal claims, foregone income etc. Some practitioners propose to translate each form of impact into money terms since this makes it easier to express the damage in numerical terms and to compare the related risks with other assessed risks (e.g., [Broekman 2003]). Others suggest qualifying the impact of a risk on project categories such as cost, performance, schedule and support (e.g., [USAF 1988]).

During a risk assessment the *overall risk exposure* is usually calculated (e.g., [USAF 1988], [Jones 1994], [Reynolds 1996], [Broekman 2003]). Using estimates on risk probabilities and impacts, the overall risk to the project is gauged. In figuring out the overall risk, the impact on other risks of the project is usually considered. Risks are rarely stand-alone. Interrelationships often exist. In order to ease the calculation of the overall risk the USAF handbook [USAF 1988] suggests a qualitative categorization of probabilities (very low, low, medium, high, and very high) and impact (negligible, marginal, critical, and catastrophic) and to use a matrix for representing the risk spectrum. An example of such a matrix is given in Table 2-6.

Table 2-6: Probability and Impact

Impact / Probability	**Very High**	**High**	**Medium**	**Low**	**Very Low**
Catastrophic	High	High	Moderate	Moderate	Low
Critical	High	High	Moderate	Low	None
Marginal	Moderate	Moderate	Low	None	None
Negligible	Moderate	Low	Low	None	None

For example, if a risk has a *High* probability and a *Marginal* impact, the overall risk would be *Moderate*. Once the assessment is complete, a certain number of the highest overall impact risks should be selected for planning. In the context of architecture development, appropriate design decisions for risk avoidance and mitigation are planned and incorporated during the next architecture design iterations. For those risks whose overall impact is *Low* or *None*, it makes little sense to squander resources in the planning stage.

2.3.3 Evaluation Support Concepts

In this section, we present important concepts supporting architecture evaluation. In particular, we explain the purpose of using scenarios during the evaluation of an architecture. We further sketch the role of use case/scenario maps and discuss questioning and measurement techniques.

2.3.3.1 Scenarios

Scenario-based techniques have gained increased attention within the requirements engineering community (e.g., [Weidenhaupt 1998]). They have also been proven useful during architecture evaluation (e.g., [Kazman 1996], [Kazman 2000], [Smith 2002], [Clements 2002]). A scenario is defined as a short statement describing an interaction of one of the stakeholders (or another software system) with the system under consideration [Kazman 2000]. Both functional and quality requirements can be described as scenarios. The former are often called use case or *usage scenarios*, the latter *quality (attribute) scenarios*. Usage scenarios make functional requirements concrete by describing a sequence of actions that use a piece of functionality of the system to provide the user with a valuable result [Kruchten 2000]. Quality scenarios make quality requirements explicit by prescribing a structure to document those requirements [Kazman 2000].

Bass et al. [Bass 2003] propose a structure for describing quality scenarios. It consists of six parts:

- *Source of stimulus.* Entity (e.g., human, computer system, or other actors) that generated the stimulus.
- *Stimulus.* Condition that needs to be considered when it arrives at a system.
- *Environment.* Certain condition in which the system is while the stimulus arrives. The system may be in an overload condition or may be running when the stimulus occurs, or some other conditions may be true.
- *Artifact.* Element of the system that is stimulated. This may be the whole system or some part of it.
- *Response.* Activity undertaken after the arrival of the stimulus.
- *Response measure.* When the response occurs, it should be measurable in some fashion so that the requirement can be tested.

Figure 2-9 illustrates the elements of a quality scenario. An example of a quality requirement and its reformulation as concrete quality attribute scenario is given in Table 2-7.

During architecture evaluation, scenarios serve as a vehicle for asking questions about how the architecture under review reacts in various situations. If the response specified in the scenario description cannot be achieved then this is an indication that there is something wrong with the architecture. Consequently, the architecture needs to be investigated in more detail in order to uncover potential risks and drawbacks.

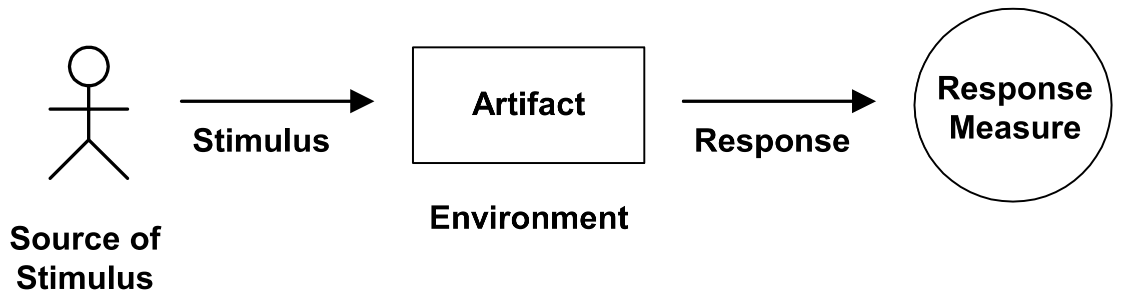


Figure 2-9: Elements of a Quality Attribute Scenario [Bass 2003]

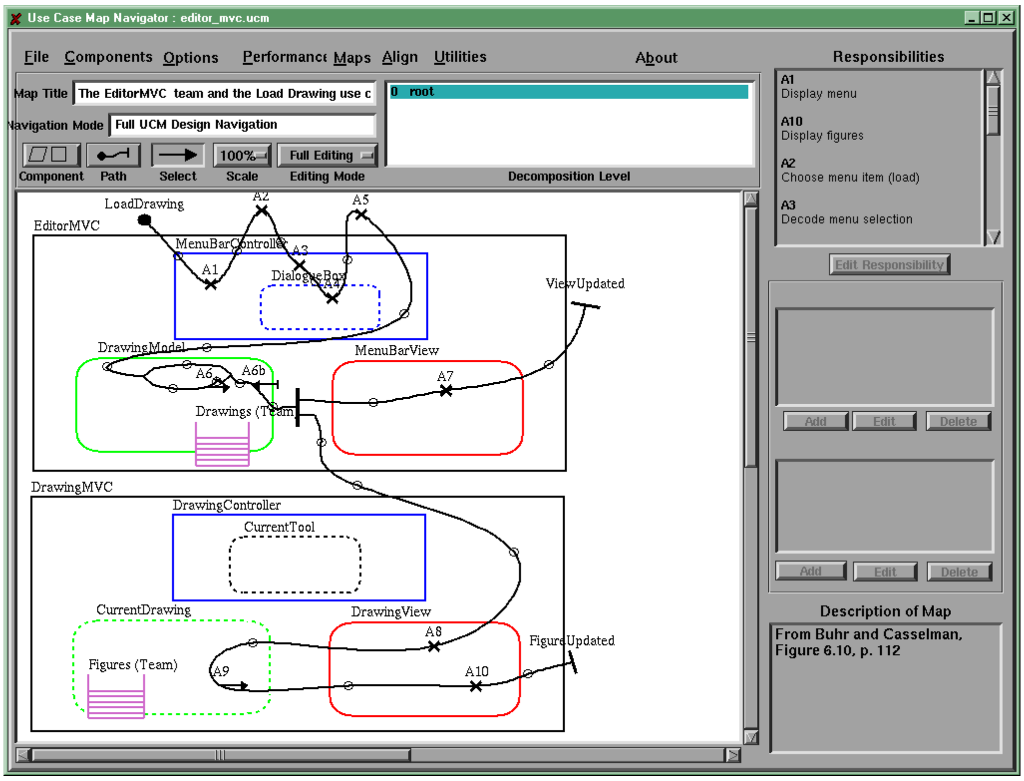
Table 2-7: Quality Attribute Scenario

Quality requirement	The User Interface shall be modifiable.
Quality attribute	Modifiability
Quality scenario	A developer shall be able to change the User Interface code at design time with no side effect changes in less than three hours.
Source of stimulus	Developer
Stimulus	Changes to the User Interface code
Environment	Design time
Artifact	Code
Response	Modification with no side effect changes
Response measure	Three hours

2.3.3.2 Use Case/Scenario Maps

A use case map is a graphical path-based notation used for specifying causal sequences of actions through systems [Buhr 1996]. It is a high-level excerpt of a design model that helps humans express and reason about a system's large-grained structural and behavior patterns.

Use case maps are composed of paths that traverse elements. They are intended to be used at the requirements level and for architectural design (e.g., [Buhr 1998]). An example of a use case map at the design level is shown in Figure 2-10.



Map Title: The EditorMVC team and the Load Drawing use c

Navigation Mode: Full UCM Design Navigation

Responsibilities

A1
Display menu

A10
Display figures

A2
Choose menu item (load)

A3
Decode menu selection

Description of Map

From Buhr and Casselman, Figure 6.10, p. 112

Figure 2-10: Use Case Map [Buhr 1996]

According to [Buhr 1996], the basic symbols and notations used in representing use case maps are as follows:

- *Path:* A path may have any shape as long as it is continuous. A path may even cross itself, but this can create visual ambiguity related to other aspects of the notation, so the crossing must be distinguished by a small crossover arc or a break in one of the crossed lines.
- *Waiting place:* A filled circle represents a start point in all the examples we have seen so far. In general, a start point is a waiting place for a stimulus to start the path. We use the same symbol for waiting places along paths, for example, to wait for events from other paths.
- *Timer:* A timer is a generalized waiting place that expresses the idea that there is a time limit on waiting. It may be used anywhere a waiting place symbol is used.
- *Bar:* A bar ends a path or marks a place where concurrent path segments begin or end.
- *Basic path:* The most basic, complete unit of a map is a path with a start marked by a waiting place and an end marked by a bar.
- *Direction (optional):* Direction is indicated by the positioning of the start and end points but it is sometimes useful to show local direction in a complicated map or in an incomplete fragment of a larger map.

Use case maps can also be used for analysis purposes [Buhr 1996]. Especially, the map can be used to express how a particular scenario performs in an architecture. The path in the map would then show which architectural design elements are triggered by the scenario in which sequence. Since the map then represents the execution of a scenario, we would refer to it as *scenario map*.

Note that currently, there is no architecture evaluation approach known to the author that applies use case maps for analysis. We have adopted use case maps in the research contribution of this work (cf. Chapter 6).

2.3.3.3 Questioning and Measuring Techniques

Clements et al. [Clements 2002] discuss specific techniques for analyzing software architectures. They distinguish between questioning techniques and measuring techniques. The former generate qualitative questions about the software architecture and aim at stimulating a discussion about the architecture. The latter provide quantitative results and answer questions that the evaluation team may have about the architecture. These techniques differ from each other in applicability and cost, but they are all used to elicit information about the architecture and increase understanding of the architecture’s fitness with respect to its requirements.

The group of questioning techniques consists of scenarios, questionnaires and checklists. The benefits of using scenarios have been discussed in Section 2.3.3.1. A questionnaire is a list of general questions for probing an architecture [Clements 2002]. The questions concern both the process that created the architecture and the resulting architecture. A checklist contains more detailed questions, which result from experience with architecture evaluation in a certain domain (e.g., [MITRE 1996]).

At AT&T, for example, checklists are used as a basis for their software architecture evaluation practice [AT&T 1993]. These checklists include questions like the following

- Can the system be online as changes are made to the application software, or is it taken down to install a new release?
- Is there anything that prevents the application from running on a multiprocessor (for example, use of shared memory)?

The group of measuring techniques contains metrics, simulations, prototypes and experiments. Metrics use measurements on the software architecture to make predictions about certain properties of the architecture. Examples are fan-in/fan-out, complexity, and coupling of design elements (e.g., [Henry 1981], [Fenton 1991], [Heyliger 1994]). With measuring techniques, the evaluation needs to focus not only on the results of the measurement/metric, but also on the assumptions under which the technique was used. For example, a calculation of performance characteristics makes assumptions about patterns of resource utilization. Coupling and cohesion metrics make assumptions about the types of functions embodied in the components being examined. However, the evidence of measuring techniques being used to answer particular analysis questions usually depends on precise environmental data, which is difficult to gather in practice (e.g., [Smith 1990]).

Alternatively, experiments can be done with the architecture by building a simulation or prototype of the system. Building a prototype or a simulation of the system may help creating and clarifying the architecture. A prototype whose components consist of functionless stubs represents a simple model of the architecture. Performance models such as those discussed by Smith [Smith 1994] are examples of a simulation. The creation of a detailed simulation or prototype just for review purposes is typically expensive. On the other hand, adequate artifacts often exist as a portion of the normal development process. In this case, using these artifacts during a review or to answer questions that come up during the review becomes a normal and natural procedure. An existing simulation or prototype may be an answer to an issue raised by a questioning technique. For example, if the evaluation team asks, “What evidence do you have to support this assertion,” one valid answer would be the results of a simulation (e.g., [Kazman 1996]).

2.3.4 Architecture Evaluation Methods

In this section, we provide an overview of methods for architecture evaluation. Similar to architecture design, there is no generally accepted standard process for evaluating an architecture. Next, existing architecture evaluation methods are discussed in more detail. In Section 2.3.5, a summary of the commonalities and differences of these methods is provided. The methods are then evaluated according to the research gaps of this thesis (cf. Section 1.3.1)

2.3.4.1 AT&T’s Best Current Practices for Software Architecture Evaluation (SAE)

AT&T’s Best Current Practices for Software Architecture Evaluation (SAE) [AT&T 1993] were described in the early 1990s and can be seen as one of the first methods that take a systematic approach to software architecture evaluation. It is a checklist-based method that has been developed by AT&T to analyze software architectures in the telecommunication domain. The goal of the method is to investigate whether the software architecture is a reasonable and *cost-effective* solution of the system requirements. The method description contains a checklist that codifies specific knowledge about the concerns that are usually important for systems in the telecommunication domain. These concerns are *error recovery, operation, administration, maintenance,* and *performance*. The architecture evaluation is performed in a two-day meeting. Representatives of the project management, development team, and architecture team are usually involved.

SAE is performed in five main steps:

1. *Contact and planning:* The development team contacts the organizational unit that is responsible for architecture reviews. A review is scheduled and organized.
2. *Form evaluation team:* The evaluation team is formed. The team lead of the evaluation team should ensure that experts for all concerns to be addressed participate in the team.
3. *Preparation:* The evaluation team provides the development team with a sample agenda for the review, a list of sample questions that may be asked during the review, and other preparation guidelines. Based on these pieces of information the development team prepares the required input documents for the review. The evaluation team reviews the documents that have been provided by the project team and prepares questions that the project team has to answer during the analysis in the next step. The checklist serves as a starting point to prepare these questions.
4. *Analysis:* The questions that have been prepared by the evaluation team are answered by the development team. If scenarios have been defined, the architect explains how the scenarios are addressed in the architecture.
5. *Present Results:* The answers provided during the analysis in step 4 and the input documents provided by the development team are used to create an evaluation report that contains the strengths and the weaknesses of the architecture.

SAE provides information about architecture misfits. These misfits represent problems in the architecture and indicate that the anticipated solution does not adequately address the requirements.

2.3.4.2 MITRE's Architecture Quality Assessment (AQA)

The purpose of MITRE's Architecture Quality Assessment (AQA) [MITRE 1996] is to provide an objective and repeatable basis for assessing six architectural quality areas:

- *Understandability:* Is the architecture recorded such that it may be communicated to others?
- *Feasibility:* Does the architecture provide a sufficient basis upon which to develop the system?
- *Openness:* Does the architecture yield a system which can operate in an open system environment?
- *Maintainability:* Does the architecture yield a system which is maintainable?
- *Evolvability:* Does the architecture exhibit a degree of changeability to meet new user or client needs?
- *Client Satisfaction:* Does the architecture result in a system which meets its requirements in the context of its mission?

The assessment is used to highlight potential risks from which to formulate risk reduction strategies, or as a basis for other programmatic decisions relating to acceptance of the architecture, program schedule, or progress payments. The AQA is designed to be applied in various ways; it may be used to: evaluate a candidate architecture; review the technical progress of an ongoing architecture development; assess a complete, delivered architecture prior to its acceptance and system implementation; or, compare two or more alternate architectures in a consistent fashion.

The steps of an AQA are as follows:

1. *Perform a needs analysis:* The concerns of the evaluation are recorded. Needs analysis produces three work products: a needs repository, a goals statement, and a vision statement. The needs repository contains needs of the system. Needs may be thought of as distilled, architecture-relevant system requirements. The goals statement captures the client's success criteria. The vision statement reflects the client's long-term direction for system and its evolution. The needs analysis is then used to: inventory system-specific items for assessment; determine relevance and applicability of measures (specific questions) to the system under assessment; and determine weighting, thresholds and prioritization of measures and factors for the system under assessment.
2. *Gather relevant documents related to the architecture:* Important documents that describe the architecture and its context are collected. Since architectural documentation may take many forms the AQA does not presume or prescribe the form or content of it. Much of this information is likely to be in an informal graphical or textual medium.
3. *Evaluate documentation against measures and score results:* When the AQA evaluators are sufficiently familiar with the architectural documentation, the actual assessment can begin. This involves evaluating different measures. Measures take the form of questions, such as "What views are evident in the architecture description?" To simplify the scoring, aggregation and interpretation of results, all questions are mapped to a set of discrete values (see Table 2-8).
4. *Interpret results and identify architecture-related risks:* Once all applicable questions have been answered, the results of individual measures are aggregated to yield scores for each quality factor. Using the weightings determined from the needs analysis, the quality area scores are then calculated using the factor inputs. At this stage, architecture-related risks may be identified.
5. *Document results for client:* There are three primary products of the AQA review: an executive summary, a detailed evaluation and interpretation of results, and a set of open issues and questions. The executive summary provides an overview of the overall quality results and any outstanding issues, relative to the system's needs analysis. The detailed evaluation provides the scores at all levels of aggregation, the weightings used, and the identified risks. In some cases, clients want to take the results of the AQA back to the architect to provide feedback, or to highlight unresolved issues in the architecture. For this purpose, open issues and questions are documented in a form that may be submitted to the architect.

Table 2-8: AQA Scoring Schema [MITRE 1996]

Value	**Description**
IDEAL	Subject is explicitly addressed, and receives best possible treatment.
GOOD	Sound treatment of the subject relative to expectations.
MARGINAL	Treatment of subject meets minimal expectations
UNACCEPTABLE	Treatment of subject does not meet minimal expectations.
INCOMPLETE	Treatment of subject exhibits a level of detail, which is insufficient to make a judgment.
NON-APPLICABLE	This measure is not applicable to the subject of interest

AQA has been applied to air mobility command information processing systems, standard terminal automation replacement systems, and battle management systems [MITRE 1996]. The method is narrowly focused on architecture-related artifacts. It specifically excludes other concerns and sources of risk not related to architecture, including: the overall system development process, the capabilities of the developers to carry out that process, the validity and relevance of requirements which led to the architecture, and the resulting system design and implementation.

2.3.4.3 Software Architecture Analysis Method (SAAM)

The Software Architecture Analysis Method (SAAM) [Kazman 1996] is a method for analyzing the *modifiability* of a software architecture. In SAAM, a scenario-based approach is employed. This means that the software architecture is analyzed by defining scenarios and exploring their effect on the system. A scenario is defined as "a brief narrative of expected or anticipated use of a system from both development and end-user viewpoints [Kazman 1996]."

SAAM can be used both for analyzing modifiability issues of a single software architecture and for comparing a number of software architectures with respect to this quality attribute. When analyzing a single software architecture, the goal is to determine whether the architecture satisfies its modifiability requirements. When several candidate architectures are analyzed, the goal is to choose the most adequate candidate.

The method consists of the following six steps:

1. *Develop scenarios:* Identify possible events that may occur in the life cycle of the system.
2. *Describe candidate architecture(s):* Give a common representation of the candidate architecture(s).
3. *Classify scenarios:* Determine for each scenario whether it requires modifications to the system. Scenarios that do not require any modifications are called direct scenarios and scenarios that do require modifications are called indirect scenarios.
4. *Perform scenario evaluations:* Determine for each indirect scenario the required modifications by listing the components and connectors that are affected.
5. *Reveal scenario interaction:* Two (or more) scenarios that affect the same component are said to interact. Interaction of unrelated scenarios could indicate a poor separation of functionality. The purpose of this step is to expose these interactions.

6. *Overall evaluation:* By assigning weights to the scenarios and scenario interactions in terms of their relative importance, this step aims to come to an overall evaluation of the candidate software architecture(s).

The results of a SAAM analysis are twofold. Capturing the system's requirements in scenarios clarifies and prioritizes a system's requirements and the evaluation of the scenarios provides insight into the degree to which these requirements have been satisfied. As such, SAAM is suitable for both internal and external analyses.

SAAM has been applied to systems in different domains, such as a telecommunications system [Kazman 1996], a financial information system [Bass 1998], and graphical debuggers [McCrickard 1996]. Its main contribution is that it offers a step-wise method for analyzing the modifiability of software architectures.

2.3.4.4 Architecture-Level Modifiability Analysis (ALMA)

The Architecture-Level Modifiability Analysis (ALMA) method [Lassing 2002] focuses on the analysis of *modifiability* properties of a software architecture. It uses multiple analysis goals, explicit assumptions, and well-documented analysis techniques for performing the evaluation [Lassing 2002].

ALMA consists of five main steps:

1. *Set goal:* Determine the aim of the analysis. Maintenance cost prediction, risk assessment for architectural changes, and software architecture selection are three specific modifiability goals ALMA can deal with.
2. *Describe software architecture(s):* Give a description of the relevant parts of the software architecture(s).
3. *Elicit change scenarios:* Define the set of relevant change scenarios
4. *Evaluate change scenarios:* Determine the effect of the set of change scenarios on the architecture.
5. *Interpret the results:* Draw conclusions from the analysis results. The interpretation of the results depends on the goal of the analysis and the given system requirements.

When performing an analysis, the separation among the tasks in ALMA is not very strict. It is often necessary to iterate over various steps. For instance, when performing change scenario evaluation, a more detailed description of the software architecture may be required or new change scenarios may come up. ALMA has been applied to an application framework, a business information system, and a declarations processing system [Lassing 2002].

2.3.4.5 Architecture-Level Prediction of Software Maintenance (APSM)

The Architecture-Level Prediction of Software Maintenance (APSM) method [Bengtsson 1999] is aimed at evaluating software architectures with respect to *maintainability*. More specifically, the goal is to estimate the maintenance effort required for a system. Two kinds of maintenance tasks are distinguished in APSM: adaptive maintenance (i.e., adapting the software to changes in the environment) and perfective maintenance (i.e., adapting the software to new or changed user requirements).

The prediction method allows comparing the predicted maintenance effort for two or more candidate architectures. Alternatively, the prediction method may be used to assess a single candidate architecture to determine whether the predicted maintenance effort is acceptable. In both cases, the method follows these steps:

1. *Identify categories of maintenance tasks:* Based on the domain or application a number of classes of expected change categories are defined.
2. *Synthesize scenarios:* For each of the maintenance categories, a representative set of scenarios is elicited from the stakeholders.
3. *Assign each scenario a weight:* Each scenario is assigned a weight based on its relative probability of occurrence during a particular time interval. If historical data from similar applications is available, this data used as a basis for determining the weights. Otherwise, the weights are estimated by the stakeholders.
4. *Estimate the size of all elements:* The size of each component (footprint) is estimated.
5. *Script the scenarios:* Determine the effect of each scenario. The effect consists of the components that have to be adapted, new components that have to be added and obsolete components that may be deleted. Based on the size estimates of existing components (step 4) and the estimated size of the new components, the size of the required adaptations for the scenario is then calculated.
6. *Calculate the predicted maintenance effort:* Based on the probability of the scenarios and the size of the required adaptations, the average effort that is required for a maintenance task is calculated.

APSM has been applied to two embedded control systems – a haemo dialysis machine and a beer can inspection system [Bosch 2000].

2.3.4.6 Performance Assessment of Software Architectures (PASA)

The Performance Assessment of Software Architectures (PASA) method [Williams 2002] focuses on the evaluation of software architectures with respect to *performance* objectives. In particular, PASA determines whether the architecture will support the system’s scalability and responsiveness objectives. It follows a scenario-based approach in which scenarios for important workloads are identified and documented. These scenarios then provide a means of reasoning about the performance of the software.

A PASA assessment consists of the following nine steps:

1. *Present process overview:* The assessment process begins with a presentation designed to familiarize both managers and developers with the reasons for an architectural assessment, the assessment process, and the outcomes.
2. *Present architecture overview:* In this step, the development team presents the current or planned architecture.
3. *Identify critical use cases:* The externally visible behaviors of the software that are important to responsiveness or scalability are identified.
4. *Select key performance scenarios:* For each critical use case, the scenarios that are important to performance are identified.
5. *Identify performance objectives:* Precise, quantitative, measurable performance objectives are identified for each key scenario.

6. *Clarify and discuss architecture:* Participants conduct a more detailed discussion of the architecture and the specific features that support the key performance scenarios. Problem areas are explored in more depth.
7. *Analyze architecture:* The architecture is analyzed to determine whether it will support the performance objectives.
8. *Identify alternatives:* If a problem is found, alternatives for meeting performance objectives are identified.
9. *Present results:* Results and recommendations are presented to managers and developers.

During architecture analysis PASA uses general software and system execution models that may be solved analytically or via simulation [Smith 2002]. PASA has been applied to assess several industrial systems, including order-processing systems, stock market data processing systems, and payment posting systems [Williams 2002].

2.3.4.7 Family Architecture Assessment Method (FAAM)

The Family Architecture Assessment Method (FAAM) [Dolan 2001] is a software architecture evaluation method that is aimed at information systems. The method emphasizes the strategic aspects that are associated with this class of systems and addresses the evolutionary capabilities of information systems.

FAAM focuses on two quality attributes: *interoperability* and *extensibility*. In [Dolan 2001] interoperability is defined as "the ability of two or more systems or components to exchange information and to use the information that has been exchanged in order to support a set of coherent business processes that require cooperation of the systems or components." Extensibility is defined as "the ability of the system to accommodate the addition of new functionality." The method uses scenarios to analyze these quality attributes. To stress that these scenarios refer to the evolution of the information systems, FAAM refers to these as change cases.

The description of FAAM mainly focuses on the process that should be followed during an analysis. It consists of the following seven activities:

1. *Define assessment:* Set the scope of the assessment. This activity consists of determining the family that the analysis will focus on and the requirements that will be addressed. Based on these, stakeholders are interviewed for change cases and asked to prioritize them.
2. *Prepare system-quality requirements*: Specify the change cases in line with the assessment goals set in the first step. The specification of requirements is the responsibility of the stakeholders; the analyst acts as a facilitator.
3. *Prepare software architecture:* Create a representation of the software architecture in line with the assessment goals, allowing for assessment against the stakeholder requirements.
4. *Review/refine artifacts:* Review the material gathered in the previous steps and verify with the stakeholders that the requirements, the change cases and the architecture description are correct. This step aims to establish commitment from stakeholders.
5. *Assess software architecture conformance:* The architecture is investigated based on the change cases defined in earlier steps. To evaluate the change cases, FAAM uses the evaluation techniques from SAAM.

6. *Report results and proposals:* Document the results of the analysis and formulate proposals to improve the architecture. These are then reported to the stakeholders and the sponsor of the analysis. In addition, there is an ongoing activity that lasts throughout the whole analysis:
7. *Facilitate architecture assessment:* The facilitator supports the participants in implementing and reporting on the analysis.

FAAM has been applied to a medical imaging system and an enterprise information system [Dolan 2001].

2.3.4.8 Architecture Tradeoff Analysis Method^{SM} (ATAM)

The Architecture Tradeoff Analysis Method^{SM} (ATAM) [Clements 2002] is an evolution of SAAM (cf. Section 2.3.4.3). The goal of the ATAM is to reveal how well an architecture satisfies particular quality objectives. It provides insight into how those quality goals interact with one another – i.e., how they trade off against one another.

ATAM consists of the following nine steps:

1. *Present the ATAM:* The analysis team familiarizes the stakeholders with the process of the ATAM.
2. *Present business drivers:* The project manager of the system under analysis gives an overview of the system: the system's main functionality, goals, constraints, stakeholders and architectural drivers (major quality requirements that shape the architecture).
3. *Present architecture:* The architect of the system presents the architecture of the system.
4. *Identify architectural approaches:* The architecture is investigated to identify the architectural decisions that are made for the system.
5. *Generate quality attribute utility tree:* A utility tree is created that includes and prioritizes the system's most important quality attribute goals specified down to the level of scenarios. The utility tree guides the remainder of the analysis; it tells the analysis team which parts of the architecture to focus on.
6. *Analyze architectural approaches:* Based on the utility tree, the architectural approaches distinguished and documented experiences with architectural styles, the software architecture is analyzed to find out whether the selected styles 'hold significant promise for meeting the attribute-specific requirements for which it is intended'.
7. *Brainstorm and prioritize scenarios:* Together with the 'widest possible group of stakeholders' a set of scenarios is elicited and prioritized. The scenario priorities and the utility tree are compared to harmonize these.
8. *Analyze architectural approaches:* Step 6 is reiterated, but now the highly ranked scenarios from the previous step are used as test cases.
9. *Present results:* The analysis team presents the results of the analysis to the stakeholders.

The ATAM distinguishes two types of scenarios: use case scenarios and change scenarios. Use case scenarios coincide with direct scenarios in SAAM and change scenarios coincide with indirect scenarios. Use case scenarios describe a typical interaction of the user with the system. They are used to elicit the software architecture. Change scenarios can be subdivided in growth scenarios and exploratory scenarios. The former represent typical anticipated future changes to a system and the latter stress the system and are used to identify risks.

An ATAM analysis is usually conducted either by the development team itself or by an external evaluation team. The method has been applied in different types of systems, such as a defense system and a satellite control system [Clements 2002]. Its main contribution is that it provides a method that uses knowledge of architectural approaches to analyze architectural decisions. Thereby, it takes a different approach than SAAM, where the resulting architecture and not the individual decisions are analyzed.

2.3.5 Comparison of Evaluation Methods

Table 2-9 provides an overview of the evaluation methods investigated in the previous section. In the following the commonalities and differences of the methods are discussed. The methods are then evaluated with respect to the research gaps of this thesis (cf. Section 1.3.1).

2.3.5.1 Commonalities and Differences

One main difference among the methods is their evaluation scope. There are two categories of evaluation methods: single attribute analysis and multiple attributes analysis methods. Single attribute analysis methods are focused on evaluating the architecture with respect to a particular quality attribute. Multiple attribute analysis methods deal with the assessment of multiple quality attributes at the same time and determine how competing quality requirements trade off against one another.

SAAM, ALMA, APSM, and PASA each are focused on analyzing an architecture with respect to a single quality attribute. SAAM and ALMA both address modifiability, APSM is concerned with maintainability, and PASA considers performance. SAE, AQA, FAAM, and ATAM are dealing with the evaluation of two or more attributes.

Concerning the evaluation goals, seven of the eight methods (SAAM, ALMA, PASA, SAE, AQA, FAAM, and ATAM) are focused on analyzing the architecture's strengths and, particularly, its weaknesses/risks with respect to the quality attributes of scope. APSM differs from this goal as it focuses on the estimation of the average effort required for achieving the quality attribute of interest (maintainability). FAAM additionally includes a step for gathering proposals that support mitigating the identified risks. All methods assume that a (preliminary) description of the candidate architecture to be evaluated exists.

Further, two methods (PASA and ATAM) exclusively support the evaluation of a single architecture, six methods additionally (SAAM, ALMA, APSM, SAE, AQA, and FAAM) support the comparative evaluation of two or more architectures. All methods use scenario-based techniques for analysis, except AQA, which applies questionnaires only. Questionnaires are also supported in SAE and ATAM. PASA uses techniques such as execution path modeling [Smith 2002] for more detailed investigations of the architecture.

Concerning applicability, six of the eight methods (SAAM, ALMA, APSM, PASA, AQA, and ATAM) can be applied to software architectures of any domain. In contrast, the evaluation activities of SAE are particularly restricted to telecommunication systems, and FAAM focuses on information systems. This means that SAE and FAAM are domain-specific methods and thus are less flexible to apply to systems of a different domain.

Table 2-9: Capability Summary of Architecture Evaluation Methods

Methods	**SAAM**	**ALMA**	**APSM**	**PASA**	**SAE**	**AQA**	**FAAM**	**ATAM**
Method's Scope	Single attribute evaluation	Single attribute evaluation	Single attribute evaluation	Single attribute evaluation	Multiple attribute evaluation	Multiple attribute evaluation	Multiple attribute evaluation	Multiple attribute evaluation
Method's Goals	Identification of architectural fitness with respect to modifiability	Identification of architectural fitness with respect to modifiability	Estimation of average effort required for maintenance	Identification of architectural fitness with respect to performance	Identification of architectural strenghts and weaknesses	Identification of architectural strenghts and weaknesses	Identification of architectural strenghts and weaknesses; proposals for improvement	Identification of architectural strenghts and weaknesses
Quality Attributes	Modifiability	Modifiability	Maintenance	Performance	Error recovery, operation, administration, maintenance, performance, cost	Understandability, feasibility, openness, maintainability, evolvability, client satisfaction	Interoperability, extensibility	Multiple quality attributes
Evaluation Extent	Single architecture, multiple architectures	Single architecture, multiple architectures	Single architecture, multiple architectures	Single architecture	Single architecture, multiple architectures	Single architecture, multiple architectures	Single architecture, multiple architectures	Single architecture
Evaluation Techniques	Scenarios	Scenarios	Scenarios	Scenarios, execution models	Questionnaire, scenarios	Questionnaire	Scenarios	Questionnaire, scenarios
Application Domain	Any	Any	Any	Any	Telecommunication systems	Any	Information systems	Any
Method's Validation	Various systems in different domains	Application framework, business information system, declaration processing system	Two embedded control systems	Various systems in different domains	No information available	Three defense systems	Medical imaging system, enterprise information system	Various systems in different domains

All methods have been validated, either in case studies or in industrial projects which gives clues to the relative high maturity of the methods. Only for SAE, no record on validation experience could be found in literature. Further, except for SAAM no information about tool support for the investigated methods could be found.

As a conclusion, the main limitation of single attribute analysis methods in contrast to multiple attribute methods is that they only provide a restricted view on an architecture's quality. As discussed in Section 2.2, a system is rarely driven by exactly one quality attribute but a set of critical quality requirements. Restricting the evaluation to a particular quality attribute may result in identifying risks of a specific type only. It may not allow for a more representative (overall) assessment of the architecture.

Moreover, the findings that result from the application of single attribute methods may be used as a suggestion to improve the architecture for a particular quality attribute. This approach is critical, since optimizing a particular quality attribute of an architecture in isolation usually has a negative impact on other architectural qualities. This is because quality requirements are competing (cf. Section 2.2.1.3). Consequently, optimizing one quality on the one hand may lead to a deterioration of the overall architectural quality on the other hand. Tradeoff decisions are not sufficiently considered.

2.3.5.2 Evaluation Against Research Gaps

In this section, we provide an evaluation of the methods against the research gaps of this thesis:

- *Gap 1: Lack of guidance for identifying requirements that are essential for architecture development.*

  Research gap 1 can be confirmed. Most of the investigated approaches base the evaluation on a particular quality goal or set of quality goals. Whether the goal(s) are the most significant (i.e., drive the architecture) is not analyzed in detail. Only one method (ATAM) includes steps to prioritize and select evaluation scenarios that address the most important architectural concerns.

- *Gap 2: No systematic support for making adequate design decisions in order to implement architecturally relevant requirements.*

  This research gap is not applicable to architecture evaluation approaches.

- *Gap 3: Insufficient support for the identification and mitigation of unwanted effects of design decisions in architecture development.*

  Research gap 3 can be confirmed. Most methods restrict the evaluation to a particular quality attribute or a specific set of attributes. Since an architecture is rarely driven by a single quality attribute, many risks may remain uncovered. Moreover, the findings that result from the application of those methods may be used as a suggestion to improve the architecture for a specific quality attribute. This approach is critical, since optimizing a particular quality attribute of an architecture in isolation usually has a negative impact on other architectural qualities. This is because quality requirements are competing. For example, the optimization of an architecture with respect to performance often has a negative impact on modifiability or portability (cf. Section 2.2.1.3). Therefore, optimizing one quality on the one hand may lead to a deterioration of the overall architectural quality on the other hand. The evaluation of tradeoff decisions is not considered.

The multiple attribute analysis methods provide a more general evaluation of two or more quality attributes and may support uncovering some of the side effects in the architecture. However, only ATAM allows starting with the most critical set of quality attributes, which provides a good basis for identifying major architectural problems. Furthermore, only a few of the methods deal with risk mitigation at all (PASA and FAAM). Unfortunately, PASA is focused on performance and thus risk mitigation is primarily considered in the performance context. FAAM deals with interoperability and extensibility requirements of information systems. Thus, the risk mitigation strategies of FAAM are also restricted to these kinds of attributes and systems. The other methods do not include risk mitigation activities but focus on the identification of risks only. However, none of the methods includes steps for identifying those decisions in the architecture that lead to risks.

Finally, all investigated evaluation methods are focused on *software* architectures. This means that they do not sufficiently consider hardware aspects during the analysis (e.g., captured in the hardware architecture view). Thus, risks related to a software-hardware mismatch may remain uncovered.

2.4 Summary

In this chapter, related work in the area of architecture development has been presented. In particular, basic principles and important terms concerning software architecture and architecture development have been introduced. Furthermore, architecture design activities and artifacts have been described in more detail. The chapter has characterized the role of functional and quality requirements in architecture design as well as tradeoffs among requirements. In addition, important parts of an architecture description such as architectural views and decisions have been discussed. The chapter has further covered architectural strategies and mechanisms and their usage during architecture design. Moreover, a comprehensive overview of existing architecture design methods has been provided and a comparison of these methods with respect to the research gaps has been given. The chapter has also provided a comprehensive overview of architecture evaluation. Architectural risks, which are the primary output of an evaluation, have been discussed and important evaluation support concepts such as scenarios, use case maps, and questioning techniques have been described. Finally, existing architecture evaluation methods have been presented and compared with the research gaps of this thesis. The chapter has demonstrated that the gaps are not sufficiently covered by current research results.

In the next chapter we will introduce the approach proposed in this work for bridging the research gaps.

3 Bridging the Research Gaps with QUADRAD

As demonstrated in the previous chapter, the research gaps cannot sufficiently be addressed by related work. There is still a lack of guidance for identifying requirements that are essential for architecture development. The approaches investigated in the previous chapter do not provide adequate steps for prioritizing requirements that drive the architecture design. Most of them do not even sufficiently analyze the quality goals for architecture evaluation.

Further, the approaches do not adequately support decision making for implementing the architecturally driving requirements. Moreover, an adequate documentation of architectural decisions that lead to architectural views is largely unsupported by the approaches. The number and type of views used for documenting the architecture differs. This increases the risk that the architecture is not correctly implemented during detailed design and coding.

There is also an insufficient support for the identification and mitigation of side effects within an architecture. The approaches do not include combined design and evaluation steps. There is a high probability that unexpected behavior of the architecture cannot be identified before implementation. Furthermore, some of the evaluation approaches are restricted to a particular quality attribute. Since an architecture is rarely driven by a single quality, critical risks may remain uncovered.

These gaps demonstrate that the transition from requirements to architecture is still a highly critical phase where some of the most fundamental mistakes of a development effort are made (cf. Chapter 1). As discussed, the mistakes may lead to the fact that the intended business goals of an organization cannot be achieved, which commonly results in fundamental financial loss. This work focuses on bridging the gaps because this would support making the transition from requirements to architecture much more systematic and less error-prone. This, in turn, would assist an organization in meeting their business goals more accurately.

The chapter is organized as follows: Section 3.1 elaborates key concepts to address the development problems and research gaps of scope. It particularly shows how an architecture can be better aligned to architecturally significant requirements, which supports building systems that meet the intended business goals more accurately. In Section 3.2, the main concepts are summarized in a metamodel, which is then used to derive the basic approach to address the respective research gaps of this work. Section 3.3 elaborates a set of key requirements to further guide the refinement and packaging of the approach in a framework. The requirements are directly evolved from fundamental limitations of existing work in architecture development. In Section 3.4, the resulting framework (QUADRAD, Quality-Driven Architecture Development) is introduced. The framework systematically addresses key limitations of existing approaches and supports bridging the research gaps. Finally, in Section 3.5 the main points of the chapter are summarized.

3.1 Key Concepts to Address Research Gaps

In Chapter 1, we have reported about two major problems that still exist in current software system development (cf. Sections 1.2 and 1.3):

- *Problem 1:* Software system architectures are primarily created or evolved based on wrong requirements.
- *Problem 2:* Making appropriate design decisions during architecture development is not or only vaguely understood. This results in serious development risks.

In the context of these problems, we have further identified three important research gaps (cf. Section 1.3.1):

- *Research Gap G1:* Lack of guidance for identifying requirements that are essential for architecture development.
- *Research Gap G2:* No systematic support for making adequate design decisions in order to implement architecturally relevant requirements.
- *Research Gap G3:* Insufficient support for the identification and mitigation of unwanted effects of design decisions in architecture development.

The gaps have been further discussed and confirmed in the previous chapter (cf. Sections 2.2.5 and 2.3.5)

As a starting point for bridging the gaps, we have elaborated a set of key concepts and important interrelations between these concepts. The concepts and interrelations build up on the limitations of related work in architecture development provided in Chapter 2. Thus, they represent a fundamental basis of the research work of this thesis.

In particular, the following observations have been made:

- Architectural decisions have impact on the overall system design
- Architectural decisions are based on architectural drivers
- Architectural strategies and mechanisms support decision making
- Architectural decisions influence each other

These observations are next explained in more detail.

3.1.1 Architectural Decisions Have Impact on the Overall System Design

As discussed in Chapter 2, architecture deals with decisions that need to be made from a broad-scoped or system perspective. These decisions are known as *architectural decisions* (cf. Section 2.2.2.2). Architectural decisions are strategic decisions made during architecture design. They have implications on the overall system design and therefore have a high impact on the achievement of business goals. Any decision that could be made from a more narrowly scoped, local perspective is not architectural (at the current level of system scope).

This allows us to distinguish between detailed design and implementation decisions on the one hand and architectural decisions on the other—the former have local impact, and the latter have systemic impact, that is, huge parts of the system are affected (e.g., must be changed). A broad-scoped perspective is required to take this impact into account and to make the necessary trade-offs across the system.

Figure 3-1 illustrates the scope and impact of design decision based on the work of [Malan 2002]. Architectural decisions are shown in the upper right quarter of Figure 3-1. These decisions are systemic decisions and have a high business impact. They are made to satisfy high priority requirements that are important to the business success of the system. However, during architecture development the architect must also cope with decisions that have a different scope and impact than architectural decisions.

Usually, the architect must also make decisions that have a systemic scope but have a low business impact. In this case he or she has to provide a solution for requirements that are difficult to implement but that are not key to selling the system. These decisions are shown in the upper left quarter of Figure 3-1. They are not architectural decisions according to our definition.

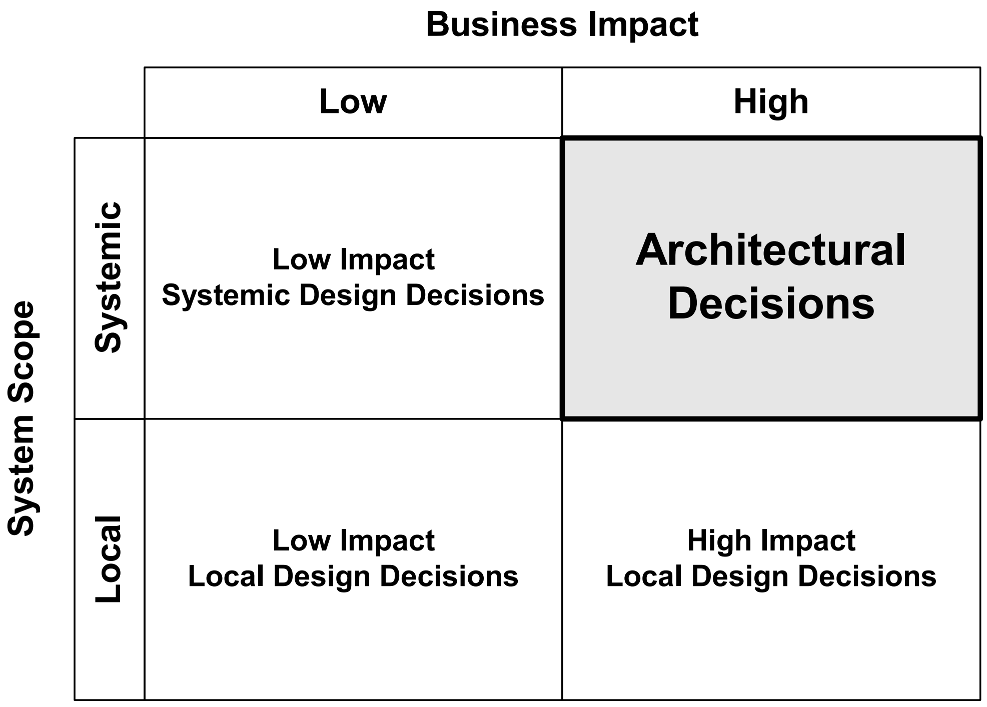

Business Impact
Low
High
System Scope
Systemic
Local
Low Impact Systemic Design Decisions
Architectural Decisions
Low Impact Local Design Decisions
High Impact Local Design Decisions

Figure 3-1: Decision Scope and Impact [Malan 2002]

On the other hand, the architect must also make decisions that have a strong impact on the business strategy but only affect local parts of the system (see lower right quarter of Figure 3-1). These decisions are also not architectural. Typically, they are tactical in nature. This means that the decisions set architectural guidelines and policies, which are usually implemented during detailed design or coding.

Finally, decisions with a low system and business impact are usually made during detailed design (see lower left quarter of Figure 3-1). They do not address the primary concerns of architecture development but may refine architectural decisions at a detailed design level.

3.1.2 Architectural Decisions are Based on Architectural Drivers

Consequently, the design of a successful system involves making appropriate architectural decisions. Generally speaking, design decisions are made to achieve particular requirements. As previously discussed, some requirements may be important for the business perspective but only have a local impact on the system design – these requirements do not drive architecture design. Others may have a systemic impact but are of low business importance – these requirements are also not key for architecture development.

Consequently, only a particular portion of the given requirements is important from an architectural perspective. These requirements are called architecturally driving requirements or *architectural drivers* (cf. Section 2.2.1.4). Typically, the set of architectural drivers consists of a combination of quality requirements that address attributes such as modifiability, performance, safety, reliability, and security.

3.1.3 Architectural Strategies and Mechanisms Support Decision Making

In the previous sections, we have discussed that it is important to capture the most critical architectural design requirements. The next step is to identify and implement appropriate solutions for those requirements at the architecture level. In other words, we have to make *appropriate* decisions in order to achieve this set of driving requirements.

Making architectural decisions to accomplish a particular architectural driver involves two fundamental design activities:

- Introduce new design elements into the architecture
- Refine or change the behavior of existing design elements

Applying these activities leads to architecture releases with new capabilities. A new release, for example, may satisfy new architecturally relevant requirements or it may improve a specific behavior according to particular quality attributes. The process from one release to another is called architecture iteration.

But how do we know that we have made appropriate decisions? According to our observation obtained from different industrial projects (e.g., [Thiel 2001a, Thiel 2001b, Thiel 2002b]) as well as from observations of other authors (e.g., [Hofmeister 2000], [Bass 2003], [Garland 2003]), experienced architects apply specific principles or *architectural strategies* to solve design problems that have been induced by requirements. An architectural strategy describes a general principle to solve a particular class of design problems (cf. Section 2.2.3.1). For example, improving the communication efficiency among software components is a general strategy for addressing performance problems.

Whereas architectural strategies describe the basic approach to be used to address design problems, an *architectural mechanism* provides a concrete design solution for that problem – it *implements* architectural strategies. This implementation is often documented by a collection of component and connection types and their basic responsibilities. Architects frequently make use of particular mechanisms included in architectural styles [Shaw 1996] and patterns [Gamma 1994, Buschmann 1996] to provide an architectural solution of one or more strategies in a given context (cf. Section 2.2.3.2).

Consider the example given in Figure 3-2. Assume an architect has to satisfy the *portability* requirement that "the system shall execute on different hardware platforms." An appropriate strategy to achieve portability could be to *decouple software functionality from the underlying hardware*. This strategy can be accomplished by a *virtual machine* mechanism [Buschmann 1996]. The decision to implement this mechanism would cause a change of the current architecture release. This change involves the introduction of a virtual machine layer into the architecture as well as the adaptation of the application and communication subsystems, as illustrated in Figure 3-2. In summary, the portability requirement has been fulfilled by providing a strategy that supports portability and a mechanism that implements this strategy.

3.1.4 Architectural Decisions Influence Each Other

Most architectural decisions involve tradeoffs. For example, the architect of Figure 3-3 decides to introduce a virtual machine to achieve the given portability requirement. However, this decision involves a performance tradeoff – system calls from the application and communication layer are interpreted by the virtual machine. This interpretation is slower than serving the calls directly and may have a significant impact on the system's run-time.

This example shows that making an architectural decision to achieve a particular requirement usually has impact on more than one system quality. Because some requirements – like those for portability and performance above – are competing, introducing a design solution to support one of them may have a negative impact on other qualities.

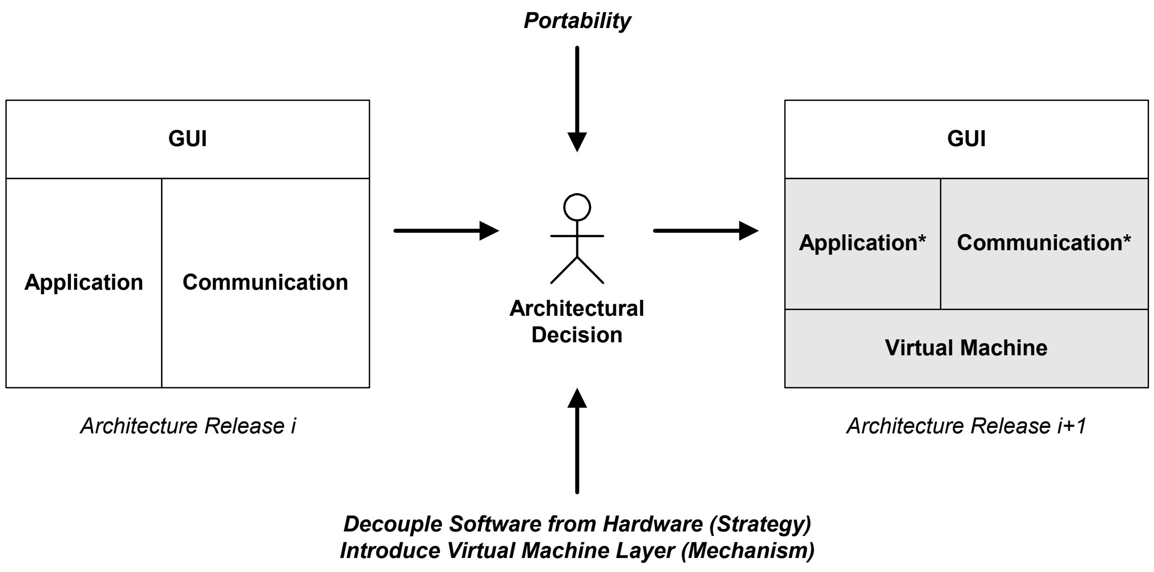


Figure 3-2: Applying a Virtual Machine Mechanism

Figure 3-3 illustrates this fact in a Kiviat diagram (e.g., [Lanza 2003]).[2] The diagram shows that an architectural decision may support portability, interoperability, and modifiability but it may conflict with performance and security. Thus, the tradeoffs of each decision must be carefully considered.

Another problem is related to the fact that architectural design is a decision making process in which the architect is faced with hundreds of design problems to resolve. Some of the problems may be known beforehand, others may show up after several iterations or releases such that the decisions cannot be all made in concert but must be made incrementally. This is natural in practice. As a consequence, the decision making process cannot be controlled completely and may introduce additional side effects into the architecture. These side effects may induce architecture risks that prevent that the adequate system can be built based on the architecture (cf. Section 2.3.2.1).

Risks can creep in an architecture in the following way: During architecture design, strategies and mechanisms are usually determined for each particular requirement under consideration, i.e., iteratively and more or less separated from other (perhaps competing) requirements. The decisions that result from applying a mechanism usually include a solution relative to the design problem induced by the requirements. The composition of mechanisms that are introduced through the decision making process usually leads to sweeping architectural implications. These implications are difficult to manage and control. Consequently, they may result in unexpected behavior of the architecture.

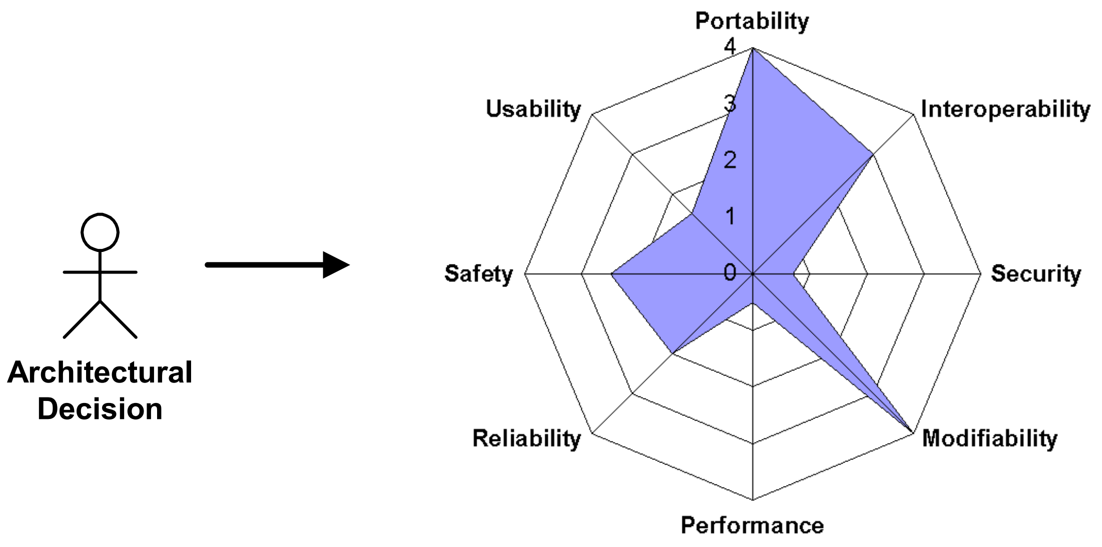


Figure 3-3: Impact of Architectural Decisions on Multiple Quality Attributes

[2] A Kiviat diagram, also called radar or net chart, resembles a wagon wheel. Each spoke represents one factor or characteristic of analysis. The degree to which a factor is positioned is displayed at a point on a spoke. Zero position is at the center (hub) of the diagram, with the highest position being at the rim of the wagon wheel. The connected points result in a polygon.

3.2 Metamodel and Approach

Figure 3-4 summarizes the observations elaborated in the previous section in an UML-based metamodel. The metamodel includes the following circumstances:

- Architectural decisions address architecturally driving requirements (cf. Section 3.1.2).
- Architectural decisions are based on architectural strategies and architectural mechanisms (cf. Section 3.1.3).
- Architectural mechanisms provide implementation guidelines for architectural strategies (cf. Section 3.1.3).
- Architectural decisions may influence each other (cf. Section 3.1.4).
- The application of architectural decisions leads to an architecture release, which represents a particular status of the architecture (cf. Section 3.1.4).
- An architecture release satisfies a set of architecturally driving requirements (cf. Section 3.1.1).
- An architecture release may also include risks that preclude the achievement of particular architecturally driving requirements (cf. Section 3.1.4).

By taking the metamodel of Figure 3-4 into account we can derive the following five fundamental steps as part of the approach in order to address the research gaps (see Figure 3-5):

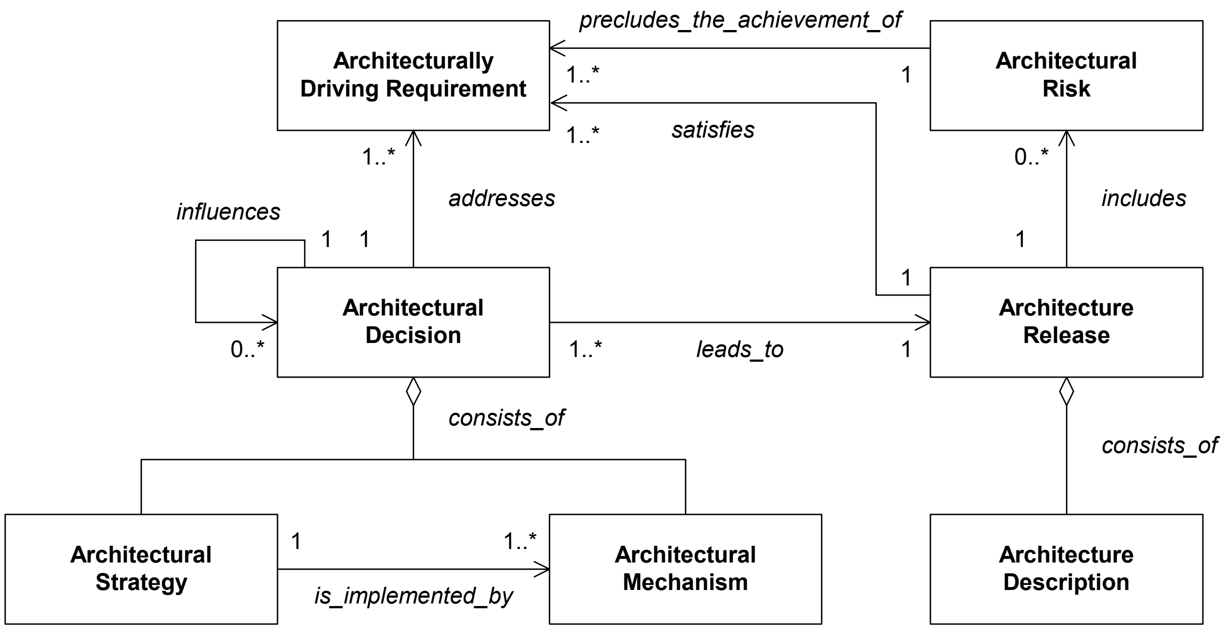


Figure 3-4: Key Concepts of the Approach

Step 1: Identify the requirements that have architectural relevance. Determine the architecturally driving requirements that shape the architecture. These drivers typically consist of a combination of quality requirements that address attributes such as modifiability, performance, safety, reliability, and security. They are a prerequisite for building architectures that meet business goals and market needs. The result of this step is the set of architectural drivers that is used throughout architecture development.

Step 2: Derive appropriate architectural decisions. Find strategies and mechanisms that address architecturally relevant requirements. Architectural strategies describe an architect's basic principles for addressing a design problem induced by the architectural driver under consideration. Based on the strategies, appropriate architectural mechanisms can be derived. These mechanisms include particular decisions that help to address the drivers, which is the result of this step.

Step 3: Apply the decisions at the architectural level. Applying the decisions at the architectural level means to introduce new design elements or modify the behavior of existing design elements according to the decision. This activity usually affects different concerns or views of the architecture (e.g., software, hardware, and mechanics) simultaneously. The result of this step is a description of the architecture. Typically, steps 2 and 3 are performed highly iterative.

Step 4: Evaluate the effects of architectural decisions from an overall system perspective. Consider the consequences of architectural decisions since they concern the overall system design. Locate hidden implications and side effects that prevent the architecture satisfying the architectural drivers. The result of this step is the set of risks that stem from inappropriate decisions or from decisions that have not been made yet.

Step 5: Iterate architecture development. Use the feedback from step 4 to eliminate risks and revise the architecture. Iterate architecture development when feedback from implementation (of parts of the system) is available or when requirements or business goals have been changed.

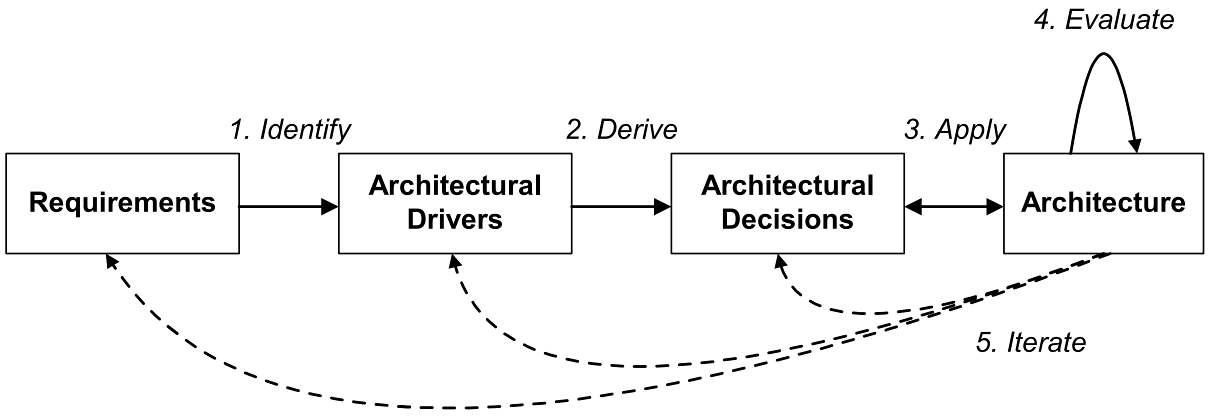


Figure 3-5: Fundamental Approach

3.3 Refining and Packaging the Approach

In order to guide the refinement of the approach provided in the previous section into a framework, we have elaborated a set of key requirements. The elicitation of those requirements is based on the three research gaps relevant to this work. Table 3-1 shows the eight most significant requirements to be considered by the approach. Each requirement has an associated description and addresses a particular research gap.

Table 3-1: Requirements Derived From Research Gaps

Framework Requirements		**Description**	**Related Research Gap**
FR1	Prioritization of Requirements	The approach shall support the prioritization of requirements in order to evaluate their importance for architecture design.	Gap 1
FR2	View Type Definition	The approach shall support the definition of architecture view types in order to allow a language for documenting relevant architecture views.	Gap 2
FR3	Multiple View Support	The approach shall support architecture modeling of multiple views in order to improve the accuracy of the architecture description.	Gap 2
FR4	Appropriateness of Architectural Decisions	The approach shall support the identification of appropriate decisions for implementing the architecturally relevant requirements.	Gap 2
FR5	Traceability of Architectural Decisions	The approach shall support the documentation of traceability information between architecturally relevant requirements and their implementation in architecture.	Gap 2
FR6	Multiple Attribute Evaluation	The approach shall support architecture evaluation with respect to the achievement of multiple quality attribute requirements.	Gap 3
FR7	Architecture Impact Analysis	The approach shall support the analysis of the implementation of architectural decisions based on the architecture description.	Gap 3
FR8	Identification of Architectural Risks and Mitigation Strategies	The approach shall support the identification of risks associated with the implementation of an architectural decision. It shall also include activities for recording risk mitigation strategies.	Gap 3

The particular rationales that lead to the specification of the requirements are as follows:

- **FR1: Prioritization of Requirements**

  Rationale: None of the investigated approaches provides steps for an adequate prioritization of requirements that drive the architecture design (cf. Section 2.2.5).

- **FR2: View Type Definition**

  Rationale: The number and type of supported views in the investigated approaches differs. The view types are not adapted to those perspectives that need to be represented when the significant requirements are implemented (cf. Section 2.2.5).

- **FR3: Multiple View Support**

  Rationale: Since the representation of architectural views in the investigated approaches is restricted, the accuracy of the architecture description is limited. This increases the risk that the architecture is not correctly implemented during detailed design and coding (cf. Section 2.2.5).

- **FR4: Appropriateness of Architectural Decisions**

  Rationale: None of the investigated approaches does directly document the architectural decisions that lead to the architectural views. Thus, much of the information about why the architecture is the way it is gets lost. This is critical because then a sufficient understanding of the architecture cannot be achieved (cf. Section 2.2.5).

- **FR5: Traceability of Architectural Decisions**

  Rationale: None of the investigated approaches deal with documenting the rationales behind architectural decisions (cf. Section 2.2.5).

- **FR6: Multiple Attribute Evaluation**

  Rationale: Most methods restrict the evaluation to a particular quality attribute or a specific set of attributes. Since an architecture is rarely driven by a single quality attribute, many risks may remain uncovered (cf. Section 2.3.5).

- **FR7: Architecture Impact Analysis**

  Rationale: None of the methods makes it explicit how to identify the decisions in the architecture that lead to risks (cf. Section 2.3.5).

- **FR8: Identification of Architectural Risks and Mitigation Strategies**

  Rationale: Most of the investigated approaches do not include risk mitigation activities but focus on the identification of risks only. The approaches that deal with mitigating risk are restricted to a particular quality attribute or small set of attributes (cf. Section 2.3.5).

The requirements help to package the approach into a framework. We have called the framework QUADRAD (Quality-Driven Architecture Development). An overview of QUADRAD is given in the next section.

3.4 An Overview of the QUADRAD Framework

To address the requirements and to bridge the research gaps elaborated in the previous sections, we propose the QUADRAD framework. QUADRAD supports the Quality-Driven Architecture Development of software-intensive systems. It provides essential activities that guide the design, assessment, and documentation of architectures.

For illustrating the activities within QUADRAD we adopted the data flow diagram (DFD) notation proposed by Gane and Sarson [Gane 1979]. Data flow diagramming is a widely used structured technique for showing the activities performed by a system and the data flowing into, out of, and within it. In this context "system" means the QUADRAD framework. Table 3-2 summarizes the DFD notation used throughout this work.

The general principle in DFD is that a system can be decomposed into subsystems, and subsystems can be decomposed into lower level subsystems, and so on. Each subsystem represents an activity in which data is processed. The top-level DFD (also known as level-0 diagram) only contains one activity that generalizes the function of the entire system in relationship to external entities. The first level DFD (level-1 diagram) shows a decomposition of the level-0 activity in main level-1 activities. Each of the activities can be further decomposed into finer-grained activities. At the lowest level, activities can no longer be decomposed. Each activity is then described in textual form to capture the essence of how the activity is performed.

Just as a system must have input and output (if it is not dead), so an activity must have data flowing into or from that activity. Data can enter the system from the environment and from data flows between activities within the system. Data is produced as output from the system. Figure 3-6 shows the top-level DFD of the framework illustrating the external entities and data stores as well as the data flows that enter and leave it.

In particular, the following data items are provided by external entities:

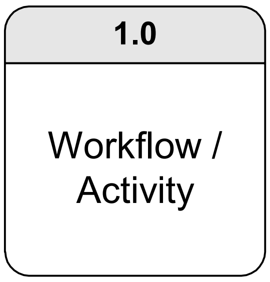
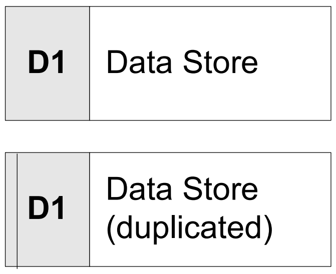
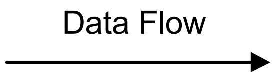
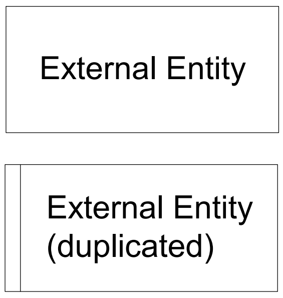
- Analysis questions and evaluation skills (architecture evaluation team)
- Domain knowledge (project stakeholders)
- Requirements specification (requirements engineer)
- Business goals (business executive)
- Architecture and domain knowledge (software architect)

QUADRAD produces the following results:

- D1: Architectural drivers and view types
- D2: Architecture description
- D3: Design issues
- D4: Architectural risks
- D5: Architectural solutions

QUADRAD also provides architecture adaptation requests to be managed by the software architect.

Table 3-2: DFD Modeling Notation [Gane 1979]

Element	Notation	Description
Workflow / Activity		Workflows and activities transform or manipulate data. Each box has a unique number as identifier and a unique name. Workflows and activities transform or manipulate input data to produce output data.
Data Store		Data Stores are locations where data is held temporarily or permanently. It is common practice to have duplicates of data stores to make a diagram less cluttered.
Data Flow		Data Flows depict data or information flowing to or from an activity. The arrows must either start or end at an activity box. It is impossible for data to flow from data store to data store except via an activity. External entities are not allowed to access data stores directly. The arrows are labeled with the name of the data that moves through it.
External Entity		External entities are objects outside the system (e.g., people or machines) which contribute data or information to the system or which receive data or information from it. As with data stores duplication of external entities is possible to improve readability of diagrams.

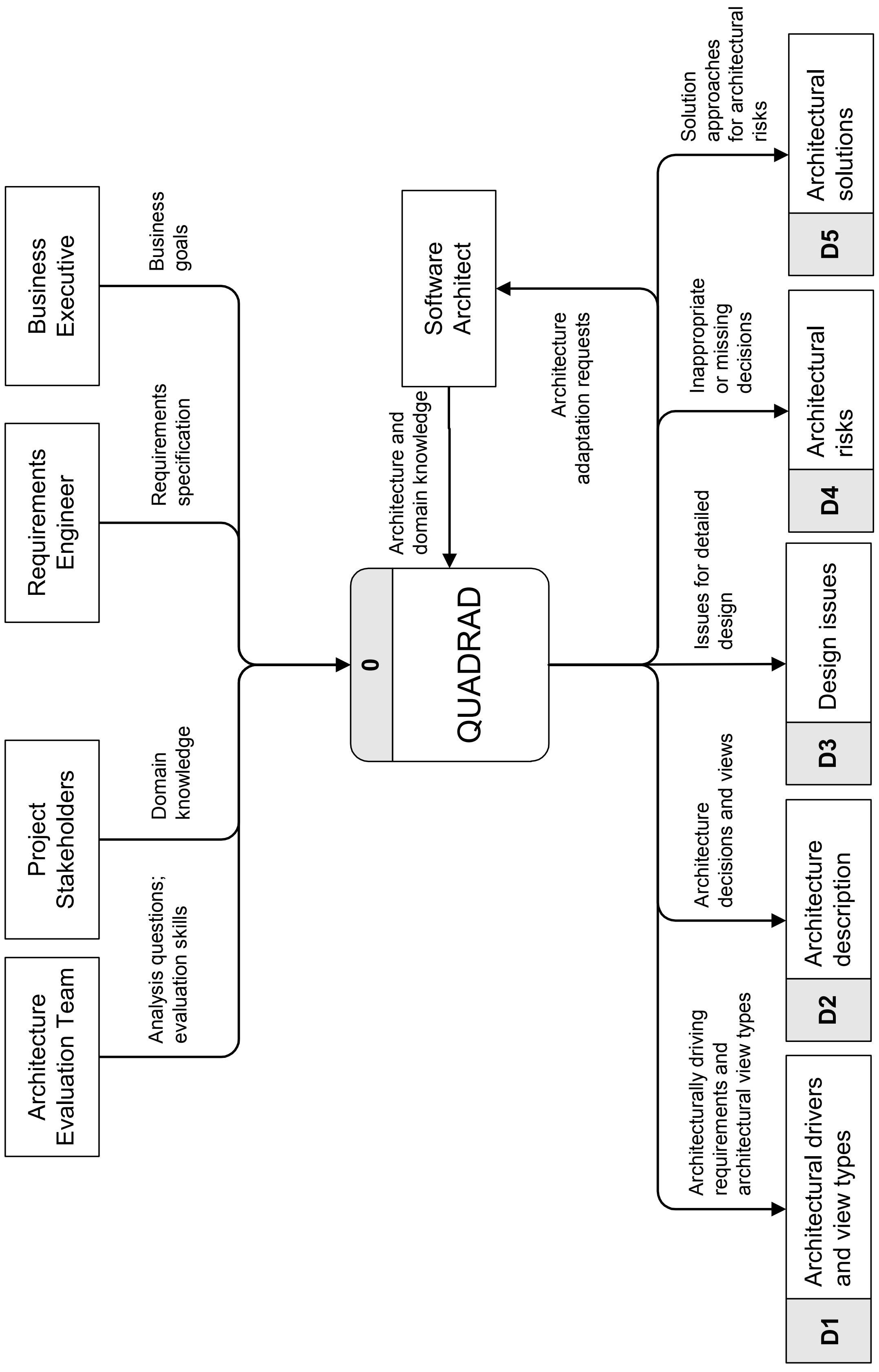

Business Executive
Business goals
Requirements Engineer
Requirements specification
Project Stakeholders
Domain knowledge
Architecture Evaluation Team
Analysis questions; evaluation skills
Software Architect
Architecture and domain knowledge
Architecture adaptation requests
0
QUADRAD
Solution approaches for architectural risks
Inappropriate or missing decisions
Issues for detailed design
Architecture decisions and views
Architecturally driving requirements and architectural view types
D5 Architectural solutions
D4 Architectural risks
D3 Design issues
D2 Architecture description
D1 Architectural drivers and view types

Figure 3-6: QUADRAD Context Diagram

Figure 3-7 shows the first decomposition of QUADRAD into the three core workflows Preparation, Modeling, and Evaluation. Each workflow consists of a sequence of activities that produce essential results for driving architecture development.

The purpose of the *Preparation* workflow (activity 1.0) is to support early architectural considerations. In particular, it includes activities for identifying architectural drivers and for defining architectural view types of concern for representing the architecture. Details of the Preparation workflow are described in Chapter 4.

The purpose of the *Modeling* workflow (activity 2.0) is to provide techniques and guidelines for modeling architectural views and to record the decisions that lead to those views. It provides essential steps to make this process more systematic and repeatable. The activities of this workflow include determining strategies and mechanisms for architectural drivers, creating an appropriate architectural infrastructure, and refining the architecture description. The Modeling workflow relies partially on artifacts of the Preparation workflow. Details of the Modeling workflow are described in Chapter 5.

The purpose of the *Evaluation* workflow (activity 3.0) is to analyze the architecture with respect to the achievement of essential quality attributes. The Evaluation workflow enables to make the consequences of architectural decisions in architectural views explicit and helps to locate side effects caused by the application and composition of design mechanisms. It also assists in identifying architecturally important decisions that have not been made yet and solutions to critical design risks. The main activities of this workflow are defining the evaluation goals, eliciting evaluation scenarios, mapping scenarios and identifying decisions, analyzing the architecture and with respect to quality attribute goals, providing solution approaches, and summarizing the findings. Details of the Evaluation workflow are described in Chapter 6.

The architecting workflows of QUADRAD are interrelated, as shown by the data flow arrows in Figure 3-7. In order to enable support of incremental architecture development, the workflows must not necessarily be performed in a specific sequence. For example, starting at a given status of the architecture (which has been developed, e.g., in the Modeling workflow), one could decide to continue with Evaluation in order to assess the architectural properties of the current design iteration. Another possibility could be to go back to the Preparation workflow to reconsider some of the architecturally relevant requirements. In order to adapt the requirements specification, requirements change requests might be considered by the requirements engineer.

The requirements of the approach derived from research gaps (cf. Table 3-1) are addressed by the QUADRAD workflows. This will be further shown at the end of the Chapters 4, 5, and 6. In Chapter 7 we will present a tool that supports essential activities of the framework.

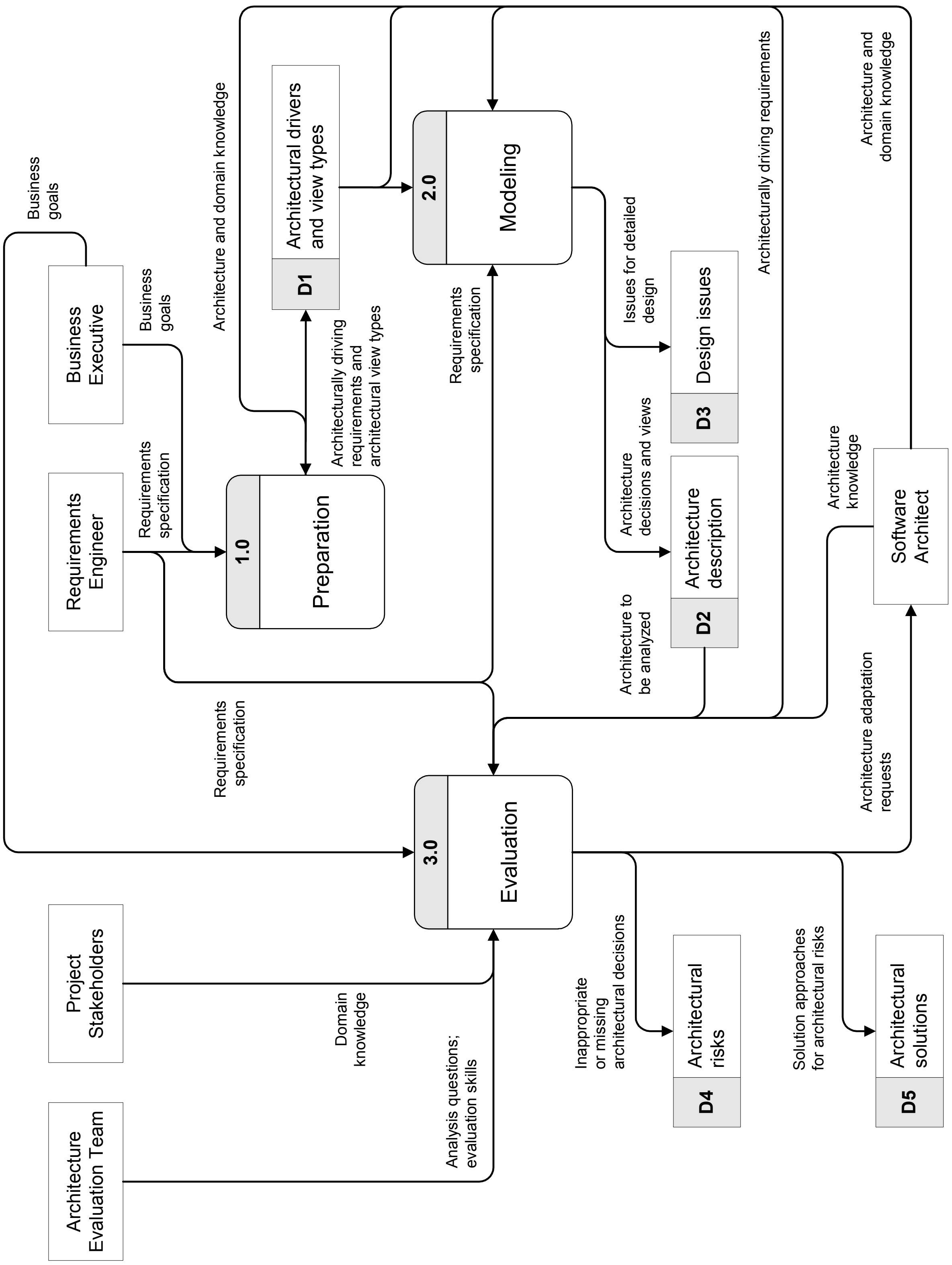

Business Executive
Requirements Engineer
Project Stakeholders
Architecture Evaluation Team
Business goals
Business goals
Requirements specification
Requirements specification
1.0
Preparation
D1
Architectural drivers and view types
Architecture and domain knowledge
Architecturally driving requirements and architectural view types
2.0
Modeling
Requirements specification
Issues for detailed design
D3
Design issues
Architecture decisions and views
D2
Architecture description
Architecture to be analyzed
Architecturally driving requirements
Architecture and domain knowledge
Architecture knowledge
Software Architect
3.0
Evaluation
Architecture adaptation requests
Domain knowledge
Analysis questions; evaluation skills
Inappropriate or missing architectural decisions
D4
Architectural risks
Solution approaches for architectural risks
D5
Architectural solutions

Figure 3-7: Core Workflows of QUADRAD (Level-1 Diagram)

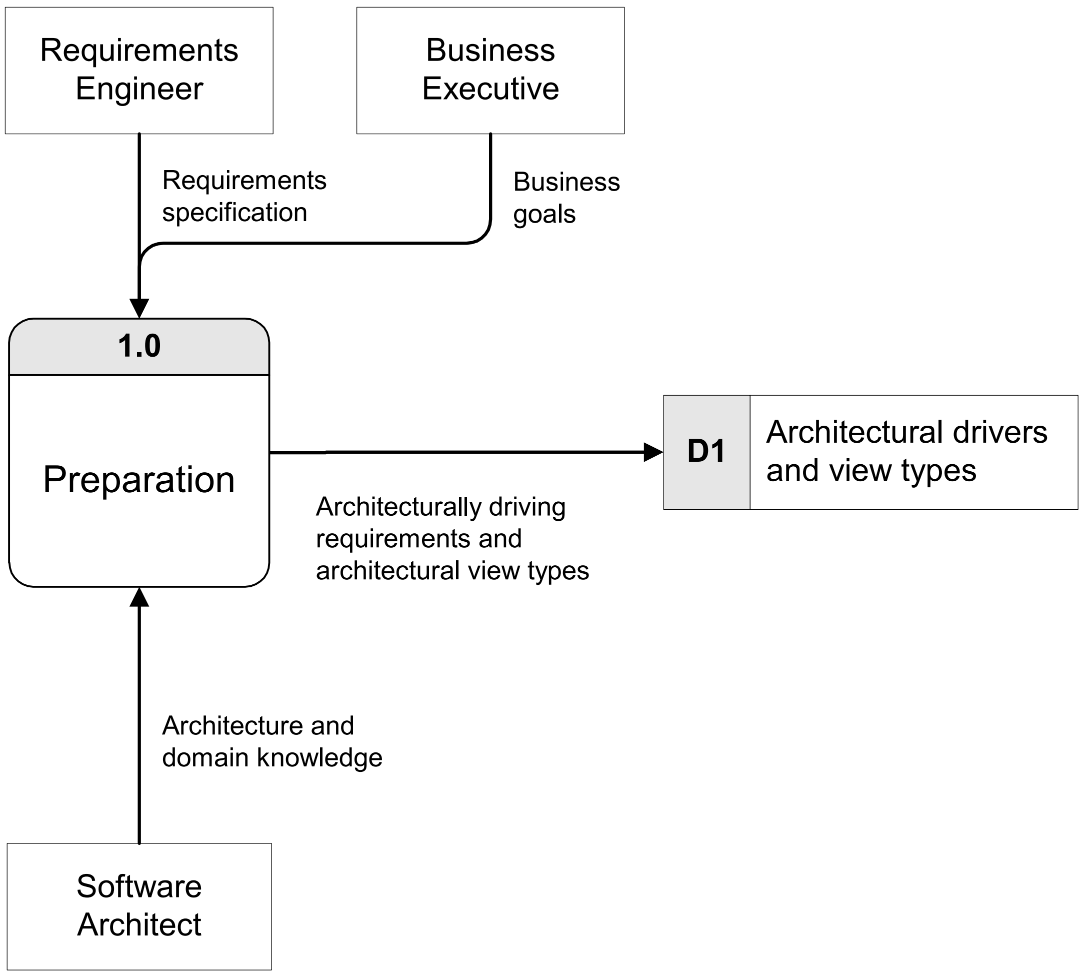
3.5 Summary

In this chapter, key concepts to address the development problems and research gaps of scope have been elaborated. In particularly, it has been outlined how an architecture can be better aligned to architecturally significant requirements and how this supports building systems that meet the intended business goals more accurately. Furthermore, a metamodel that summarizes the main concepts has been presented. This metamodel has been used to derive the basic solution approach to address the respective problems and research gaps of this work. In addition, a set of key requirements to further guide the refinement and packaging of the approach has been elaborated. Further, it has been shown how the requirements have directly been evolved from limitations of existing work in architecture development. Finally, an overview of the QUADRAD framework has been given. Particularly, the notation used to describe QUADRAD activities has been explained and the core workflows have been briefly introduced.

In the subsequent chapters, we will describe the activities of the QUADRAD framework in more detail.

4 The QUADRAD Preparation Workflow

This chapter describes the QUADRAD Preparation workflow (QUADRAD-P). The purpose of the workflow is to support early architectural considerations. QUADRAD-P aims at bridging the gap between requirements specification and architecture design by providing activities that support architects in preparing the essential design requirements systematically. An overview of the major input and output artifacts is shown in Figure 4-1.



Requirements Engineer
Business Executive
Requirements specification
Business goals
1.0
Preparation
Architecturally driving requirements and architectural view types
D1
Architectural drivers and view types
Architecture and domain knowledge
Software Architect

Figure 4-1: The QUADRAD Preparation Workflow (Level-1 Diagram)

This chapter is organized as follows: Section 4.1 provides an overview of the Preparation workflow. In Section 4.2 activities for selecting the set of requirements that are most essential for developing an adequate architecture are described. Activities for determining architectural view types addressed by architectural drivers are provided in Section 4.3. Section 4.4 describes how the QUADRAD-P activities address the research gaps in this thesis. Finally, Section 4.5 summarizes the main issues of this chapter.

4.1 Workflow Overview

Figure 4-2 shows the first decomposition of QUADRAD-P. It comprises two activities:

1. Identify architectural drivers
2. Determine architectural view types

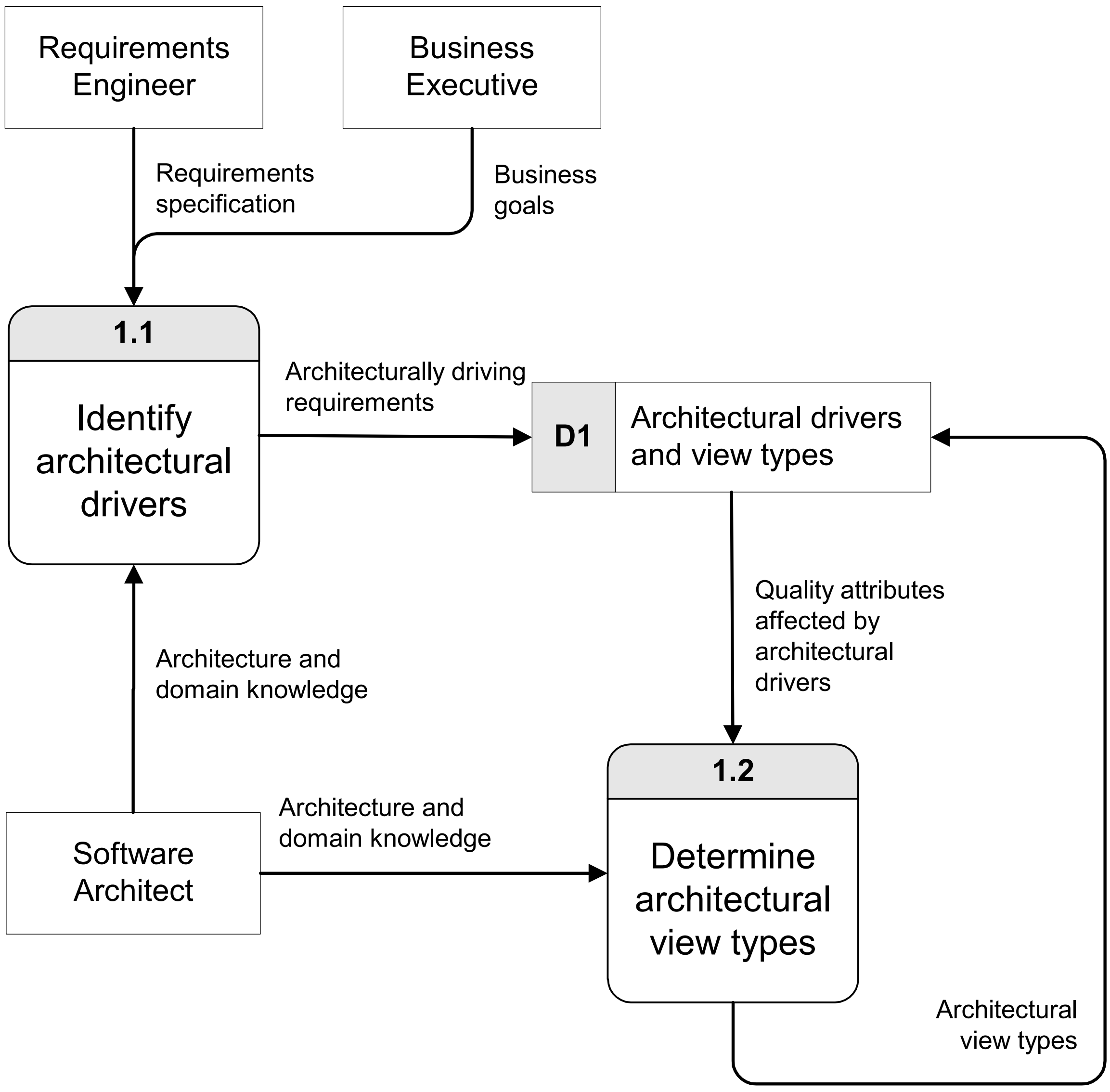

Requirements Engineer
Business Executive
Requirements specification
Business goals
1.1
Identify architectural drivers
Architecturally driving requirements
D1
Architectural drivers and view types
Quality attributes affected by architectural drivers
Architecture and domain knowledge
1.2
Determine architectural view types
Software Architect
Architecture and domain knowledge
Architectural view types

Figure 4-2: The QUADRAD Preparation Workflow (Level-2 Diagram)

1. Identify architectural drivers

The objective of this activity is to identify the set of requirements that have a strong impact on the architecture and that are important from the business perspective.

The inputs to the activity are the requirements specification (provided by the requirements engineer), the business goals of the system (provided by the business executive) as well as architecture and domain knowledge (provided by the software architect). The output from the activity is a list of architecturally driving requirements.

2. Determine architectural view types

The objective of this activity is to specify the view types that are necessary for representing the relevant architectural views for the set of architectural drivers.

The inputs to the activity are the quality attributes affected by the architecturally driving requirements (provided by the activity "Identify architectural drivers") as well as architecture and domain knowledge (provided by the software architect). The output from the activity is a list of architectural view types which is relevant for representing the architectural views for the set of architectural drivers.

The activities of QUADRAD-P are next described in more detail.

4.2 Identify Architectural Drivers

The objective of this activity is to identify the set of requirements that have a strong impact on the overall software architecture and that are essential for achieving the business goals of the system under consideration. These requirements are known as architectural drivers. The identification of such drivers is based on the fact that some requirements are more influential on the design of an architecture than others. Architectural drivers *shape* the architecture and make the system unique, competitive, and worth building. They are a prerequisite for designing the appropriate architecture and thus support systematic system development. An overview of the input and output data flow for identifying architectural drivers is shown in Figure 4-3.

Figure 4-4 shows how this activity is decomposed into the following two subordinate activities:

1. Calculate priorities
2. Determine high-priority requirements

These activities are next described in more detail.

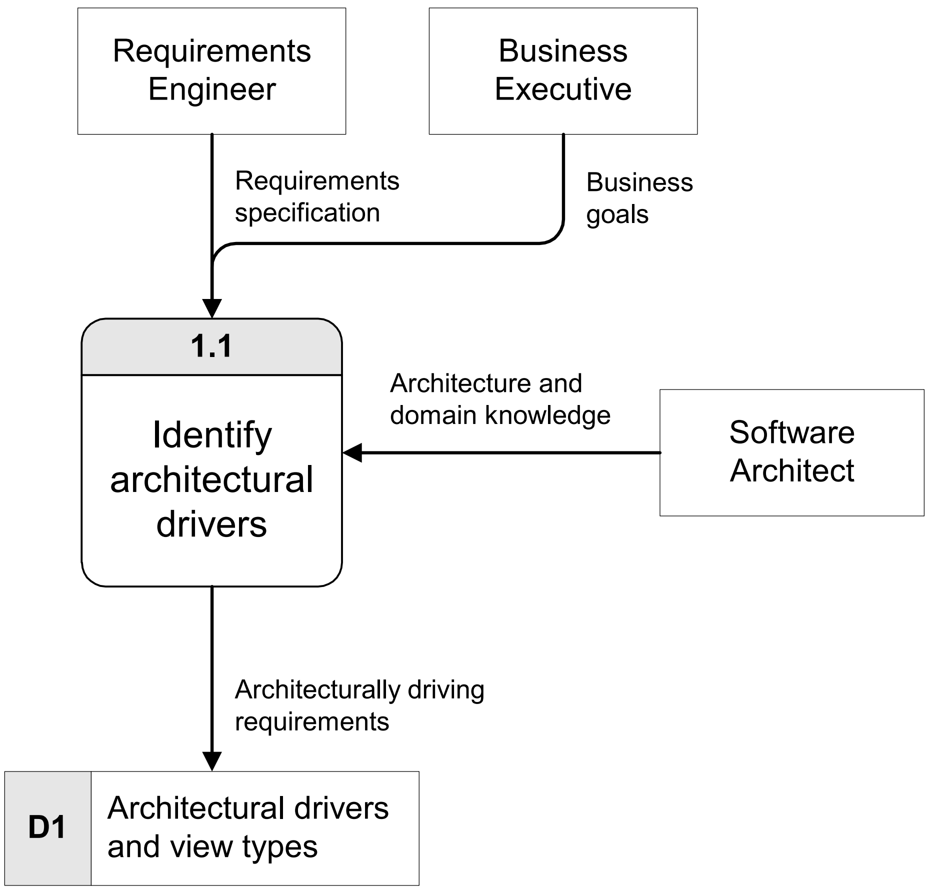

Requirements Engineer
Business Executive
Requirements specification
Business goals
1.1
Identify architectural drivers
Architecture and domain knowledge
Software Architect
Architecturally driving requirements
D1
Architectural drivers and view types

Figure 4-3: Identifying Architectural Drivers (Level-2 Diagram)

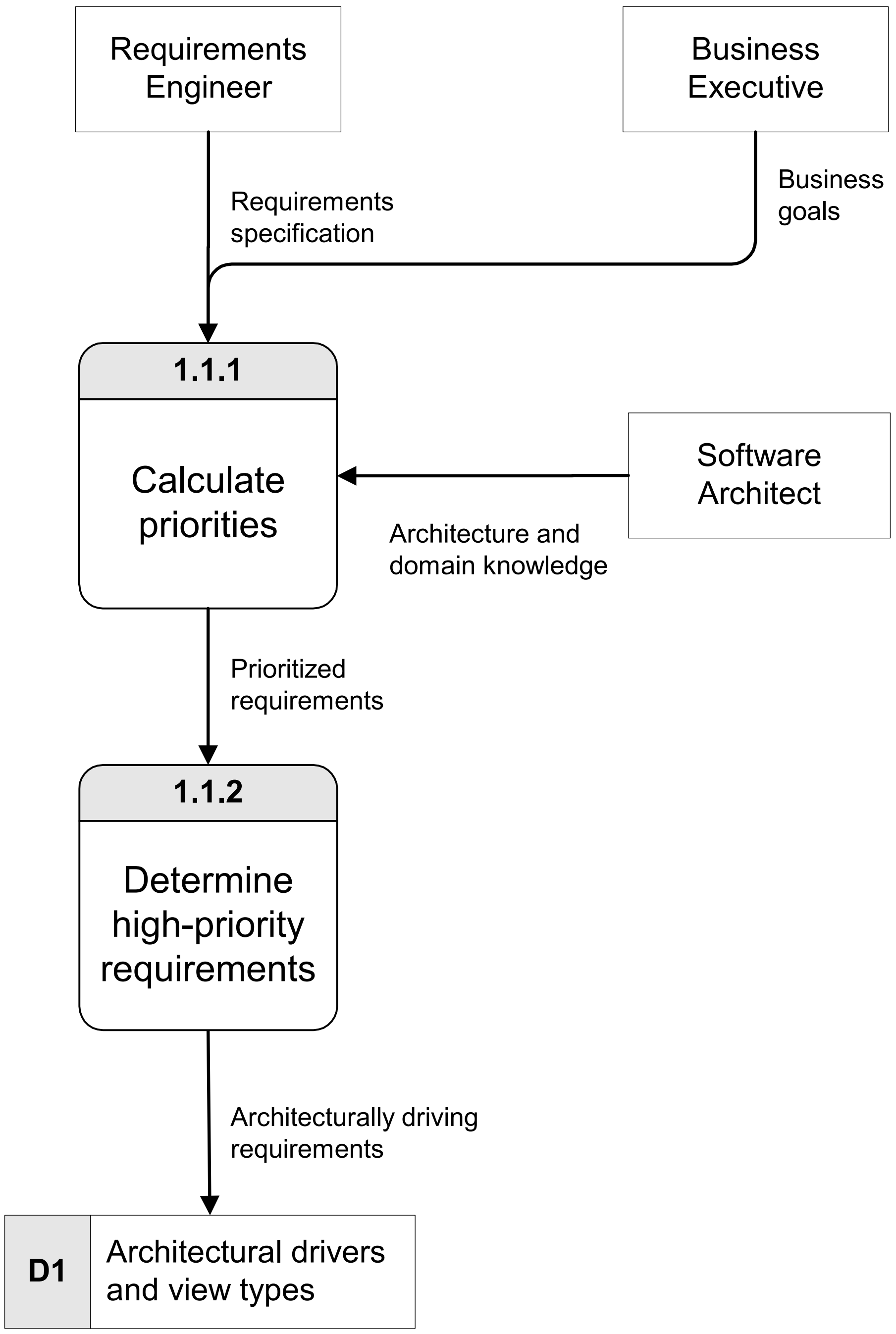

Requirements Engineer
Business Executive
Requirements specification
Business goals
1.1.1
Calculate priorities
Software Architect
Architecture and domain knowledge
Prioritized requirements
1.1.2
Determine high-priority requirements
Architecturally driving requirements
D1
Architectural drivers and view types

Figure 4-4: Identifying Architectural Drivers (Level-3 Diagram)

4.2.1 Calculate Priorities

The objective of this activity is to assign priorities to the given set of requirements. For each requirement a rank that reflects the importance of the requirements for architecture design is assigned.

In order to identify the relevance of requirements for architecture design we suggest the consideration of two criteria for prioritization: Business importance and architectural impact (cf. Figure 4-5).

- *Business importance:* A requirement is important for architecture design if it is essential for achieving business goals of the system. A software system is built because an organization wants to achieve particular goals with this system. The correlation of a requirement to business goals is thus a good indicator for determining the architectural design relevance of that requirement. Examples of business goals include the following:
  - Increase market share in region R by X%
  - Reduce development costs by Y%
  - Achieve market price of P€
  - Guarantee system availability of Z%
  - Establish standard S
- *Architectural impact:* A requirement is important for architecture design if it has a high impact on the structure of the architecture. This means that the implementation of the requirement forces the architect to make design decisions that affect large parts of the system's structure. The decisions then have implications on the overall system and thus represent key advances in the course of design. For example, a requirement that requires the architect to introduce a standard communication mechanism for all the software components within a system is more important for architecture design than a requirement that is concerned with changing colors in the graphical user interface.

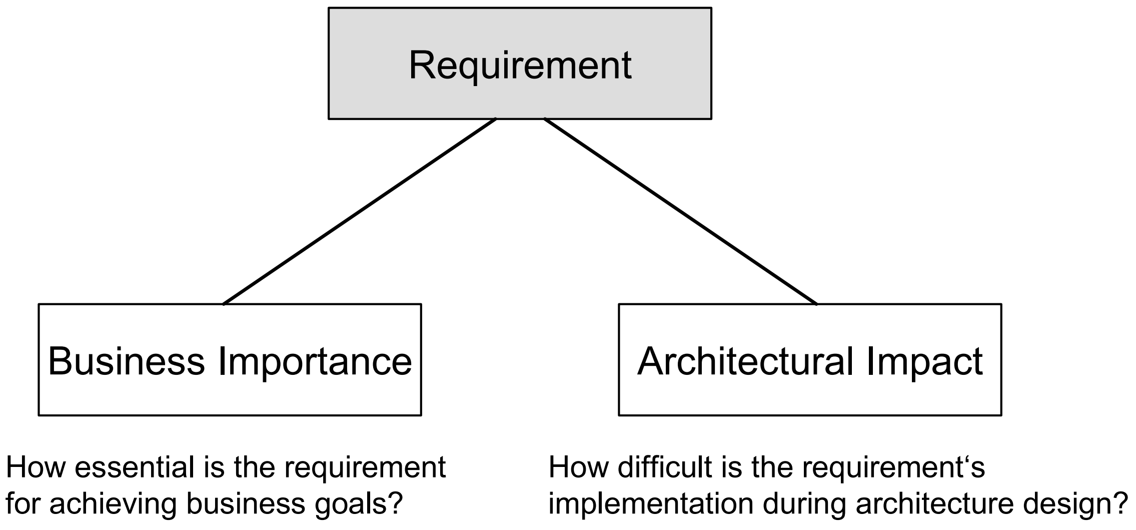

Requirement
Business Importance
Architectural Impact
How essential is the requirement for achieving business goals?
How difficult is the requirement‘s implementation during architecture design?

Figure 4-5: Criteria for Identifying Architectural Drivers

The activity comprises two steps which are performed in concert for prioritizing each requirement in the requirements specification:

1. Assign business importance
2. Assign architectural impact

1. Assign business importance

The goal of this step is to prioritize a requirement according to its business relevance. For each requirement business representatives, such as the business executive or marking manager for the system, judge the business relevance of that requirement. We suggest using the following simple scale to reflect the business importance of each scenario:

- *High* – the requirement is essential for the business success of the system
- *Medium* – the requirement is important but less essential for business success
- *Low* – the requirement is not essential for the business success of the system

The scale helps the business representative to manage assigning the business relevance for each requirement. In order to keep the assignment understandable we suggest that the business representative gives a short rationale why a particular requirement is essential or not essential for the business success of the system. The rationale is recorded for future reference.

For example, assume that one of the business goals of a development effort for airbag systems is to increase market share in the Asian region by factor two. Assume further that the legal regulations in Asia require a safe deployment of the airbag before disposal. Then a requirement which states that "the system shall include a disposal functionality for safe airbag deployment" would be highly important for business success. If the requirement is not achieved, then the airbag systems cannot be sold and the market will be lost.

2. Assign architectural impact

The goal of this step is to prioritize a requirement according to its architectural impact. For each requirement, the software architect determines the ranking for the system impact of each requirement. We suggest using a similar scale to that described in the previous step to reflect the architectural impact of a scenario:

- *High* – the requirement has a high impact on the architecture; it is difficult to achieve the requirement in the architecture
- *Medium* – the requirement has a moderate impact on the architecture; it is less difficult to achieve the requirement in the architecture
- *Low* – the requirement only has a low impact on the architecture; it is easy to achieve the requirement in the architecture

As explained before, a high architectural impact means that the implementation of the requirement will force the architect to make design decisions that affect large parts of the system's structure. According to our experience this explanation does work very well in practice when estimating the impact of incorporating requirements into an existing architecture (or parts of the architecture). In this case it is usually easy for an architect to work out the effort involved in restructuring the existing architecture.

However, at the very beginning of a new development effort there might be no existing structure from which to start estimating the architectural impact of requirements. In this case we suggest that the architect should focus on his or her experience: Does he or she expect that the requirement can be localized to an individual design element? If yes, then the architectural impact of the requirement will be low. If not, then it seems to be a "crosscutting" problem. Fulfilling it at the architecture level might require a substantial effort. Multiple design elements might need to be specified in order to achieve the concerns of the requirement. Thus, the requirement will be assigned a medium or even high impact.

As discussed in Section 2.2.1.2, quality attribute requirements such as modifiability, performance, and safety typically have a high impact on the architecture. However, sometimes the information about whether a requirement addresses a particular quality attribute is missing in the requirements specification. In order to compensate for this, we suggest that the architect should check if the requirement addresses one of the following issues:

- Monitoring and logging failures in the system (safety, reliability)
- Recovering from failures (safety)
- Upgrading and configuring a system (modifiability, configurability)
- Adapting the system to a new hardware environment, e.g. new processor and devices (modifiability)
- Adapting the system to a new software environment, e.g. new Operating System (modifiability)
- Introducing a new communication protocol to the system (modifiability, integrability)
- Protecting the system data against disclosure, modification, or destruction (security)
- Increasing the responsiveness of the system (performance)

These issues typically involve a high architectural impact. Table 4-1 shows a template for requirements prioritization.

According to our experience there may be the case that a particular requirement cannot be prioritized by the architect because it is not understood completely. In this case a change request for improving the accuracy of the requirement description is forwarded to the requirements engineer. If use cases or scenarios have been described during requirements engineering, these inputs can often be used for clarification of requirements.

Note that sometimes it will turn out during architecture design that the assumption the architect has made about the impact of a requirement was not correct. This is quite normal in practice. Our advice here is to reconsider the requirement priorities and rationales after each substantial design iteration of the architecture. In this way a continuous improvement can be achieved.

Table 4-1: Template for Requirements Prioritization

R#	Requirement	Priority
R1	< Description of requirement >	
	Business Importance: < Description of business importance >	< Priority >*
	Architectural Impact: < Description of architectural impact >	< Priority >*

* Priority = { High \| Medium \| Low }

Example:

Table 4-2 shows an example of how requirements can be prioritized according to business importance and architectural impact.

The business importance of R1, for example, is high because integrating authorized third party components is a business goal of the organization. Since this integration affects large parts of EVCS, the architect has judged the impact of R1 as high.

On the other hand, adding a commercial database system to the EVCS is not an important issue for business success. As a consequence, R3 is not essential for business success. The architectural impact of R3 is also low because the architect knows from experience that integration of a database system only has a local architectural impact for the current architecture.

Table 4-2: Prioritization of Requirements

REQ#	Requirement	Priority
REQ1	**Support integration of 3rd party software for air condition control**	
	Business Importance: Customizing the systems with authorized third party software components is part of the organization's business strategy. Therefore the requirement is assigned a high business priority.	High
	Architectural Impact: The implementation of the requirement requires standardized component interfaces for the third party software as well as component certification and registration mechanisms. This has an overall impact on the system design and involves a high effort for architecture development. Therefore the architectural impact of the requirement is high.	High
REQ2	**Support integration of 3rd party internet access software**	
	Business Importance: In-vehicle internet access is a less important product feature for the customers of the system. Therefore the requirement gets a low business priority.	Low
	Architectural Impact: The third party components must be integrated in the standard communication layer and respective application components must be provided for controlling internet access. This has an impact on a couple of architectural components. Since these components are intended to be encapsulated from the rest of the system, the development effort is moderate. The architectural impact of the scenario is therefore judged as medium.	Medium
REQ3	**Support integration of commercial embedded database system**	
	Business Importance: The system must provide a database system for storing vehicle data and personal information. Whether the database management is realized via commercial software or in-house development is not critical to the business success of the system. The business relevance of the requirement is therefore judged as low.	Low
	Architectural Impact: To implement the requirement, the internal database management, which is fully encapsulated, has to be replaced with the commercial system. This has only a local impact on the system's architecture. Thus supporting the integration of commercial embedded database system is only a low priority requirement for architecture development.	Low

4.2.2 Determine High-Priority Requirements

The goal of this activity is to identify the requirements with the highest significance for the given decision criteria. This is done by assessing the priorities assigned in the previous activity. Requirements that both have been assigned high values for business importance and architectural impact are included in the set of architectural drivers.

The activity consists of two steps:

1. Cluster requirements
2. Determine architecturally driving requirements

1. Cluster requirements

The goal of this step is to cluster the requirements according to their priorities. The purpose is to get a quick overview on the different "classes" of requirements – i.e., those that are important and those that are less important for architecture design. This is especially useful if the number of requirements is very high. We suggest using a portfolio diagram for representing the impact-to-importance ratios for the requirements. Figure 4-6 shows a template for such a diagram.

A portfolio diagram clearly facilitates the selection process for architecturally driving requirements, since it distinguishes most important requirements (shown in the upper right quarter of Figure 4-6) from less significant requirements (shown in the lower left quarter). Further, the upper left quarter of the diagram contains requirements that are difficult to implement but not essential for achieving business goals. It is worth negotiating these requirements with the customers and within the organization in order to clarify why they should be included in the system. The information should be forwarded to requirements engineering for further activities. Moreover, the lower right quarter contains requirements that are essential for achieving business goals and that are easy to implement in the architecture. These requirements are said to be "low hanging fruits" since their implementation is straight forward and highly effective.

2. Determine architecturally driving requirements

The goal of this step is to determine the architectural drivers. As mentioned in the previous step, the architectural drivers are shown in the upper right quarter of the portfolio diagram. According to our experience it may be the case that the number of drivers is very high. In order to adequately support the guidance of design activities it is worthwhile to further arrange the drivers according to the relative importance among each other. We suggest using the Analytic Hierarchy Process (AHP) [Saaty 1994] (cf. Section 2.2.1.4) for this purpose. The AHP relies on the pair-wise evaluation of alternatives. In our case the alternatives are the architecturally driving requirements. We propose to use the top-10 drivers with the highest relative importance for the first architecture design iteration (cf. Chapter 5). The implementation of these drivers will lead to an initial architectural structure. It is worthwhile then to check the priority assignments of the requirements for validity (cf. Section 4.2.1), to iterate clustering, and to seek for new high-priority requirements. After that the next design iteration with the most driving requirements should be performed.

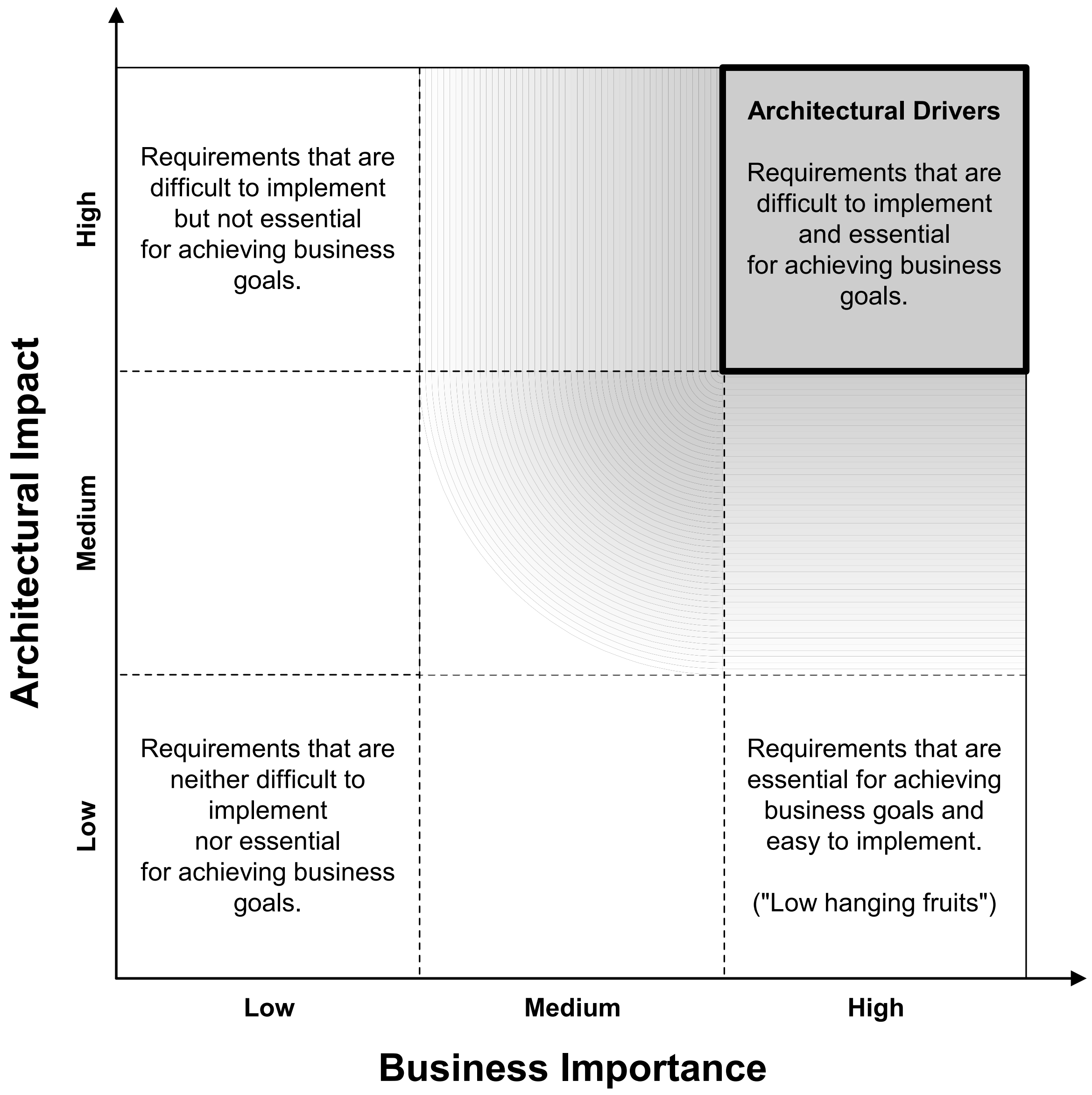

Architectural Impact
High
Medium
Low
Requirements that are difficult to implement but not essential for achieving business goals.
Architectural Drivers
Requirements that are difficult to implement and essential for achieving business goals.
Requirements that are neither difficult to implement nor essential for achieving business goals.
Requirements that are essential for achieving business goals and easy to implement.
("Low hanging fruits")
Low
Medium
High
Business Importance

Figure 4-6: Requirements Portfolio Diagram Template

Example:

Assume a company has decided to develop an innovative embedded vehicle control system (EVCS). The EVCS shall comprise entertainment, information, and communication functionality to improve passenger comfort, safety, economy, and security in an automobile. Assume further that the EVCS shall be built based on the software architecture of the current car navigation system. Table 4-3 illustrates the EVCS requirements provided by requirements engineering and the priorities prepared during the QUADRAD activity "Calculate priorities" (cf. Section 4.2.1). Each requirement is ranked according to its business importance and architectural impact. The characters "H", "M", "L" denote a high, medium, low business importance and architectural impact, respectively.

Table 4-3: EVCS Requirements

REQ#	Quality Attribute	Requirement	Priority
REQ1	Integrability	Support integration of 3rd party software for air condition control	[H, H]
REQ2	Integrability	Support integration of 3rd party internet access software	[L, M]
REQ3	Integrability	Support integration of commercial embedded database system	[L, L]
REQ4	Modifiability	Support software control for customer-specific CD players	[H, H]
REQ5	Modifiability	Support different vehicle communication bus systems	[M, H]
REQ6	Modifiability	Support easy update of entertainment software components	[M, M]
REQ7	Modifiability	Support different phone service providers based on localization	[L, M]
REQ8	Modifiability	Support easy configuration of low-end system at end-of-line	[H, H]
REQ9	Performance	Support watching rear camera images without significant delay	[M, L]
REQ10	Performance	Support quick system start-up (less than five seconds)	[H, H]
REQ11	Portability	Support different versions of the Windows CE Operating System	[M, H]
REQ12	Safety	Support detection of software errors in 3rd party software	[H, M]
REQ13	Safety	Support detection of system failures during start-up	[H, H]
REQ14	Security	Support protection of personal data stored in system	[L, M]
REQ15	Security	Support encryption of car information sent to service station	[M, L]
REQ16	Security	Prevent installation of non-certified software	[M, L]
REQ17	Security	Support remote diagnosis	[L, H]
REQ18	Usability	Support easy change from German to English language	[H, M]
REQ19	Usability	Support automatic adaptation of currency units (e.g., from Euro to Pound Sterling)	[L, L]
REQ20	Usability	Support training mode in order to allow user to get familiar with the system in less than two hours	[H, L]

* Priorities: [A, B] = { A = Business Importance, B = Architectural Impact }

For example, requirement REQ17 “Support remote diagnosis” has been assigned a low business importance and a high architectural impact. This means that performing remote diagnosis is a less important feature for the business success of the system but would be difficult to integrate in the architecture. On the other side, requirement REQ20 “Support training mode” has been assigned a high business importance but a low architectural impact. This indicates that usability is an important system feature but not a major architecture design issue. Rather, usability has to be achieved by a consistent and easy-to-use Human-Machine-Interface.

Figure 4-7 represents the portfolio diagram which clusters the impact/importance priorities of the requirements. The upper right quarter contains the following five architectural drivers with high business relevance and strong architectural impact:

- REQ1: Support integration of 3rd party software for air condition control (integrability),
- REQ4: Support software control for customer-specific CD players (modifiability),
- REQ8: Support easy configuration of low-end system at end-of-line (modifiability),
- REQ10: Support quick system start-up (performance), and
- REQ13: Support detection of system failures during start-up (safety).

Requirements with a high-to-medium importance and impact are REQ5 (modifiability), REQ11 (portability), REQ12 (safety), and REQ18 (usability). They are located left to and below the architectural drivers.

REQ20 is important for business success and can easily be implemented ("low hanging fruit"), whereas REQ17 is difficult to achieve but not important for achieving business goals. The requirements REQ3 and REQ19 are neither difficult to implement nor important from a business perspective and thus subject for negotiation.

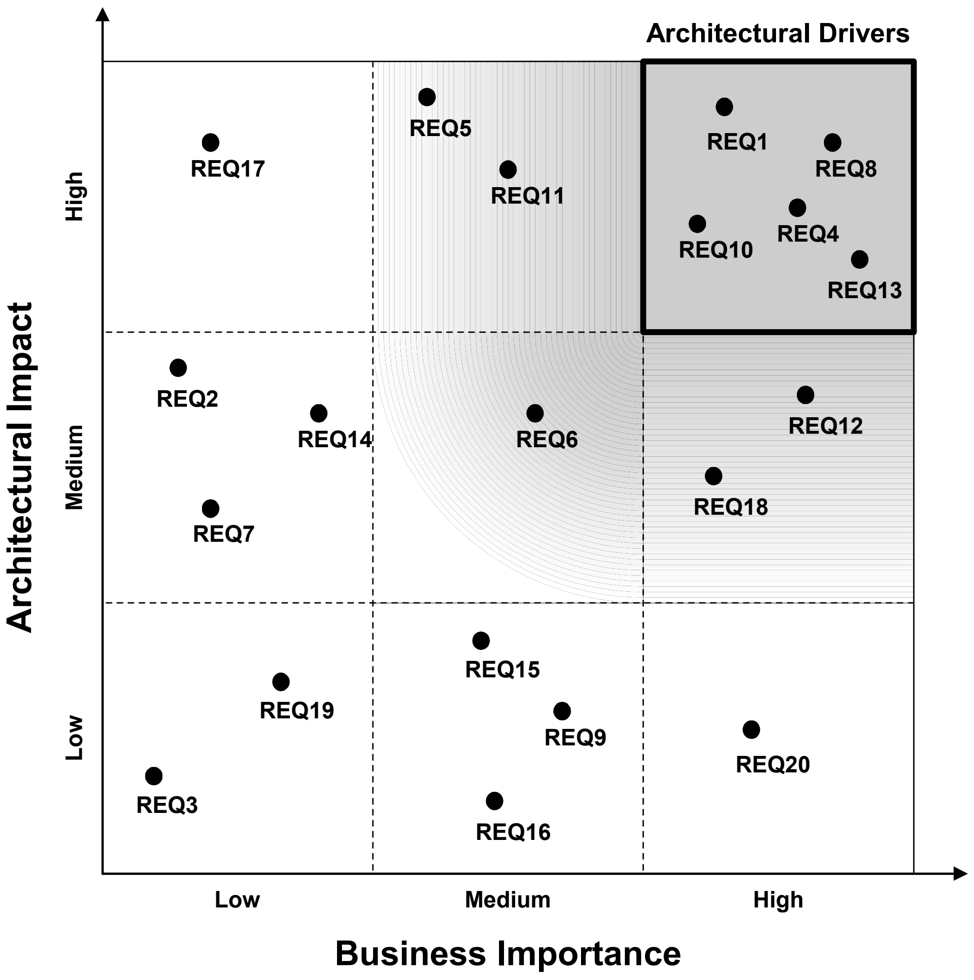


Figure 4-7: Portfolio Diagram of the EVCS Requirements

4.3 Determine Architectural View Types

The goal of this activity is to specify the architectural view types that are necessary for representing the architectural views that are relevant for implementing the architectural drivers. An overview of the input and output data flow for defining architectural view types is shown in Figure 4-8. The activity consists of the following two steps:

1. Cluster drivers according to quality attributes
2. Select view types

1. Cluster drivers according to quality attributes

The goal of this step is to organize the architectural drivers according to the quality attributes they address. This is useful because the quality attributes spell out system concerns that, on the other side, must be represented adequately by an architecture description. As discussed in Section 2.2.2.1, an architecture description includes a representation of multiple architectural views. By determining view types, the architect decides in which views the architecture is documented during design. An adequate starting set of views ideally provides the possibility to represent how the concerns of architecturally driving requirements have been addressed.

To perform the step the architectural drivers are clustered according to the quality attributes they address. If the attributes are not yet specified we suggest that the architect should do this first together with the requirements engineer before proceeding with this step. Note that this is not shown in Figure 4-8 for the sake of simplicity.

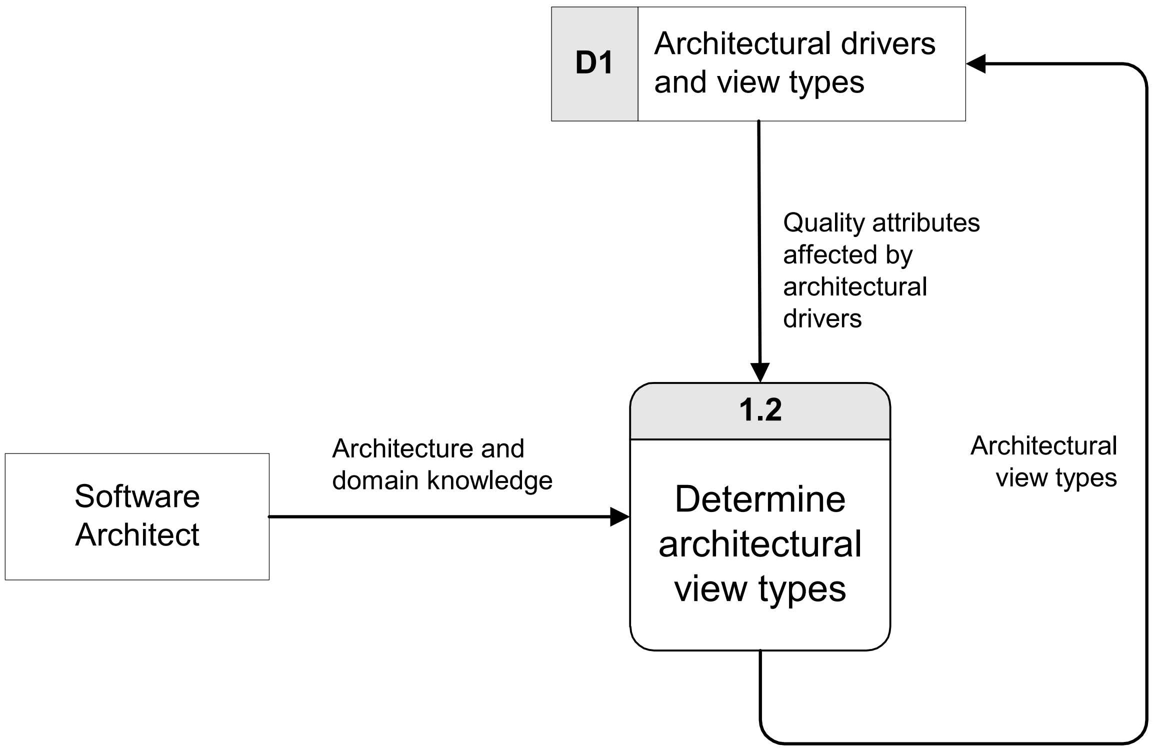

D1
Architectural drivers and view types
Quality attributes affected by architectural drivers
1.2
Determine architectural view types
Architectural view types
Software Architect
Architecture and domain knowledge

Figure 4-8: Determining Architectural View Types (Level-2 Diagram)

Example:

Table 4-4 shows the organization of the architectural drivers for the EVCS example introduced in the previous section according to the three quality attribute clusters modifiability/integrability, performance, and portability.

Table 4-4: Quality Attribute Clusters for Architectural Drivers of EVCS

Quality Attributes	**Architectural Drivers**
Modifiability, integrability	REQ1, REQ4, REQ8
Performance	REQ10
Portability	REQ13

2. *Select view types*

The goal of this step is to select the view types relevant for representing the implementation of the architectural drivers. As mentioned in Section 2.2.2.1, Bass et al. [Clements 2003] propose to use a set of eleven view types for representing different concerns of an architecture. We have adopted the list of view types and have added the information about how the view types correlate to quality attribute concerns. Table 4-5 shows the view types and their correlation to quality attributes.

To perform this step the set of view types for the clustered quality attributes are selected according to Table 4-5. The information about which view type specific elements are used to represent the architectural can be obtained from literature. We recommend the work of Clements et al. [Clements 2003] which contains detailed instructions for documenting architectural views.

Note that there might be the case that the list of architectural view types to be considered for architecture design is impracticably large. This is, for example, the case when the architectural drivers address various quality attributes. We suggest following the advice of Clements et al. [Clements 2003] to combine and prioritize the views in order to approach the minimum set of view types needed (cf. [Clements 2003, pp. 305-306]).

Example:

Table 4-6 shows the view types selected for capturing the architecture design for the EVCS. For the first design iteration only a subset of five view types (decomposition, uses, layered, concurrency, and deployment) have been chosen to reduce documentation overhead.

Table 4-5: Architectural View Types and Quality Attributes

View Type	Concerns	Quality Attributes addressed
Decomposition	Resource allocation; encapsulation; configuration control	Modifiability, configurability
Uses	Dependencies among components; separation in processes/tasks	Modifiability, performance
Layered	Encapsulation; virtual machines	Modifiability, portability
Class	Instances of components; shared methods	Modifiability, performance
Client-Server	Distributed operation; separation of concerns; performance analysis; load balancing	Performance, modifiability
Concurrency	Thread analysis; resource contention analysis	Performance, safety, reliability
Shared Data	Data producer/consumer analysis	Performance, modifiability
Deployment	Allocation of software to hardware nodes; safety/security analysis	Performance, availability, safety, security
Implementation	Configuration control; integration; test activities	Modifiability, integrability. Configurability
Work Assignment	Assignment of software components to development team; best use of expertise; management of commonality	N/A

Table 4-6: Architectural View Types to be Considered for EVCS

Quality Attributes	Architectural Drivers	Architectural View Types
Modifiability, integrability	REQ1, REQ4, REQ8	Decomposition Uses Layered Concurrency Deployment
Performance	REQ10	
Portability	REQ13	

4.4 Preparation Activities and Research Gaps

The provided activities of the QUADRAD Preparation workflow represent major research contributions of this work. Table 4-7 shows how these activities support bridging two of the research gaps described in Chapter 1. In addition, the table shows which of the framework requirements defined in Table 3-1 are achieved by those activities.

Table 4-7: Mapping Preparation Activities to Research Gaps and Requirements

Preparation Workflow Activities	Research Gaps	Framework Requirements
Identify architectural drivers	G1	FR1
Determine architectural view types	G2	FR2, FR3

Contributions of the QUADRAD Preparation workflow with respect to bridging the research gaps:

Research Gap 1: Lack of guidance for identifying requirements that are essential for architecture development.

The "Identify architectural drivers" activity (cf. Section 4.2) improves the identification and clarification of quality requirements. It supports identifying a system's essential quality requirements for attributes such as performance, safety, security, and modifiability. These requirements prevent creation of an architecture that is overly complex or that strives for unnecessary elegance at the expense of critical system properties. They rather provide a basis for developing an adequate architecture, i.e. an architecture that permits building a system that meets the intended business goals and the most essential market needs. The "Identify architectural drivers" activity supports the clarification of these requirements and provides techniques to rank them with respect to business impact and architectural importance.

Research Gap 2: No systematic support for making adequate design decisions in order to implement architecturally relevant requirements.

The "Determine architectural view types" activity (cf. Section 4.3) provides a basis for the comprehensive documentation of multiple architectural views. It makes explicit which view types should be used for which concerns. The treatment of multiple views is a central issue in architecture design and leads to a more comprehensive and concise documentation of the high-level design elements of the system. Describing multiple views improves the understanding of how the architecture responds to critical system qualities and shortens the learning time for other stakeholders.

4.5 Summary

In this chapter the activities and artifacts of the QUADRAD Preparation workflow (QUADRAD-P) have been described. The purpose of this workflow is to support early architectural considerations. In particular, activities for selecting the set of requirements that are most essential for developing an adequate architecture have been described. The description includes steps for assigning priorities to requirements and for identifying the requirements with the highest relevance for architecture design. Furthermore, activities for determining architectural view types addressed by architectural drivers are discussed. Finally, it has been shown how the QUADRAD-P activities support bridging the research gaps of this thesis.

In the next chapter, we will discuss activities for creating an architecture systematically based on the set of architectural drivers.

5 The QUADRAD Modeling Workflow

This chapter describes the QUADRAD Modeling workflow (QUADRAD-M). The purpose of QUADRAD-M is to support the definition of architectural views and to record the decisions that lead to those views. As designing an architecture remains a creative process, QUADRAD-M provides essential steps to make this process more systematic and repeatable. An overview of the major input and output artifacts is shown in Figure 5-1.

The chapter is organized as follows: Section 5.1 provides an overview of the Modeling workflow. In Section 5.2 activities for identifying architectural strategies and mechanisms that support the adequate implementation of architectural drivers are described. Activities for defining and refining the architecture are provided in Section 5.3 and 5.4, respectively. Section 5.5 describes how the QUADRAD-M activities address the research gaps in this thesis. Finally, Section 5.6 summarizes the main issues of this chapter.

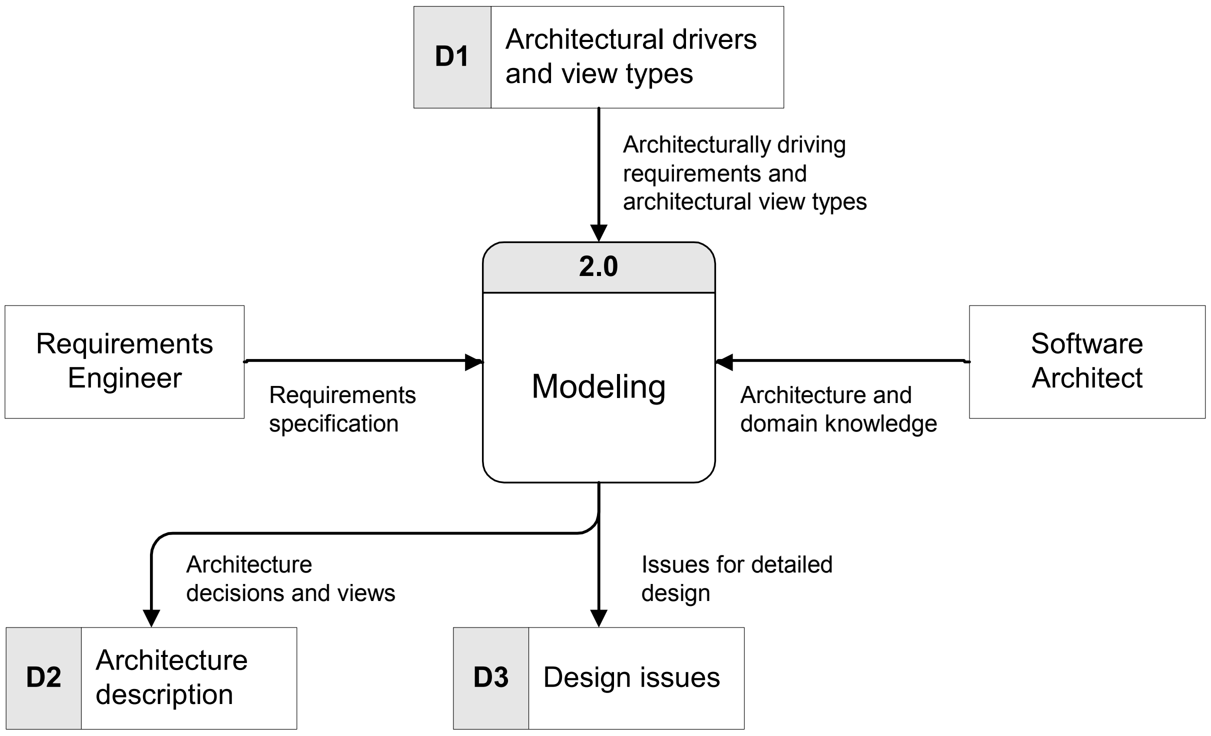

D1
Architectural drivers and view types
Architecturally driving requirements and architectural view types
2.0
Modeling
Requirements Engineer
Requirements specification
Software Architect
Architecture and domain knowledge
Architecture decisions and views
Issues for detailed design
D2
Architecture description
D3
Design issues

Figure 5-1: The QUADRAD Modeling Workflow (Level-1 Diagram)

5.1 Workflow Overview

Figure 5-2 shows the first decomposition of QUADRAD-M. It comprises three activities:

1. Define architectural strategies and mechanisms
2. Model the architectural infrastructure
3. Refine the architecture

1. Define architectural strategies and mechanisms

The objective of this activity is to determine architectural strategies and an adequate combination of mechanisms to implement the architectural drivers.

The inputs to the activity are architecturally driving requirements (provided by QUADRAD-P) as well as architecture and domain knowledge (provided by the software architect). The outputs from the activity are architectural strategies and mechanisms that are relevant for addressing the architecturally driving requirements.

2. Model the architectural infrastructure

The objective of this activity is to create an architectural infrastructure by considering architectural drivers, strategies, and mechanisms.

The inputs to the activity are architectural strategies and mechanisms that are relevant for addressing the architecturally driving requirements (provided by the activity “Define architectural strategies and mechanisms”), architecturally driving requirements and view types (provided by QUADRAD-P) as well as architecture and domain knowledge (provided by the software architect). The outputs from the activity are decisions and views for shaping the architecture infrastructure as well as issues for detailed design.

3. Refine the architecture

The objective of this activity is to refine the architecture and to incorporate the application functionality of the system by including design elements that address non-driving architectural requirements.

The inputs to the activity are decisions and views for shaping the architecture infrastructure as well as issues for detailed design (provided by the activity “Model the architectural infrastructure”), the requirements specification (provided by the requirements engineer) as well as architecture and domain knowledge (provided by the software architect). The outputs from the activity are refined architectural decisions and views as well as further issues for detailed design.

The activities are next described in more detail.

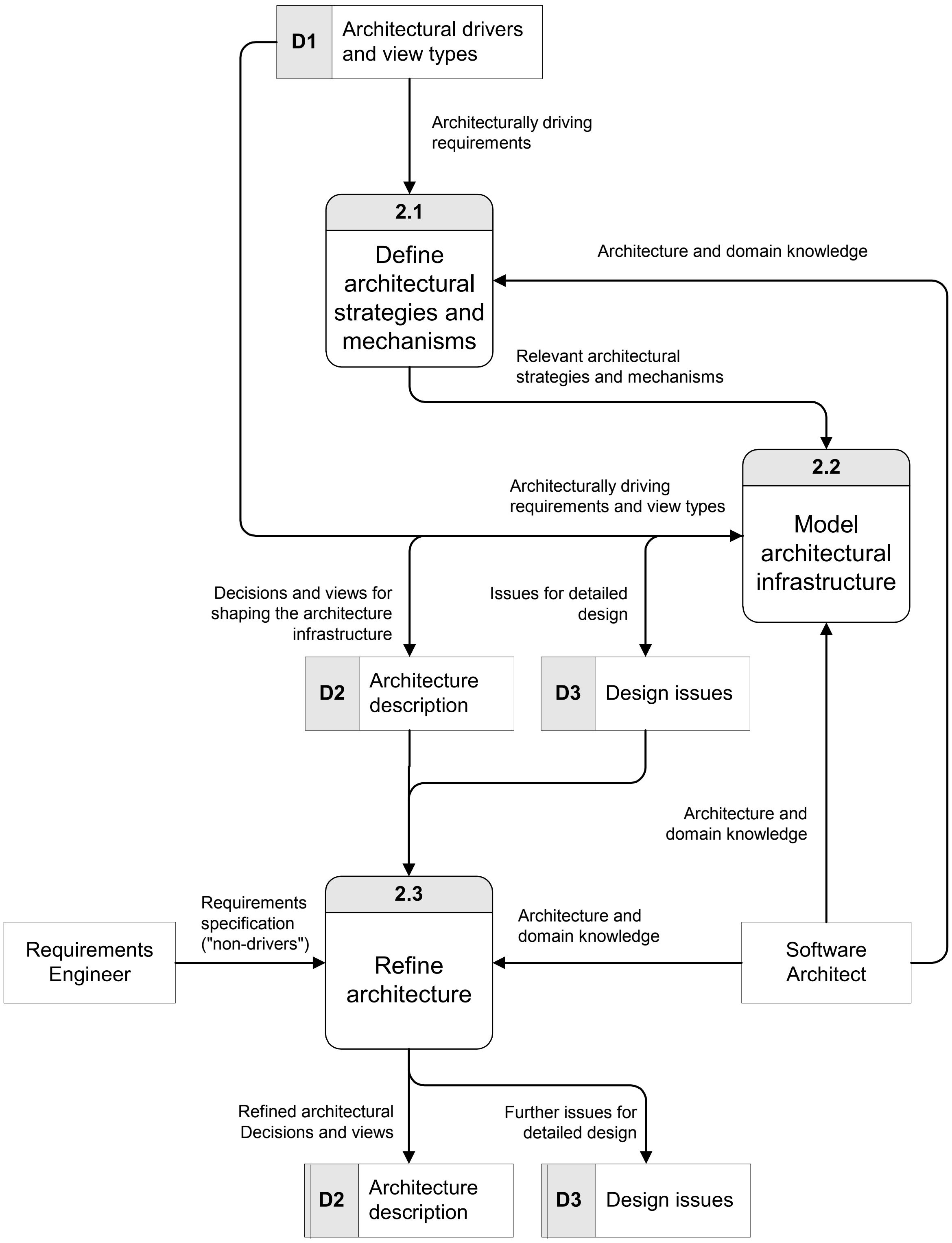

D1
Architectural drivers and view types
Architecturally driving requirements
2.1
Define architectural strategies and mechanisms
Architecture and domain knowledge
Relevant architectural strategies and mechanisms
2.2
Model architectural infrastructure
Architecturally driving requirements and view types
Decisions and views for shaping the architecture infrastructure
Issues for detailed design
D2
Architecture description
D3
Design issues
Architecture and domain knowledge
2.3
Refine architecture
Requirements specification ("non-drivers")
Requirements Engineer
Architecture and domain knowledge
Software Architect
Refined architectural Decisions and views
Further issues for detailed design
D2
Architecture description
D3
Design issues

Figure 5-2: The QUADRAD Modeling Workflow (Level-2 Diagram)

5.2 Define Architectural Strategies and Mechanisms

The objective of this activity is to determine adequate architectural strategies and mechanisms that contribute to the achievement of the given set of architectural drivers (cf. Section 4.2). This involves an exploration of effects to architectural drivers that result from combining and composing strategies and mechanisms.

An overview of the input and output data flow for determining architectural strategies and mechanisms is shown in Figure 5-3.

Figure 5-4 illustrates the decomposition of the activity into the following three subordinate activities:

1. Determine architectural strategies
2. Determine architectural mechanisms
3. Analyze suitability

The activities are next described in more detail.

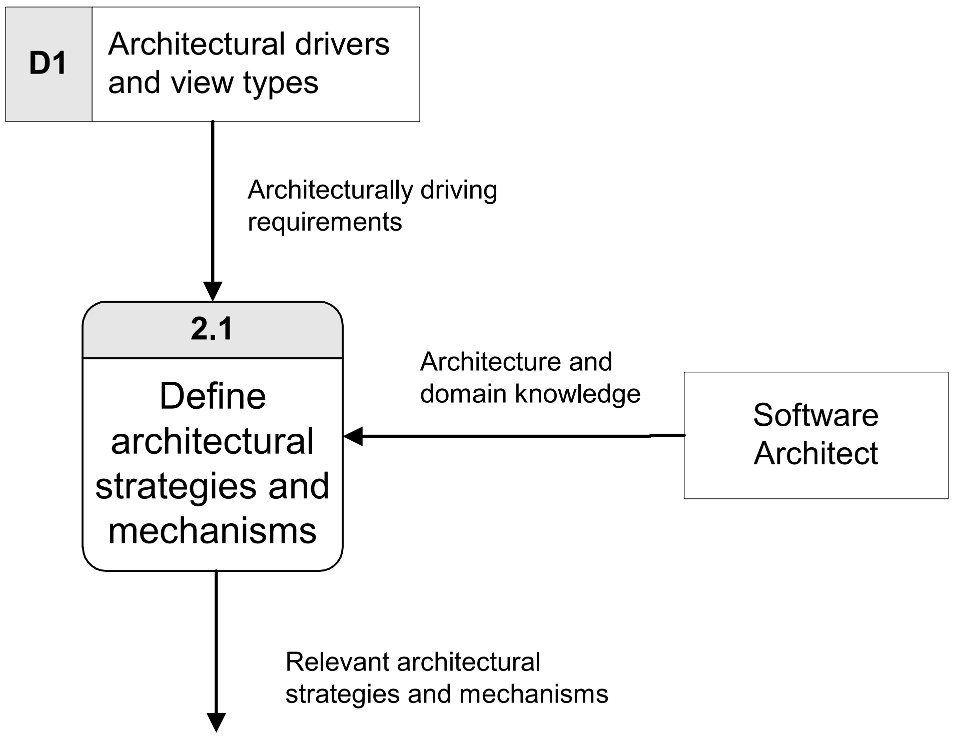

D1
Architectural drivers and view types
Architecturally driving requirements
2.1
Define architectural strategies and mechanisms
Architecture and domain knowledge
Software Architect
Relevant architectural strategies and mechanisms

Figure 5-3: Defining Architectural Strategies and Mechanisms (Level-2 Diagram)

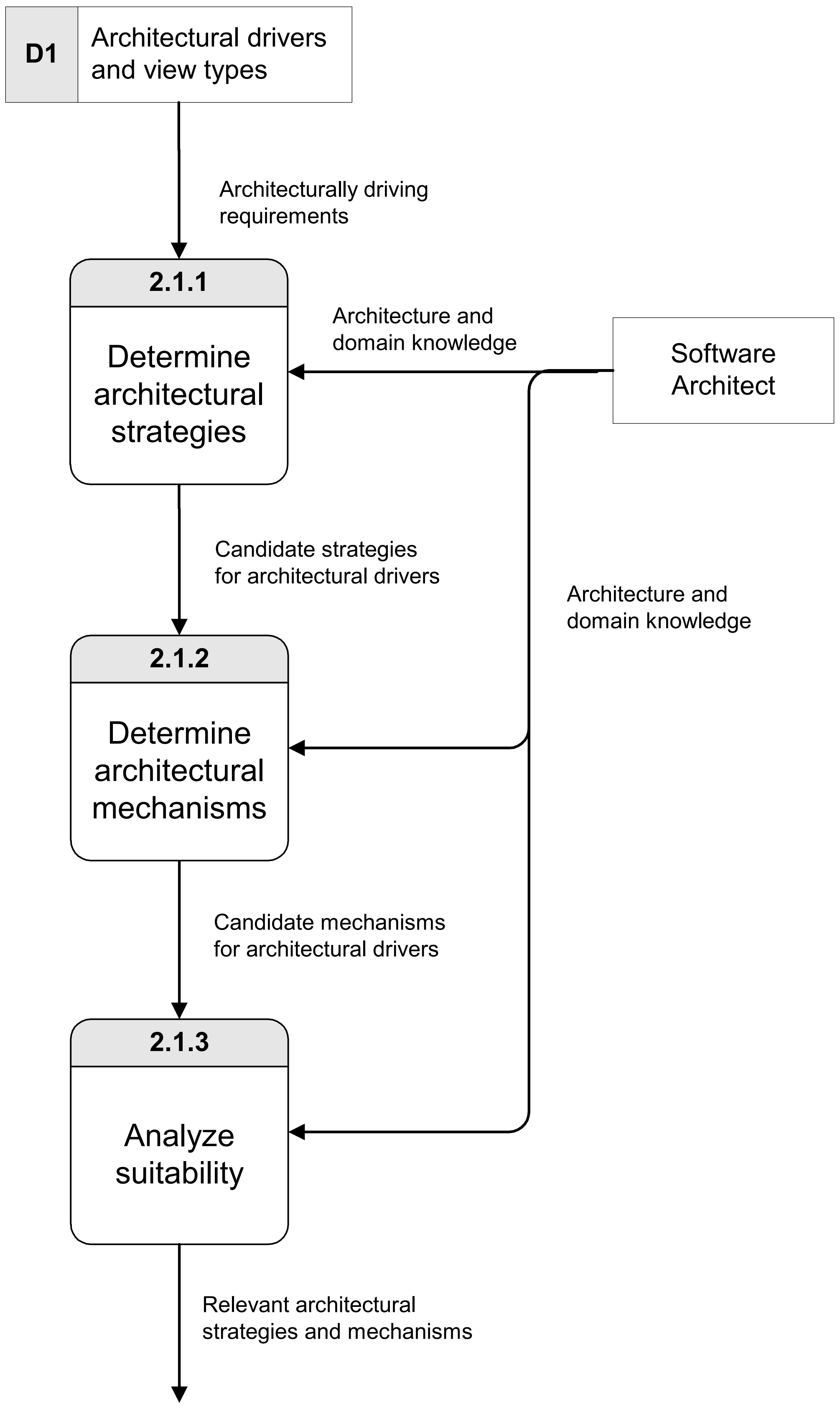

D1
Architectural drivers and view types
Architecturally driving requirements
2.1.1
Determine architectural strategies
Architecture and domain knowledge
Software Architect
Candidate strategies for architectural drivers
Architecture and domain knowledge
2.1.2
Determine architectural mechanisms
Candidate mechanisms for architectural drivers
2.1.3
Analyze suitability
Relevant architectural strategies and mechanisms

Figure 5-4: Defining Architectural Strategies and Mechanisms (Level-3 Diagram)

5.2.1 Determine Architectural Strategies

The goal of this activity is to determine architectural strategies that support the achievement of the set of architecturally driving requirements. As discussed in Section 2.2.3.1, an architectural strategy describes a general design principle to solve a particular class of design problems. Strategies are the first step in systematically establishing appropriate solutions that fulfill the set of requirements in the given system context. The activity is focused on describing basic approaches to be used in order to address the design problems related to a particular driver. Proposing complete design solutions for the problem should be avoided at this point since there is the risk that the solution does not properly match the design problem related to the driver.

In particular, there are four steps that must be performed in order to determine architectural strategies for the set of architecturally driving requirements:

1. Increase understanding of design problems
2. Cluster design problems
3. Brainstorm/select architectural strategies
4. Correlate strategies to architectural drivers

1. Increase understanding of design problems

The goal of this step is to increase the understanding of the fundamental design problems related to the architectural drivers. We suggest that the architect should carefully recapitulate the rationales he or she has given for determining the architectural impact during requirements prioritization (cf. Section 4.2). According to our experience the rationale for the architectural impact usually includes a characterization of the major problem as understood by the architect. The design problem needs to be refined with respect to the root problem. The additional consideration of the affected quality attributes for that problem can support this task.

In many cases more than one design problem is related to a given driver. In this case we suggest that the architect tries to describe any design problem he identifies and explains how they differ from each other in terms of system aspects. Note that the architect in this course might uncover aspects of the design problem which relate to quality attributes other than those identified during requirements prioritization. This is normal in practice since the design problem is better understood over time. We propose to record the new quality attributes together with the design problems identified. Table 5-1 shows a template for capturing the design problems of architectural drivers.

Additional note: As discussed in step 2 of Section 4.2.2 (“Determine architecturally driving requirements”), the architectural drivers might have been organized according to their relative importance. If the architect has uncovered a lot of new design problems during the current step which relate to new quality attributes he or she has not considered before, the relative importance between the drivers might have been significantly changed. We suggest repeating the pair-wise weighting of the drivers in this case because this guarantees that the architect starts designing the architecture by considering the most fundamental problems first.

Table 5-1: Template for Capturing Design Problems of Architectural Drivers

<Architectural Driver >		
Design Problem	**Description**	**Quality Attributes**
P1	< Description of design problem P1 >	< Quality attributes affected by P1 >
P2	…	…

Example:

Recall the EVCS example introduced in Chapter 4. Table 5-2 summarizes the fundamental design problems related to the architectural drivers of EVCS. As illustrated, the design problem of each of the five drivers is further refined. For the drivers REQ1 and REQ13, additional design problems that have an effect on quality attributes other than those identified during requirements prioritization (cf. Section 4.2.1) have been uncovered. For REQ1 an additional safety problem (P2) has been captured. For REQ13, a second design problem that affects performance issues of the system (P7) has been recorded.

2. *Cluster design problems*

The goal of this step is to organize the design problems in clusters that represent similar classes of problems. We suggest organizing the design problems with respect to affected quality attributes. The reason for this is that the effort and complexity involved in the identification of solution approaches (architectural strategies, architectural mechanisms) can be significantly reduced by such a clustering. According to our experience, an architect selects architectural approaches based on their effect on quality attributes rather than based on a specific application domain. However, the knowledge of the domain helps the architect to think of typical design problems that the domain includes with respect to particular qualities.

For example, assume the architect has uncovered the following performance design problem for a particular architectural driver of a web-based broker system:

The communication between client and server application is critical since each request for stocks must be handled within strict timing requirements.

Typically, the architect will think of solutions for reducing the communication overhead between client and server and thus will focus on performance strategies. In this case it is secondary if the application domain is a broker system, a document management system, or an automotive system. This is because the selection of a particular strategy for addressing a design problem is independent from the application domain. Mapping the design problems to a particular set of quality attributes helps organizing the problems is a way that is manageable for the architect. Table 5-3 shows a template for clustering design problems.

Table 5-2: Design Problems for Architectural Driver REQ1

Design Problem	**Description**	**Quality Attributes**
REQ1: Support integration of 3rd party software for air condition control		
P1	3rd party software shall be seamlessly integrated to the EVCS without requiring changes in on-board software components.	Modifiability, integrability
P2	On-board software of the EVCS shall not be affected by malfunctions of 3rd party software.	Safety
REQ4: Support software control for customer-specific CD players		
P3	The CD player control interface varies between different customers. The software control must be easily adapted to the respective interface.	Modifiability
REQ8: Support easy configuration of low-end system at end-of-line		
P4	The configuration affects different applications. This must be done in a comfortable way and without inducing configuration errors.	Modifiability
REQ10: Support quick system start-up		
P5	The applications of the system must be available within two seconds after starting the vehicle.	Performance
REQ13: Support detection of system failures during start-up		
P6	The system requires self-test capabilities for analyzing installed software components during start-up. There is a tradeoff between performing self-tests and quick initialization of the system.	Safety
P7	The detection of system failures must be done in less than two seconds.	Performance

Table 5-3: Template for Clustering Design Problems

Cluster	**Design Problem**	**Architectural Driver**
< Quality attribute cluster 1 of design problems >	< Design problem concerning cluster 1 >	< Quality attributes affected >
< Quality attribute cluster 2 of design problems >	…	…

Example:

Table 5-4 shows how the design problems related to architecturally driving requirements of the EVCS are clustered according to the affected quality attributes.

Table 5-4: Clustered Design Problems of the EVCS

Cluster	Design Problem	Architectural Driver
Modifiability	P1 P3 P4	REQ1 REQ4 REQ8
Safety	P2 P6	REQ1 REQ13
Performance	P5 P7	REQ10 REQ13

3. Brainstorm/select architectural strategies

The goal of this step is to determine architectural strategies that could be used to address the design problem clusters documented in the previous step. As introduced in Section 2.2.3.1, an architectural strategy represents a general design principle which is used by the architect to solve a particular class of design problems that pertain to a set of quality attributes. In order to identify a candidate set of architectural strategies we suggest performing the following two tasks:

- *Brainstorm strategies:* In this task the architect collects architectural strategies for the design problem clusters in a brainstorming process. We suggest that the lead developers and quality attribute experts should participate in the brainstorming. According to our experience it is wise to narrow the brainstorming to the quality attribute clusters in order to focus the identification of solutions. Thus, for each quality attribute cluster candidate strategies are collected that support the treatment of the design problems associated to that cluster.
- *Select strategies from literature:* In this task the brainstormed set of strategies is further extended by incorporating strategies documented in literature. A comprehensive list of architectural strategies for availability, modifiability, security, usability, and performance problems can be found in [Bass 2003, pp. 101-123] and [Smith 2002]. Table 2-4 in Section 2.2.3.1 shows architectural strategies provided by Bass et al. [Bass 2003] addressing availability problems.

Table 5-5 shows a template for linking candidate strategies to design problem clusters.

Table 5-5: Template for Linking Architectural Strategies to Design Problem Clusters

Cluster	Design Problem	Architectural Strategy
< Quality attribute cluster 1 of design problems >	< Design problem concerning cluster 1 >	< Set of architectural strategies >
< Quality attribute cluster 2 of design problems >	…	…

4. Correlate strategies to architectural drivers

The goal of this step is to finally correlate the candidate strategies elicited in the previous step to the architecturally driving requirements. This is worthwhile in order to provide an overview of how the drivers can be addressed in the course of design. During this step the strategies captured in the previous step are organized according to the set of architectural drivers. Table 5-6 shows a template for correlating architectural drivers and strategies.

Table 5-6: Template for Correlating Architectural Drivers with Strategies

Architectural Driver	**Design Problem**	**Architectural Strategy**	**Quality Attribute**
< Architectural driver 1 >	< Set of design problems >	< Set of strategies >	< Quality attributes >
< Architectural driver 2 >	…	…	…

Example:

The following list summarizes architectural strategies provided by [Bass 2003] that address performance and modifiability design problems.

Performance

- *S1: Reduce communication overhead:* The goal of this strategy is to reduce communication overhead in order to increase execution performance. Computational work usually requires operating system and middleware services to manage process interaction and communication. Adjusting the assignment of responsibilities to processes influences the need for communication between those processes. For example, using Remote Method Invocation (RMI) for requesting a service from a process will incur more overhead than using a method of a Java class within the same process. Another example is the communication overhead induced by context-switching to balance concurrent requests for processor time.
- *S2: Limit execution time:* The goal of this strategy is to place a limit on how much execution time is used to respond to an event. For a data dependent, iterative algorithm, bounding execution time can be accomplished by limiting the number of iterations. However, this strategy might imply trading off accuracy for improved latency.
- *S3: Increase locality:* The goal of this strategy is to create actions, functions, and data that are “close” to physical computer resources (such as files, databases, processors, and storage devices). For example, if a desired screen result is identical to the physical database row that produces it, then the locality is good. Similarly, the locality is better if the data to be presented is in the processor’s main memory, rather than on a disk drive attached to different machine

Modifiability

- *S4: Isolate expected changes:* The goal of this strategy is to relate responsibilities to classes of expected change. Separating responsibilities that are likely to change from those which are unlikely to change divides an architecture into fixed and variant parts. More design effort can be devoted to make changes to the modules that are likely to change, or will change very frequently, as easy as possible.
- *S5: Break the dependency chain:* This strategy refers to the use of an intermediary to keep one module from being dependent on another. The intermediary will break the dependency chain. The name of the intermediate, typically, depends on type of dependency. For example, a name server breaks a dependency on the run time location of a module, or a database breaks the dependency that a consumer of a data item has to know the identity of its producer.

Table 5-7 shows how the architectural strategies can be used to address the design problems involved in the architecturally driving requirements of the EVCS. For example, in order to address the modifiability design problem P1 of REQ1, the architectural strategy S5 can be chosen as starting point for further design considerations. For REQ10 the architectural strategies S1, S2, and S3 are candidates for coping with the performance design problem. The safety design problems P2 and P6 of REQ1 and REQ13 cannot be addressed by the given set of strategies.

Table 5-7: Correlation of Architectural Drivers and Strategies

Architectural Driver	Design Problem	Architectural Strategy	Quality Attribute
REQ1	P1 P2	S5 --	Modifiability Safety
REQ4	P3	S4	Modifiability
REQ8	P4	S4	Modifiability
REQ10	P5	S1, S2, S3	Performance
REQ13	P6 P7	-- S1, S2, S3	Safety Performance

5.2.2 Determine Architectural Mechanisms

The objective of this activity is to identify architectural mechanisms that implement the strategies. The strategies help to narrow the scope of appropriate solutions for the given set of architectural drivers. An architectural mechanism refines one or more corresponding strategies and adds concrete detail to its implementation at the architecture level (cf. Section 2.2.3.2).

The activity consists of two steps:

1. Brainstorm/select mechanisms
2. Create strategy-mechanism matrix

1. Brainstorm/select mechanisms

The goal of this step is to collect architectural mechanisms that could be used to implement the strategies determined in the previous activity. As discussed in Section 2.2.3.2, an architectural mechanism describes a collection of component and connection types (and their basic responsibilities) to use for application to the given design problem. It thus pre-selects a set of architectural decisions. Applying the mechanism in a given context refines the decisions and leads to a refinement/extension of the architecture. The refinement/extension describes how a particular driver is implemented in the architecture. It defines what kind of "functionality" must be provided by a certain collection of design elements and how those design elements must cooperate. The Appendix provides a comprehensive list of patterns which can be examined to identify appropriate mechanisms for architecture design.

As in the previous activity we suggest both a brainstorming session and a literature search in order to identify a candidate set of architectural mechanisms. We suggest that the architect should recapitulate the design problem clusters and the quality attributes affected. The mechanism descriptions that can be found in literature usually contain information about the problems that can be solved by the mechanism, about the affected quality attributes, as well as about the basic strategies used to implement the mechanism. This information supports selecting a set of candidate mechanisms. Details are not so important at this stage. The goal is rather to have a starter list of mechanisms which are somehow relevant to the design problem clusters captured in the previous activity and which provide means for refining the architectural strategies. Evaluating the suitability of these mechanisms for addressing the architectural drivers is part of the forthcoming activity "Determine suitability".

However, there may be some design problem clusters or strategies for which no relevant mechanism can be identified. In this case we suggest leaving mechanism selection open where necessary and using the identified architectural strategies as an input for driving subsequent design activities.

2. Create strategy-mechanism matrix

The goal of this step to create a strategy-mechanism matrix which captures information about whether a mechanism supports a particular strategy. The matrix correlates the mechanisms identified in the previous step to the candidate strategies provided by the preceding activity. For each mechanism the architect evaluates if it provides a solution/refinement of the architectural strategies. If a particular mechanism supports the implementation of a strategy then the architect records this information in the matrix. The resulting matrix then shows the most adequate solution approaches (mechanisms) to be used to address the architecturally driving requirements. Table 5-8 shows a template for creating a strategy-mechanism matrix.

Table 5-8: Template for Creating a Strategy-Mechanism Matrix

Mechanisms	**Architectural Strategies**			
	S1	S2	S3	…
M1	*	*	*	…
M2	*	*	*	…
M3	*	*	*	…
…	…	…	…	…

Legend: * = [Supports | <Blank>]

Supports = Mechanism supports the implementation of the strategy.
<Blank> = Mechanism does not support the strategy or no information.

Example:

Candidate mechanisms that provide an architectural solution to the strategies S1, S2, S3, S4, and S5 given in the previous example are (cf. Appendix):

- *M1: Slicing* [Buschmann 1996]. Slicing supports a relaxed layering of a system.
- *M2: Caching* [Kircher 2004]. Caching describes how to avoid expensive re-acquisition of resources by not releasing the resources immediately after their use.
- *M3: Layering* [Buschmann 1996]. Layering helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.

Introducing vertical slices (M1) [Buschmann 1996] is a mechanism that can be used to reduce communication overhead (S1) in a layered system. Caching (M2) [Kircher 2004] can be used to both limit execution time (S2) and to increase data locality (S3). Layering (M3) [Buschmann 1996] supports maintaining semantic coherence and can be applied to break the dependency chain of components (S5). It can also be used to isolate expected changes (S4). For example, component changes in the presentation layer can be isolated from changes in the application layer. Table 5-9 illustrates how the architectural mechanisms support the given strategies.

Table 5-9: Architectural Strategies and Supporting Mechanisms

Mechanisms	Architectural Strategies				
	S1	S2	S3	S4	S5
M1	Supports				
M2		Supports	Supports		
M3				Supports	Supports

5.2.3 Determine Suitability

The goal of this activity is to evaluate overall mechanism suitability. For each mechanism selected in the previous step, the effect on the set of architectural drivers is determined. The activity consists of the following steps:

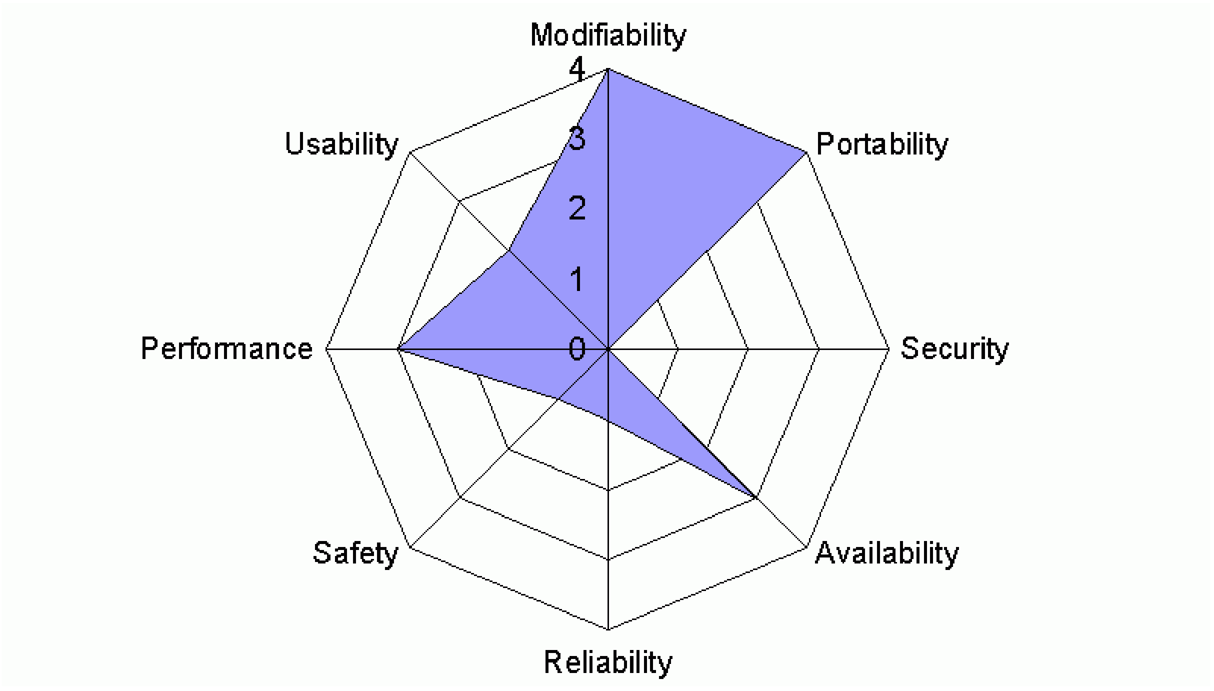
1. Analyze consequences of mechanisms
2. Create driver-mechanism matrix
3. Determine suitability of mechanisms

1. Analyze consequences of mechanisms

The goal of this step is to analyze the consequences of applying the mechanisms identified in the previous activity. The application of a mechanism involves tradeoffs with respect to quality attributes (cf. Section 2.2.1.3). The task of this step is to make these tradeoffs visible to the architect such that he or she can estimate the architectural impact of applying the mechanisms. Architectural mechanisms documented in the literature usually include a description of consequences (see, for example, [Buschmann 1996], [Schmidt 2000], [Fowler 2002], [Kircher 2004]). These descriptions should be considered during this step. If the architect has defined his or her own mechanisms then the consequences of these mechanisms must be evaluated here.

We suggest using Kiviat diagrams [Lanza 2003] in order to visualize the tradeoffs of mechanisms with respect to quality attributes. The benefit of using Kiviat diagrams is to gain a quick overview of the "strengths" and "liabilities" of a mechanism. A Kiviat diagram thus represents a mechanism's quality tradeoff profile.

Figure 5-5 shows a sample Kiviat diagram of a mechanism illustrating how it scores within the set of typical quality attributes. As depicted, the application of the mechanism would strongly support modifiability and portability of an architecture. It would also support availability and performance. On the other side, the mechanism would strongly conflict with security issues of the architecture. It would also negatively influence reliability and safety but would be neutral to usability aspects of the architecture.



<u>Legend</u>: 0 = Strong conflict, 1 = conflict, 2 = neutral,
3 = support, 4 = strong support

Figure 5-5: Kiviat Diagram of an Architectural Mechanism

We suggest storing the diagram for each mechanism for later use. We also propose the accumulation of a mechanism library in order to support future design activities. The mechanism library should include a short description of the mechanism, the Kiviat diagram showing the mechanism's quality tradeoff profile, and a reference where the architect can get detailed information about the implementation of the mechanism. Having such a library available would significantly speed up the process of identifying the most adequate solutions for architecture design. However, the detailed specification of this library is beyond the scope of this work.

Example:

Figure 5-6 shows three Kiviat diagrams for the mechanisms identified for the EVCS in the previous activity:

- M1: Slicing
- M2: Caching
- M3: Layering

The diagrams depict the consequences of applying the mechanisms with respect to quality attributes. The benefits and tradeoffs are as follows:

- *M1: Slicing.*

  The slicing mechanism (M1) supports modifiability and portability. Its application results in a relaxed layered system that is less restrictive about the relationship between layers. In a sliced system, each layer may use the services of all layers below it, not only of the next lower layer. Slicing is applied at the cost of performance since some requests will still need to be transferred through a number of intermediate layers. It does not affect security, availability, reliability, safety, and usability issues. Note that the modifiability – and especially, maintainability – of a sliced system is worse than that of a strictly layered system (see M3).

- *M2: Caching.*

  The caching mechanism (M2) supports performance and availability. Fast access to frequently used resources is an explicit benefit of caching. Caching ensures that the resources maintain their identities. Therefore, when the same resource needs to be accessed again, the resource need not be acquired or fetched from somewhere—it is already available. The mechanism has also a positive effect on reliability. Since Caching reduces the number of releases and re-acquisitions of resources, it reduces the chance of memory fragmentation leading to greater system stability. Caching has a tradeoff with security since there is the probability that unauthorized persons can gain access to cached data if the data is not properly protected. Caching is largely neutral to modifiability, portability, safety, and usability.

- *M3: Layering.*

  The layering mechanism (M3) supports modifiability and portability. Individual layer implementations can be replaced by semantically-equivalent implementations with limited effort. If the connections between layers are hard-wired in the code, these can be updated with the names of the new layer's implementation. If an individual layer embodies a well-defined abstraction and has a well-defined and documented interface, the layer can be reused in multiple contexts. Layering also provides that dependencies are kept local. Standardized interfaces between layers usually confine the effect of code changes to the layer that is changed. Changes of the hardware, the operating system, the window system, special data formats and so on often affect only one layer, and you can adapt affected layers without altering the remaining layers. This especially supports the portability of a system.

  However, layering has a negative effect on performance. A layered architecture is usually less efficient than, say, a monolithic structure or a 'sea of objects'. If high-level services in the upper layers rely heavily on the lowest layers, all relevant data must be transferred through a number of intermediate layers, and may be transformed several times. The same is true if all results or error messages produced in lower levels that are passed to the highest level. Communication protocols, for example, transform messages from higher levels by adding message headers and trailers.

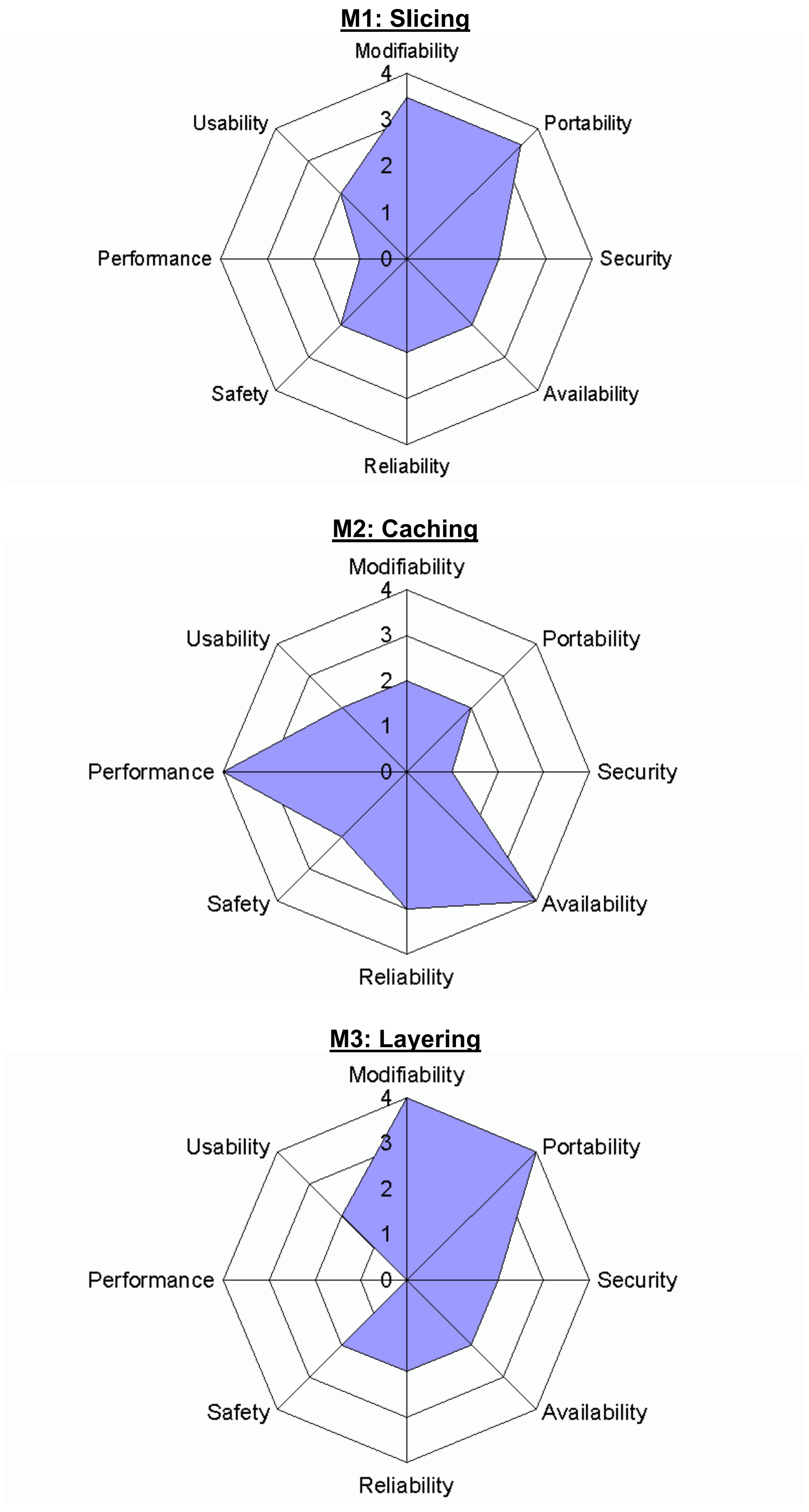


Figure 5-6: Quality Profiles for Slicing, Caching, and Layering

2. Create driver-mechanism matrix

The goal of this step is to create a matrix that correlates the architectural mechanism analyzed in the previous step to the set of architecturally driving requirements. In the previous step, each mechanism has been evaluated against the general set of quality attributes. In this step, the task is to get an understanding of how well the total set of mechanisms fit to those quality attributes that are addressed by the architectural drivers.

In order to have an overall view of the relevant set of quality attributes we suggest creating a driver-mechanism matrix that additionally includes links to the design problems identified during the "Determine strategies" activity. In this way, each quality attribute that is affected by the drivers is represented in the matrix. The mechanisms are then evaluated against these quality attributes by considering the quality profiles recorded in the previous step. Table 5-10 shows a template for creating a driver-mechanism matrix.

In order to fill the entries of the matrix we suggest that the architect should scan through the Kiviat diagrams of the mechanisms which were prepared in the previous step and record in the driver-mechanism matrix how the mechanisms contribute to the driving quality attributes. If it (strongly) supports an attribute then the architect should note this in the matrix by putting in a "+" or "++", respectively. If it does not support the attribute then he or she should record either a "–" (for a minor conflict) or "––" (for a strong conflict). Finally, if the mechanism does not have any effect on the quality attribute then the architect should make this explicit by noting a "0".

Table 5-10: Template for Creating a Driver-Mechanism Matrix

Mechanisms	**Architectural Drivers**			
	REQ1		REQ2	…
	Design Problem P1	Design Problem P2	Design Problem P3	…
	Quality Attribute of P1	Quality Attribute of P2	Quality Attribute of P3	…
M1	*	*	*	…
M2	*	*	*	…
M3	*	*	*	…
…	…	…	…	…

Legend: * = [++ | + | 0 | – | ––]

++ = Strong support, + = support, **0** = neutral,
– = conflict, – – = strong conflict

Example:

Table 5-11 shows the driver-mechanism matrix for the EVCS. It correlates the architectural drivers REQ1, REQ4, REQ8, REQ10, REQ13 and associated design problems P1, P2, P3, P4, P5, P6, and P7 to the candidate mechanisms M1, M2, and M3. The matrix thus shows the architectural implications of the mechanisms.

For example, slicing (M1) and layering (M3) support modifiability design problem P1 (seamless integration of 3rd party software into the EVCS) of architectural driver REQ1. They also support P3 (easy adaptation of CD player control interface) and P4 (easy configuration) of REQ4 and REQ8, respectively. Caching (M3) improves performance and availability. It thus supports P5 (quick availability of applications after start-up) and P7 (quick detection of failures) of REQ10 and REQ13, respectively. Slicing and layering have a negative effect on performance if they are not carefully applied in the architecture. Safety aspects are not considered by the current set of mechanisms.

Table 5-11: Driver-Mechanism Matrix of the EVCS

Mechanisms	**Architectural Drivers**						
	REQ1		REQ4	REQ8	REQ10	REQ13	
	P1	P2	P3	P4	P5	P6	P7
	Modifiability	Safety	Modifiability	Modifiability	Performance	Safety	Performance
M1: Slicing	+	**0**	+	+	–	**0**	–
M2: Caching	**0**	**0**	**0**	**0**	++	**0**	++
M3: Layering	++	**0**	++	++	––	**0**	––

<u>Legend:</u> ++ = Strong support, + = support, **0** = neutral,
– = conflict, –– = strong conflict

3. Determine suitability of mechanisms

The goal of this step is to determine the suitability of the mechanisms with respect to the architectural drivers. There are three different tasks that must be performed in this step:

a) Identify quality attributes that are not supported by the mechanisms
b) Identify quality attributes that are completely supported by the mechanisms
c) Identify quality attributes that are supported by but also have conflicts with mechanisms

a) Identify quality attributes that are not supported by the mechanisms:

If a quality attribute is not supported by any mechanism then this means that the design problems identified and clustered during the "Determine strategies" activity are not addressed, yet. We suggest that the architect should repeat the activity "Determine mechanisms" and focus the brainstorming/literature research on mechanisms that address the unsupported quality. Alternatively, the architect can also go back to the "Determine strategies" activity and try to get more familiar with the design problems related to the unsupported quality attribute. There is also the chance that the architect will identify additional architectural strategies during this activity, which help him or her in defining/selecting adequate mechanisms.

b) Identify quality attributes that are completely supported by the mechanisms

If a quality attribute is supported by the mechanisms then this means that the design problems identified and clustered for that attribute can be addressed. In this case, the mechanisms are selected for further design activities (see next activity "Model architectural infrastructure"). Note that there is the opportunity to reduce the number of supporting mechanisms by removing those with similar quality profiles.

c) Identify quality attributes that are supported by but also have conflicts with mechanisms

If a quality attribute is supported by particular mechanisms but also conflicts with other mechanisms then this is an indicator for architectural tradeoff decisions. Tradeoff decisions are quite normal during architectural design. Quality attributes compete against each other (cf. Section 2.2.1.3) and thus architectural mechanisms compete against each other, too. Conflicts between mechanisms cannot always be avoided. The objective is rather to minimize such conflicts and make them explicit.

Before accepting the tradeoff decision, we suggest that the architect should repeat the "Determine mechanisms" activity and check if there are alternative mechanisms with similar profiles but better scores for the conflicting quality attribute. Then the mechanisms are selected for further design activities (see next activity "Model architectural infrastructure"). We suggest that the architecture should keep in mind that deriving design decisions for those mechanisms will lead to quality tradeoffs in the architecture.

Table 5-12 shows a template for documenting the suitability of the architectural mechanisms.

Table 5-12: Template for Documenting the Suitability of Mechanisms

Results	**Quality Attribute**	**Design Problem**	**Mechanism**	**Architectural Driver**
Not Supported	< Quality attributes not supported >	< Associated design problems >	< Associated mechanisms >	< Associated drivers >
Supported	< Supported quality attributes >	< Associated design problems >	< Associated mechanisms >	< Associated drivers >
Tradeoffs	< Quality attribute tradeoffs >	< Associated design problems >	< Associated mechanisms >	< Associated drivers >

Example:

Table 5-13 shows the suitability of the architectural mechanisms with respect to the driving requirements of the EVCS. As illustrated, safety issues are covered by none of the mechanisms. Consequently, the "Determine mechanisms" activity should be repeated for this attribute in order to identify appropriate solutions. Modifiability aspects are supported by the mechanism M1 and M3. These mechanisms have also a performance tradeoff with M2.

Table 5-13: Suitability of Mechanisms for the EVCS

Results	**Quality Attribute**	**Design Problem**	**Mechanism**	**Architectural Driver**
Not Supported	Safety	P2, P6	--	REQ1, REQ13
Supported	Modifiability	P1, P3, P4	M1, M3	REQ4, REQ8
Tradeoffs	Performance vs. Modifiability	P5, P7	M2 vs. M1, M2 vs. M3	REQ10, REQ13

5.3 Model the Architectural Infrastructure

The objective of this activity is to design the architectural infrastructure of the system. The architectural infrastructure comprises those design elements that are essential for achieving the most important quality properties and business goals. As the infrastructure is shaped by the set of architecturally driving requirements, it provides a solution for the most strategic design decisions. Those decisions have the highest impact on structural aspects of the architecture and are the most difficult to change.

The architectural infrastructure consists of those portions that every application functionality requires to execute. The operating system, communication protocols, middleware, and system-specific supporting functionality are all infrastructure elements. Starting with a specification of the architectural infrastructure, the architect is forced to focus on the most difficult design problems first (the ones that cannot be built in later and that are essential for the system), prior to investigating application functionality. It results in a partial architecture where the application functionality can be implemented incrementally. The set of architectural mechanisms identified in the previous activity is an ideal starting point for modeling infrastructure concerns of the architecture.

An overview of the input and output data flow for modeling the architectural infrastructure is shown in Figure 5-7.

Figure 5-8 illustrates the decomposition of the activity into the following two subordinate activities:

1. Derive architectural decisions
2. Apply decisions and adapt architecture

The activities are next described in more detail.

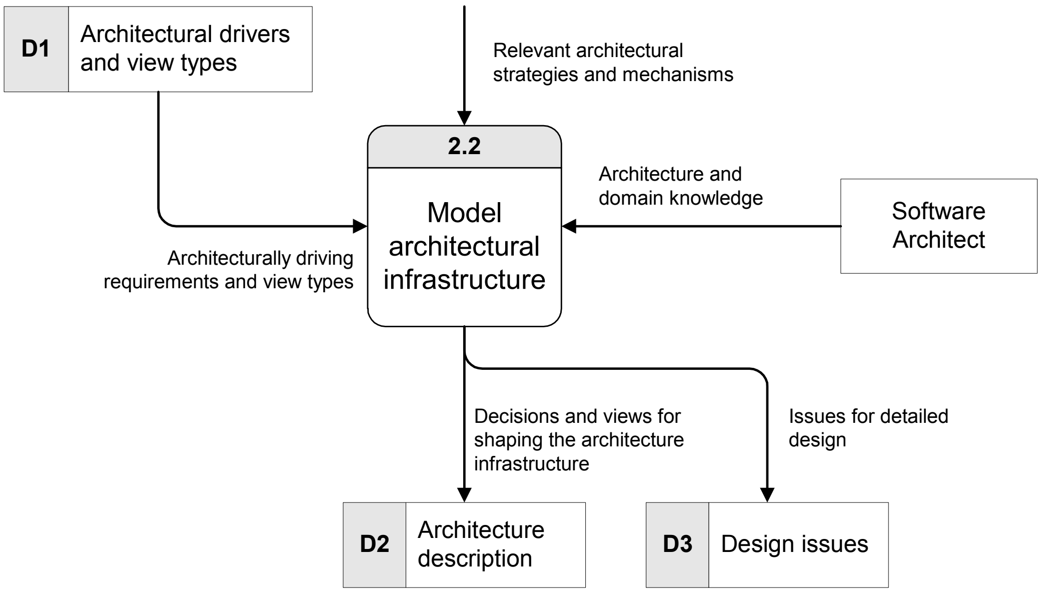

D1
Architectural drivers and view types
Relevant architectural strategies and mechanisms
2.2
Model architectural infrastructure
Architecture and domain knowledge
Software Architect
Architecturally driving requirements and view types
Decisions and views for shaping the architecture infrastructure
Issues for detailed design
D2
Architecture description
D3
Design issues

Figure 5-7: Modeling the Architectural Infrastructure (Level-2 Diagram)

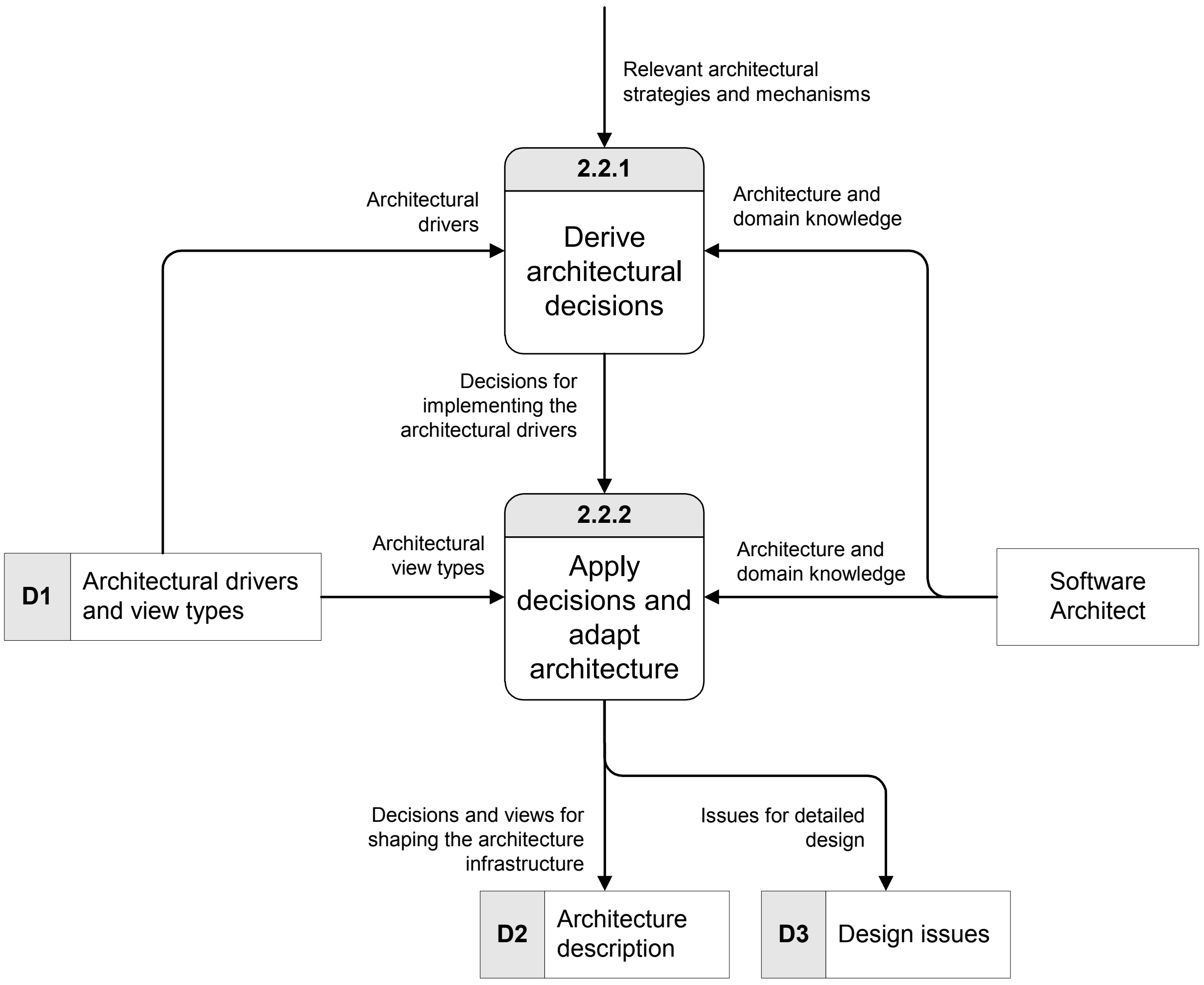

Relevant architectural strategies and mechanisms
2.2.1
Derive architectural decisions
Architectural drivers
Architecture and domain knowledge
Decisions for implementing the architectural drivers
2.2.2
Apply decisions and adapt architecture
Architectural view types
Architecture and domain knowledge
D1
Architectural drivers and view types
Software Architect
Decisions and views for shaping the architecture infrastructure
Issues for detailed design
D2
Architecture description
D3
Design issues

Figure 5-8: Modeling the Architectural Infrastructure (Level-3 Diagram)

5.3.1 Derive Architectural Decisions

The goal of this activity is to derive adequate architectural decisions for addressing the architectural drivers. The decision making process is supported by the set of architectural strategies and mechanisms identified and evaluated in the previous activity.

The activity consists of two steps:

1. Specify architectural decisions
2. Identify corresponding views

1. Specify architectural decisions

The goal of this step is to specify the architectural decisions that support the implementation of the architectural drivers in the architecture. For each architectural driver the associated mechanisms and/or strategies identified in the previous activity are used to derive the architectural decisions that address the drivers. We suggest documenting these decisions together with basic traceability information in order to improve the understanding of the architecture design. As discussed in Section 2.2.2.2, it is essential to document architectural decisions explicitly because they usually cannot be extracted from the description of architectural views anymore. The knowledge about why the architecture is the way it is can only be preserved if the fundamental decisions are recorded. This is also important because refining and especially optimizing the architecture with respect to driving quality attributes can only be achieved systematically if the architect knows about the design decisions and the implications of those decisions.

We suggest using the template given in Table 5-14 for documenting architectural decisions. According to this template, the documentation of an architectural decision should include an identifier for unique reference, the title, as well as a short description of the decision. The description should make clear what the essence of the decision is. Further, the documentation should include the set of architecturally driving requirements or quality attributes that are addressed by the decision as well as the mechanisms that have been used to derive the decision. This supports traceability of the decision. Particularly, it records the rationale why the decision is important for the architecture and why it is specified the way it is specified. Information about the attributes the decision addresses and the approach it uses to implement can be obtained from the results of the QUADRAD-M activity "Define Strategies and Mechanisms" described in the previous section.

The documentation of each architectural decision should also include a discussion of the benefits and drawbacks of the decision. The benefit of the decision can be characterized by referring to the respective design problem it solves (e.g., a performance problem, which is related to a particular architectural driver). Many decisions also have drawbacks because they are competing with other decisions to achieve different quality goals. It is wise to make the drawbacks explicit such that the consequences (e.g., side effects) of the decision can be better understood.

We also suggest that the documentation should include individual comments on the decision. A comment that it is worthwhile to capture is, for example, in which systems the same decision has been made or why the current decision is slightly different from a similar decision of another system. This also supports understandability of the design created based on the decisions. Comments can also be open issues to be resolved during design.

Finally, we propose that the documentation should also contain information about when and by whom or which team the decision has been made. In practice, it is very useful to capture the history of design decisions. For example, it can happen that a decision is made without calling in a person from another project group (e.g., hardware design) that has a stake in the decision (e.g., the decision has severe implications on the hardware, which has not been captured in the documentation). The information about which person or team is responsible for the decisions helps to identify such situations. It also helps to uncover decisions that need to be carefully reviewed by the project stakeholders. Whenever a decision is modified for a particular reason this should also be recorded in the documentation.

According to our experience it is very likely that issues dealing with detailed design and implementation are raised during the decision making process. We suggest capturing the essence of these issues in an accompanied chapter or document. We also advise that these issues should not be investigated in detail during architectural modeling as long as they are not essential for shaping the architecture. Rather, the architectural modeling should be kept focused on the major design problems that affect the overall system. The design issues should be discussed later – i.e., in subsequent detailed design iterations. Table 5-15 shows a template for documenting design decisions.

Table 5-14: Template for Documenting Architectural Decisions

Item	**Description**
AD#	< Unique identifier of the decision. >
Title	< Title of the decision. >
Description	< Short description of the decision. >
Addresses	< Architectural driver or other (quality attribute) requirements the decision relates to. >
Approach	< Short outline of the solution approach. The approach may include architectural mechanisms or strategies that are implemented by the decision. >
Benefits / Drawbacks	< An evaluation of benefits and drawbacks of the decision. >
Comments	< Comments regarding the decision. >
Date and Author	< Creation date and author of the decision. >
Revision	< Revision date and reason for revising the decision. >

Table 5-15: Template for Documenting Design Issues

Design Issue#	**Description**
DI1	< Issues that must be considered during detailed design and implementation regarding an architectural decision AD*i*. >
DI2	< Issues that must be considered during detailed design and implementation regarding an architectural decision AD*j*. >
…	…

Example:

Table 5-16 and Table 5-17 show the documentation of two architectural decisions for the EVCS. The decisions address modifiability issues and support the implementation of the architecturally driving requirement REQ4 (Support software control for customer-specific CD players) which includes the design problem P3. Table 5-16 and Table 5-17 include a short description about each decision and provide information about benefits and drawbacks. The tables also describe open issues related to the decisions (see paragraph "Comments") and contain information about when and by whom the decision has been made.

Table 5-18 summarizes design issues that are related to the decisions.

Table 5-16: Architectural Decision AD001 for the EVCS

Item	**Description**
AD#	AD001
Title	Organize software in 3-tiers
Description	The software architecture is organized according to three layers (3-tier architecture): user interface, application, and system layer. The user interface layer is responsible for presenting information. The application layer includes the application logic and performs computations. The system layer is a hardware abstraction layer and includes low-level functionality for accessing devices connected to the system.
Addresses	Modifiability
Approach	*Mechanism:* • M3: Layering *Strategies:* • S4: Isolate expected changes • S5: Break the dependency chain
Benefits / Drawbacks	*Benefits:* The decision greatly improves the modifiability of the system. It leads to a layered system which abstracts from the underlying hardware and which clearly separates presentation from application functionality. *Drawback:* The decision has performance tradeoffs for messages that need to be processed in realtime. The communication between system layer and presentation layer may be too slow.
Comments	The worst-case communication time between system layer and presentation layer components has to be estimated. (See design issue DI001)
Date and Author	Decision made on January 20, 2005 by the software architect.
Revision	--

Table 5-17: Architectural Decision AD002 for the EVCS

Item	**Description**
AD#	AD002
Title	Implement CD Player as virtual device
Description	The software control for CD player is implemented as virtual device in the driver layer. The virtual device is designed to include a standardized audio interface for accessing CD Player relevant functions. The application functionality and presentation of the CD Player is implemented as separate components. The virtual device design hides implementation details about the communication with hardware devices.
Addresses	• Modifiability • REQ4, P3: Support software control for customer-specific CD players. The CD player control interface varies between different customers. The software control must be easily adapted to the respective interface.
Approach	*Mechanism:* • M3: Layering *Strategies:* • S4: Isolate expected changes • S5: Break the dependency chain
Benefits / Drawbacks	*Benefits:* The decision addresses the design problem P3 of architectural driver REQ4. It improves the modifiability of the CD Player control implementation. The implementation can be adapted to different customers without changing application components. *Drawback:* The responsiveness for accessing the CD Player via the user interface might be limited.
Comments	Check if responsiveness for operating the CD Player via the user interface is sufficient. This affects usability of the system. (See design issue DI002)
Date and Author	Decision made on January 20, 2005 by the software architect.
Revision	Revised on January 21, 2005 by the software architect: Included the drawback/comment on responsiveness.

Table 5-18: Design Issues of the EVCS

Design Issue#	Description
DI001	The communication time between system layer components and presentation layer components has to be tested under high load conditions in order to estimate the worst-case timing. (See architectural decision AD001)
DI002	Check responsiveness of the CD Player application. (See architectural decision AD002)

2. *Identify corresponding views*

The goal of this step is to identify the architectural views that will be affected when the decisions are applied to the system. As discussed in Section 2.2.2.2, applying an architectural decision usually has effect on multiple architectural views. This step helps the architect in the course of adapting those views of the architecture that are affected by a decision. We suggest that architect should make explicit which architectural views are affected by each of the decisions documented in the previous step. This means he or she should get an understanding about which view descriptions are to be modified in order to represent the decisions adequately and completely. In order to do so we recommend that the architect should carefully review the set of architectural view types that have been selected as relevant for the system during the QUADRAD-P activity "Determine architectural view types".

For example, the decision to replicate an architectural design element for safety reasons may affect different views: the software module view, process view, and the deployment view. In the software module view, the functionality may be replicated, e.g., by modeling one or more backup components as well as a decision maker. In the process view, two new processes may be introduced – one for the backup components, another one for the decision maker. This would allow a safe switching to the backup components when the process running the original functionality crashes. Finally, in the deployment view the backup components may reside on an extra processor node to provide a more reliable and faster operation. In order to represent the decision correctly in the architecture, the architect may need to change different views.

Thus we suggest that for each decision the architect should note for which view types he or she will need to make changes. Table 5-19 shows a template for documenting architectural decisions and the corresponding view types that will be affected when the decision is applied.

Note that this step must not necessarily performed for all documented decisions at the same time. We rather recommend that it should be done for a single or a small set of decisions. The architect should then apply the decision and adapt the architecture (i.e., the architectural views), as described in the next section. He or she should then come back to this step in order to successively consider the remaining decisions.

Table 5-19: Template for Documenting Architectural Decision and Affected View Types

Architectural Decision#	Architectural View Types
AD001	< List of view types that need to be modified in order to represent the decision in the architecture. >
AD002	…
…	…

Example:

Table 5-20 shows that the architect will adapt the layered and deployment view when the architectural decisions AD001 and AD002 defined in the previous step will be represented in the architecture.

Table 5-20: View Types Affected By Architectural Decisions AD001 and AD002

Architectural Decision#	Architectural View Types
AD001	Layered, deployment
AD002	Layered, deployment

5.3.2 Apply Decisions and Adapt Architecture

The goal of this activity is to apply the decisions specified in the previous activity to the current architecture and to document/update the resulting architecture views. During this activity the architectural decisions are made effective and persistent in the architecture. Making an architectural decision initiates an architectural transformation where new design elements are added to the architecture and/or existing elements are modified. Design elements (i.e., components and connectors) with predefined responsibilities are incorporated to each architecture view type that is affected by the decision. The responsibilities of the component and connectors *implement* the solution according to the design decision specified in the previous activity. The design decisions, on the other side, conform to a particular architectural mechanism and/or follow architectural strategies which have been defined before. In this way the architecturally driving requirements are incorporated into the architecture in a systematic way. Figure 5-9 illustrates the traceability chain from architectural drivers down to the architecture.

Applying a design decision involves two fundamental design activities:

- Introduce new design elements into the architecture
- Refine or change the behavior of existing design elements

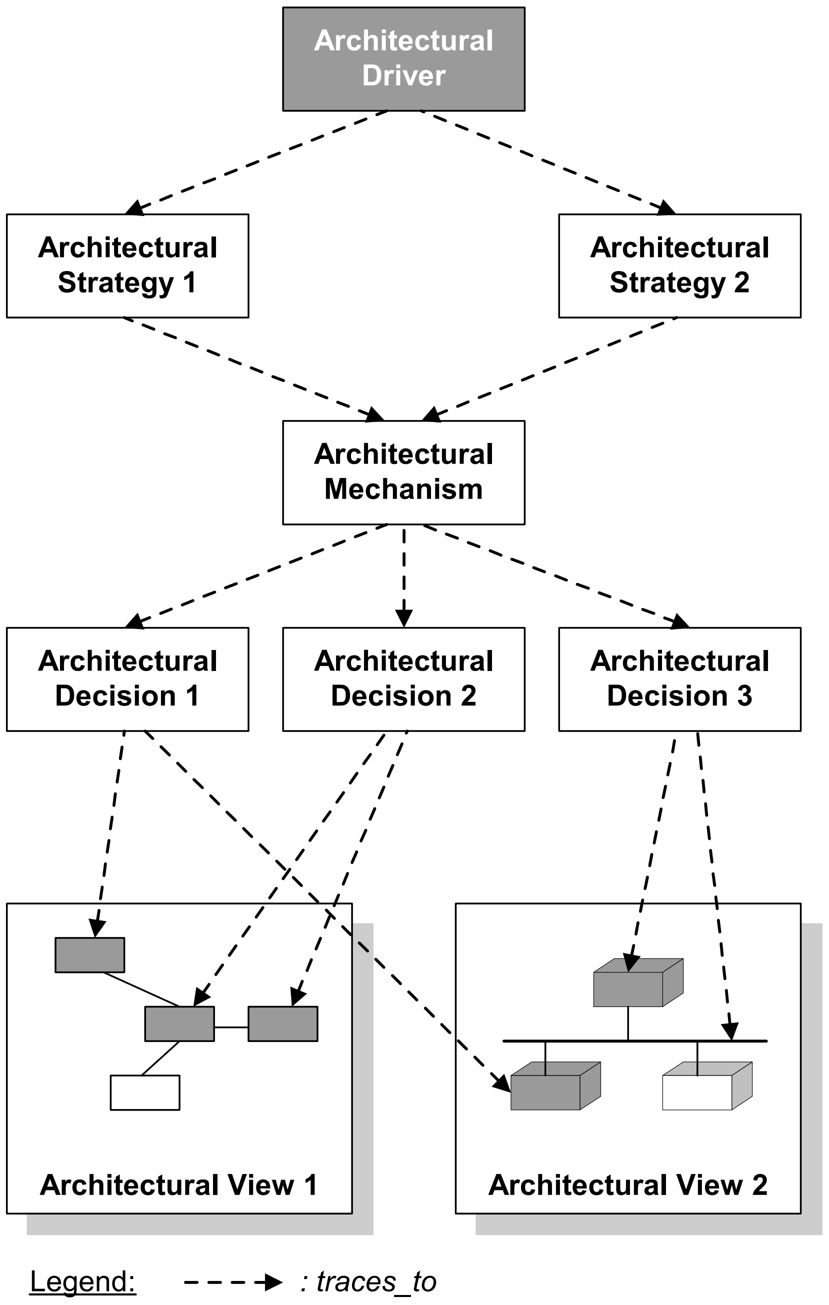


Figure 5-9: Traceability Chain From Architectural Driver to Architectural Views

These activities lead to new architecture releases with different properties. The architectural properties depend, in turn, on the properties of the strategies and mechanisms to which the architectural decisions conform. A property of the architecture could be that it can be easily modified with respect to changes in the user interface. This property may be introduced in the architecture by making decisions that comply to the Model-View-Controller pattern [Buschmann 1996].

A new architecture release may improve a particular property of the architecture. For example, it may improve the architecture's robustness according to failures of the environment and thus exhibits a better reliability. Generally speaking, the new release refines or extends the properties of the previous architecture (release). The evolution from one release to another is called architectural iteration. Each iteration represents a stepwise refinement or extension of the architecture. In summary, during an architectural iteration an architecture is evolved from the current state of properties to a higher elaborated version with additional or improved properties. However, the quality of the architecture may degrade with new releases if the decisions are not systematically derived from driving quality attributes (cf. Section 5.2).

We recommend documenting the behavior/responsibilities and collaborators of each design element with a short description. Table 5-21 shows a CRC template adopted from [Beck 1989] for documenting the behavior/responsibilities and collaborations of design elements. The documentation is usually supported by modern computer-aided software engineering tools as well (e.g., [Kruchten 2000]). The description should state what the element does to fulfil a particular requirement. All responsibilities should be treated as externally accessible (i.e., each design element can use all services of other elements). This is because we suggest not caring about public or private functionality in this early phase of design. In later development phases (i.e., detailed design) the responsibilities should be further refined. Then it also useful to elaborate which responsibilities of a design element really needs to become externally accessible (interface description) and which only provide internal services.

The application of the complete set of architectural decisions specified for the architecturally driving requirements results in the first release of the architecture. This release provides adequate infrastructure design elements for implementing the architectural drivers of the system. The architecture release builds the basis for further refinement (cf. Section 5.4) as well as for detailed design activities.

Before continuing the effort in architecture refinement we recommend performing a review of the architectural infrastructure (cf. QUADRAD-Evaluation, discussed in Chapter 6). In this review the architecture is assessed according to its fitness for achieving the essential quality requirements. If the review does uncover critical risks we recommend to repeat the QUADRAD-M activities "Define architectural strategies and mechanisms" (cf. Section 5.2) and "Model the architectural infrastructure" (cf. Section 5.3) in order to resolve the risks. If the risks have been resolved we suggest continuing with architecture refinement as discussed in Section 5.4.

Table 5-21: Template for Documenting Design Elements

Design Element	**Collaborators**
< Name of design element>	< List of other elements that collaborate with the design element >
Responsibilities	
< Description of the services the design element provides >	

Example:

Figure 5-10 and Figure 5-11 show the layered and deployment view of the EVCS architecture after the decisions AD001 and AD002 have been applied. The notation is based on [Clements 2003]. As depicted in Figure 5-10 the architect has introduced three software design elements: a *CD Player User Interface*, a *CD Player Application*, and a *Virtual CD Player* device driver. The *CD Player User Interface* encapsulates services for presenting the graphical user interface of a CD Player. *The CD Player Application* provides services for operating the CD Player. The *Virtual CD Player* includes services for the low-level communication with the CD Player device.

Figure 5-11 shows the initial deployment view of the architecture. It depicts an active hardware component (*Main Processor*) and a hardware device (*CD Player Device*) that is connected via a communication bus. The deployment view also shows that the complete software of the system resides on the *Main Processor* and that there is only one process (*Software*) which executes the complete software of the system. Table 5-22 summarizes the description of the design elements.

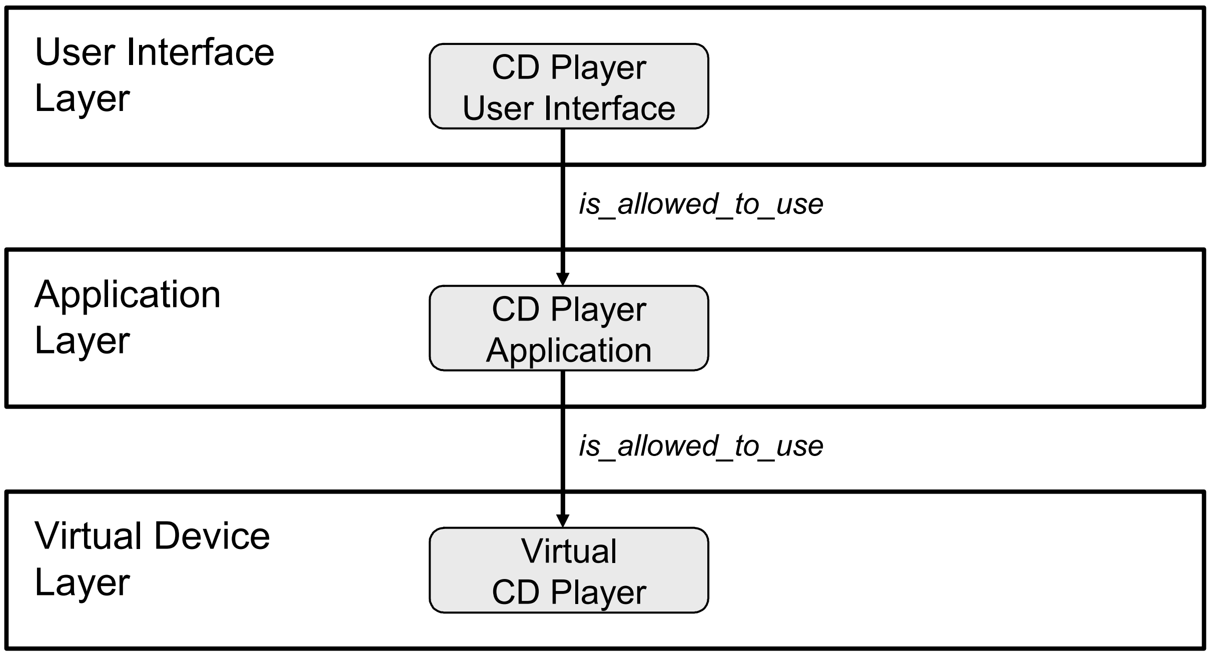


Figure 5-10: Layered View of the EVCS Architecture

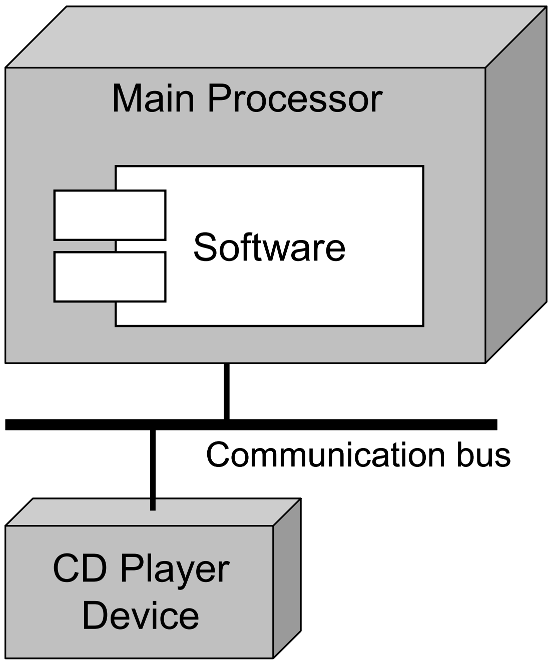


Figure 5-11: Deployment View of the EVCS Architecture

Table 5-22: Documentation of EVCS Design Elements

Design Element	**Description**
CD Player User Interface	*Responsibilities:* Encapsulates services for presenting the graphical user interface of a CD Player. *Collaborators:* CD Player Application
CD Player Application	*Responsibilities:* Encapsulates services for operating the CD Player. *Collaborators:* CD Player User Interface, Virtual CD Player
Virtual CD Player	*Responsibilities:* Encapsulates services for the low-level communication with the CD Player Device. *Collaborators:* CD Player Application
Software	*Responsibilities:* Main process which executes the software. (Design decisions for decomposing the main process in sub-tasks are not yet defined.) *Collaborators:* --
Main Processor	*Responsibilities:* Main processor of the system where the software resides on. *Collaborators:* --
CD Player Device	*Responsibilities:* CD Player Hardware Device for playing Audio and Navigation CDs. *Collaborators:* --

5.4 Refine the Architecture

The objective of this activity is to refine the architecture by incorporating application functionality and the remaining quality attributes to the infrastructure. This activity is started when the architecturally driving requirements have been successfully implemented in the architecture. The architecture refinement stops when implementation details about design elements are specified, since this is the goal of detailed/component design.

An overview of the input and output data flow for the activity is shown in Figure 5-12. The activity is performed by considering the following activities

1. Determine architectural view types (cf. Section 4.3)
2. Define architectural strategies and mechanisms (cf. Section 5.2)
3. Refine the architectural infrastructure (based on Section 5.3)

1. Determine architectural view types

The goal of this activity during architecture refinement is to specify the architectural view types that are relevant for implementing the remaining requirements. The business importance and architectural impact of the requirements have already been estimated during the QUADRAD-P activity "Identify architectural drivers" (cf. Section 4.2). This activity has also uncovered which requirements are architecturally important and which requirements are less critical for architecture development. For the "non-driving" requirements it is checked if the architectural view types defined for the architectural drivers are sufficient. If view types are missing for an adequate representation of the architecture then these view types are added (refer to Section 4.3 for the steps to be performed for determining architectural view types).

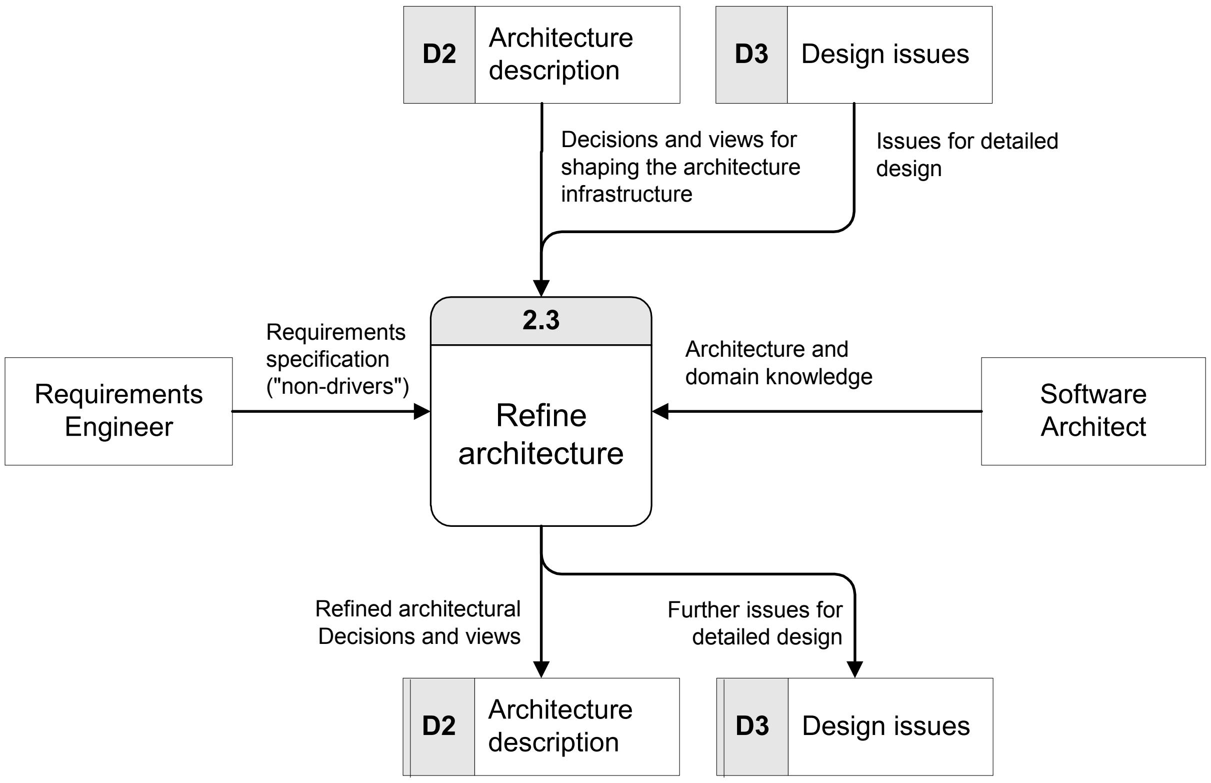

D2
Architecture description
D3
Design issues
Decisions and views for shaping the architecture infrastructure
Issues for detailed design
2.3
Refine architecture
Requirements specification ("non-drivers")
Requirements Engineer
Architecture and domain knowledge
Software Architect
Refined architectural Decisions and views
Further issues for detailed design
D2
Architecture description
D3
Design issues

Figure 5-12: Refining the Architecture (Level-2 Diagram)

2. *Define architectural strategies and mechanisms*

The goal of this activity during architecture refinement is to determine adequate architectural strategies and mechanisms that contribute to the fulfillment of the set of “non-driving” requirements. For each of these requirements the following subordinate activities are performed:

- *Determine architectural strategies:* Determine architectural strategies that support the achievement of the set of “non-driving” requirements.
- *Determine architectural mechanisms:* Identify architectural mechanisms that implement the strategies.
- *Analyze suitability:* Evaluate the consequences of applying the mechanism in the architecture.

According to our experience it may not be necessary to brainstorm strategies and mechanisms for every requirement because some of them may easily be integrated directly into the architectural infrastructure. We suggest that in this case the three activities above should be skipped and the architect should continue with decision making (see next activity). We also recommend proceeding with the next activity when the architect cannot identify appropriate strategies or mechanisms for “non-driving” quality requirements. He or she should then carefully evaluate the consequences of those decisions.

3. *Refine the architectural infrastructure*

The goal of this activity during architecture refinement is to refine the architectural infrastructure of the system that is provided in the first architecture release. During this activity adequate architectural decisions for addressing the “non-driving” requirements are defined. The decisions are then applied to the given architectural infrastructure and the affected views are further updated and refined.

The application of the “Refine architecture” activity results in a second, more elaborated, release of the architecture. This release provides a basis for detailed design activities. However, we recommend performing an overall evaluation of this release in order to analyze the consequences of architectural decisions and to identify potential side effects. Side effects may be introduced to the system because the architect may need to compose different mechanisms. It is important to analyze whether the side effects are critical for achieving the essential quality requirements of the system. We propose to again apply the QUADRAD-Evaluation workflow discussed in Chapter 6 for this purpose.

Example:

Assume the architectural infrastructure for the CD Player of the EVCS (cf. Figure 5-10 and Figure 5-11) shall be refined by incorporating a non-driving requirement “Support basic CD Player functionality”. Table 5-23 shows how the architect could have extended the design elements specified in Table 5-22 in order to incorporate the requirement in the architecture design. Figure 5-13 shows a refined version of the layered view of the architecture.

Table 5-23: Refinement of EVCS Design Elements

Design Element	Description
CD Player User Interface	*Responsibilities:* Encapsulates services for presenting the graphical user interface of a CD Player. The services include showing the CD Player menu (*Show*) and hiding the menu (*Hide*). *Collaborators:* CD Player Application
CD Player Application	*Responsibilities:* Encapsulates services for operating the CD Player. The services include playing Audio CDs (*Play*), stop playing (*Stop*), skipping to the previous audio track (*Previous_Track*), and skipping to the next track (*Next_Track*). *Collaborators:* CD Player User Interface, Virtual CD Player
Virtual CD Player	*Responsibilities:* Encapsulates services for the low-level communication with the CD Player Device. The services include setting a track (*Set_Track*), retrieving the current track number (*Get_Track*), setting the operation mode (*Set_Operation_Mode*), as well as retrieving the current mode of operation (*Get_Operation_Mode*). *Collaborators:* CD Player Application

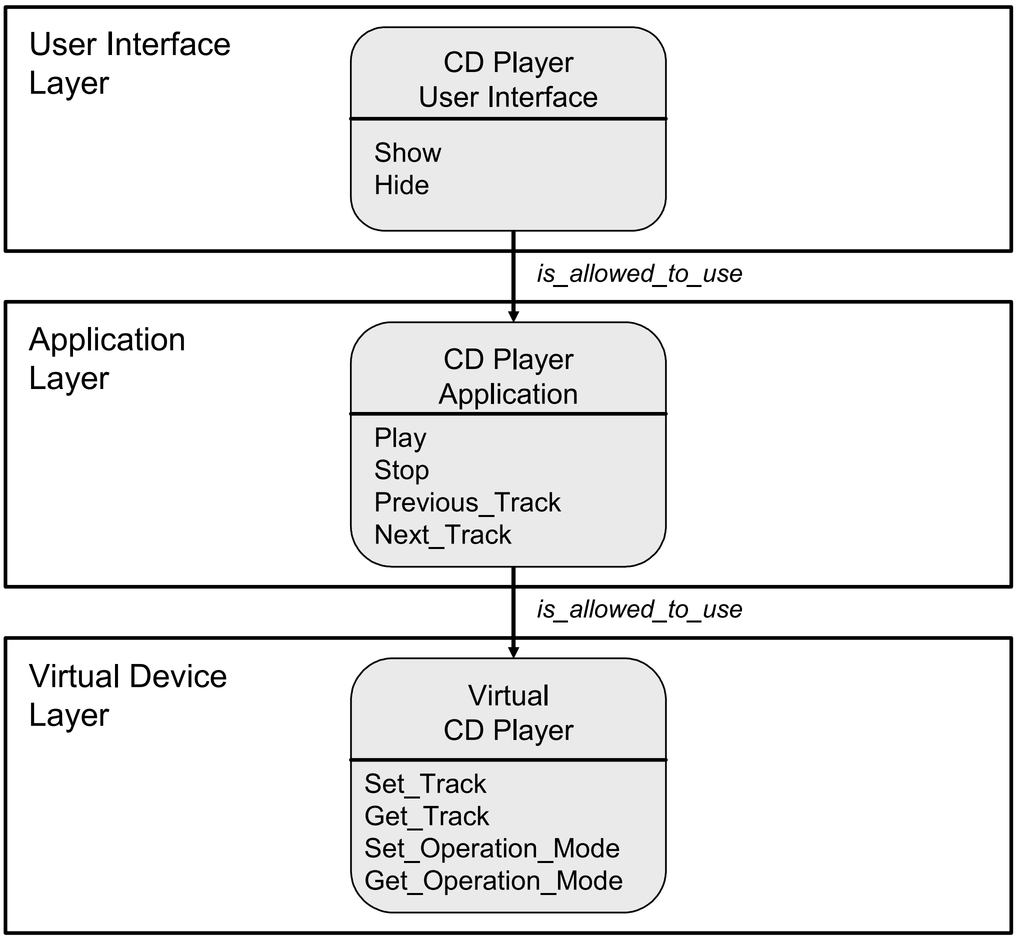


Figure 5-13: Layered View of the EVCS Architecture (Refined)

5.5 Modeling Activities and Research Gaps

The provided activities of the QUADRAD Modeling workflow represent major research contributions of this work. Table 5-24 shows how these activities support bridging the research gap G2 described in Chapter 1. In addition, the table shows which of the framework requirements defined in Table 3-1 are achieved by those activities.

Table 5-24: Mapping Modeling Activities to Research Gaps and Requirements

Modeling Workflow Activities	Research Gaps	Framework Requirements
Define architectural strategies and mechanisms	G2	FR4, FR5
Model the architectural infrastructure	G2	FR3, FR4, FR5
Refine the architecture	G2	FR3, FR4, FR5

Contributions of the QUADRAD Modeling workflow with respect to bridging the research gaps:

Research Gap 2: No systematic support for making adequate design decisions in order to implement architecturally relevant requirements.

The "Define architectural strategies and mechanisms" activity (cf. Section 5.2) effectively allows the identificantion of those design solutions that contribute to the achievement of the essential design problems. Thus, it enables the architect to focus on the hardest design problems first. It allows the direct relation of architectural strategies and mechanism to architecturally driving requirements and thus supports traceability between major artifacts of the problem and solution space.

The "Model the architectural infrastructure" and "Refine the architecture" activities (cf. Section 5.3 and 5.4) both allow for a systematic documentation of architectural decisions. They make effective use of architectural strategies and mechanisms that have been proven useful to achieve the architecturally driving requirements. The activities support the architect in designing the architectural infrastructure of the system. The architectural infrastructure comprises those design elements that are essential for achieving the most important quality properties and business goals. As the infrastructure is shaped by the set of architecturally driving requirements, it provides a solution for the most strategic design decisions. Those decisions typically have the highest impact on structural aspects of the architecture and are the most difficult to change.

In addition, the activities of the QUADRAD Modeling workflow explicitly consider making architectural decisions to refine the architecture in multiple architectural views. The types of architectural view that are important in a specific system context have been previously defined in the Preparation workflow. The consideration of multiple views during architecture design is a central issue. The Modeling workflow provides a template for documenting the architectural decisions in each view. It thus supports a systematic and concise documentation of the rationales that lead to the different architectural structures. This, for example, improves the understanding of how the architecture responds to critical system qualities. It also shortens the learning time for other stakeholders.

The QUADRAD Modeling activities also support traceability between architectural strategies/mechanisms and design elements. The trace information among architectural mechanisms, decisions, and the resulting design elements are recorded. Together with the traceability support provided by the Preparation workflow, QUADRAD allows to trace a particular requirement to those elements in the architecture that contribute to the implementation of that requirement and vice versa. With this tracing capability, more control over the architecture as well as a better understanding of the main architectural interrelations can be achieved.

5.6 Summary

In this chapter the activities and artifacts of the QUADRAD Modeling workflow (QUADRAD-M) have been described. The purpose of this workflow is to support the modeling of architectural views and to record the decisions that lead to those views. In particular, activities for creating the architectural infrastructure of the system, which comprises those design elements that are essential for achieving the most important quality requirements, have been described. Furthermore, steps for refining this infrastructure by incorporating application functionality and remaining quality attributes to the infrastructure have been presented. Finally, it has been shown how the QUADRAD-M activities support bridging the research gaps of this thesis.

In the next chapter, we will discuss activities for evaluating an architecture with respect to the achievement of driving requirements.

6 The QUADRAD Evaluation Workflow

The purpose of the QUADRAD Evaluation workflow (QUADRAD-E) is to understand the consequences of architectural decisions made by the architect, to analyze them with respect to the achievement of essential quality requirements, and to propose architectural improvements. QUADRAD-E is facilitated by an external architecture evaluation team in order to guarantee objective results. An overview of the major input and output artifacts is shown in Figure 6-1.

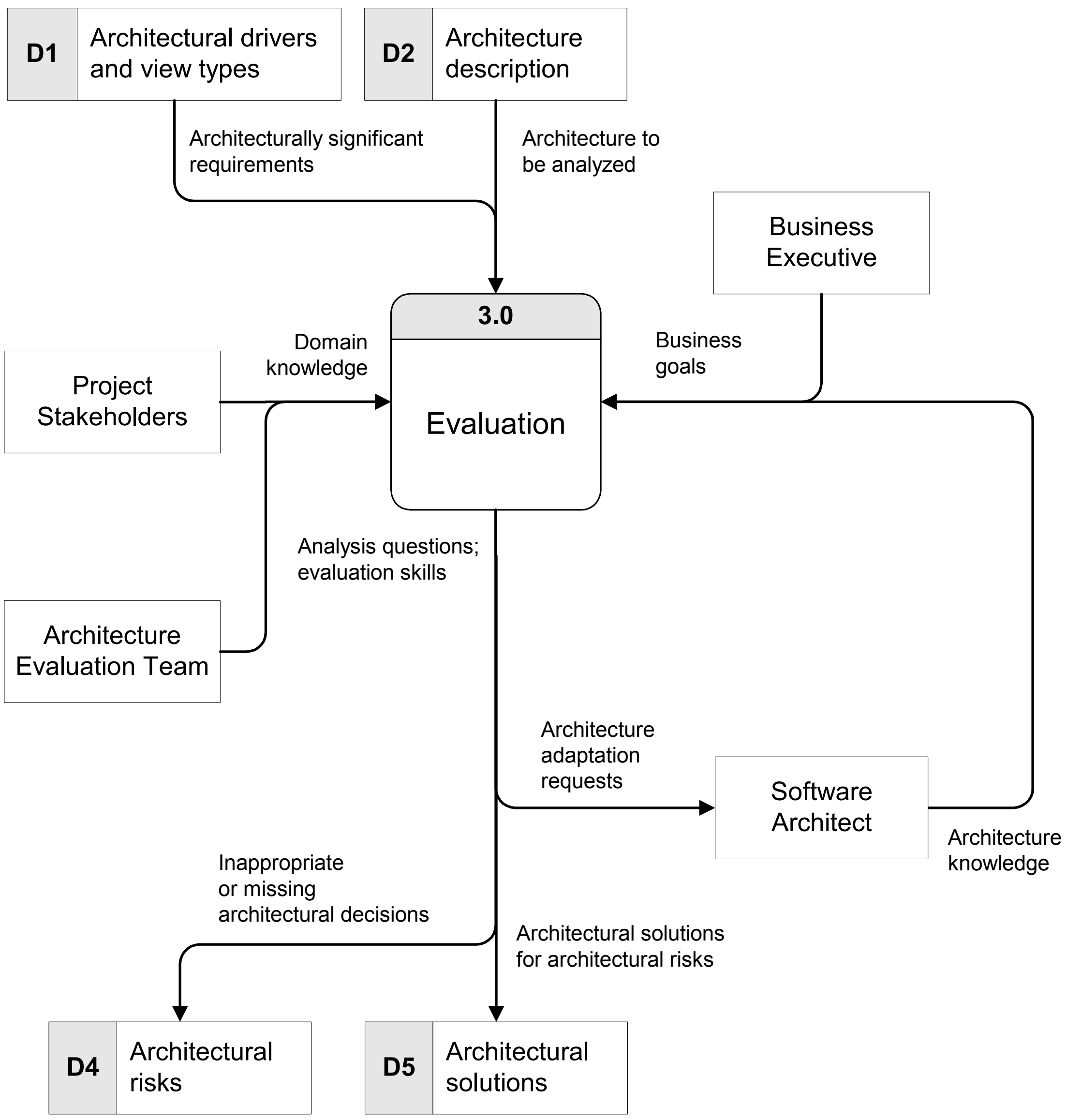

D1 Architectural drivers and view types
D2 Architecture description
Architecturally significant requirements
Architecture to be analyzed
Business Executive
3.0
Evaluation
Domain knowledge
Business goals
Project Stakeholders
Analysis questions; evaluation skills
Architecture Evaluation Team
Architecture adaptation requests
Software Architect
Architecture knowledge
Inappropriate or missing architectural decisions
Architectural solutions for architectural risks
D4 Architectural risks
D5 Architectural solutions

Figure 6-1: The QUADRAD Evaluation Workflow (Level-1 Diagram)

The chapter is organized as follows: Section 6.1 provides an overview of the Evaluation workflow. In Section 6.2 activities for defining the goals of the architecture evaluation are described. Section 6.3 explains how evaluation scenarios are elicited and prioritized based on the goals. Activities and techniques for making those parts of the architecture explicit that implement the most driving requirements concerning the evaluation goals are described in Section 6.4. In Section 6.5 activities for analyzing the appropriateness of the architectural decisions made by the architect, for proposing improvements, as well as for organizing the findings systematically for risk mitigation are described. Section 6.6 explains how QUADRAD-E activities address the research gaps in this thesis. Finally, Section 6.7 summarizes the main issues of this chapter.

6.1 Workflow Overview

Figure 6-1 shows the first decomposition of QUADRAD-E. It consists of four activities:

1. Define evaluation goals
2. Elicit and prioritize evaluation scenarios
3. Map scenarios and identify decisions
4. Analyze decisions and summarize findings

1. Define evaluation goals

The objective of this activity is to define the context of the evaluation. This includes the clarification of the specific evaluation goals and the documentation of rationales for those goals.

The inputs to the activity are business goals (provided by the business executive), architecture knowledge (provided by the software architect) as well as evaluation skills (provided by the architecture evaluation team). The outputs from the activity are quality attribute goals that are relevant for the evaluation.

2. Elicit and prioritize evaluation scenarios

The objective of this activity is to elicit and prioritize scenarios to be used to analyze the architecture with respect to the evaluation goals.

The inputs to the activity are the quality attribute goals for evaluation (provided by the activity “Define evaluation goals”), architecturally driving requirements (either provided by QUADRAD-P or from requirements engineering if QUADRAD-E is performed independently), knowledge about the business goals of the system (provided by the business executive) as well as architecture and domain knowledge (provided by the software architect and project stakeholders that are involved in the system’s development). The outputs from the activity are a prioritized list of evaluation scenarios to be used to analyze the architecture’s fitness.

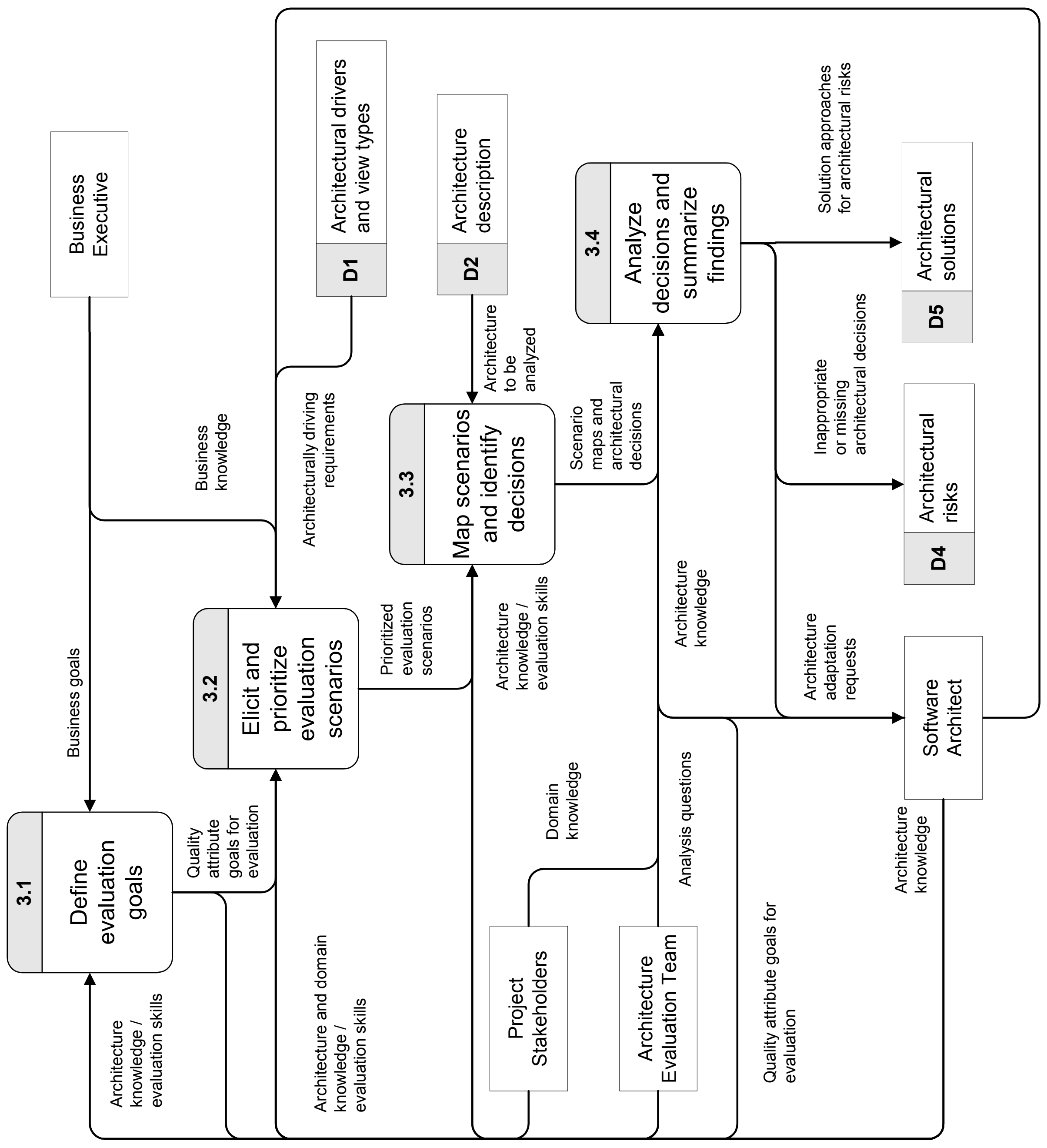

Business Executive
D1 Architectural drivers and view types
D2 Architecture description
3.4 Analyze decisions and summarize findings
Solution approaches for architectural risks
D5 Architectural solutions
Business knowledge
Architecturally driving requirements
Architecture to be analyzed
3.3 Map scenarios and identify decisions
Scenario maps and architectural decisions
Inappropriate or missing architectural decisions
D4 Architectural risks
Business goals
3.2 Elicit and prioritize evaluation scenarios
Prioritized evaluation scenarios
Architecture knowledge / evaluation skills
Architecture knowledge
Architecture adaptation requests
Software Architect
3.1 Define evaluation goals
Quality attribute goals for evaluation
Domain knowledge
Analysis questions
Architecture knowledge
Architecture knowledge / evaluation skills
Architecture and domain knowledge / evaluation skills
Project Stakeholders
Architecture Evaluation Team
Quality attribute goals for evaluation

Figure 6-2: The QUADRAD Evaluation Workflow (Level-2 Diagram)

3. Map scenarios and identify decisions

The objective of this activity is to make those parts of the architecture explicit that are affected by the evaluation scenarios. This includes the identification of design decisions that contribute to the achievement of the evaluation scenarios.

Inputs to the activity are the prioritized evaluation scenarios (provided by the activity “Elicit and prioritize evaluation scenarios”), the architecture description (either provided by QUADRAD-M or from an external source such as the software architect if QUADRAD-E is performed independently), architecture knowledge (provided by the software architect), as well as evaluation skills (provided by the architecture evaluation team). Outputs from the activity are scenario maps that document the design elements and responsibilities affected by the scenarios in different architectural views as well as associated architectural decisions.

4. Analyze decisions and summarize findings

The objective of this activity is firstly to analyze the appropriateness of the decisions made in the architecture to satisfy the evaluation scenarios and secondly to propose architectural improvements. For each scenario map we assess to what extent the responsibilities of the design elements contribute to the achievement of the scenario. This involves an explicit identification of missing or inappropriate decisions and a systematic clustering of findings.

Inputs to the activity are the scenario maps and architectural decisions identified for the evaluation scenarios (provided by the activity "Map scenarios and identify decisions"), architecture and domain knowledge (provided by the software architect and project stakeholders) as well as evaluation skills (provided by the architecture evaluation team). Outputs from the activity are a list of architectural risks associated to the decisions made in the architecture and a list of solution approaches to address the risks. The outputs are a prerequisite for architectural change requests and are communicated to the software architect.

As mentioned in the activity overview, QUADRAD-E can be applied independently from QUADRAD-P (cf. Chapter 4) and QUADRAD-M (cf. Chapter 5). It provides explicit steps for defining the evaluation goals and scenarios that serve for the analysis of any existing architecture. QUADRAD-E also supports documentation (of a weakly described architecture), especially with respect to the architectural drivers and important decisions used to create the architecture. However, the benefits and limitations of using QUADRAD-E for reconstructing an architecture description are not treated in detail in the remainder of this chapter because this topic is not part of the research problems addressed in this work.

We suggest that the architecture evaluation should be led by an independent evaluation team. An independent team supports the activities of an architecture evaluation from a neutral position. It helps to gather facts about the realization of the architecture and guarantees objective results for the evaluation. Social skills must also be provided by the evaluation team. For example, the evaluation team will have to protect the architect should the audience want to blame him or her for wrong decisions. However, discussing details about professional and social requirements for the evaluation team is beyond the scope of this work.

The activities of QUADRAD-E are next described in more detail.

6.2 Define Evaluation Goals

The objective of this activity is to clarify the specific goals of the architecture evaluation and to document a rationale for those goals. An overview of the input and output data flow is shown in Figure 6-3.

Basically, there are three possibilities to evaluate an architecture. It can be evaluated against

1. the achievement of a particular quality attribute (e.g., performance)
2. the achievement of multiple quality attributes (e.g., modifiability, safety, and reliability)
3. the achievement of those quality attributes that are affected by the architecturally driving requirements (specialization of the previous case)

We suggest considering the second or third alternative for driving an evaluation in order to perform an overall evaluation against the business goals. As discussed in Section 2.3, evaluating an architecture against a single quality attribute can be critical if the architecture relies on multiple competing qualities. A subsequent optimization of the architecture with respect to that quality then will probably have a negative impact on other system qualities. Assessing dedicated system qualities should thus be done very carefully when an overall evaluation is not feasible (e.g., because of time constraints).

For each of the three cases the evaluation goals should be chosen in a way that one can make a statement about how the architecture should support the business goals of the system. For example, if the business goal is to provide an engine control system, which can be adapted flexibly to individual emission regulations of different markets, then the architecture should be evaluated against its fitness for modifiability with respect to emission control. If the business strategy is to provide a system that is 99.95% available for the customer then the architecture should be evaluated against quality attributes such as availability, reliability, and safety. We suggest capturing a rationale or specific questions to be answered for each evaluation goal in order to make explicit why it is important to analyze the architecture with respect to those goals. Table 6-1 shown a template for capturing goals for an evaluation.

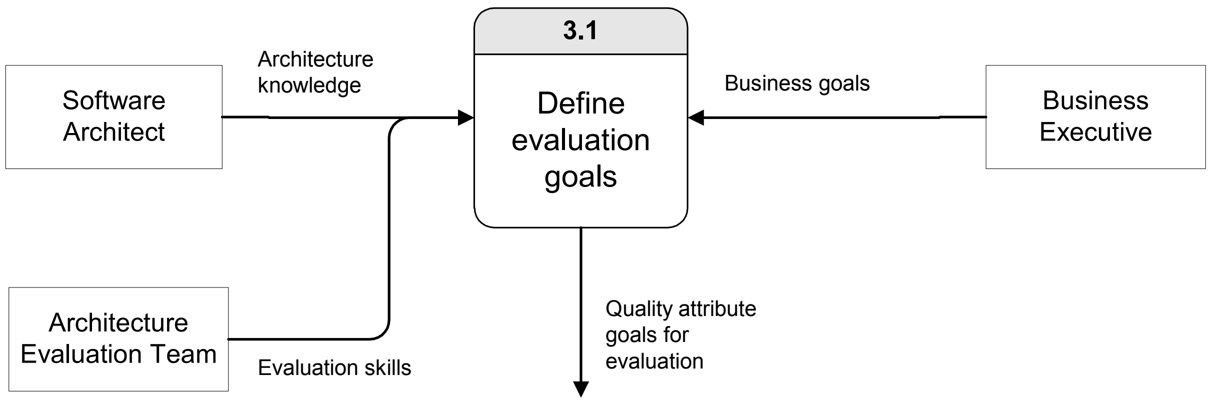


Figure 6-3: Defining Evaluation Goals (Level-2 Diagram)

Table 6-1: Template for Capturing Evaluation Goals

Evaluation Goals	Description
< Quality attribute 1 >	< Rationale 1.1 / evaluation question 1.1>
	< Rationale 1.2 / evaluation question 1.2>
	…
< Quality attribute 2 >	< Rationale 2.1 / evaluation question 2.1>
…	…

Example:

Table 6-2 shows a list of four goals to be used for evaluating the EVCS system with respect to the architecturally driving requirements (cf. Section 4.2).

Table 6-2: Evaluation Goals for the EVCS architecture

Evaluation Goals	Description
Integrability	Does the architecture support an easy integration of 3rd party software for air condition control?
Modifiability	Does the architecture support an easy modification of software control components for customer-specific CD players?
	Does the architecture support an easy configuration of low-end system at end-of-line?
Performance	Does the architecture support a quick system start-up?
Safety	Does the architecture support the detection of system failures during start-up?

6.3 Elicit and Prioritize Evaluation Scenarios

The objective of this activity is to elicit and prioritize scenarios to be used for analyzing the architecture with respect to the evaluation goals. This is achieved based on a description of evaluation scenarios that decompose the goals into concrete statements, which can be analyzed in the architecture. An overview of the input and output data flow of this activity is shown in Figure 6-4.

Figure 6-5 shows how this activity is decomposed into the following two subordinate activities:

1. Brainstorm and organize evaluation scenarios
2. Calculate scenario priorities

These activities are next described in more detail.

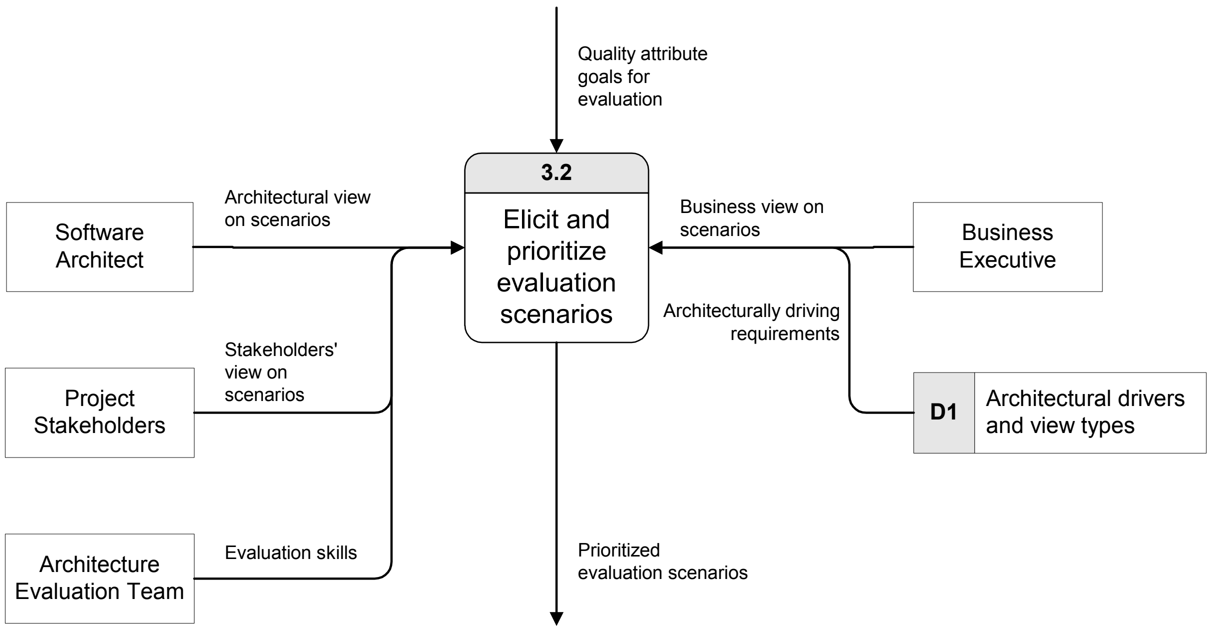

Quality attribute goals for evaluation
3.2
Elicit and prioritize evaluation scenarios
Software Architect
Architectural view on scenarios
Business view on scenarios
Business Executive
Architecturally driving requirements
Stakeholders' view on scenarios
Project Stakeholders
D1
Architectural drivers and view types
Architecture Evaluation Team
Evaluation skills
Prioritized evaluation scenarios

Figure 6-4: Eliciting and Prioritizing Evaluation Scenarios (Level-2 Diagram)

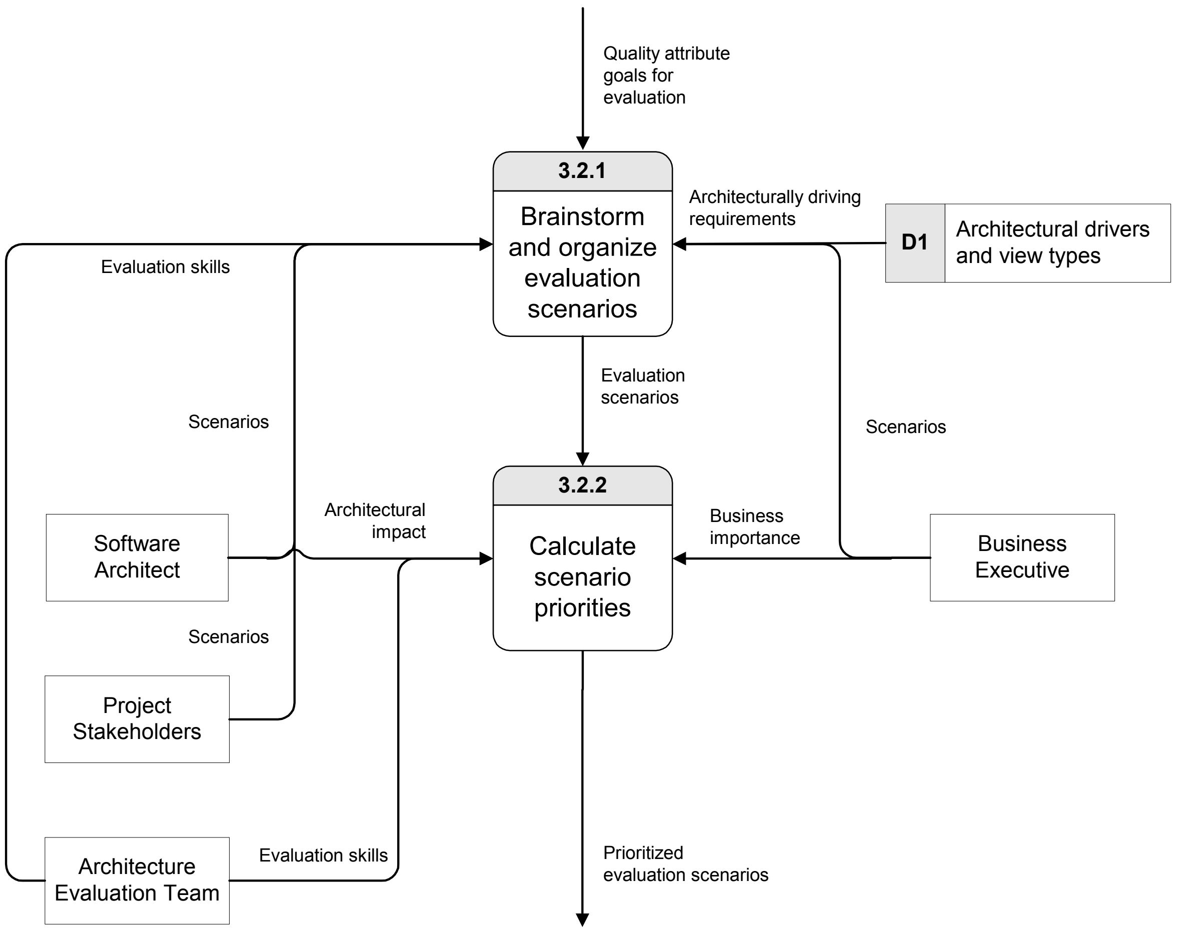

Quality attribute goals for evaluation
3.2.1
Brainstorm and organize evaluation scenarios
Architecturally driving requirements
D1
Architectural drivers and view types
Evaluation skills
Scenarios
Evaluation scenarios
Scenarios
3.2.2
Calculate scenario priorities
Architectural impact
Software Architect
Business importance
Business Executive
Scenarios
Project Stakeholders
Architecture Evaluation Team
Evaluation skills
Prioritized evaluation scenarios

Figure 6-5: Eliciting and Prioritizing Evaluation Scenarios (Level-3 Diagram)

6.3.1 Brainstorm and Organize Evaluation Scenarios

The goal of this activity is to collect a list of evaluation scenarios that cover the goals of the evaluation. The activity consists of two steps:

1. Brainstorm scenarios for evaluation
2. Check scenario coverage according to evaluation goals

1. Brainstorm scenarios for evaluation

Based on the evaluation goals, candidate usage, change, and stress scenarios are brainstormed in order to clarify how the architecture should be assessed. As introduced in Section 2.3.3.1, usage scenarios describe a stakeholder's intended interaction with the system. They are used for getting a basic understanding of the system. Change scenarios cover anticipated future changes to a system. Stress scenarios deal with extreme situations that are expected to "stress" the system in order to expose the limits or boundary conditions of the current design.

Each scenario should be described such that it becomes clear how to use it for evaluating the architecture. We suggest using the guidelines proposed by [Bass 2003] (cf. Section 2.3.3.1) in order to describe scenarios effectively for architecture evaluation. Table 6-3 shows which elements should be included in a scenario description.

The brainstorming process involves the business executive, the software architect, project stakeholders of the system development core team, as well as the architecture evaluation team. The brainstorming process can be facilitated in three different ways:

- *Brainstorming by scenario type:* In this alternative, the scenarios are brainstormed along particular types of scenario. This means that, for example, in a first phase the brainstorming process allows only the capturing of usage scenarios. In a second phase, only change scenarios are elicited. Finally, in the third phase stress scenarios are added to the list of evaluation scenarios.
- *Brainstorming by quality attribute:* The second alternative is concerned with driving the brainstorming process along the quality attribute goals. For example, if the goal of the architecture evaluation is to analyze modifiability and safety issues, then the brainstorming can be dedicated to capturing modifiability scenarios first and safety scenarios in a second phase.
- *Brainstorming by scenario types and quality attributes:* This represents a mixture of the previous two brainstorming alternatives.

Whether one alternative of facilitating the brainstorming works better than another alternative depends on the individuals that participate in the process. Many software architects, for example, think in terms of quality attributes. They may prefer brainstorming scenarios in this way. A business executive, on the other hand, often tries to stress the architecture with respect to "fancy" features. He or she thus may like brainstorming according to scenario types. In case the brainstorming process does not work well, we suggest not restricting the elicitation but allowing individuals to bring in their scenarios as they come to mind. If the QUADRAD-P workflow (cf. Chapter 4) has been performed prior to architecture evaluation, then the architectural drivers that match with the evaluation goals can be used as additional input for seeding the scenario brainstorming process.

Table 6-3: Template for Describing Evaluation Scenarios

Element	Description
Source of stimulus	Entity (e.g., human, computer system, or other actors) that generated the stimulus.
Stimulus	Condition that needs to be considered when it arrives at a system.
Environment	State of the system when the stimulus arrives. The system may be in an overload condition or may be running when the stimulus occurs, or some other conditions may be true.
Artifact	Element of the system that is stimulated. This may be the whole system or some part of it.
Response	Activity undertaken after the arrival of the stimulus.
Response measure	When the response occurs, it should be measurable in some fashion so that the requirement can be tested.

2. Check scenario coverage according to evaluation goals

After a first brainstorming iteration, it should be checked whether every quality attribute has been represented by at least one sound scenario that supports assessing the architecture. If one or more attributes are not represented by scenarios then the brainstorming should be repeated for those attributes. It is best to choose the "brainstorming by quality attribute" alternative described in the previous step to focus the elicitation process. Note that it is best to have more than one scenario for each quality goal (up to five is a good rule of thumb) since then the probability of performing the evaluation more broadly is increased. Table 6-4 shows a template for checking the coverage of scenarios with respect to evaluation goals.

We also suggest restricting the brainstorming process in time. Depending on the complexity of the system and the number of different goals to be achieved by the evaluation, two hours are generally a good time frame for eliciting scenarios. Checking coverage should be done at least once within the brainstorming process (e.g., in the middle of the session) in order to keep the process on track.

Table 6-4: Template for Checking Quality Coverage of Evaluation Scenarios

Evaluation Goals	Evaluation Scenarios	Total#
< Quality Attribute 1 >	< List of evaluation scenarios addressing Quality Attribute 1 >	< Number of scenarios 1>
< Quality Attribute 2 >	< List of evaluation scenarios addressing Quality Attribute 2 >	< Number of scenarios 2>
…	...	...

Example:

Table 6-5 shows different types of evaluation scenarios for the quality attribute goals performance, usability, reliability, safety, availability, security, and modifiability. The scenarios are described by taking the scenario description guidelines proposed in Table 6-3 into account. For example, the different elements of the scenario

ES1: Remote user requests a database report via the Web during peak period and receives it within five seconds.

are as follows:

Source of stimulus	Remote user
Stimulus	Requests database report via the Web
Environment	During peak period and
Artifact	(database report)
Response	Receives it (the database report)
Response measure	Within five seconds.

Note that filling out all the elements of the description template does not make sense for every scenario. The important point here is that the template should serve as a *guideline* for scenario description rather than a strict rule. During brainstorming, there is the tradeoff between frequency of eliciting scenarios and accuracy of the scenario description. The evaluation team should be aware of this tradeoff and facilitate the stakeholders in formulating scenarios according to the elements of the description guideline. We suggest that they should recapitulate that, in order to guarantee useful scenarios, it must be clear what the stimulus is, what the environmental conditions are, and what the measurable manifestation of the response is.

Table 6-6 shows an organization of the evaluation scenarios of Table 6-5 according to quality attributes. As illustrated, the scenarios cover modifiability, performance, and reliability aspects of the system very well. These quality attributes also represent the goals for evaluating the architecture in the example. Safety, availability, and security aspects are only touched on lightly. If these attributes were part of the evaluation goals then the brainstorming process should be focused on the attributes in a second iteration.

Table 6-5: Examples of Evaluation Scenarios

ES#	Evaluation Scenario	Quality Attribute
Usage Scenarios		
ES1	*Request database report:* Remote user requests a database report via the Web during peak period and receives it within five seconds.	Performance
ES2	*Change graph layout:* The user changes the graph layout from horizontal to vertical. The graph is redrawn in one second.	Performance
ES3	*Recalculate route:* Recalculate route within two seconds after an incoming radio data message has been received.	Performance
ES4	*Occurrence of an exception:* A data exception occurs and the system notifies a defined list of recipients by e-mail and displays the offending conditions in red on data screens.	Reliability
ES5	*Switch caching system:* The caching system will be switched to another processor when its processor fails, and will do so within one second.	Reliability
Change scenarios		
ES6	*Add new message type:* Add a new message type to the system's repertoire in less than a person-week of work.	Modifiability
ES7	*Add new map and interface:* Add a new 3-D map feature, and a virtual reality interface for viewing the maps in less than five person-months of effort.	Modifiability
ES8	*Add planning capability:* Add a collaborative planning capability, where two planners at different sites collaborate on a plan, in less than a person-year of work.	Modifiability
ES9	*Double database size:* Double the size of existing database tables while preserving an average retrieval time of one second.	Performance, reliability
ES10	*Add data server:* Add a new data server to reduce latency to 2.5 seconds within one person-week.	Modifiability, performance
ES11	*Migrate to new operating system:* Migrate software to a new release of the existing operating system in less than a person-year of work.	Modifiability, performance, security, reliability
Stress scenarios		
ES12	*Reuse legacy software:* Integrate a 25-years-old software in a new generation automotive system within one month.	Modifiability, performance, safety, security, reliability
ES13	*Reduce time for displaying route:* The time for displaying changed route data is reduced by a factor of 10.	Performance
ES14	*Improve availability:* Improve the system's availability from 98% to 99.999%.	Availability, performance, reliability
ES15	*Half the servers:* Half of the servers go down during normal operation without affecting overall system availability.	Availability, performance, reliability

Table 6-6: Quality Coverage of Evaluation Scenarios

Quality Attributes	Evaluation Scenarios	Total#
Modifiability*	ES6, ES7, ES8, ES10, ES11, ES12	6
Performance*	ES1, ES2, ES3, ES9, ES10, ES11, ES12, ES13, ES14, ES15	10
Reliability*	ES4, ES5, ES9, ES11, ES12, ES14, ES15	7
Safety	ES12	1
Availability	ES14, ES15	2
Security	ES11, ES12	2

* = Evaluation Goals

6.3.2 Calculate Scenario Priorities

The goal of this activity is to prioritize the scenarios according to their importance for evaluating the architecture. The activity consists of two steps:

1. Prioritize evaluation scenarios
2. Sort evaluation scenarios according to their priorities

1. Prioritize evaluation scenarios

In this step, the evaluation scenarios elicited in the previous activity are prioritized in order to achieve a ranking for driving the evaluation. Similar to the ranking of requirements (cf. Chapter 4), judging the importance of a particular scenario requires a multi-criteria decision making process. We suggest using the same criteria to prioritize the scenarios as for ranking requirements (namely, business importance and architectural impact). These criteria apply here since they assist in focusing on those scenarios that are essential for assessing the architecture.

For example, if a particular scenario is important for achieving the business goals (and some of the business goals are often goals of the evaluation) then we should look at it. Further, if that scenario is a usage or stress scenario and has at the same time a high impact on the architecture then it clearly was difficult to realize in the architecture. We should definitely recommend the scenario for evaluation in order to analyze whether it has been properly implemented. If the scenario, on the other side, is a change scenario then a high architectural impact does reflect the fact that it will be difficult to achieve (e.g., integrating new functionality or changing the existing functionality is difficult). If at the same time that scenario is important for business success then we should recommend it for evaluation since it seems that the software architect might have missed to make the appropriate decisions during architecture design.

We also suggest using an absolute weighting approach with an easy weighting scale (e.g., High, Medium, Low) if there is limited time for scenario prioritization. With an absolute weighting approach each scenario is evaluated independently against the criteria. Similar to the approach of identifying architecturally driving requirements (cf. Section 4.2), the business executive judges the business importance of the scenario and the software architect assesses its architectural impact. Table 6-7 shows a template for prioritizing evaluation scenarios.

Table 6-7: Template for Scenario Prioritization

ES#	Business Importance*	Architectural Impact*
ES1	< Business importance of evaluation scenario ES1 >	< Architectural impact of evaluation scenario ES1 >
ES2	< Business importance of evaluation scenario ES2 >	< Architectural impact of evaluation scenario ES2 >
…	...	...

* Weighting scale = { High, Medium, Low }

In some cases where it is not obvious why a scenarios has a high/low business importance or architectural impact, we suggest documenting the rationale for the weighting. Then the assigned weights can be better understood by the evaluation stakeholders throughout QUADRAD-E.

2. Sort evaluation scenarios according to their priorities

In this step, the scenarios are sorted in descending order of priority. The scenarios that have been ranked with highest weights for the criteria are the best candidates for evaluation (as explained above). However, there may be certain limitations for selecting candidate scenarios for further exploration. For example, if time is a problem then using a threshold for selection (e.g., the first three highest priority scenarios) may be appropriate. On the other side, if a scenario can only be explored with the support of particular experts but these experts do not participate in the evaluation, then this scenario should be omitted and put into a backlog, even if it is highly important. It should not be chosen because an objective or in-depth evaluation would not be possible. In any case, the decisions concerning why a particular scenario has not been selected for evaluation should be recorded. This allows focusing on backlog scenarios in subsequent iterations of QUADRAD-E.

Example:

Table 6-8 shows the top-5 high priority scenarios ES3, ES9, ES10, ES12, and ES14 to be chosen for architecture evaluation. The scenarios are important for achieving the business goals and they have a high impact on the design of the architecture.

Table 6-8: High Priority Evaluation Scenarios

ES#	Business Importance	Architectural Impact
ES3	High	High
ES9	High	High
ES10	High	High
ES12	High	High
ES14	High	High

6.4 Map Scenarios and Identify Decisions

The objective of this activity is to make those parts of the architecture explicit that are affected by the evaluation scenarios. This includes capturing design elements and identifying design decisions that contribute to the achievement of the evaluation scenarios. Briefly, we go through the architecture description in this activity and isolate those design elements and approaches that contribute to the fulfillment of the scenario under consideration. Note that this activity is performed in concert with the next activity ("Analyze decisions and summarize findings") for every evaluation scenario to be considered in the evaluation.

An overview of the input and output data flow of mapping scenarios and identifying decisions is shown in Figure 6-6. The activity consists of two steps:

1. Select evaluation scenario for investigation
2. Identify contributing design elements and capture decisions

1. Select evaluation scenario for investigation

In this step an evaluation scenario for further investigation is selected in descending order of the assigned priority. We suggest that the evaluation team asks the audience if the scenario is completely understood. If the scenario description is not clear to the people then it should be refined without changing the original intention (especially, stimulus and response). If the scenario cannot be refined without changing the intention we suggest making the description clear first and then prioritizing the modified scenario again, as defined in Section 6.3.2.

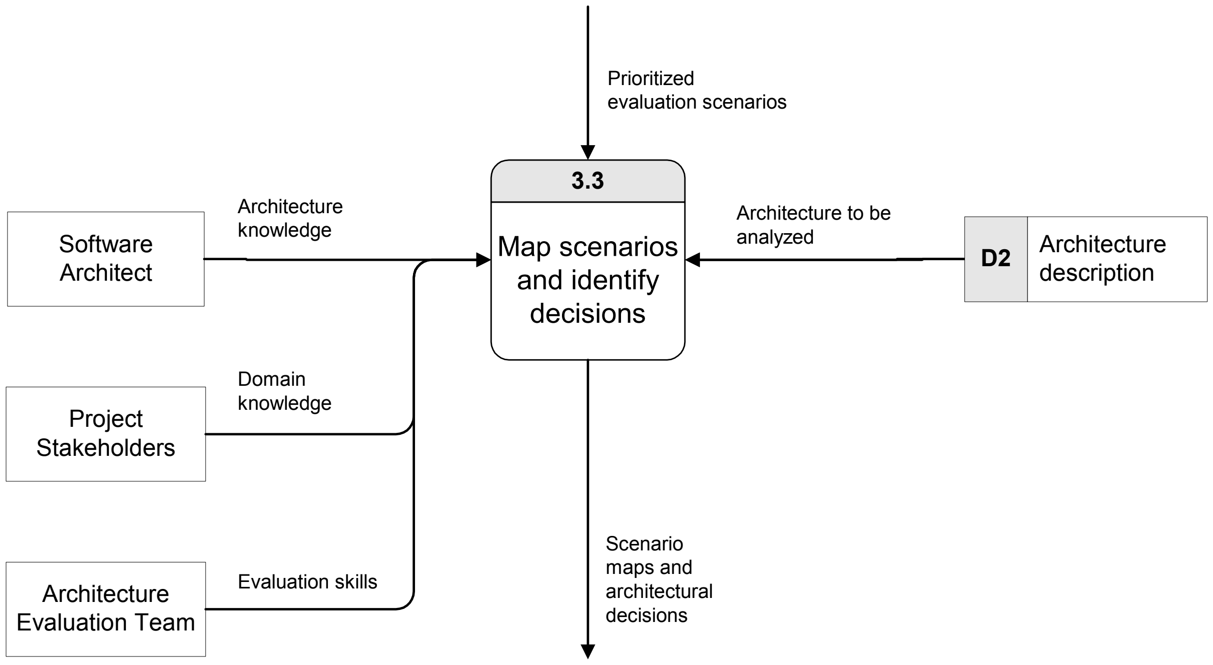

Prioritized evaluation scenarios
3.3
Map scenarios and identify decisions
Software Architect
Architecture knowledge
Architecture to be analyzed
D2
Architecture description
Project Stakeholders
Domain knowledge
Architecture Evaluation Team
Evaluation skills
Scenario maps and architectural decisions

Figure 6-6: Mapping Scenarios and Identifying Decisions (Level-2 Diagram)

2. Identify contributing design elements and capture decisions

The task of this step is to identify those design elements that are important for achieving the scenario. Achieving the scenario means that the architecture is capable of providing the required response for the stimulus as defined in the scenario description (cf. Table 6-3). For example, in the case of the following scenario:

ES1: Remote user requests a database report via the Web during peak period and receives it within five seconds

the architect would need to demonstrate that the architecture is capable of sending database reports via the Web during a peak period within five seconds. In this step the evaluation team should take care that the evaluation stakeholders are focused on getting a better understanding of the architect's solution, rather than discussing how good the chosen solution is.

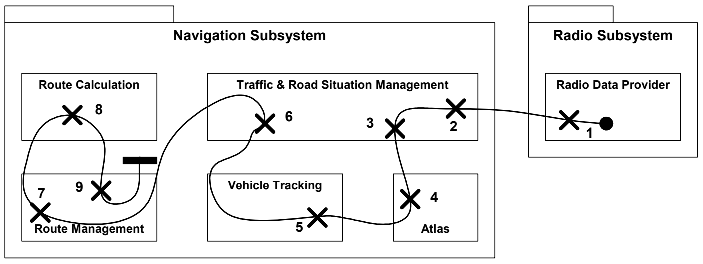
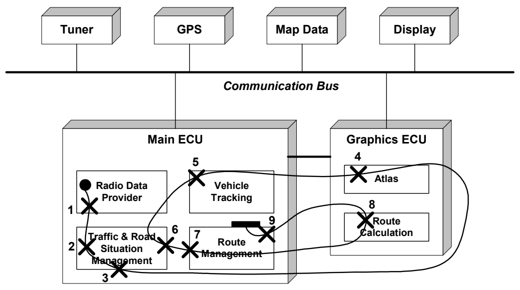
In order to support focusing on the current solution, we suggest that the evaluation team starts asking the software architect which approaches he or she has been taken or which design elements have been incorporated to address the scenario. Usually, the architect then starts explaining the solution he or she has introduced in the architecture. It is worthwhile capturing and visualizing the design elements that participate in the solution. We recommend the use of scenario maps (cf. Section 2.3.3.2) for this purpose. Note that the creation of scenario maps is supported by tools such as the Use Case Navigator (UCM) [Miga 1998]. The UCM, for example, can be applied during evaluation to facilitate a "realtime" visualization of the scenario maps.

For each design element identified (and captured in the scenario map), the major responsibilities are documented. The responsibilities describe the behavior of the design elements in the scenario map. A responsibility can be expressed in textual form or in terms of an interface specification. The important point here is that the description should help to clarify the role of the element in context of the given scenario.

To improve the understanding of how the scenario under consideration is realized in the architecture, the associated "execution path" through the design elements that are involved in or affected by the scenario are visualized in the scenario map. An execution path shows in which sequence the design elements are triggered (cf. Section 2.3.3.2). Note that there can be more than one execution path within a scenario map since particular activities of the design elements might be performed in parallel.

We also suggest recording the architectural decisions which are not obvious and which are not explicitly shown and described in the scenario map. In order to improve the description accuracy of the architectural solution of a particular scenario we further suggest capturing scenario maps for any particular architecture view that is relevant for the scenario. In many cases, it is beneficial to record more than one single view of the solution in order to identify "hidden" risks. Table 6-9 shows a template for documenting the architectural solution of an evaluation scenario.

Table 6-9: Template for Documenting the Achievement of an Evaluation Scenario

Scenario*	< Evaluation scenario# >
Quality Attributes	< Quality attributes addressed by the scenario >
Scenario Maps	< Scenario map represented in architectural view 1 >  Navigation Subsystem Radio Subsystem Route Calculation Traffic & Road Situation Management Radio Data Provider Route Management Vehicle Tracking Atlas 1 2 3 4 5 6 7 8 9
	< Scenario map represented in architectural view 2 >  Tuner GPS Map Data Display Communication Bus Main ECU Graphics ECU Radio Data Provider Vehicle Tracking Traffic & Road Situation Management Route Management Atlas Route Calculation 1 2 3 4 5 6 7 8 9
	…
Description	< Responsibility description of how the design elements behave >
Architectural Decisions*	AD1: < Description of architectural decision 1>
	AD2: < Description of architectural decision 2>
	…

Legend: * = Unique identifier recommended for better reference.

Note that the architect normally can describe very well the solution he or she has built into the architecture to address a particular scenario, no matter if it partially or completely solves the problem associated with the scenario. However, there will be cases in which a scenario has thus far been totally ignored in the architecture design. For example, it is likely that an anticipated change scenario (e.g., to support an easy integration of a particular COTS component) will not have been addressed by the architect. In this case creating a scenario map might not be possible. We suggest eliciting the changes the architect needs to introduce for achieving the scenario in the given architecture. The list of changes can then be reconsidered during risk analysis of the scenario.

Example:

An example of a scenario map for the Vehicle Navigation Subsystem (VNS) of the EVCS is shown in Figure 6-7. In this example, the high priority scenario:

ES3: Recalculate route within two seconds after an incoming radio data message has been received.

which affects performance concerns (cf. Table 6-5) is mapped on the architecture's logical software structure (subsystem-and-component view model). As illustrated in Figure 6-7, five components of the *Navigation* and one component of the *Radio* subsystem are involved in this scenario. A cross represents particular design element responsibilities that are affected by the scenario.

Figure 6-8 shows the same scenario mapped on the physical structure of the architecture. As illustrated, the software components reside on two different hardware nodes: a *Main ECU* (electronic control unit) and a *Graphics ECU*. The components *Atlas* and *Vehicle Tracking* as well as *Route Calculation* and *Route Management* communicate via remote procedure calls since they operate on different ECUs. The communication between the software components and the connected devices (*Tuner*, *GPS*, *Map Data*, and *Display*) is not shown for simplicity.

Table 6-10 summarizes the responsibilities of the respective components as well as five additional decisions that correspond to the design elements represented in the scenario map.

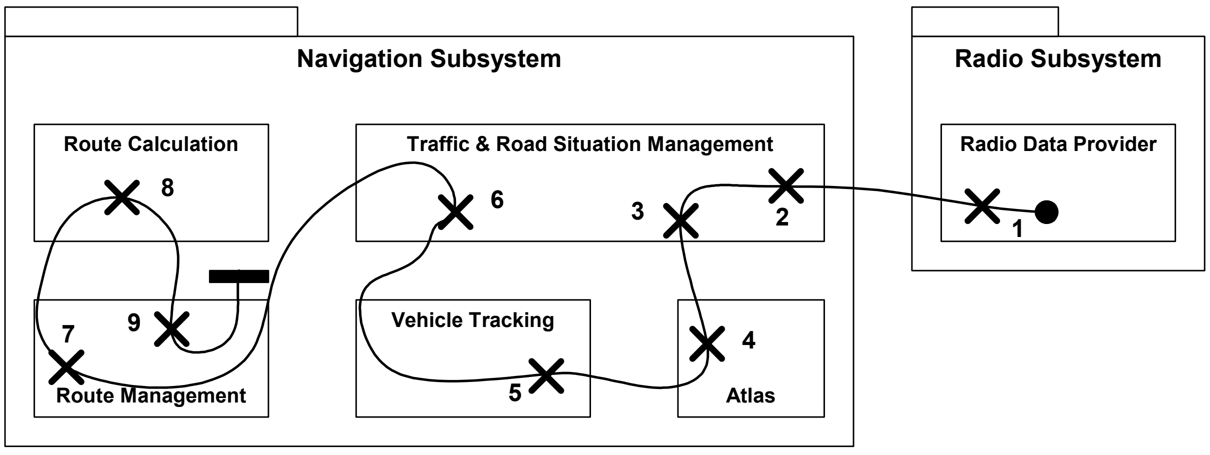


Figure 6-7: Scenario Map for Scenario ES3 (Software View)

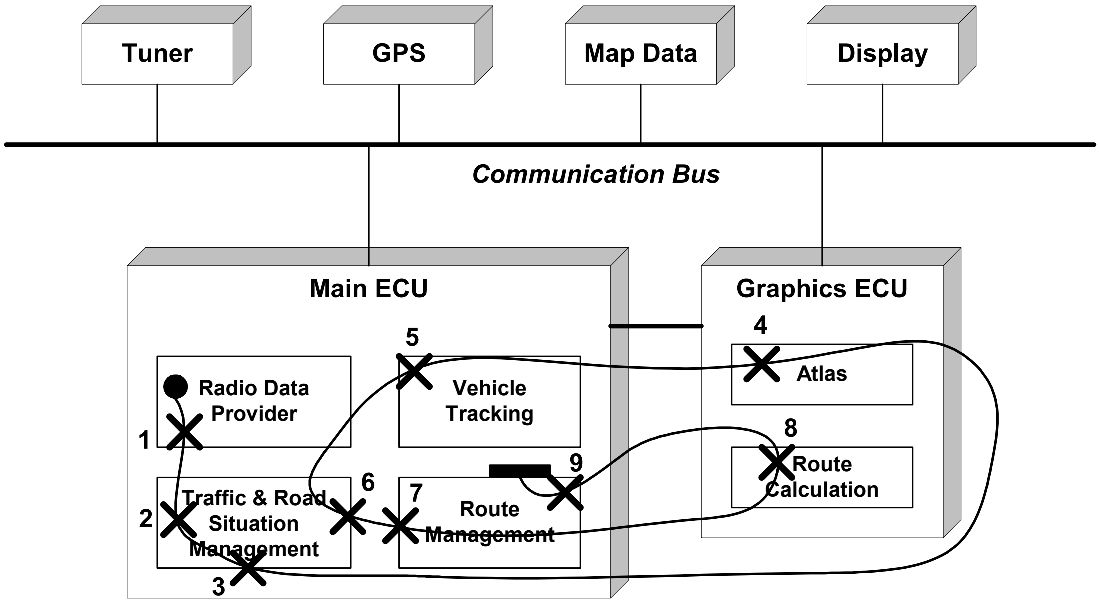


Figure 6-8: Scenario Map for Scenario ES3 (Physical View)

Table 6-10: Design Element Responsibilities and Decisions for Scenario ES3

Description		
Design Element	*R#*	*Responsibility*
Radio Data Provider	1	Receive radio data message
TRS Management	2 3 6	Decode and convert data into internal format Accumulate message and replace outdated data Check data filtering
Atlas	4	Update street elements
Vehicle Tracking	5	Report position
Route Management	7 9	Check if recalculation is necessary Notify interested parties (subscription mechanism)
Route Calculation	8	Recalculate route
Architectural Decisions	AD1:	Software components that are concerned with navigation functions are encapsulated in a Navigation subsystem.
	AD2:	The Radio Data Provider component is part of a separate Radio subsystem.
	AD3:	Vehicle Tracking is executed in an exclusive task (not shown in scenario map).
	AD4:	Atlas and Route Calculation are assigned to an extra Graphics ECU for performance reasons.
	AD5:	Main ECU and Graphics ECU are connected via a high speed communication bus.

6.5 Analyze Decisions and Summarize Findings

The objective of this activity is to analyze the appropriateness of the architectural decisions made for the high priority scenarios under consideration and to propose architectural improvements. For each scenario map it has to be evaluated if and to what extent the responsibilities of the design elements contribute to the achievement of a scenario. This involves an explicit identification of architecturally important decisions that have not been made yet. The findings are organized according to similar classes of problems in order to support the planning of adequate risk mitigation strategies. An overview of the input and output data flows is shown in Figure 6-9.

Figure 6-10 shows how this activity is decomposed into the following four subordinate activities:

1. Analyze scenario maps
2. Refine scenario analysis
3. Identify solution approaches
4. Cluster findings

These activities are next described in more detail.

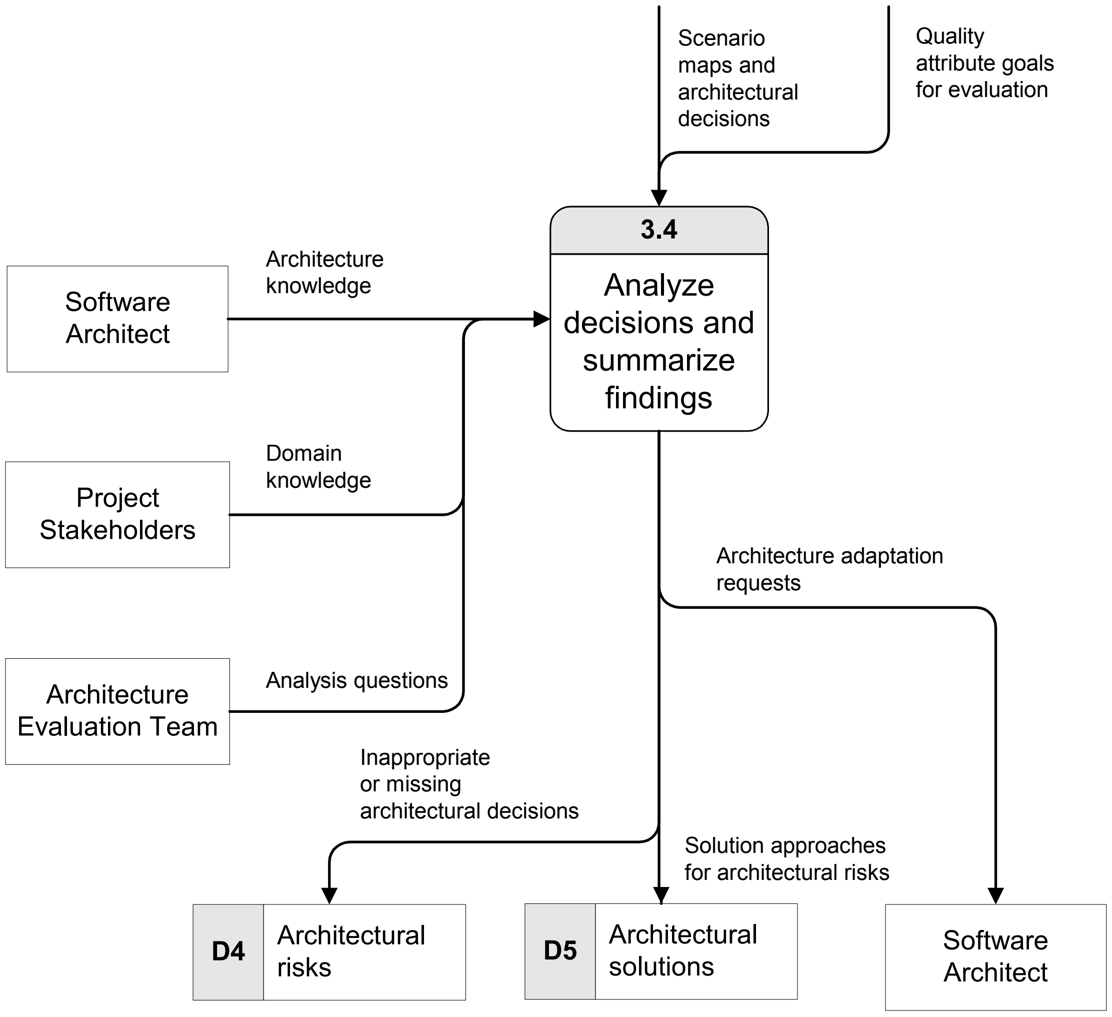

Scenario maps and architectural decisions
Quality attribute goals for evaluation
3.4
Analyze decisions and summarize findings
Software Architect
Architecture knowledge
Project Stakeholders
Domain knowledge
Architecture Evaluation Team
Analysis questions
Architecture adaptation requests
Inappropriate or missing architectural decisions
Solution approaches for architectural risks
D4 Architectural risks
D5 Architectural solutions
Software Architect

Figure 6-9: Analyzing Decisions and Summarizing Findings (Level-2 Diagram)

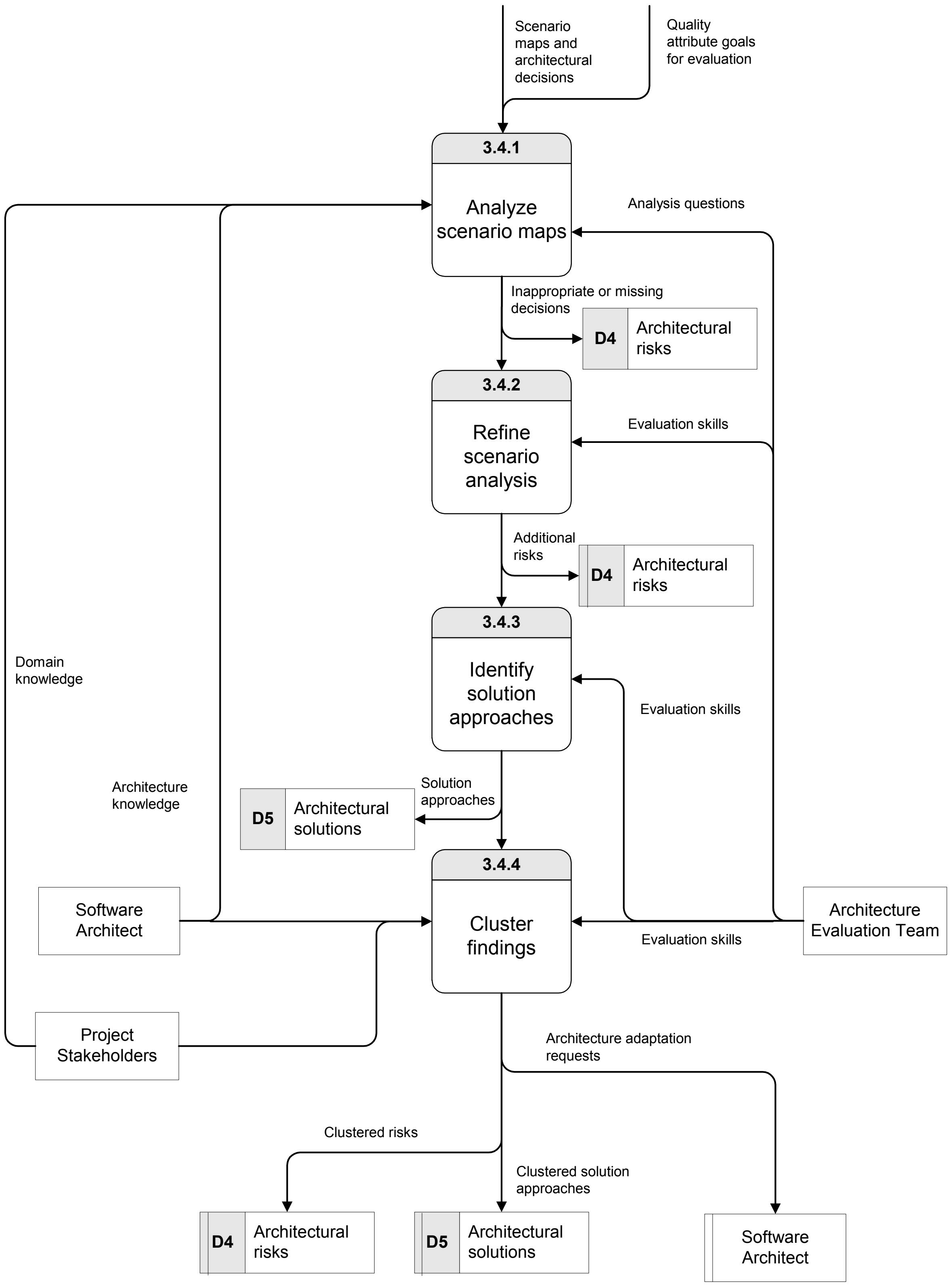


Figure 6-10: Analyzing Decisions and Summarizing Findings (Level-3 Diagram)

6.5.1 Analyze Scenario Maps

The goal of this activity is to evaluate if the architectural solution expressed in the scenario map is adequate for achieving the required response of the scenario. The activity consists of two steps:

1. Evaluate the design elements of the scenario map
2. Document findings for each scenario map

1. Evaluate the design elements of the scenario map

For each scenario map under consideration, it is evaluated if and to what extent the responsibilities of the identified design elements contribute to the achievement of the scenario. In order to support the evaluation process, we suggest applying questioning techniques (cf. Section 2.3.3.3). The use of architecture analysis questions is an appropriate vehicle in this respect. There are two types of analysis questions that can be applied to evaluate an architecture: general and quality-specific analysis questions.

- *General analysis questions:* General analysis questions are universally applicable for architecture evaluation. The questions consider the way the architecture was created and documented. They also focus on the details of the architecture description itself (by asking, for example, if all user interface aspects of the system are separated from functional aspects). Usually, the evaluation team is looking for a favorable architecture response and will probe a single question to a level of detail that is necessary to satisfy their concern. General analysis questions are a vehicle to get a basic overview on important properties of the architecture. They usually apply to any quality attribute. A set of general questions that have been proven useful for architecture evaluation are given in Table 6-11.
- *Quality-specific analysis questions:* In contrast to general analysis questions, quality-specific analysis questions address specific aspects of quality attribute requirements (e.g., security, performance, modifiability, safety etc.) and support a quality-based assessment of the architecture. The questions are used to stress a particular (part of the) architecture since they trigger the investigation as to whether and to what extent the aspects addressed can currently be accomplished. Table 6-12, Table 6-13, Table 6-14, and Table 6-15 show useful questions for analyzing the achievement of security, performance, modifiability, safety and reliability aspects of an architecture.

For a proper evaluation, the analysis questions of scope (i.e., those that address the evaluation goals) must be applied to the scenario maps. Since these questions stress specific aspects of a quality, they can be used to investigate whether and to what extent the responsibilities of the design elements included in the scenario map give a solution to those aspects. The risks discovered during this process should then be recorded in the list of findings (see next step).

Table 6-11: General Analysis Questions

General Analysis Questions
• Which are the important services of the system (i.e., services that are essential for some quality attribute requirement)?
• What are the consequences to an important service from the low quality of a non-important service? (For example, personal email might not be important except that it might be assumed to exist in an important service.)
• Are there important services that depend on a less important service?
• What services can operate in degraded modes?
• What are these degraded modes (e.g., X% speed, online upgrade)?
• What are the conditions or events that might lead to a service degradation (e.g., message sent at the wrong time, incorrect operator action, supplier going out of business)?
• What are the consequences of not meeting the quality requirements in various degraded modes (e.g., catastrophe, annoyance, minor inconvenience)?

Table 6-12: Questions for Analyzing Security Aspects

Security-Related Questions
• How is user authentication and authorization information maintained?
• How is access from outside the network managed?
• How is sensitive information protected (e.g., encryption)?
• What approach is used to protect that information (e.g., type of encryption)?
• Which user actions are logged?
• What kind of monitoring and access controls exists at the network boundaries?
• What kind of data is permitted through or filtered out?

Table 6-13: Questions for Analyzing Performance Aspects

Performance-Related Questions
• How do the time-critical components communicate (e.g., protocols, #messages sent/received per time unit, message size, processing per message)?   • Are there sources of potential resource contention (e.g., specific devices, CPU cycles, network bandwidth, and memory usage) and which kinds of mechanisms are used to prevent them?   • How does a particular component respond during periods of high stress? Does an overload affect the data flowing through a particular communication link?   • How do services respond to clients and can they handle multiple clients?

Table 6-14: Questions for Analyzing Modifiability Aspects

Modifiability-Related Questions
• What strategy is used to upgrade versions of systems containing multiple components released independently?   • What are the limitations on upgrades or modifications? Can it be done online (e.g., during execution) or must it be done offline?   • What are the conditions or events outside our control that might lead to a limitation on upgrades or modification (e.g., changes in standards, suppliers merging, supplier going out of business)?

Table 6-15: Questions for Analyzing Safety and Reliability Aspects

Safety/Reliability-Related Questions
• What type of faults can occur in each component or connection involved in a service?   • What services cannot be allowed to fail (e.g., authentication services)?   • What kind of strategies are used to detect software/hardware failures?   • When is a particular failure detected by the system (e.g., during start-up or during shut down)?   • What mechanisms are used to recover from failures?

2. Document findings for each scenario map

The goal of this step is to document the findings and open issues that result from analyzing a scenario. Findings represent architectural risks that are associated with the current implementation of the scenario in the architecture. There is a risk in the architecture if the response defined in the scenario description cannot be (completely) achieved by the design elements of an associated scenario map. For example, consider scenario ES5 of Table 6-5:

ES5: The caching system will be switched to another processor when its processor fails, and will do so within one second.

There would be a risk in the architecture if the analysis of the scenario map were to uncover that there is no second processor which can be used for switching the caching system. There would also be a risk if a backup processor were available but the architecture did not consider using it when there was a problem with the caching system or if failures of processor or caching system were not recognized.

We suggest documenting the affected quality attributes of each identified risk and, if possible, the design element, which represents the root cause for that risk. In addition, we suggest capturing any open issue that emerged during analysis of a particular scenario and that could not be resolved by the audience. This is worthwhile because then these issues can be considered during subsequent design iterations. Moreover, open issues often turn out to include hidden risks of the architecture. Thus, it is important to capture them for further treatment. We also suggest adding roles or names of individuals that should check the issues after the evaluation. Table 6-17 shows a template for documenting findings and open issues of a scenario analysis.

Table 6-16: Template for Documenting Findings and Open Issues

Scenario	< Evaluation scenario# >
Quality Attributes	< Quality attributes addressed by the scenario >
Findings*	Risk 1: < Description of architectural risks 1, affected qualities, root cause design element >
	Risk 2: < Description of architectural risks 2, affected qualities, root cause design element >
	…
Open Issues*	O1: < Open Issue 1, roles/individuals who should check the issue >
	O2: < Open Issue 2, roles/individuals who should check the issue >
	…

Legend: * = Unique identifier recommended for better reference.

Example:

Two risks and two open issues for the scenario map of Figure 6-7 are shown in the findings list of Table 6-17. Risk 1 is concerned with the fact that too many processes are created when multiple routes are active in the system. The risk affects performance and modifiability properties of the system. The current specification of the *Route Management* and *Atlas* components represent the root cause for that risk. High memory consumption of map data is the reason for risk 2. This affects performance and efficiency properties of the system. The *Atlas* component again represents the root cause for risk 2. The two open issues O1 and O2 are concerned with missing information about the maximum number of parallel processes and the total memory consumption of the software. The information will be later provided by the software architect.

Table 6-17: Documentation of Findings Identified during Architectural Analysis

Findings			
F#	**Description of Findings**	**Quality Attribute**	**Root Cause Element**
Risk 1	*Too many processes spawned for routes:* The Route Management component must be re-entrant because there may be multiple routes active simultaneously. This may introduce a dependency on the given operating system since there are only a limited number of processes available with one address space per process.	Performance, modifiability	Route Management, Atlas
Risk 2	*High memory consumption of map data:* The current map data from the Atlas is stored in RAM. System RAM is limited. Thus, sufficient free memory may not be available for storing all the map data of the current position.	Performance, memory efficiency	Atlas

Open Issues		
O#	**Description of Issue**	**To be checked by**
O1	The maximum number of parallel processes that can be handled by the system is unknown.	Software architect
O2	The total memory consumption of the software is not known yet.	Software architect

6.5.2 Refine Scenario Analysis

The goal of this activity is to refine the analysis in order to estimate the average execution time of *performance* scenarios. Estimating execution times is especially important if the architecture has to meet certain timing constraints. Timing constraints are usually expressed by response measures within the scenario description (cf. Section 6.3). Table 6-5, for example, includes the following performance-related scenarios that include execution time response measures:

ES1: Remote user requests a database report via the Web during peak period and receives it within ***five seconds****.*

ES2: The user changes the graph layout from horizontal to vertical. The graph is redrawn in ***one second****.*

ES3: Recalculate route within ***two seconds*** *after an incoming radio data message has been received.*

ES9: Double the size of existing database tables while preserving an average retrieval time of ***one second****.*

ES10: Add a new data server to reduce latency to ***2.5 seconds*** *within one person-week.*

ES13: The ***time*** *for displaying changed route data* ***is reduced by a factor of 10****.*

In order to calculate the average execution time of a scenario, environmental data from the scenario must be captured. Table 6-18 shows data items we suggest be gathered in order to perform an estimation.

It is worthwhile to mention that the purpose of this activity is not to make precise statements about the execution time but to achieve a rough estimate. The objective is to come to some first numbers without going beyond the scope of the architecture evaluation. The estimation should be further refined within design and code reviews.

There are four steps that support calculating the execution time of a scenario:

1. Capture execution structure and environment
2. Determine resource usage
3. Estimate average execution time
4. Document additional findings

Table 6-18: Environmental Data for Estimating Execution Times

Data Source	Data Description
Execution structure	• The software components that execute • The order of execution • Component repetition and conditional execution
Execution environment	• The hardware configuration upon which the system will execute • The abstract machine (i.e., the operating system and other support software routines that interface between the new software and the hardware)
Resource usage	• The number of instructions to be executed • The number of I/O operations for each device • The number of types of abstract machine service routines used • The amount of memory for code and for data

1. *Capture execution structure and environment*

The goal of this step is to capture the execution structure and execution environment for the respective performance scenarios. A starting point for specifying the execution structure and environment is the description of the scenario map (cf. Section 6.4). As discussed earlier, a scenario map shows the design elements in a particular architectural view that execute in order to achieve the response of the underlying scenario description. The execution path illustrates the order in which the design elements are triggered, if there are repetitions, and if there are parallel paths of execution. Consequently, a scenario map and its associated description support capturing the execution structure of the scenario in this activity. However, quantitative data concerning the hardware configuration and software environment upon which the scenario executes is often not part the scenario description. In this step, this kind of information must be captured. We suggest eliciting the following fundamental data:

- CPU speed in Million instructions per second (MIPS) – $p_{CPU} = \frac{1}{n_{INSTR\_CPU}}$
- Time to access, read from, and write to I/O devices – $t_{IO,access}$, $t_{IO,read}$, $t_{IO,write}$
- Time for switching between tasks/processes – t_{TASK}
- Time to make a remote procedure call (RPC) – t_{RPC}
- Compiler overhead (Ratio of high-level to machine instructions) – c

2. Determine resource usage

The goal of this step is to determine the resource usage for the scenarios. The resource usage can be estimated based on the functions that are triggered in the design elements by the scenarios under considerations. For this purpose, we have to get an understanding about how much time is spent for each function triggered. We suggest concentrating on gathering the following relevant data for calculating the execution time:

- Number of high-level instructions to be executed for each function – n_{INSTR_H}
- Number of I/O operations for each device used – $n_{IO,access}$, $n_{IO,read}$, $n_{IO,write}$
- Number of remote procedure calls made by the functions – n_{RPC}
- Type of CPU used for executing the functions (if deployed on multiple CPUs)

Determining the resource usage is an approximation process, especially in an early phase of architecture development. However, in many cases parts of the architecture are coded for testing purposes. The compiled code can be used as basis for extrapolating the resource usage of the functions triggered in the scenario maps. Table 6-19 shows a template for capturing the resource usage of a scenario.

Table 6-19: Template for Describing the Resource Usage of an Evaluation Scenario

Scenario	< Evaluation scenario# >				
Quality Attributes	< Quality attributes addressed by the scenario >				
Functions		**High-Level Instructions**	**#I/O**	**#RPC**	**CPU**
1	< Function 1 >	< High-level instruction for function 1 >	< Number of I/O access in function 1 >	< Number of RPC access in function 1 >	< CPU used for executing function 1 >
3	…	…	…	…	…

3. Estimate execution time

The goal of this step is to estimate the average execution time for the scenarios. For each function that is triggered in the scenario the execution time is calculated based on the hardware/software environment and resource usage recorded in the previous steps.

The execution time for each function triggered within a scenario can be estimated as follows:

$T(F) = T_{IO}(F) + T_{RPC}(F) + T_{INSTR}(F)$: Execution time for each scenario function

where

$T_{IO,access}(F) = n_{IO,access} \cdot t_{IO,access}$: Time for device access
$T_{IO,read}(F) = n_{IO,read} \cdot t_{IO,read}$: Time for reading from a device
$T_{IO,write}(F) = n_{IO,write} \cdot t_{IO,write}$: Time for writing to a device
$T_{IO}(F) = T_{IO,access}(F) + T_{IO,read}(F) + T_{IO,write}(F)$: Total time for device I/O
$T_{RPC}(F) = n_{RPC} \cdot t_{RPC}$: Time for making remote procedure calls
$T_{INSTR}(F) = n_{INSTR\_M} \cdot p_{CPU}$: Time for executing the machine instructions
$n_{INSTR\_M} = c \cdot n_{INSTR\_H}$: Number of machine instructions executed

The execution time of a scenario can then be calculated as

$T_{SCENARIO} = \sum_{i=1}^{m} T(F_i)$: Execution time for a scenario

with

m : Number of functions triggered by the scenario

Note that $T_{SCENARIO}$ represents a rough estimation of a scenario's execution time. Repetitions of functions and parallel executing functions are calculated in a similar way. In the first case, the time for repeating function is accumulated, in the latter case only the thread with the longest execution time is included in $T_{SCENARIO}$.

In order to refine the estimation the system's workload can be considered. Workload-related information includes:

- Number of average (minimum/maximum) users of the system
- Number of system requests that the system has to process per minute/hour?
- Rate at which functions are requested
- Special patterns of requests

However, workload specification and its consideration in execution time estimation are beyond the scope of this work. Advanced performance measurement techniques can be obtained from the performance engineering community (e.g., [Smith 2002]). These techniques include shortest and longest time computation, conditional computations, and queuing models.

4. Document additional findings

The goal of this step is to document additional findings that result from estimating the average execution time of a scenario. It is worthwhile to analyze how well the response measure of the scenario could be achieved by the architecture. In this respect, it is important to capture if the estimated response is only slightly higher than the required response or if it is higher by orders of magnitude. This information is especially useful in order to plan the effort for risk mitigation strategies such as restructuring the architecture.

Example:

In this example an estimation of the average execution time for the scenario ES3 of the VNS is illustrated:

ES3: Recalculate route within two seconds after an incoming radio data message has been received.

The estimation is based on the scenario map introduced in the example of Section 6.5.1. Table 6-20 shows the environmental data gathered for the scenario. Table 6-21 summarizes the resource usage of ES3 in terms of high-level instructions, number of input/output-operations, number of RPCs, and CPU types used.

Based on this data, the average execution time for scenario ES3 can be estimated according to the equations given above, as shown in Table 6-22. The resulting execution time is 1.157 seconds, which is below the required response measure of 2 seconds defined in the scenario description. Consequently, there is no risk added to the list of findings for ES3.

Table 6-20: Environmental Data for the VNS Example

Environmental Data	
CPU speed (Main ECU, MECU)	10 MIPS
CPU speed (Graphics ECU, GECU)	50 MIPS
I/O devices	50ms per request
Remote procedure call overhead	10ms per call/return
Ratio high-level to machine instructions	1:20

Table 6-21: Resource Usage of the Scenario ES3

Functions		High-Level Instructions	#I/O	#RPC	CPU
1	Receive radio data message	1.000	2 (Tuner)	--	MECU
2	Decode and convert data into internal format	4.000	--	--	MECU
3	Accumulate message and replace outdated data	2.500	--	1 (call)	MECU
4	Update street elements	2.500	5 (Map Data)	--	GECU
5	Report position	2.000	2 (GPS) 1 (Display)	1 (return)	MECU
6	Check data filtering	1.000	--	--	MECU
7	Check if recalculation is necessary	250	--	1 (call)	MECU
8	Recalculate route	20.000	8 (Map Data) 2 (Display)	--	GECU
9	Notify interested parties	250	1 (Display)	1 (return)	MECU

Table 6-22: Execution Time of the Scenario ES3

Functions		Machine Instructions n_{INSTR_M}	Time for Instructions $T_{INSTR}(F)$ [ms]	Time for I/O $T_{IO}(F)$ [ms]	Time for RPC $T_{RPC}(F)$ [ms]
1	Receive radio data message	20.000	2	100	--
2	Decode and convert data into internal format	80.000	8	--	--
3	Accumulate message and replace outdated data	50.000	5	--	10
4	Update street elements	50.000	5	250	--
5	Report position	40.000	4	150	10
6	Check data filtering	20.000	2	--	--
7	Check if recalculation is necessary	5.000	0.5	--	10
8	Recalculate route	400.000	40	500	--
9	Notify interested parties	5.000	0.5	50	10
Intermediate Total [ms]			67	1050	40
Scenario Execution Time $T_{SCENARIO}$ [s]					**1.157**

6.5.3 Identify Solution Approaches

The goal of this activity is to identify and describe solution approaches for the architectural risks captured during the scenario map analysis (cf. Section 6.5.1 and 6.5.2). According to our experience it is very likely that during analysis of a scenario will be possible not only to identify risks associated with the current implementation but also to discover how to cope with those risks. This knowledge is very valuable for driving the revision of the architecture and should not be lost during the evaluation.

Therefore, we suggest reserving some time for identifying solution approaches for the identified risks of each scenario. The important thing here is *not* to focus on solutions before the risks have been documented and completely understood by the audience. This is why we propose to perform the activity *after* the risks of a particular scenario have been identified. Only once the risks are known and understood brainstorming for solutions does make sense. However, finding a solution to each risk should not be strongly forced within this activity. Instead, the solutions that people had in mind during scenario analysis should be recorded here concisely.

This activity consists of two steps, which are performed in concert:

1. Document basic solution approach for a risk
2. Capture benefits and drawbacks of the solution

1. Document basic solution approach for a risk

The goal of this step is to give a short outline of a potential solution for the risks identified for a particular scenario during analysis. It is important to note that no detailed solution description (e.g., no programming code) should be given here but rather that the basic idea of the solution approach should be captured. The documentation should be concise enough such that it is possible to investigate the approach in depth after the architecture evaluation. As a matter of course, more that one solution alternative can be captured for a risk.

2. Capture benefits and drawbacks of the solution

The goal of this step is to describe the benefits and potential drawbacks of each solution briefly. Since each solution will probably have a tradeoff (e.g., undesired effect on the system) it is important to describe this tradeoff here as a drawback in order to guide the selection of an optimal solution. It is essential to get evidence that a particular solution does focus on addressing a risk without degrading the overall system quality and without inducing new risks. It is also worthwhile to form a rough estimate of the effort and cost involved in changing the architecture to incorporate the solution. Table 6-23 shows a template for documenting solution approaches for identified risks.

Table 6-23: Template for Documenting Solution Approaches

Scenario	< Evaluation scenario# >
Quality Attributes	< Quality attributes addressed by the scenario >
Solution Approaches*	SA1: < Description of solution alternative 1, benefits/drawbacks, links to risks >
	SA2: < Description of solution alternative 2, benefits/drawbacks, links to risks >
	…

Legend: * = Unique identifier recommended for better reference.

Example:

Table 6-24 shows a solution approach for addressing risk 2 of the VNS example given in Section 6.5.1:

Risk 2: High memory consumption of map data. The current map data from the Atlas is stored in RAM. System RAM is limited. Thus, sufficient free memory may not be available for storing all the map data of the current position.

Table 6-24: Solution Approaches for Risk 2 of the VNS Example

Solution Approaches		
SA#	**Description of Solution Approaches**	**Addressed Risk**
SA1	Only the vector information about the current situation is stored in main memory. A bitmap of the map is generated and presented by the system on demand. • *Benefits:* Memory consumption for storing map data is significantly reduced; effort for implementing solution is low (less than two person days) • *Drawback:* Responsiveness of displaying maps to user is decreased	Risk 2

6.5.4 Cluster Findings

The goal of this activity is to organize the findings according to specific clustering criteria. This is especially helpful if the list of risks is very long. The clustering criteria help to organize the findings into a few major risk themes. The risk themes represent abstractions for risks that fall into similar problem areas.

The activity consists of two steps:

1. Identify clustering criteria
2. Organize findings and solution approaches to clusters

1. *Identify clustering criteria*

The goal of this step is to identify criteria that can be used to cluster the identified risks in similar problem areas. In order to find adequate criteria we suggest starting to organize the risks according to similar system aspects or functions. The clustering should provide the opportunity to understand to which extent the architecture achieves the evaluation goals stated in Section 6.2. In other words, the clustering should illustrate in which system aspects the architecture must be improved.

If the risks cannot be further subsumed by system aspects we suggest organizing the remaining risks with respect to similar quality attributes. Usually, the latter provides the opportunity to bring together those risks that primarily have an effect on modifiability issues, performance issues, safety issues etc. of the system.

2. *Organize findings and solution approaches to clusters*

The goal of this step is to organize the findings (and solution approaches) to the clusters. For each cluster identified, the set of associated risks and solution approaches are recorded as a reference. The risk clusters then represent the major problem areas of the architecture. The problem areas must be improved during subsequent design iterations (e.g., by performing QUADRAD-M activities). Table 6-25 shows a template for documenting risk clusters.

Note that the previous activity “Identify solution approaches” (cf. Section 6.5.3) could also be applied after the risks have been clustered. The benefit would be to understand better how the risks relate to each other and what overall system implications they have. The drawback is that the clustering can only be done *after* the scenarios have been analyzed. It is very likely that many of the solution approaches for the risks cannot be recalled anymore by the audience. Many approaches might get lost. Thus, we would not recommend performing the “Identify solution approaches” activity after the risks have been clustered but *before* clustering. We further suggest checking the solution approaches that apply to similar classes of risks. The opportunity here is to extend the solutions to broader solution patterns that can be applied to address a complete risk cluster.

Table 6-25: Template for Clustering Risks

RC#	**Risk Clusters**	**Associated Findings**	**Associated Solution Approaches**
RC 1	< Description of problem area which is similar for the associated findings >	< List of risks that match the problem area of the risk cluster RC 1>	< Solution approaches of the risks in that cluster >
…	…	…	…

Example:

Table 6-26 shows six additional risks identified during the architecture evaluation of the VNS example. Table 6-27 depicts an organization of these risks in two risks clusters. For risk cluster RC1 the risks 4, 7, 8, and 10 have been organized according to the system aspect "software certification for 3rd party components". The risks 5 and 12 have been summarized according to the quality attributes safety and reliability in risk cluster RC2.

Table 6-26: List of Findings for the VNS Evaluation

Findings		
F#	**Description of Findings**	**Quality Attributes**
Risk 4	*Tracing errors not possible:* The architecture contains no explicit mechanism to locate errors in the system. This means that integration of COTS introduces a risk since failures that result from COTS malfunctions cannot be traced to its origin.	Safety, reliability
Risk 5	*No adequate hardware monitoring:* The system does not include adequate components for monitoring hardware defects.	Safety, reliability
Risk 7	*Strong dependency on Operating System:* The customer for the system requires Windows CE. If another customer requires a different Operating System then the software will need to be reworked completely.	Portability
Risk 8	*No rules for defensive programming:* No rules for defensive programming of the components that must interact with third party components are specified. Techniques such as pre-conditions, assertions, and contracts are not used to help isolate errors and keep them from propagating.	Safety, reliability
Risk 10	*No certification process:* There is a strong dependency on the certification process of COTS for security reasons. An adequate process that guarantees security has not been defined, yet.	Security
Risk 12	*No adequate component self-test:* Critical software components do not include an adequate self-test functionality.	Safety, reliability

Table 6-27: Risk Clusters

RC#	**Risk Clusters**	**Associated Findings**	**Associated Solution Approaches**
RC 1	*Certification of external software:* Risks arising from the incorporation of 3rd party software.	Risks 4, 7, 8, 10	--
RC 2	*Safety and reliability of software and hardware:* Risks arising from guaranteeing safety and reliability of software and hardware components.	Risks 5, 12	--

6.6 Evaluation Activities and Research Gaps

The provided activities of the QUADRAD Evaluation workflow represent major research contributions of this work. Table 6-28 shows how these activities support bridging the research gap G3 described in Chapter 1. In addition, the table shows which of the framework requirements defined in Table 3-1 are achieved by those activities.

Table 6-28: Mapping Evaluation Activities to Research Gaps and Requirements

Evaluation Workflow Activities	Research Gaps	Framework Requirements
Define evaluation goals	G3	FR6
Elicit and prioritize evaluation scenarios	G3	FR5, FR6
Map scenarios and identify decisions	G3	FR6, FR7
Analyze decisions and summarize findings	G3	FR6, FR7, FR8

Contributions of the QUADRAD Evaluation workflow with respect to bridging the research gaps:

Research Gap 3: Insufficient support for the identification and mitigation of unwanted effects of design decisions in architecture development.

The QUADRAD Evaluation workflow allows for an improved support for a scenario-based architecture evaluation. It uses different types of scenarios (usage, change, stress, and configuration scenarios) to support the elicitation of significant requirements for evaluation. The scenario types give hints during elicitation as to what kind of aspects (e.g., user interaction with the running system, anticipated future system changes, possibility to derive a product with particular capabilities etc.) should be addressed with respect to functional or non-functional requirements.

In particular, the "Define evaluation goals" activity (cf. Section 6.2) allows defining the specific goals of the evaluation and documenting the rationales for those goals. An evaluation can thus be adapted to a particular quality attribute ("restricted evaluation") or it can be optimized to multiple attributes – i.e., the set of architecturally driving requirements ("overall evaluation"). The benefit here is that one can individually adapt the evaluation according to available resources, time, or selected quality aspects of the current architecture design.

The "Elicit and prioritize evaluation scenarios" activity (cf. Section 6.3) allows to select those scenarios that are important for the quality of the architecture. These scenarios also most likely contribute to the achievement of the business goals. An explicit ranking of scenarios additionally supports the selection process. It considers criteria which allow judging a scenario as relevant or irrelevant. Thus, an evaluation is not started with an arbitrary scenario but with those scenarios that have a high priority for the success of the system.

The "Elicit and prioritize evaluation scenarios" activity also supports different weighting schemas to calculate high priority scenarios. Besides the benefit of explicitly judging the relevance of scenarios, different approaches for the decision making process are proposed.

The selection of a decision-making method (e.g., absolute weighting or comparative weighting) can thus be adapted according to the evaluation goals and the available stakeholders.

The "Map scenarios and identify decisions" activity (cf. Section 6.4) allows for the systematic identification and visualization of relevant design elements in multiple architectural views. It supports a systematic isolation of design elements that contribute to a scenario. For this purpose, analysis questions are introduced. These questions can be used to narrow or focus the scope of the evaluation by improving the precision for isolating the affected design elements of a scenario. The activity also improves the understanding of how evaluation scenarios are realized in the architecture. It proposes scenario maps that visualize the "path" through the design elements that are involved in or affected by a scenario. A scenario map shows the mapping of the scenario onto a particular architecture view, i.e., the "implementation" of the scenario. Again, this eases the identification of potentially high-risk parts in the architecture as well as the sequence of design elements that are triggered by a scenario.

The "Analyze decisions and summarize findings" activity (cf. Section 6.5) allows for an analysis of the architecture's properties. It uses attribute-specific questions such as how many users are logged in the system or how sensitive information is protected in order to stress a particular (part of the) design – e.g., the part defined by a scenario map. Attribute-specific questions support the qualitative analysis and trigger the investigation about if and to what extent the aspects addressed by the questions can currently be accomplished. They also help to identify side effects between obviously unrelated architectural decisions. The activity also guides in determining solution alternatives that focus on addressing the identified risks.

The qualitative analysis capabilities of QUADRAD-E also provides an excellent basis for further (quantitative) design investigations. Since the design elements that contribute to a scenario are already isolated in a scenario map, the gathering of more precise data about the environment as well as detailed information about the design and implementation of those elements can be supported and coordinated. In particular, the activity "Analyze decisions and summarize findings" provides steps for estimating the average execution time of performance-related scenario. This is especially useful for understanding how close the current architectural solution is to the required response time. It thus supports planning and staffing improvement efforts.

Finally, the "Analyze decisions and summarize findings" activity also includes steps for clustering findings and candidate solutions according to similar problem classes. This helps in getting a quick executive overview about the major problems in the architecture and about how these problems can be tackled systematically in subsequent design iterations.

6.7 Summary

In this chapter the activities and artifacts of the QUADRAD Evaluation workflow (QUADRAD-E) have been described. The purpose of this workflow is to understand the consequences of architectural decisions made by the architect, to analyze them with respect to the achievement of essential quality requirements, and to propose architectural improvements. In particular, activities for defining the goals of the evaluation, for documenting related architectural decisions, and for analyzing those parts of the architecture that implement the most driving requirements concerning the evaluation goals have been provided. Furthermore, steps for analyzing the appropriateness of the architectural decisions made by the architect, for determining approaches to improve the architecture as well as for organizing the findings systematically to support planning risk mitigation strategies have been described. Finally, it has been shown how the QUADRAD-E activities support bridging the research gaps of this thesis.

In the next chapter we will introduce the research tool AET (Architecture Exploration Tool) which supports essential architecture design and evaluation activities of QUADRAD.

7 QUADRAD Tool Support

This chapter provides an overview of the Architecture Exploration Tool (AET). AET is a research tool that supports essential activities of QUADRAD-E. The chapter is organized as follows: Section 7.1 briefly introduces AET and illustrates which QUADRAD activities are supported and automated by the tool. In Section 7.2, important requirements for developing AET are presented. It is also shown how the requirements are derived from QUADRAD activities. Section 7.3 provides an overview of the AET software architecture and data model. In Section 7.4, it is described how AET can be applied to support a QUADRAD-E architecture evaluation workshop. Finally, Section 7.5 summarizes the main points of this chapter.

7.1 Architecture Exploration Tool

The Architecture Exploration Tool (AET) is a research tool developed by the author and colleagues at the Software Technology department of Robert Bosch. It supports an architect or a review team in managing information of an architecture evaluation and in documenting design results. In particular, AET supports capturing requirements and refining them with the help of scenarios. It further supports the prioritization of scenarios with respect to their significance for development and allows for a description of the major design decisions made to implement each scenario. AET also supports the documentation of trace information. Especially, it allows the traceability between quality requirements, refined scenarios, design decisions that pertain to a particular scenario, and risks associated with particular design decisions recorded during an architecture evaluation. Table 7-1 shows the QUADRAD activities that are supported by AET.

Table 7-1: QUADRAD Activities Supported by AET

Workflow	Activities	AET Support
Preparation	Identify architectural drivers	Supported
	Determine architectural view types	Partially Supported
Modeling	Determine architectural strategies and mechanisms	Supported
	Model the architectural infrastructure	Partially Supported
	Refine the architecture	Partially Supported
Evaluation	Define evaluation goals	Supported
	Elicit evaluation scenarios and identify decisions	Supported
	Map scenarios onto the architecture	Supported
	Analyze the architecture	Supported

As illustrated, the activities of QUADRAD-E are completely supported by AET. However, since the tool helps in documenting major issues of an architecture, this is also beneficial for QUADRAD-P and QUADRAD-M activities.

In the following we describe important requirements that have been considered during the development of AET.

7.2 AET Requirements

The requirements for developing AET are derived from the description of the activities defined in the QUADRAD framework. As mentioned in the previous section, the first release of the AET, which is discussed in this thesis, has been primarily focused on supporting the architecture evaluation activities of QUADRAD-E.

Table 7-2 shows the most important functional requirements of AET. Each requirement is directly deduced from a particular QUADRAD activity. In particular, the requirement AET1 supports the "Define evaluation goals" activity (cf. Section 6.2). AET2 and AET3, deal with the activity "Brainstorm and organize evaluation scenarios" (cf. Section 6.3.1), AET4, AET5, AET6, and AET7 with "Calculate scenario priorities" (cf. Section 6.3.2). The activities "Map scenarios and identify decisions" (cf. Section 6.4), "Analyze scenario maps" (cf. Section 6.5.1), and "Cluster findings" (cf. Section 6.5.4) is considered by AET8, AET9, AET10, and AET11. Finally, the requirement AET12 supports the complete set of QUADRAD-E activities.

However, while the development of AET is focused on QUADRAD-E, the requirements AET1 and AET8 would also support QUADRAD-P and QUADRAD-M activities. AET1, for example, provides means for capturing quality requirements. This would also support the "Identify architectural drivers" activity defined in QUADRAD-P (cf. Section 4.2). Moreover, AET8 deals with the capability to document architectural strategies and decisions. This would clearly provide input for the "Model the architectural infrastructure" activity of QUADRAD-M (cf. Section 5.3).

Besides the functional requirements, AET has also to achieve certain quality attributes. Table 7-3 illustrates the most driving quality requirements for AET. As shown, safety, reliability, modifiability, usability, and performance are important attributes that must be considered during the development of AET. The quality requirements are not related to the QUADRAD framework since they do not directly support functional capabilities that are required to perform the QUADRAD activities.

There have also been constraints to the development which should guarantee an effective implementation of AET within a limited time frame. The constraints are primarily concerned with using COTS components (Components-off-the-Shelf). For example, Microsoft Access® and ODBC (Open Database Connectivity) are used to advance the development of AET's database management capability. In addition, the Microsoft Foundation Classes (MFC) library is used for a rapid prototyping of AET's graphical user interface.

In the following we provide an overview of the AET software architecture and data model.

Table 7-2: Functional Requirements of AET

REQ#	Requirement	QUADRAD Activity
AET1	The AET shall support capturing functional and quality requirements. The quality requirements shall be represented in a quality tree. They shall be used to define the goals of the evaluation.	QUADRAD-E, Define evaluation goals (cf. Section 6.2)
AET2	The AET shall support the description of evaluation scenarios. The description shall also make explicit the stimulus and response of the scenario.	QUADRAD-E, Brainstorm and organize evaluation scenarios (cf. Section 6.3.1)
AET3	The AET shall support linking scenarios to quality requirements such that it is clear to the user, which quality requirement is further described by a scenario.	QUADRAD-E, Brainstorm and organize evaluation scenarios (cf. Section 6.3.1)
AET4	The AET shall support the prioritization of evaluation scenarios.	QUADRAD-E, Calculate scenario priorities (cf. Section 6.3.2)
AET5	The AET shall support different criteria for prioritizing scenarios such as business importance and architectural impact.	QUADRAD-E, Calculate scenario priorities (cf. Section 6.3.2)
AET6	The AET shall support different scales for prioritizing scenarios according to the criteria (e.g., high, medium, low).	QUADRAD-E, Calculate scenario priorities (cf. Section 6.3.2)
AET7	The AET shall support sorting of scenarios according to their prioritization criteria.	QUADRAD-E, Calculate scenario priorities (cf. Section 6.3.2)
AET8	The AET shall support the description of scenarios analysis results (e.g., the design decisions made and the strategies used in the architecture to address the scenario).	QUADRAD-E, Map scenarios and identify decisions (cf. Section 6.4); Analyze scenario maps (cf. Section 6.5.1)
AET9	The AET shall support capturing findings of the scenario analysis (e.g., architectural risks).	QUADRAD-E, Analyze scenario maps (cf. Section 6.5.1)
AET10	The AET shall support the linking of findings to scenarios analyzed.	QUADRAD-E, Analyze scenario maps (cf. Section 6.5.1)
AET11	The AET shall support clustering of risks to logically coupled risk themes.	QUADRAD-E, Cluster findings (cf. Section 6.5.4)
AET12	The AET shall support the generation of a report that includes the results of the architecture evaluation.	QUADRAD-E, all activities

Table 7-3: Quality Requirements of AET

REQ#	Requirement	QUADRAD Activity
AET13	The AET shall support backups of the architecture exploration (safety, reliability).	N/A
AET14	The AET shall support recovering from data failures (safety, reliability).	N/A
AET15	The AET shall support an easy adaptation to different export formats for report generation (modifiability).	N/A
AET16	The AET shall support an easy adaptation to different languages (modifiability).	N/A
AET17	The AET shall support response times that allow performing a tool-based architecture exploration efficiently (usability, performance).	N/A
AET18	The AET shall support understandability by considering common standards during the development of the graphical user interface (usability).	N/A

7.3 AET Software Architecture and Data Model

AET is implemented in C++ and runs on Microsoft® Windows operating systems. It uses a commercial database system for storing and retrieving data, which has drastically reduced the development effort. The software architecture of AET is organized in three layers (see Figure 7-1): presentation, business logic, and data management. The presentation layer is responsible for user interaction and data presentation. Data post-processing such as scenario sorting, combining data from different database tables, and report generation is done in the business application layer. The data management layer provides low-level services to access and maintain the database.

AET uses two different databases to store evaluation information: one for general data (*GeneralData*) and one for project data (*ProjectData*). The general database contains static data – i.e., data that does not depend on the specific context of an evaluation. Examples are domain-independent analysis questions such as “Does the system have to operate 24 hours a day, 7 days a week?” and evaluation scenarios such as “An error occurs during processing – how does the system detect, report, handle, and correct it?” General data can be used to support the evaluation team during the analysis of a particular system.

The project database contains project-specific information obtained during an evaluation (dynamic data). It includes data such as qualities of the system, scenarios, architectural approaches, analysis information, and risks that have been identified and examined during evaluation. The two databases can be distributed over a network or internet, as shown in Figure 7-2. This means that the AET application and the AET databases need not to execute on the same computer.

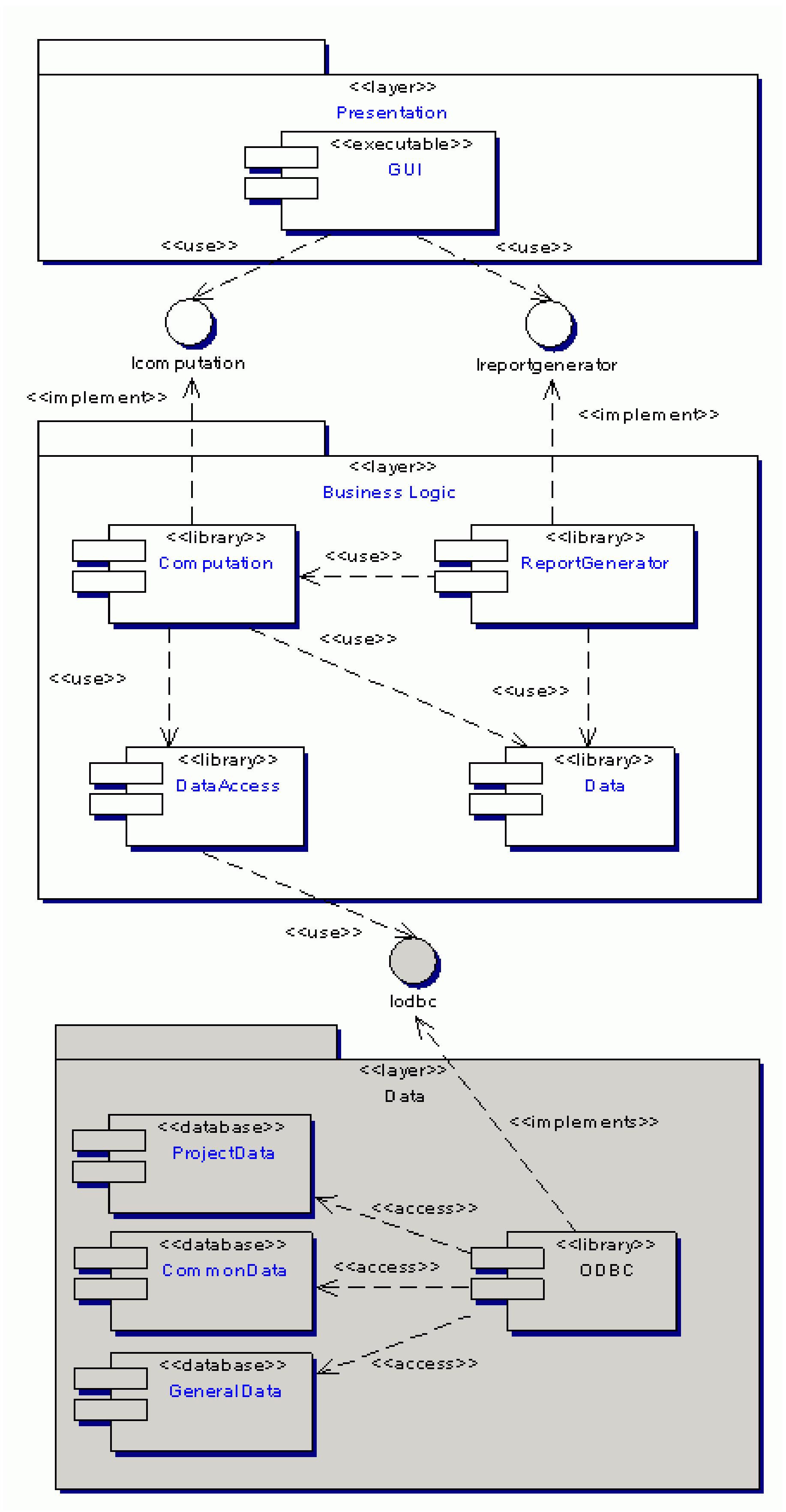


Figure 7-1: AET Software Architecture

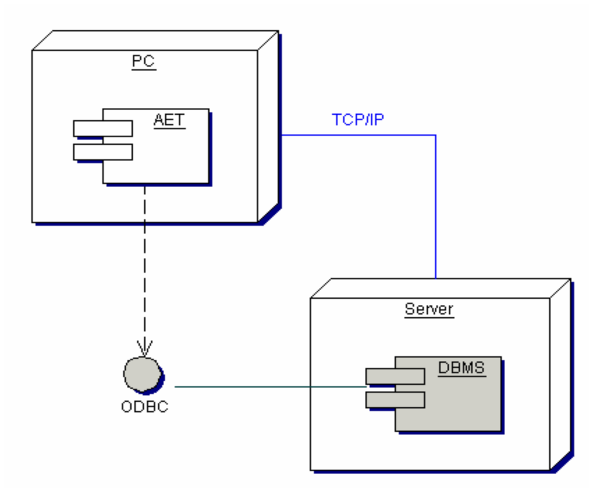


Figure 7-2: Deployment View of the AET Architecture

Figure 7-3 shows a simplified data model of the project database. For each evaluation project individual requirements, scenarios, architectural decisions, findings, and risk themes can be recorded. Priority dimensions and scales are global to a project. Each scenario can have a ranking. The ranking must conform to the global scheme defined for the project.

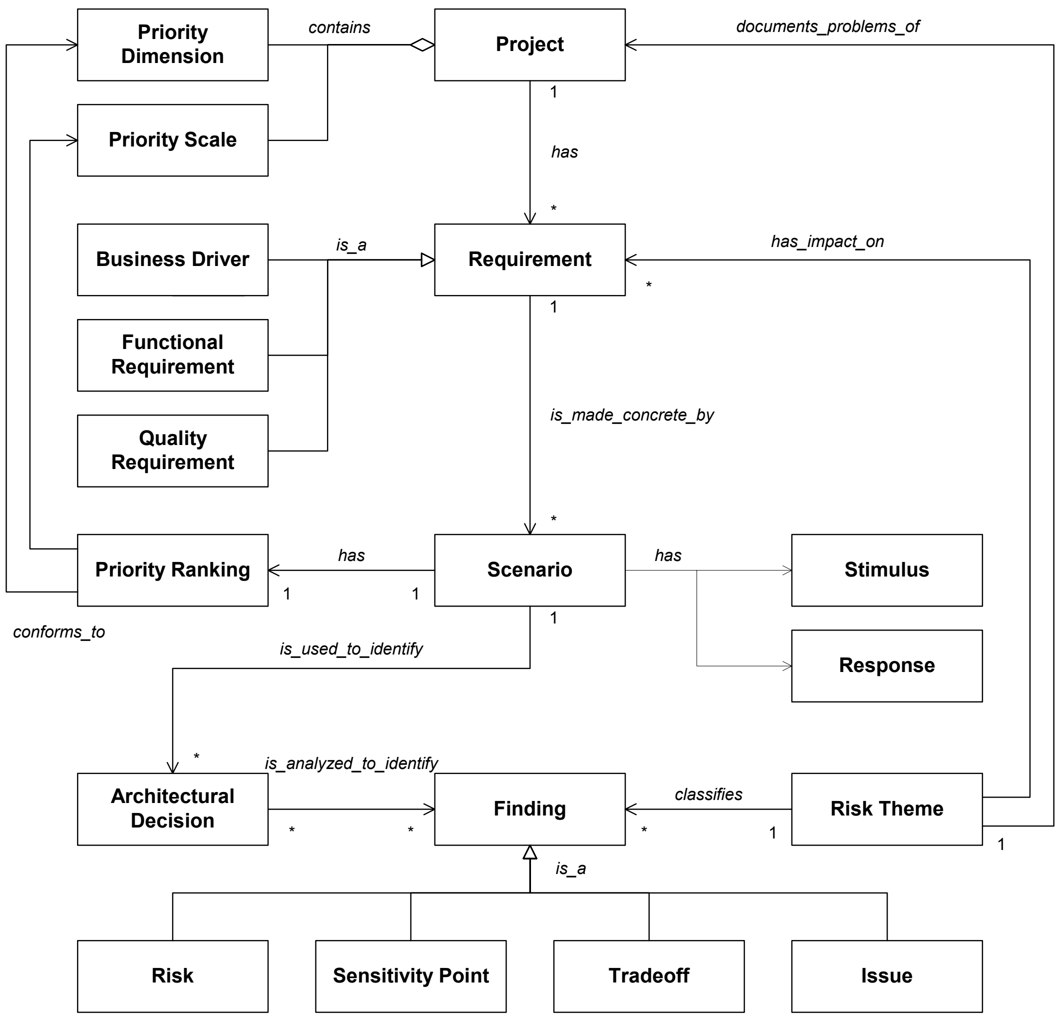


Figure 7-3: AET Data Model

Scenarios have a stimulus and a response. They are explored in order to identify architectural decisions. These decisions can further be analyzed to identify particular findings. A finding can be a risk, sensitivity point, tradeoff, or issue. Risk themes classify and summarize findings. Each risk theme has an impact on one or more requirements – this means, some of the business drivers or qualities cannot completely be met. The risk themes document the problem areas associated with the system under evaluation. The severity level of the problems indicates how close an organization is to fielding a successful system.

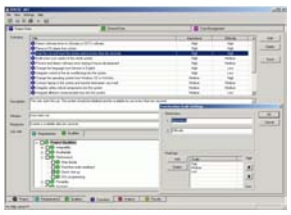
From an implementation point of view, AET is created based on 151 design classes (28 GUI, 77 application logic, and 46 other classes). The source code contains 32.054 LOC (Lines of Code).

7.4 Applying AET During Architecture Evaluation

In this section we describe how AET can be applied in practice in order to support a QUADRAD-E architecture evaluation. Table 7-4 gives an overview of the AET functionality. It also shows how the functional requirements presented in Section 7.2 are achieved in AET.[3]

A QUADRAD-E architecture evaluation is typically performed in a workshop. Participants of the workshop are usually the key stakeholders of the system and a review team that facilitates the evaluation. Table 7-5 shows typical activities of a QUADRAD-E evaluation workshop. In the next subsections, we describe how AET can be used to support the evaluation activities performed during such a workshop.

Table 7-4: AET Functionality

AET Functionality		**Requirement**
	- Capturing requirements - Documenting evaluation scenarios - Linking scenarios to quality requirements - Prioritization of evaluation scenarios - Defining criteria for scenario prioritization - Defining scales for scenario prioritization - Sorting scenarios according to priorities	AET1 AET2 AET3 AET4 AET5 AET6 AET7
	- Documenting scenario analysis results - Capturing findings - Linking findings to scenarios	AET8 AET9 AET10
	- Clustering of risks - Report generation	AET11 AET12

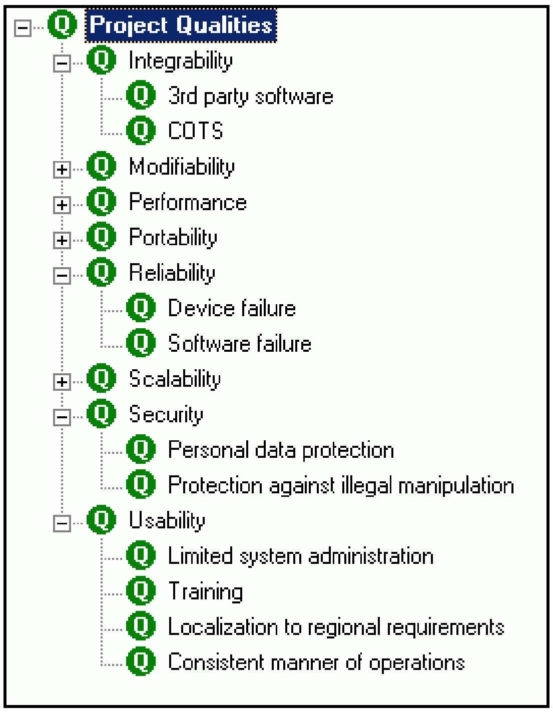
[3] The treatment of quality requirements is further discussed in the validation of this thesis (cf. Chapter 8).

Table 7-5: Typical Activities of an Architecture Evaluation Workshop

Step	Description
Presentation	
1	*Present method:* The review team describes the evaluation method to the assembled stakeholders (QUADRAD-E). Typical stakeholders of an evaluation are architects, developers, integrators, testers, sales & marketing, and management.
2	*Present business drivers:* The marketing representative describes what business goals are motivating the development effort and hence what will be the primary architectural drivers (e.g., high availability, high security, or time-to-market).
3	*Present architecture:* The architect describes the proposed architecture, focusing on how it addresses the architectural drivers.
Investigation and Analysis	
4	*Identify architectural approaches:* The architect determines the central mechanisms (architectural styles and patterns) used in the architecture.
5	*Brainstorm and prioritize scenarios:* The review team collects evaluation scenarios from the stakeholders to make business drivers and important requirements more concrete. The scenarios are then prioritized according to a ranking scheme (e.g., business importance and architectural impact).
6	*Analyze architecture:* The evaluation team and the architect analyze the architectural decisions made to achieve the high-priority-scenarios. This is supported by examining the architectural approaches from step 4 and by identifying those design elements that are affected by the scenarios. The goal is to uncover risks and to provide solution approaches for mitigating the risks.
Reporting	
7	*Present results:* The review team presents the findings (e.g., risks and solution approaches) to the audience and summarizes them in a written report.

7.4.1 Capturing Requirements and Quality Attributes

During the presentation of business drivers and the architecture (step 2 and 3), a lot of information about requirements and quality attributes is usually gathered from stakeholders. This information can be recorded in AET for later exploration and reference. Functional and quality attribute requirements can be put into a requirements list. Quality attributes can be documented in a quality tree. A sample quality tree for an Embedded Vehicle Control System (EVCS) is shown in Figure 7-4.



- Project Qualities
  - Integrability
    - 3rd party software
    - COTS
  - Modifiability
  - Performance
  - Portability
  - Reliability
    - Device failure
    - Software failure
  - Scalability
  - Security
    - Personal data protection
    - Protection against illegal manipulation
  - Usability
    - Limited system administration
    - Training
    - Localization to regional requirements
    - Consistent manner of operations

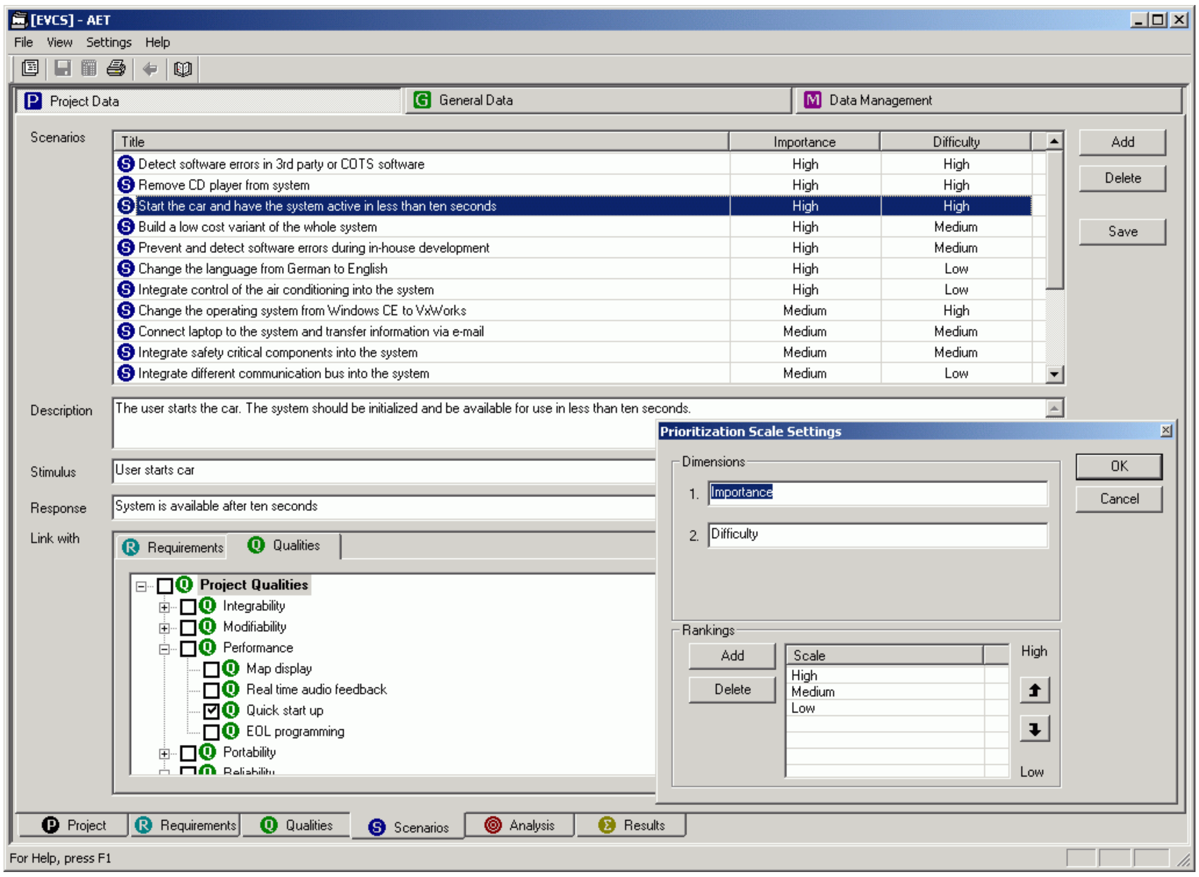
Figure 7-4: Quality Attribute Tree

Optionally, a first set of typical quality requirements and scenarios for the type of systems under evaluation (e.g., embedded automotive systems) can be generated from the general database.

Each quality attribute may contain one or more sub-factors, as shown in Figure 7-4. Sub-factors describe specific stakeholder concerns of the quality. For example, in Figure 7-4 “personal data protection” is of specific concern for security. Note that each item in the quality tree can be moved easily. This allows quick modifications during an architecture evaluation.

7.4.2 Scenarios and Prioritization

AET allows the scenarios gathered in step 5 to be recorded in a scenario list, as illustrated in Figure 7-5. Scenarios can also be described in more detail. For example, you can document potential stimuli and responses in order to make the expected behavior more concrete. In addition, scenarios can be linked to a particular quality attribute or business goal in order to document their contribution to that attribute or goal. In Figure 7-5, the selected scenario contributes to “quick start up” which is a sub-factor of performance.



[EVCS] - AET

Scenarios

Title	Importance	Difficulty
Detect software errors in 3rd party or COTS software	High	High
Remove CD player from system	High	High
Start the car and have the system active in less than ten seconds	High	High
Build a low cost variant of the whole system	High	Medium
Prevent and detect software errors during in-house development	High	Medium
Change the language from German to English	High	Low
Integrate control of the air conditioning into the system	High	Low
Change the operating system from Windows CE to VxWorks	Medium	High
Connect laptop to the system and transfer information via e-mail	Medium	Medium
Integrate safety critical components into the system	Medium	Medium
Integrate different communication bus into the system	Medium	Low

Description: The user starts the car. The system should be initialized and be available for use in less than ten seconds.

Stimulus: User starts car

Response: System is available after ten seconds

Link with: Requirements | Qualities

- ☐ Project Qualities
  - ☐ Integrability
  - ☐ Modifiability
  - ☐ Performance
    - ☐ Map display
    - ☐ Real time audio feedback
    - ☑ Quick start up
    - ☐ EOL programming
  - ☐ Portability
  - ☐ Reliability

Prioritization Scale Settings

Dimensions
1. Importance
2. Difficulty

Rankings

Scale
High
Medium
Low

For Help, press F1

Figure 7-5: Classification of Scenarios

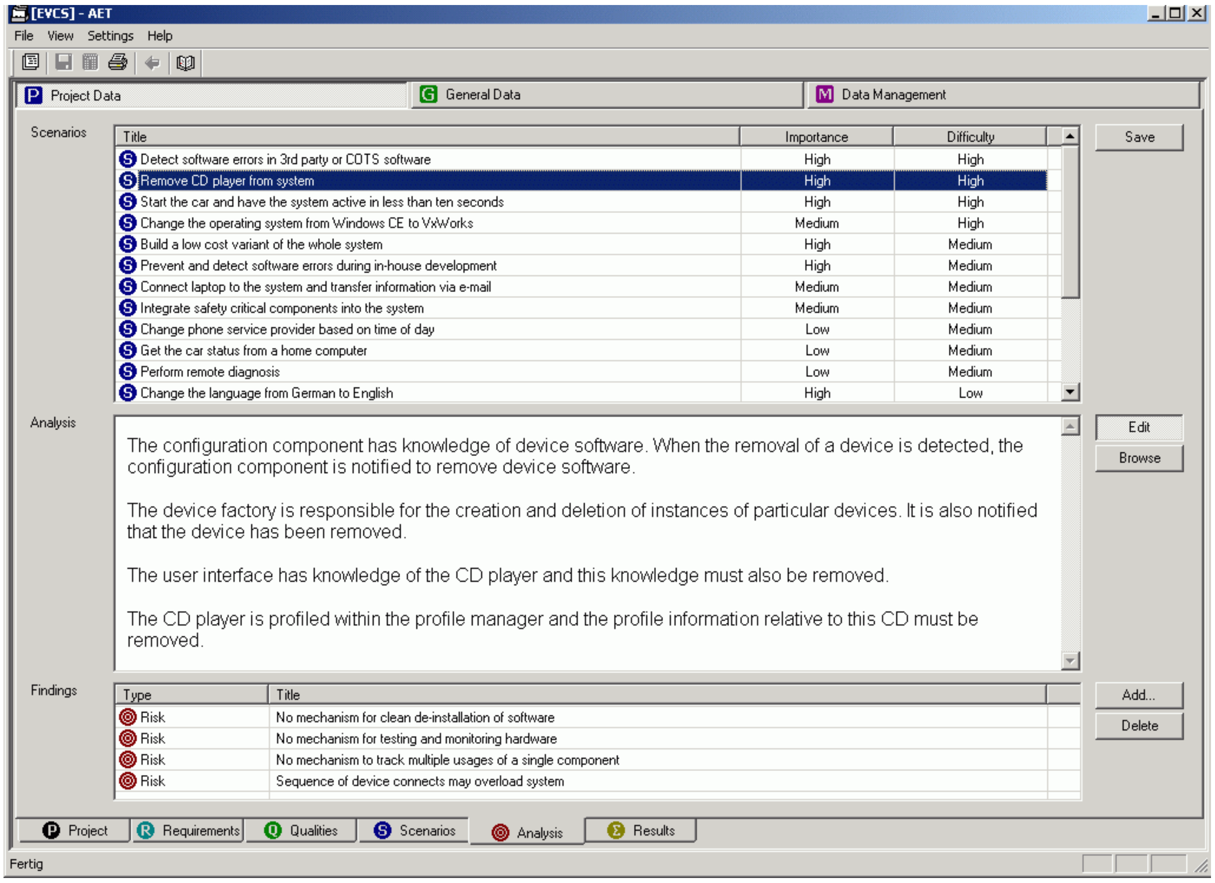
Furthermore, stakeholder priorities can be assigned to each scenario, as defined in step 5. The scenario priorities then drive the further analysis. In the example of Figure 7-5, we use two dimensions (business importance and architectural difficulty) and three values (High, Medium, Low) for prioritization.

However, AET allows adapting the dimensions and priority scale to fit individual needs of the evaluation, as shown in the lower right Figure 7-5. Scenarios can easily be sorted according their priority such that the most important ones (high importance, high difficulty) appear at the top of the list.

7.4.3 Scenario Analysis

Each scenario can be analyzed in AET. Usually, the user starts with the most critical scenarios. There is room for a detailed description of the analysis results, including text and pictures. For example, the architectural elements that contribute to a particular scenario can be described. The user may also document how the architecture would need to be changed to accommodate a scenario. The description is stored in HTML-format in the database for post-processing.

Figure 7-6 illustrates the scenario analysis capabilities of AET. As illustrated in the middle of the figure, a textual description of the analysis for the scenario "Remove CD player from system" has been recorded in AET. In addition, four risks have been identified for the scenario. They can be found at the bottom of the screen shot.



Scenarios

Title	Importance	Difficulty
Detect software errors in 3rd party or COTS software	High	High
Remove CD player from system	High	High
Start the car and have the system active in less than ten seconds	High	High
Change the operating system from Windows CE to VxWorks	Medium	High
Build a low cost variant of the whole system	High	Medium
Prevent and detect software errors during in-house development	High	Medium
Connect laptop to the system and transfer information via e-mail	Medium	Medium
Integrate safety critical components into the system	Medium	Medium
Change phone service provider based on time of day	Low	Medium
Get the car status from a home computer	Low	Medium
Perform remote diagnosis	Low	Medium
Change the language from German to English	High	Low

Analysis

The configuration component has knowledge of device software. When the removal of a device is detected, the configuration component is notified to remove device software.

The device factory is responsible for the creation and deletion of instances of particular devices. It is also notified that the device has been removed.

The user interface has knowledge of the CD player and this knowledge must also be removed.

The CD player is profiled within the profile manager and the profile information relative to this CD must be removed.

Findings

Type	Title
Risk	No mechanism for clean de-installation of software
Risk	No mechanism for testing and monitoring hardware
Risk	No mechanism to track multiple usages of a single component
Risk	Sequence of device connects may overload system

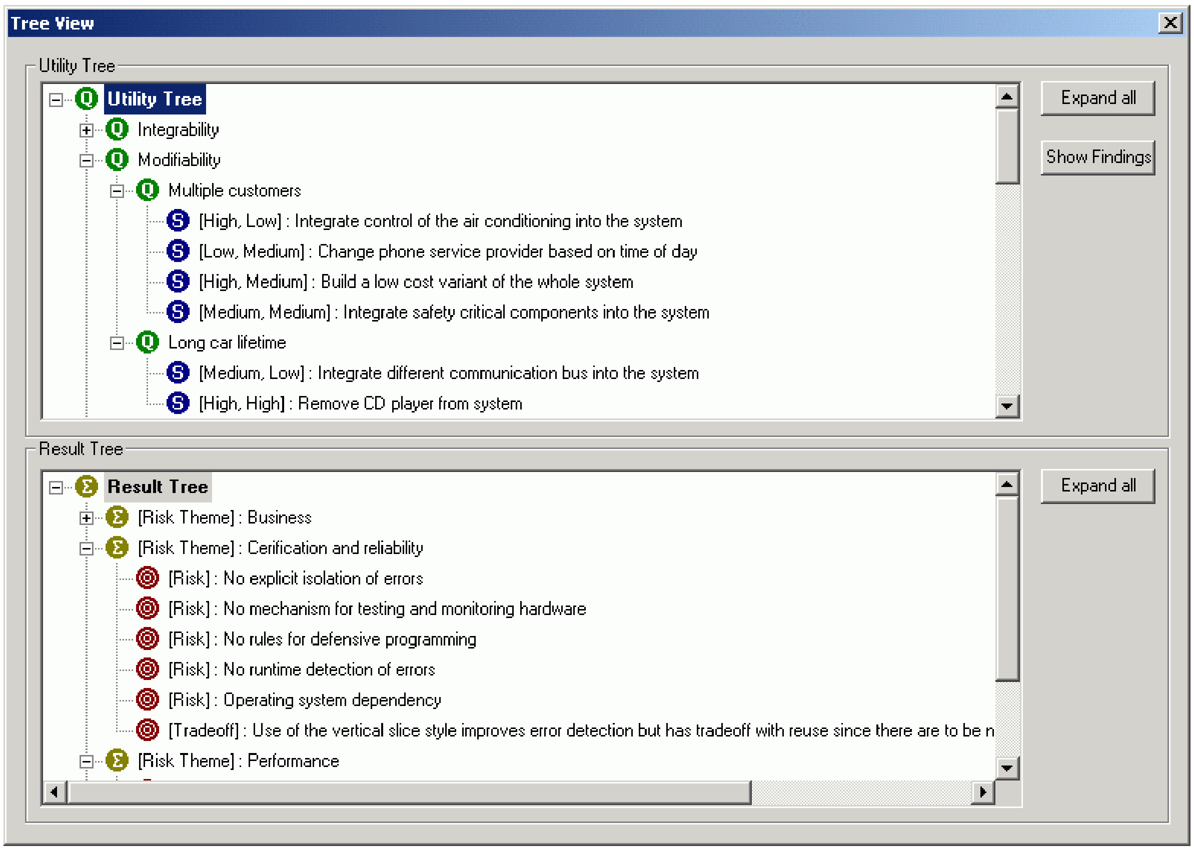
Figure 7-6: Scenario Analysis

Furthermore, AET allows the classification of important findings of the analysis. Important findings are, for example, risks or tradeoff points. Risks arise from architecturally important decisions that have not yet been made. A tradeoff point occurs when multiple quality attributes are differently affected when changing one architectural parameter. For example, improving throughput may result in reduced reliability.

When recording a finding, AET directly links the finding to the scenario under analysis. This is very useful since it allows you to easily trace back risks, tradeoffs, issues etc. to its source – the scenario. You may follow the trace to obtain a more detailed description of the analysis. Storing traces also supports statistics, for example, about which scenarios are most critical for the success of the system.

7.4.4 Clustering Findings

Since the number of risks identified during an evaluation can be high, AET allows classifying them in risk themes. Risk themes summarize key architectural issues that pose potential future problems for the success of the system. For each risk theme, AET allows to assign one or more findings. In addition there is room for a detailed discussion of the risk theme. Risks and risk themes can be clearly arranged in a "result tree," as shown in the lower part of Figure 7-7. This tree is automatically generated by AET based on the relationships between findings and risk themes.


Tree View

Utility Tree

Utility Tree
- Integrability
- Modifiability
  - Multiple customers
    - [High, Low] : Integrate control of the air conditioning into the system
    - [Low, Medium] : Change phone service provider based on time of day
    - [High, Medium] : Build a low cost variant of the whole system
    - [Medium, Medium] : Integrate safety critical components into the system
  - Long car lifetime
    - [Medium, Low] : Integrate different communication bus into the system
    - [High, High] : Remove CD player from system

Result Tree

Result Tree
- [Risk Theme] : Business
- [Risk Theme] : Cerification and reliability
  - [Risk] : No explicit isolation of errors
  - [Risk] : No mechanism for testing and monitoring hardware
  - [Risk] : No rules for defensive programming
  - [Risk] : No runtime detection of errors
  - [Risk] : Operating system dependency
  - [Tradeoff] : Use of the vertical slice style improves error detection but has tradeoff with reuse since there are to be n
- [Risk Theme] : Performance

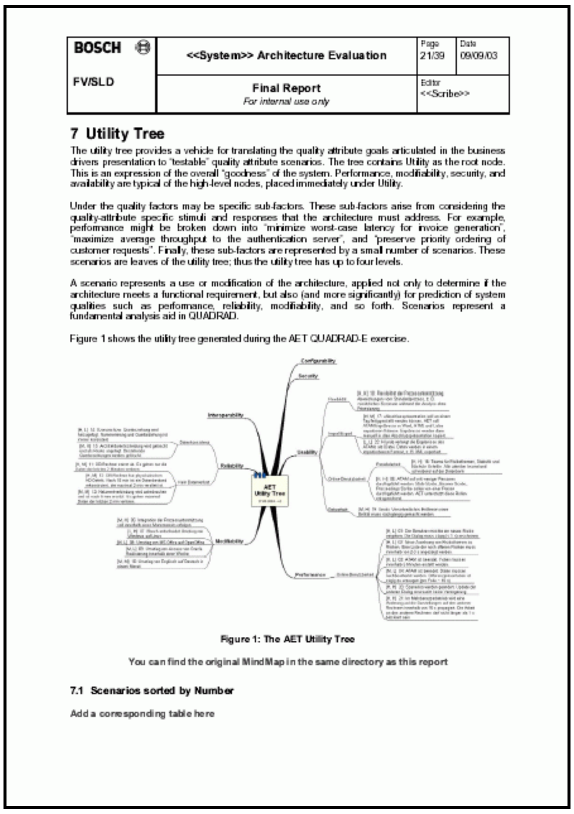
Figure 7-7: Tree View of Qualities, Scenarios, and Analysis Results

AET can also generate a "utility tree," as illustrated in the upper part of Figure 7-7. This tree represents a summary of the elicited scenarios and priorities together with the respective sub-factors and quality attributes. As shown in Figure 7-7, four scenarios have been documented to address the modifiability concern of supporting "multiple customers."

7.4.5 Report Generation

Finally, the results documented in AET can be included in a written report. AET is capable of automatically exporting the evaluation results as Microsoft Word® or Microsoft PowerPoint® document. It formats the evaluation data and includes it in user-definable documentation templates. For example, every tree view (e.g., quality tree) is converted to a MindMap [Mindjet 2002]. These trees visualize the results of the evaluation in a concise form. Furthermore, scenario descriptions and scenario analysis results can be exported in tabular form.

Parts of an AET generated report are shown in Figure 7-8. As illustrated, the report generation capabilities support clarity and understandability in the documentation of evaluation results.



BOSCH FV/SLD	<<System>> Architecture Evaluation	Page 21/39	Date 09/09/03
	Final Report *For internal use only*	Editor <<Scribe>>	

7 Utility Tree

The utility tree provides a vehicle for translating the quality attribute goals articulated in the business drivers presentation to "testable" quality attribute scenarios. The tree contains Utility as the root node. This is an expression of the overall "goodness" of the system. Performance, modifiability, security, and availability are typical of the high-level nodes, placed immediately under Utility.

Under the quality factors may be specific sub-factors. These sub-factors arise from considering the quality-attribute specific stimuli and responses that the architecture must address. For example, performance might be broken down into "minimize worst-case latency for invoice generation", "maximize average throughput to the authentication server", and "preserve priority ordering of customer requests". Finally, these sub-factors are represented by a small number of scenarios. These scenarios are leaves of the utility tree; thus the utility tree has up to four levels.

A scenario represents a use or modification of the architecture, applied not only to determine if the architecture meets a functional requirement, but also (and more significantly) for prediction of system qualities such as performance, reliability, modifiability, and so forth. Scenarios represent a fundamental analysis aid in QUADRAD.

Figure 1 shows the utility tree generated during the AET QUADRAD-E exercise.

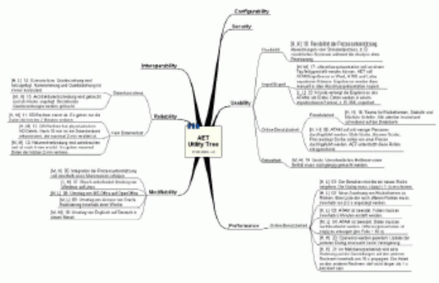
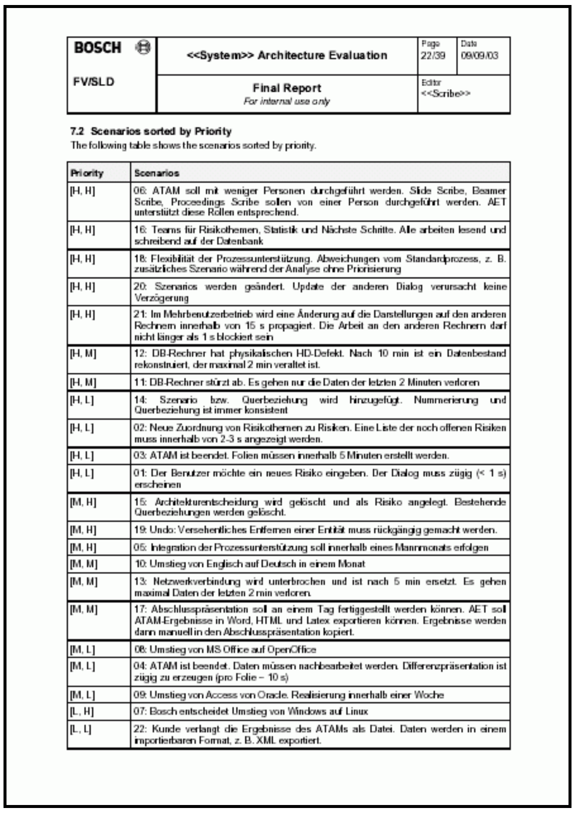


Figure 1: The AET Utility Tree

You can find the original MindMap in the same directory as this report

7.1 Scenarios sorted by Number

Add a corresponding table here



BOSCH FV/SLD	<<System>> Architecture Evaluation	Page 22/39	Date 09/09/03
	Final Report *For internal use only*	Editor <<Scribe>>	

7.2 Scenarios sorted by Priority

The following table shows the scenarios sorted by priority.

Priority	Scenarios
[H, H]	06: ATAM soll mit weniger Personen durchgeführt werden. Slide Scribe, Beamer Scribe, Proceedings Scribe sollen von einer Person durchgeführt werden. AET unterstützt diese Rollen entsprechend.
[H, H]	16: Teams für Risikothemen, Statistik und Nächste Schritte. Alle arbeiten lesend und schreibend auf der Datenbank
[H, H]	18: Flexibilität der Prozessunterstützung. Abweichungen vom Standardprozess, z. B. zusätzliches Szenario während der Analyse ohne Priorisierung
[H, H]	20: Szenarios werden geändert. Update der anderen Dialog verursacht keine Verzögerung
[H, H]	21: Im Mehrbenutzerbetrieb wird eine Änderung auf die Darstellungen auf den anderen Rechnern innerhalb von 15 s propagiert. Die Arbeit an den anderen Rechnern darf nicht länger als 1 s blockiert sein
[H, M]	12: DB-Rechner hat physikalischen HD-Defekt. Nach 10 min ist ein Datenbestand rekonstruiert, der maximal 2 min veraltet ist.
[H, M]	11: DB-Rechner stürzt ab. Es gehen nur die Daten der letzten 2 Minuten verloren
[H, L]	14: Szenario bzw. Querbeziehung wird hinzugefügt. Nummerierung und Querbeziehung ist immer konsistent
[H, L]	02: Neue Zuordnung von Risikothemen zu Risiken. Eine Liste der noch offenen Risiken muss innerhalb von 2-3 s angezeigt werden.
[H, L]	03: ATAM ist beendet. Folien müssen innerhalb 5 Minuten erstellt werden.
[H, L]	01: Der Benutzer möchte ein neues Risiko eingeben. Der Dialog muss zügig (< 1 s) erscheinen
[M, H]	15: Architekturentscheidung wird gelöscht und als Risiko angelegt. Bestehende Querbeziehungen werden gelöscht.
[M, H]	19: Undo: Versehentliches Entfernen einer Entität muss rückgängig gemacht werden.
[M, H]	05: Integration der Prozessunterstützung soll innerhalb eines Mannmonats erfolgen
[M, M]	10: Umstieg von Englisch auf Deutsch in einem Monat
[M, M]	13: Netzwerkverbindung wird unterbrochen und ist nach 5 min ersetzt. Es gehen maximal Daten der letzten 2 min verloren.
[M, M]	17: Abschlusspräsentation soll an einem Tag fertiggestellt werden können. AET soll ATAM-Ergebnisse in Word, HTML und Latex exportieren können. Ergebnisse werden dann manuell in den Abschlusspräsentation kopiert.
[M, L]	08: Umstieg von MS Office auf OpenOffice
[M, L]	04: ATAM ist beendet. Daten müssen nachbearbeitet werden. Differenzpräsentation ist zügig zu erzeugen (pro Folie – 10 s)
[M, L]	09: Umstieg von Access von Oracle. Realisierung innerhalb einer Woche
[L, H]	07: Bosch entscheidet Umstieg von Windows auf Linux
[L, L]	22: Kunde verlangt die Ergebnisse des ATAMs als Datei. Daten werden in einem importierbaren Format, z. B. XML exportiert.

Figure 7-8: Architecture Evaluation Report

AET can facilitate architecture documentation, especially during an evaluation. We have particularly used AET during the QUADRAD validation.

7.5 Summary

In this chapter the Architecture Exploration Tool (AET) has been introduced. AET supports essential architecture evaluation and documentation activities of QUADRAD. In particular, important requirements for developing AET have been presented. The requirements are directly derived from QUADRAD-E activities. Furthermore, an overview of the AET software architecture and the data model used to design the AET database have been provided. In addition, it has been demonstrated how AET can be used to support a QUADRAD-E architecture evaluation workshop.

In the next chapter, we will present the validation of the QUADRAD framework. We will describe the validation approach and the results obtained from the validation in order to prove the hypothesis of this work.

8 Validation of the QUADRAD Framework

This chapter describes the approach and results of the validation undertaken to prove the hypothesis of this work. The validation was based on three case studies that were performed during the author's involvement in two industry-driven European research projects: ESAPS and CAFE[4].

The chapter is organized as follows: Section 8.1 describes the validation goals and introduces the pilot projects used to investigate the hypothesis of this work. In Section 8.2 the framework for setting up the validation and the respective metrics are explained. Section 8.3 describes the three main phases of the validation approach. Sections 8.4, 8.5, and 8.6 provide the results obtained from the validation of the pilot projects. In particular, there is an analysis of the extent to which the architectural quality of the pilot projects could be improved by applying the QUADRAD framework. In Section 8.7 the validation results are summarized and discussed in detail. Section 8.8 covers lessons learned from the validation. Finally, Section 8.9 summarizes the results of this chapter.

8.1 Validation Goals and Settings

The goal of the validation is to prove the hypothesis stated in Chapter 1:

> *"The architecture of software-intensive systems can be improved with respect to quality based on a systematic derivation of design decisions for architecturally driving requirements and a methodical evaluation and revision of the effects of those decisions in the architecture."*

We have developed and proposed the QUADRAD framework as a research contribution for this purpose. In order to validate the hypothesis we need evidence that QUADRAD improves the architectural quality of software-intensive systems (i.e., at least those that are part of the validation). This, in turn, shows that QUADRAD is capable of improving architecture development. Thus, the validation goal can be stated as follows:

> *Goal = Estimate if the QUADRAD framework improves the architectural quality of software-intensive systems*

We have chosen case studies for empirical validation because in our context it was the most suitable approach for evaluating the new framework against the company baseline in a real-world industrial setting. Conducting an experiment with full execution and measurement

[4] ESAPS (Engineering Software Architectures, Processes, and Platforms for System Families) and CAFE (Concepts to Application in System Family Engineering) were two projects of the Information Technology for European Advancement (ITEA) programme. ITEA is dedicated to advances in engineering of software-intensive systems.

control was simply impossible in our context. Since the validation was integrated in running prototype development projects with tight schedules and limited budget, a controlled experiment would have consumed too many extra resources such that milestones of these projects might have been at risk. Instead, conducting case studies was a feasible and cost-effective alternative to gather valuable feedback from applying a new research framework in an industrial context.

To investigate the hypothesis of this work, we have selected three pilot projects A, B, and C for validation (see Table 8-1). Project A was concerned with the development of automotive multimedia systems. Project B dealt with automotive assistance systems. Finally, project C was dedicated to the implementation of an engineering support system.

A and B are representative systems from the automotive domain. Project C comprises the development of the QUADRAD support tool described in Chapter 7. The result of this project – the Architecture Exploration Tool (AET) – is part of the research contribution of this work. Note that a student of the University of Applied Science at Gießen-Friedberg has supported the development during his practical training period and diploma stay in the author's project team [Engelhardt 2003].

By choosing three pilot projects we could achieve a comparative validation. We have contrasted the results of using QUADRAD for architectural development against a baseline. For comparing the results we used the same method (QUADRAD-E) and the same baseline data (evaluation scenarios) in order to avoid bias and to ensure internal validity. Next we will describe the framework used throughout the validation.

Table 8-1: Pilot Projects of the Validation

Project / Characteristics	**A (CMS)**	**B (AAS)**	**C (AET)**
Application Domain	Car Multimedia Systems	Automotive Assistance Systems	Engineering Support Systems
Application Type	Software-Intensive System	Software-Intensive System	Software-Intensive System
Operating System	Embedded OS	Embedded Real-Time OS	Windows 2000
Programming Language	C++	C	C++
Target Software Size	2 MB	256 KB	1.8 MB
Development Team	Large (40 Persons)	Medium (10 Persons)	Small (2 Persons)

8.2 Validation Framework

The quality improvement during an architectural development effort has been estimated by taking the quality of the effort's major outcome into account: the architecture. The quality of an architecture depends on the risks included in the design decisions made in the architecture. A design decision involves a risk if it is not technically sound or if it does not support the business goals of the development effort (cf. Chapter 6).

To achieve the validation goal given in Section 8.1, quality improvements have been estimated by evaluating the risks included in the design decisions made in the architecture of each case study. This has been done by considering the risk assessment equation discussed in Section 2.3.2.2:

$$Risk = probability \times impact\text{, with}$$

Probability:	Estimates how probable it is that the risk will occur.
Impact:	Estimates the influence on the business success of the system.

According to Chapter 2, the risks included in (the design decisions of) an architecture are usually not equally critical. The equation above represents a possibility to estimate the *severity* of a risk. The equation expresses that the severity of a risk depends on its probability and impact. More precisely, the higher the probability and impact of a risk the more critical it is for the development and the more likely the architecture will fail to achieve its requirements. In other words, estimating the severity of risks in an architecture enables us to judge the *quality* of that architecture. These observations lead to the following two assumptions:

Assumption 1:

The more severe risks an architecture includes the lower is the quality of that architecture.

Assumption 2:

Reducing the severity of risks in an architecture improves the overall quality of that architecture.

As described in Chapter 6, scenarios can be used for identifying risks in an architecture during an architecture evaluation. By using the same set of scenarios for evaluation one can analyze the improvement made between two versions of an architecture (e.g., two development releases of the architecture). The quality of the two architectures becomes comparable.

The validation goal of this work requires an estimation of the improvement in quality between two architectures. If we use the same set of evaluation scenarios it is possible to analyze whether the quality between a baseline architecture (A) and an architecture revised by applying the QUADRAD framework (A*) has increased. Figure 8-1 depicts the basic validation concept.

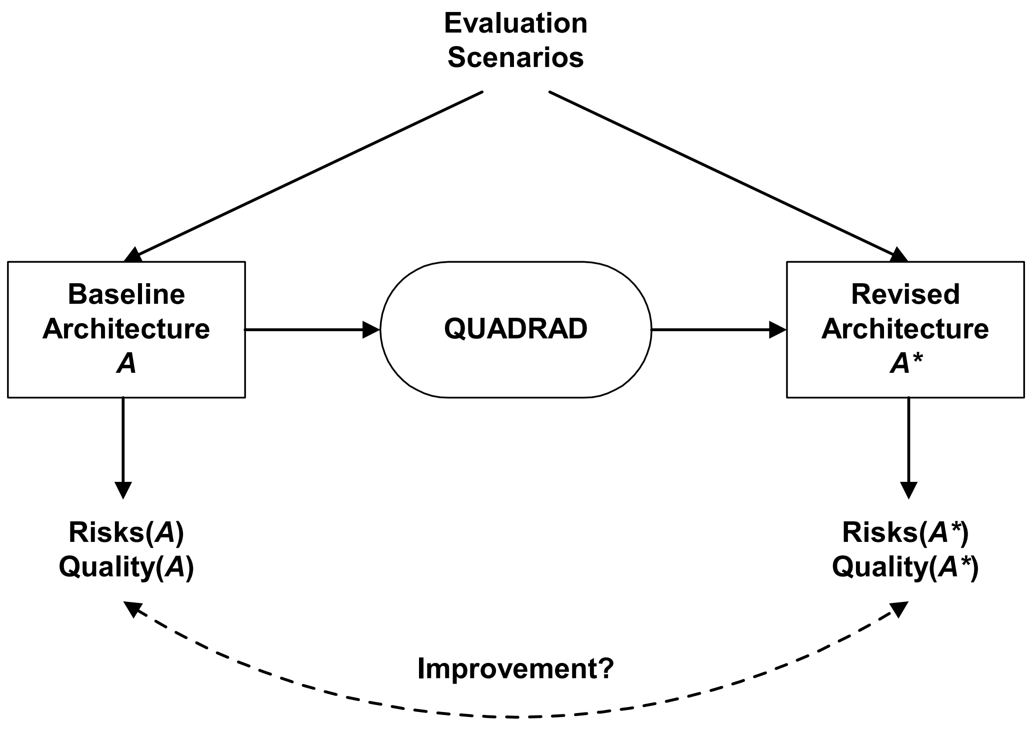


Figure 8-1: Validation Concept

For each case study project the risks included in the current release of the architecture (baseline) have been determined. For each risk the probability and impact has been determined in order to determine the severity of the risks and consequently on the overall quality of the architecture. Next, the QUADRAD framework has been applied for revising the baseline architectures of the projects. During this phase the evaluation scenarios and risks identified previously have been used as input. Finally, the effects of the QUADRAD application have been analyzed. Again the risks of the revised architectures have been evaluated and compared with the results obtained from the baseline architecture.

To define appropriate metrics for measuring the quality improvement between two architectures, the Goal Question Metric (GQM) approach [Basili 1992] has been applied. The GQM approach is based upon the assumption that for an organization to measure in a purposeful way it must first specify the goals for itself and its projects, then it must trace those goals to the data that are intended to define those goals operationally, and finally provide a framework for interpreting the data with respect to the stated goals. Thus it is important to make clear, at least in general terms, what informational needs the organization has, so that these needs for information can be quantified whenever possible, and the quantified information can be analyzed as to whether or not the goals are achieved. The result of the application of GQM is the specification of a measurement system targeting a particular set of issues and a set of rules for the interpretation of the measurement data.

Table 8-2 shows the GQM model developed as part of the validation framework. The model represents a hierarchical structure starting with a goal (specifying measurement purpose, issue, and object). In our case, the goal particularly represents the validation goal. The goal is refined into several specific questions (Q1 to Q7). Each question is then refined into metrics (M1 to M13). The metrics answer the questions stated to achieve the validation goal.

Table 8-2: GQM Model for Validation

Goal	**Purpose** **Issue** **Object**	Estimate if the QUADRAD framework improves the architectural quality of software-intensive systems
Question	Q1	What are the risks of the baseline architecture?
Metrics	M1	S: Evaluation scenarios analyzed
	M2	r_i: Risks identified in baseline architecture
Question	Q2	How severe are the risks of the baseline architecture?
Metrics	M3	$\text{Probability}(r_i)$: Probability of risks
	M4	$\text{Impact}(r_i)$: Impact of risks
	M5	$\text{Ranking}(r_i) = \text{Probability}(r_i) \cdot \text{Impact}(r_i)$: Ranking of risks
	M6	$R_S(A)$: Overall risk severity of baseline architecture
Question	Q3	What is the overall quality of the baseline architecture?
Metrics	M7	Overall quality of baseline architecture: $Q(A) = \frac{1}{R_S(A)} = \begin{cases} 0 & \text{for } R_S(A) = 0 \\ \frac{1}{R_S(A)} & \text{else} \end{cases}$
Question	Q4	What are the risks of the revised architecture?
Metrics	M1	S: Evaluation scenarios analyzed
	M7	r_i*: Risks identified in revised architecture
Question	Q5	How severe are the risks of the revised architecture?
Metrics	M8	$\text{Probability}(r_i*)$: Probability of risks
	M9	$\text{Impact}(r_i*)$: Impact of risks
	M10	$\text{Ranking}(r_i*) = \text{Probability}(r_i*) \cdot \text{Impact}(r_i*)$: Ranking of risks
	M11	$R_S(A*)$: Overall risk severity of revised architecture
Question	Q6	What is the overall quality of the revised architecture?
Metrics	M12	Overall quality of revised architecture: $Q(A*) = \frac{1}{R_S(A*)} = \begin{cases} 0 & \text{for } R_S(A*) = 0 \\ \frac{1}{R_S(A*)} & \text{else} \end{cases}$
Question	Q7	Has the quality between baseline and revised architecture been improved?
Metrics	M13	$Q(A*) < Q(A)$: The quality of $A*$ is worse than that of A. $Q(A*) = Q(A)$: The quality of $A*$ is similar to that of A. $Q(A*) > Q(A)$: The quality of $A*$ is better than that of A.

The remainder of this section provides information about how the metrics of the validation framework have been defined.

8.2.1 Probability, Impact, and Risk Ranking

This section provides details about the metrics M3, M4, M5, M8, M9, and M10.

The risk assessment equation given in Section 2.3.2.2 signifies that the severity of a risk depends on its probability and impact. For each risk we can assign a "rank", which essentially measures the severity of a risk. The more probable a risk and the bigger the impact of a risk, the more critical it is for development.

We define the ranking of a risk r_i as probability-impact product:

$$\text{Ranking}(r_i) = \text{Probability}(r_i) \cdot \text{Impact}(r_i) \tag{1}$$

During the first case study we encountered the problem that the stakeholder could not specify the probability and impact of particular architectural risks by absolute numbers. We managed this problem by providing the stakeholders a range of discrete values for probability and impact they could choose from. Table 8-3 and Table 8-4 show the values and descriptions used for specifying the probability and impact of a risk, respectively.

Table 8-3: Probability of a Risk

Probability	Value	Description
Effective	100%	The risk already represents a problem in the architecture.
High	75%	There is a high probability that the risk will become a problem in the architecture.
Medium	50%	There is a moderate probability that the risk will become a problem in the architecture.
Low	25%	There is a low probability that the risk will become a problem in the architecture.
No	0	The risk will never become a problem in the architecture. (This may happen if the architecture evaluation was based on wrong assumptions.)

Table 8-4: Impact of a Risk

Impact	Value	Description
Very High	9	The risk has a very high impact on the business success of the system.
High	6	The risk has a high impact on the business success of the system.
Low	3	The risk has a low impact on the business success of the system.
Very Low	1	The risk has a very low impact on the business success of the system.
No	0	The risk has no impact on the business success of the system.

As described in Table 8-3, we used the term “effective risk” to denote a risk with probability 1. In fact, an “effective risk” represents not so much a risk as a concrete problem in the architecture. An architectural problem usually happens if an evaluation scenario with high priority cannot be adequately implemented with the architecture under consideration. Typically, then the architecture contains major design flaws exposed by the scenario (cf. Section 8.3 and 8.4).

Additionally, we used the discrete ratios 75%, 50%, and 25% for expressing that a risk will become a problem with high, medium, and low probability in the architecture. If a risk that has been recorded during architecture evaluation was based on wrong assumptions (e.g., the evaluation scenario that lead to that risk addresses requirements out of scope) then the risk’s probability has been set to null.

For specifying a risk’s impact on business success we defined five classes: very high, high, low, very low, and no impact. According to the stakeholders’ requests, we omitted to define a medium impact. The stakeholders were of the opinion that without a “safe” medium value, they could score the impact of a risk more practically. In addition, their preference was to use non-symmetrical weights for the five impact classes. As a consequence, we used the values 9, 6, 3, 1, and 0 known from quality function deployment (QFD) evaluations [Hauser 1988].

Table 8-5 summarizes the values for the resulting risk ranking using equation (1). For example, a risk that has a high probability but a low impact has a ranking of 0.75 · 3 = 2.25. The higher the ranking of a risk, the more critical it is for the development and the more likely the architecture will fail to achieve its requirements.

Table 8-5: Risk Ranking

Probability	Effective (100%)				High (75%)			
Impact	Very High	High	Low	Very Low	Very High	High	Low	Very Low
	9	6	3	1	9	6	3	1
Ranking	**9**	**6**	**3**	**1**	**6.75**	**4.5**	**2.25**	**0.75**

Probability	Medium (50%)				Low (25%)				No (0%)
Impact	Very High	High	Low	Very Low	Very High	High	Low	Very Low	No
	9	6	3	1	9	6	3	1	0
Ranking	**4.5**	**3**	**1.5**	**0.5**	**2.25**	**1.5**	**0.75**	**0.25**	**0**

8.2.2 Severity of Risks

This section provides details about the metrics M6 and M11.

The severity of the risks in an architecture A, $R_S(A)$, depends on the risks' probability and impact, as discussed above. We assume that the risks address different aspects of the system and thus can be treated as linearly independent.

$R_S(A)$ can be determined by the average ranking of the risks r_i included in the architecture:

$$R_S(A) = \text{avg}(\text{Ranking}(r_i)) = \frac{1}{n}\sum_{i=1}^{n}\text{Ranking}(r_i) \tag{2}$$

For example, the severity of three architectural risks ranked as 6.75, 6, and 4.5 is $\frac{1}{3}\cdot(6.75+6+4.5)=5.75$.

8.2.3 Architectural Quality

This section provides details about the metrics M7 and M12.

Assumptions 1 and 2 describe the fact that the quality improvement of an architecture A can be achieved by reducing the (severity of) risks included in A. This means that in order to improve the quality of A the probability and impact of the risks included in the design decisions of A have to be mitigated.

To express this observation in mathematical terms, we define the quality of an architecture A as inversely proportional to the severity of risks included in A:

$$Q(A) \sim \frac{1}{R_S(A)} \tag{3}$$

$Q(A)$:	Quality of an architecture A
$R_S(A)$:	Severity of the risks included in architecture A

Table 8-6 shows how the different rankings have been considered to define respective names for risk severity and architectural quality.

Table 8-6: Risk Severity and Architectural Quality

Ranking Interval	[9, 6)	[6, 3)	[3, 1)	[1, 0]
Risk Severity $R_S(A)$	High	Medium	Low	Very Low
Architectural Quality $Q(A)$	Bad	Average	Good	Very Good

Legend: [a, b] = $\{\ i \mid a \ge i \ge b\ \}$, [a, b) = $\{\ i \mid a \ge i > b \}$, $a > b$

On the one end, the interval [9, 6) is mainly driven by effective or highly probable risks, which have a very high or high impact on the development and business success of the system. This counts for an architecture with high risk severity. We define the quality of an architecture with a high risk severity as bad.

On the other end, the interval [1, 0] primarily includes risks with a very low or no impact. Therefore, for this interval we define the risk severity as very low. Consequently, the quality of an architecture with a very low risk severity is very good.

In between these ends, the interval [3, 1) the probability-impact-product is mainly shaped by low impact rates or low risk probabilities. This counts for a good architecture with a low risk severity. The interval [6, 3) is principally driven by risks with a medium-scale probability and a high to low impact. Therefore we define a medium risk severity and an average architectural quality for this interval.

In order to express the quality of an architecture by a single number we define the following equation:

$$Q(A) = \frac{1}{R_S(A)} = \begin{cases} 0 & \text{for } R_S(A) = 0 \\ \dfrac{1}{R_S(A)} & \text{else} \end{cases} \qquad (4)$$

Equation (4) is derived by taking (3) and the assumptions 1 and 2 discussed above into account: The more severe risks an architecture includes, the lower is the quality of that architecture. The severity of risks in an architecture is calculated by $R_S(A)$. Reducing the severity of risks in an architecture improves the overall quality of that architecture. If $R_S(A)$ decreases, $Q(A)$ increases by the same amount. In the ideal case there are no severe risks in the architecture (i.e., $R_S(A)$=0). The quality of the architecture can be defined as very good ($Q(A)$=0, see Table 8-6) since it serves its purpose and achieves the imposed requirements without negative effects.

8.2.4 Architectural Quality Improvement

This section provides details about the metric M13.

With the help of the definitions given above, we can now compare the quality of two architectures *A* and *A**. In order to minimize comparison errors we assume that *A* and *A** rely on the same business goals and have the same architectural drivers. Further, we assume that *A* and *A** implement the same requirements. Moreover, we use the same set of evaluation scenarios to analyze the design decisions with respect to the risks. Finally, we assume that we have captured the most important risks for the given scenarios – i.e., there are no more hidden risks or side effects of risks that have a strong impact on the overall quality of the system.

With these assumptions we can state the following comparisons for the two architectures *A* and *A**:

$Q(A^*) < Q(A)$: The quality of A^* is worse than that of A.

$Q(A^*) = Q(A)$: The quality of A^* is similar to that of A.

$Q(A^*) > Q(A)$: The quality of A^* is better than that of A.

Example:

Suppose we have two architectures A and A^* that implement the same set of safety requirements but use different architectural mechanisms to achieve those requirements. Further assume we identified four risks in A and six risks in A^* with the following probability and impact values during a QUADRAD-E evaluation:

Risks of *A*	**Probability**	**Impact**	**Ranking**
R1	Effective	Low	**3**
R2	Effective	Very High	**9**
R3	Medium	Very High	**4.5**
R4	High	High	**4.5**

Risks of *A**	**Probability**	**Impact**	**Ranking**
R1	Effective	Low	**3**
R2	High	Very High	**6.75**
R3	Effective	Very High	**9**
R4	Low	Low	**0.75**
R5	Effective	High	**6**
R6	Medium	Low	**1.5**

Using equation (4) the quality of the two architectures can be calculated as:

$$Q(A) = \frac{1}{R_S(A)} = \frac{1}{\frac{1}{4}\sum_{i=1}^{4}\text{Ranking}(r_{A_i})} = \frac{1}{\frac{1}{4}(3+9+4.5+4.5)} = \frac{1}{5.25} = 0.19$$

$$Q(A^*) = \frac{1}{R_S(A^*)} = \frac{1}{\frac{1}{6}\sum_{i=1}^{6}\text{Ranking}(r_{A^*_i})} = \frac{1}{\frac{1}{6}(3+6.75+9+0.75+6+1.5)} = \frac{1}{4.5} = 0.22$$

According to Table 8-6 the average risk severity of both architectures is medium. From $R_S(A) = 5.25 > 4.5 = R_S(A^*)$ follows that $Q(A) < Q(A^*)$. This means that the quality of A is worse than to that of A^*.

8.3 Validation Approach

After clarifying what data and metrics are required for validating the hypothesis we now introduce the validation approach applied in this work. An overview of the approach is shown in Figure 8-2. This overview refines the validation concept given in Figure 8-1.

There are four steps that have been performed during the validation: Evaluating the original architectures of the pilot projects (step 1), creating revised architectures for the projects by applying QUADRAD (step 2), evaluating the revised architectures (step 3), and comparing the quality of the revised architectures with the original architectures (step 4). More details of the validation steps are explained in the remainder of this section. In the next sections the individual results of the validation are described.

8.3.1 Step 1: Evaluating the Original Architectures

The purpose of this step is to collect baseline data for the validation. In different architecture evaluation workshops we performed QUADRAD-E activities in order to identify the architectural risks that are included in the pilot systems A, B, and C. In particular, we captured the architecturally driving requirements as well as the strategies and mechanisms used to implement those drivers. Based on this information we collected a set of evaluation scenarios and analyzed the design decisions made in the architectures to achieve the scenarios. During this analysis we evaluated if risks are associated with the decisions, especially with respect to achieving the architectural drivers and the business goals of the systems (cf. Chapter 7). For each risk we determined the probability and impact in order to decide on the severity of the risks and, further, on the overall quality of the architecture.

8.3.2 Step 2: Creating the Revised Architectures

The purpose of this phase is to apply the activities of the QUADRAD framework in order to revise the architectures of A, B, and C. During this step the evaluation scenarios and risks identified in the previous step have been used as input for the architecture refinement of the systems. In general, two basic strategies for risk mitigation/resolution are possible:

- *Architecture adaptation:* The architecture is modified such that it will be capable of achieving the requirements (i.e., architectural drivers) correctly.
- *Requirements refinement and architectural adaptation:* The architectural drivers (and potentially business goals) are refined such that the respective impact of the risks is mitigated. The architecture is then changed such that it will be capable of achieving the refined drivers.

Depending on the strategies for addressing the risks different activities of QUADRAD-P and QUADRAD-M have been performed. In addition, QUADRAD-E has been applied in order to analyze the new design decisions. This has been done in mini-evaluations after larger revision iterations of the architecture. The resulting architectures A*, B*, and C* have been used as input for step 3 of the validation.

8.3.3 Step 3: Evaluating the Revised Architectures

The purpose of this step is to analyze the effects of applying QUADRAD activities during step 2. The revised architectures A*, B*, C* have once more been evaluated against the evaluation scenarios gathered in step 1. Through interviews with the lead architects and through application of the QUADRAD-E scenario mapping activities each evaluation scenario has been examined with respect to the (revised) design decisions. In particular, we evaluated if the risks identified in step 1 have been mitigated or resolved. For risks that were not resolved and for new risks we determined their probability and impact in order to calculate the severity of the risks.

8.3.4 Step 4: Comparison

In this step the evaluation results obtained for the original architectures A, B, C (step 1) and the revised architectures A*, B*, C* (step 3) are finally compared. The goal is to determine if the risks have been reduced and if the quality of the architectures has been improved.

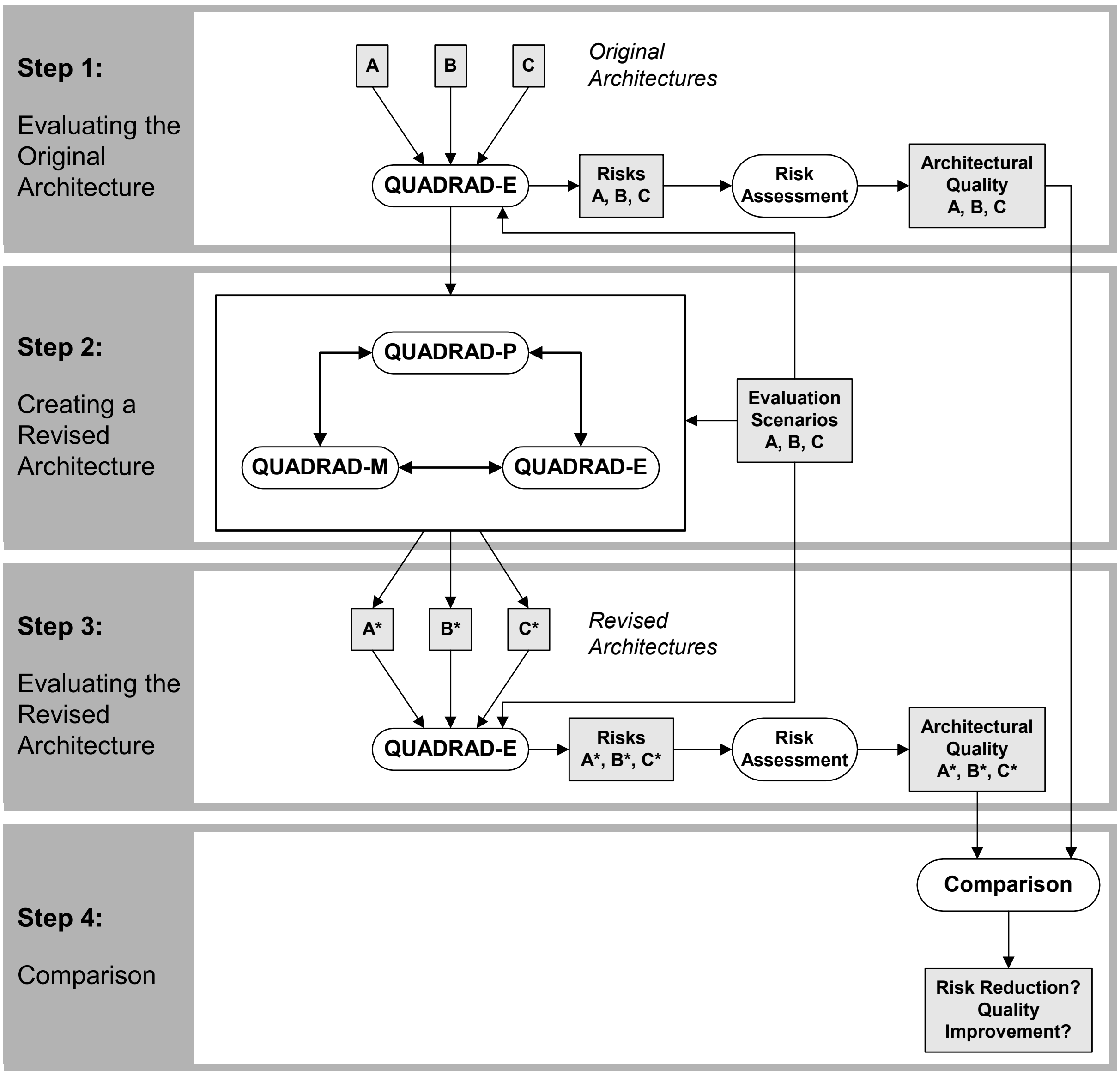

Step 1:
Evaluating the Original Architecture
A
B
C
Original Architectures
QUADRAD-E
Risks A, B, C
Risk Assessment
Architectural Quality A, B, C
Step 2:
Creating a Revised Architecture
QUADRAD-P
QUADRAD-M
QUADRAD-E
Evaluation Scenarios A, B, C
Step 3:
Evaluating the Revised Architecture
A*
B*
C*
Revised Architectures
QUADRAD-E
Risks A*, B*, C*
Risk Assessment
Architectural Quality A*, B*, C*
Step 4:
Comparison
Comparison
Risk Reduction? Quality Improvement?

Figure 8-2: Overview of the Validation Approach

8.4 Validation in Project A

8.4.1 Step 1: Evaluating the Original Architecture A

The purpose of this step is to collect baseline data for the validation of project A. To evaluate the original architecture of A we performed activities of the QUADRAD-E workflow. The evaluation was done in a workshop with the lead architect and key stakeholders of the system. The goal of the evaluation was to analyze how good the individual quality attributes of the system are supported by the current release of the architecture. For this purpose the risks of A as well as the information on probability and impact of each risk have been captured. Table 8-7 summarizes the results of the workshop. Details can be obtained in the next subsections.

Table 8-7: Evaluation Results for the Original Architecture A

Case Study	**Project A**
	Car Multimedia System (CMS)
Evaluation Goals	Quality Achievement
Number of Architectural Drivers	20
Number of Evaluation Scenarios	15
Number of Architectural Risks	19
Average Risk Severity	**3.37 (Medium)**
Architectural Quality	**0.30 (Average)**

8.4.1.1 Architecture Overview

Figure 8-3 shows the basic static structure of the architecture for project A. There are three layers: the user interface layer, the functional layer and the system layer.

The user interface layer is responsible for visualization and component control of the system's application functions. It includes the graphical user interfaces for radio and CD entertainment, navigation, telephone, vehicle comfort, and system options.

The functional layer is responsible for the management of the components. It has no autonomous operation but is invoked either from the system layer or the user interface layer. Notification of activity flows upward to the user interface; control flow comes down to the functional and system layer. The functions are packaged into closely related application groups.

Finally, the system layer manages common behavior from the functional packages. There is no specific abstraction interface for this layer but each component defines its own interface. The components of the system layer realize low-level functions necessary to communicate with connected hardware devices

8.4.1.2 Evaluation Scenarios

Table 8-8 shows the 15 scenarios that have been collected and analyzed during architecture evaluation. The scenarios affect architectural drivers of the system. Important quality attributes of project A are modifiability, portability, performance, security, safety, and reliability. During step 1 of the validation it is evaluated if and to what extent the scenarios can be supported by the original architecture. Note that the scenario descriptions have been simplified and slightly adapted to protect confidentiality.

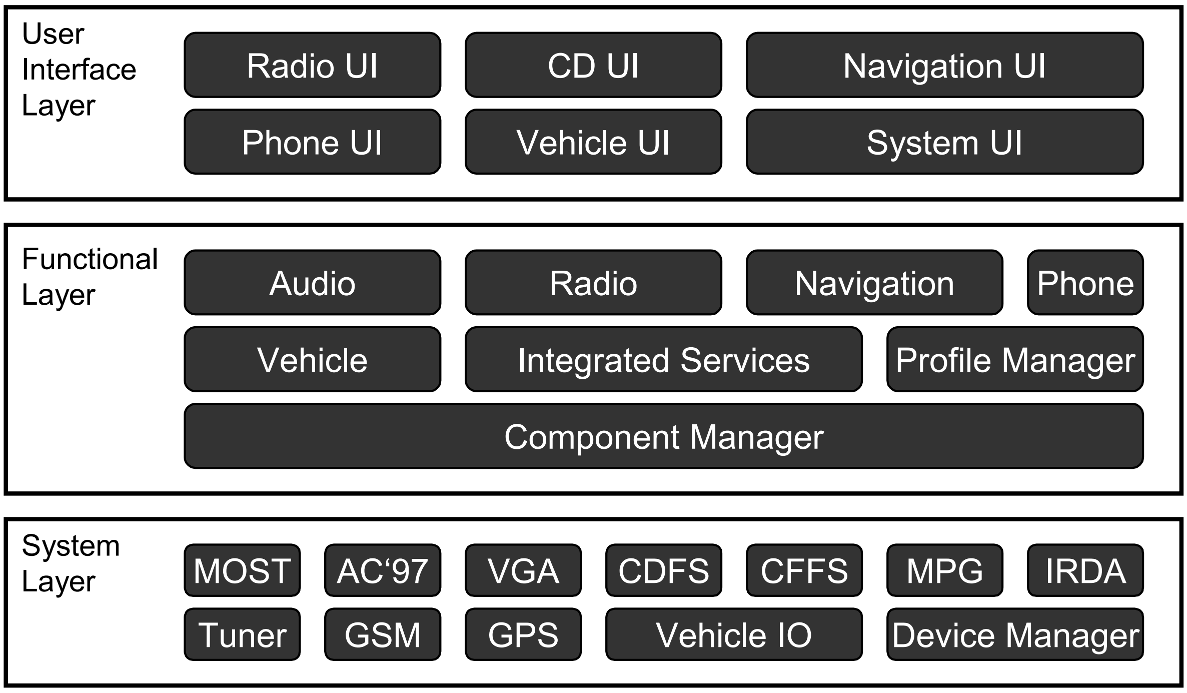


Figure 8-3: Original Architecture of Project A

Table 8-8: Evaluation Scenarios of Project A

ES#	Quality	Scenario Description
ES1	Modifiability	Integrate a new tuner to the system
ES2	Modifiability	Integrate a third party CD device to the system
ES3	Modifiability	Change the user interface to customer A
ES4	Modifiability	Adapt the system to regional requirements of Sweden
ES5	Modifiability, portability	Adapt system to Automotive JavaOS
ES6	Functionality	Mute sound due to an incoming phone call
ES7	Functionality	Change personal profile information
ES8	Functionality	Direct the radio to locate ten stations by speech command
ES9	Functionality	Make an emergency call after an accident
ES10	Functionality, performance	Update navigation map in real-time (the direction of heading must always be at the top of the map)
ES11	Performance	Startup system in less than 0.5 seconds
ES12	Security	An unauthorized user tries to get access to personal data
ES13	Security	A user tries to install/execute uncertified software
ES14	Safety, reliability	Detect and log errors in third party software components
ES15	Safety, reliability	Recover system from device errors

8.4.1.3 Mapping the Scenarios onto the Architecture

An initial step of architecture evaluation with QUADRAD-E is mapping the scenarios onto the architecture (cf. Section 6.4). Mapping scenarios helps exploring the architecture and supports the analysis of associated design decisions.

The scenario maps in Figure 8-4 and Figure 8-5 show examples of project A. They present those components of A that are addressed by the scenarios ES6 and ES9 (cf. Table 8-8), respectively. The path illustrated in each of the maps shows the sequence in which the components are triggered by the corresponding scenario and what activities are carried out in sequence and in parallel. The small circles illustrate design decisions that have been identified during the analysis of the scenario. A cross represents a major responsibility of the component that is triggered by the scenario. In the following a short description of two evaluation scenarios ES6 and ES9 is given.

Scenario ES6: Mute sound due to an incoming phone call.

As illustrated in Figure 8-4 the following activities are performed to support scenario ES6:

1. The scenario starts with the arrival of an incoming phone call. This event occurs in the *Telephone* package of the *Functional Layer*.
2. The *Telephone* package is responsible for notifying the corresponding *Telephone UI* package in the *User Interface Layer* that there is an incoming call.
3. The *Telephone UI* package tells the *Audio Request Broker (ARB)* to make the *WAV Player* play a particular sound file on the speaker device. The waveform file has been previously configured to contain the sound of a ringing telephone.
4. The *ARB* knows the state of the speakers within the vehicle and applies a set of rules that enable it to mute the radio and play the ringing file.
5. The *ARB* sends a message to the *WAV Player* to start playing, another message to the *Audio Control* to switch the source of the signal from the radio to the *WAV Player*, and a third message to the *Radio Player* to stop playing.
6. When these steps are complete, the *ARB* also notifies the *Sound Manager* of the change.
7. The *Sound Manager* forwards the profile of the current user with respect to phone ringing to the *Audio Control* for settings.
8. Now the phone is ringing.

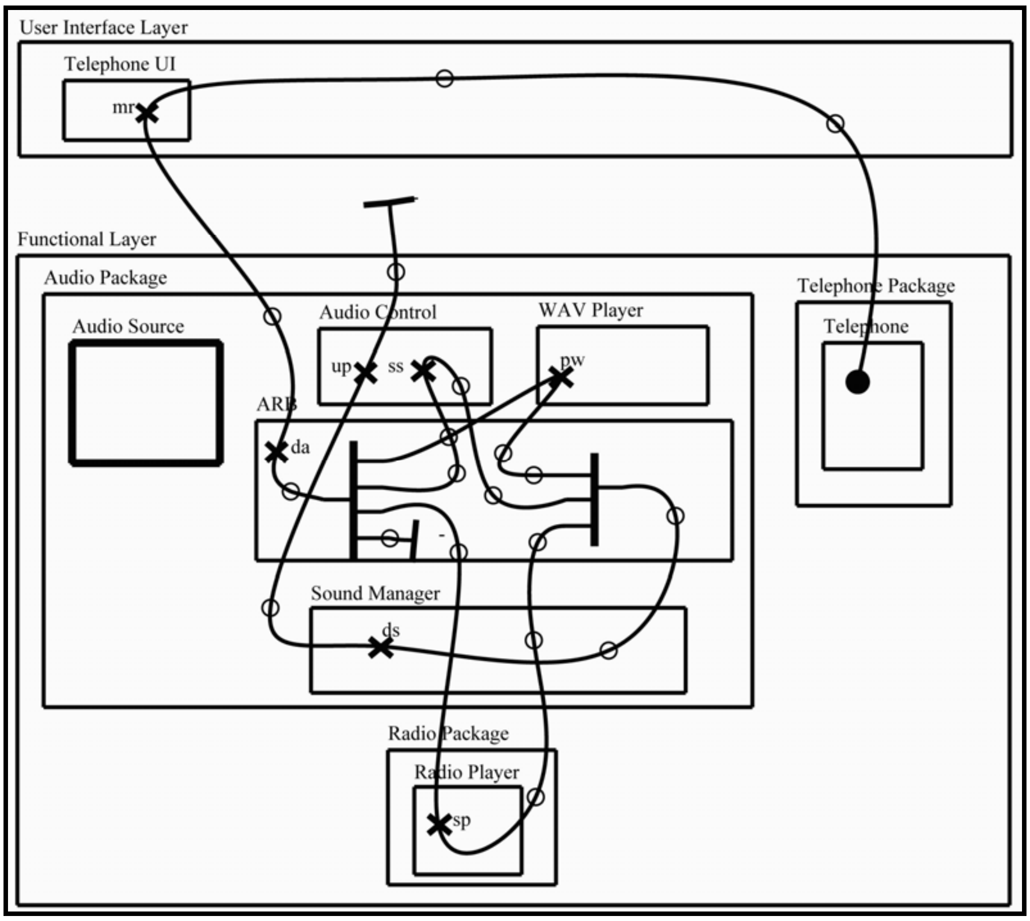


Figure 8-4: Scenario Map for ES6

Scenario ES9: Make an emergency call after an accident

As shown in Figure 8-5 there are two independent control loops to satisfy scenario ES9: one for initialization of the emergency call and one for making the call in case of an accident. Note that the correct working of this scenario depends on the availability of the system after the crash. The course of ES9 is as follows:

1. At system initialization, the *Emergency Call* package gets the information necessary to make an emergency call from the *Personal Profile*. The emergency number must have been configured beforehand.
2. If an accident happens, the *Vehicle* package is the first place in the system that is informed of a crash. It informs the *Emergency Call* package whose responsibility it is to organize the placing of the call.
3. The *Vehicle Tracking* package reports the current location of the vehicle.
4. The *Text-to-Speech* package converts this information into an audio format.
5. The basic *Telephone* package places the call (using the *GSM* driver, which is not shown in Figure 8-5).
6. The *ARB* allocates the phone line for the message, in a way similar to that discussed in scenario ES6.
7. Now the emergency call has been placed.

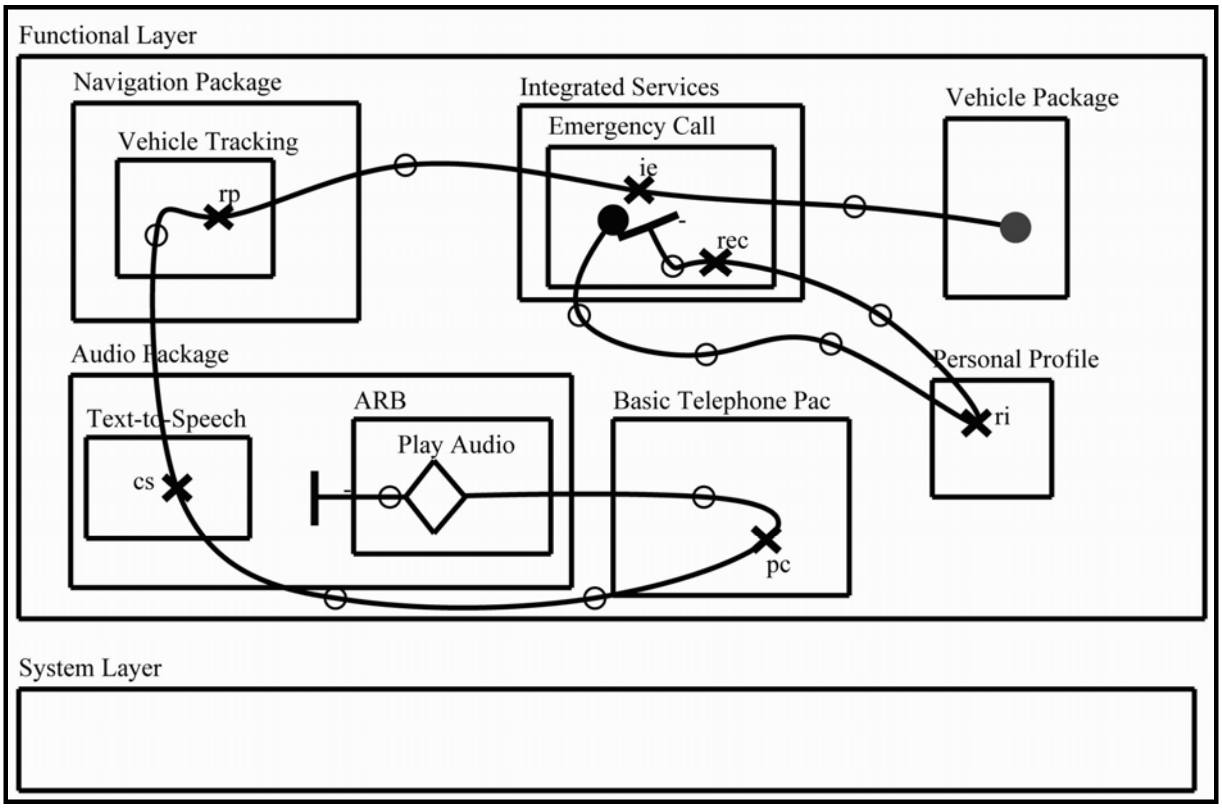


Figure 8-5: Scenario Map for ES9

8.4.1.4 Architectural Risks and Risk Assessment

During the analysis of the evaluation scenarios, 19 architectural risks have been identified in the original architecture of project A. Table 8-9 shows the probability, impact, and resulting ranking of the risks recorded during the risk assessment of A. Table 8-10 gives a detailed description of the effective risks R2, R6, R7, R8, and R14.

Table 8-9: Probability and Impact of the Risks in Project A

Risks of A	Probability	Impact	Ranking
R1	Effective	Low	**3**
R2	Effective	High	**6**
R3	Medium	Low	**1.5**
R4	Medium	Low	**1.5**
R5	High	High	**4.5**
R6	Effective	Very High	**9**
R7	Effective	High	**6**
R8	Effective	High	**6**
R9	Low	Very Low	**0.25**
R10	High	Low	**2.25**
R11	High	Low	**2.25**
R12	Low	Low	**0.75**
R13	High	High	**4.5**
R14	Effective	High	**6**
R15	Low	Low	**0.75**
R16	Low	Low	**0.75**
R17	Medium	Low	**1.5**
R18	Effective	High	**6**
R19	Medium	Low	**1.5**
Total	--	--	**64**

Table 8-10: Architectural Risks of Project A (Extract)

Risks of *A*	**Description**
R2	**Limited security for system access.**
	The *Profile Manager* does not support different levels of access rights. Once authorized to the system the user can potentially access any functionality provided. This is a risk since personal information about the car driver/owner can be obtained and uncertified software can be installed. In addition, particular pay services can be used by unauthorized persons since they are not sufficiently protected.
R6	**No overall system diagnosis for component-specific failures.**
	The availability and correct operation of safety-critical components such as the *Vehicle* component (which enables access to car dynamics) is currently not monitored. If such a component provides wrong data or fails to operate correctly then the integrity and safety of the system cannot be guaranteed. There is no mechanism to gracefully degrade the system to a safe mode.
R7	**Infotainment components depend on the underlying hardware devices.**
	The system's infotainment software in the functional layer is directly coupled to the underlying infotainment hardware devices. If the devices change (e.g., different manufacturer) then the functional components must be changed accordingly. Since there are individual devices for different customers, there is a high software change effort to support those devices. There is no appropriate abstraction that preserves the basic application functionality of the infotainment components to be independent of modified or improved software to access the hardware devices.
R8	**Change intensive adaptation of the look-and-feel to customer demands.**
	The user interface layout and control is encapsulated to software components in the *User Interface* layer. If the look-and-feel needs to be adapted to individual customer requirements then the complete layout and control functionality has to be modified. This requires a high change effort of the User Interface layer from one customer variant to another, even if the application functionality does not change. Since changing the look-and-feel of the system is very likely for different car manufacturers, the high change effort represents an architectural risk.
R14	**System software cannot be updated by user.**
	Currently there is no possibility for a user to update the system's software. Software updates can only be done by service technicians in a certified garage.

According to the metrics M6 and M7 given in Table 8-2, the average risk severity and architectural quality of the original architecture of project A is calculated as follows:

Average risk severity of *A*: $R_S(A) = \frac{1}{19}\sum_{i=1}^{19} \text{Ranking}(r_{A_i}) = \frac{1}{19} \cdot 64 = 3.37$

Architectural quality of *A*: $Q(A) = \frac{1}{R_S(A)} = 0.30$

Figure 8-6 illustrates the risk rankings and the average risk severity of *A* in a bar chart.

The average risk severity of 3.37 leads to an architectural quality of 0.30. According to Table 8-6 the baseline architecture of A has an average quality.

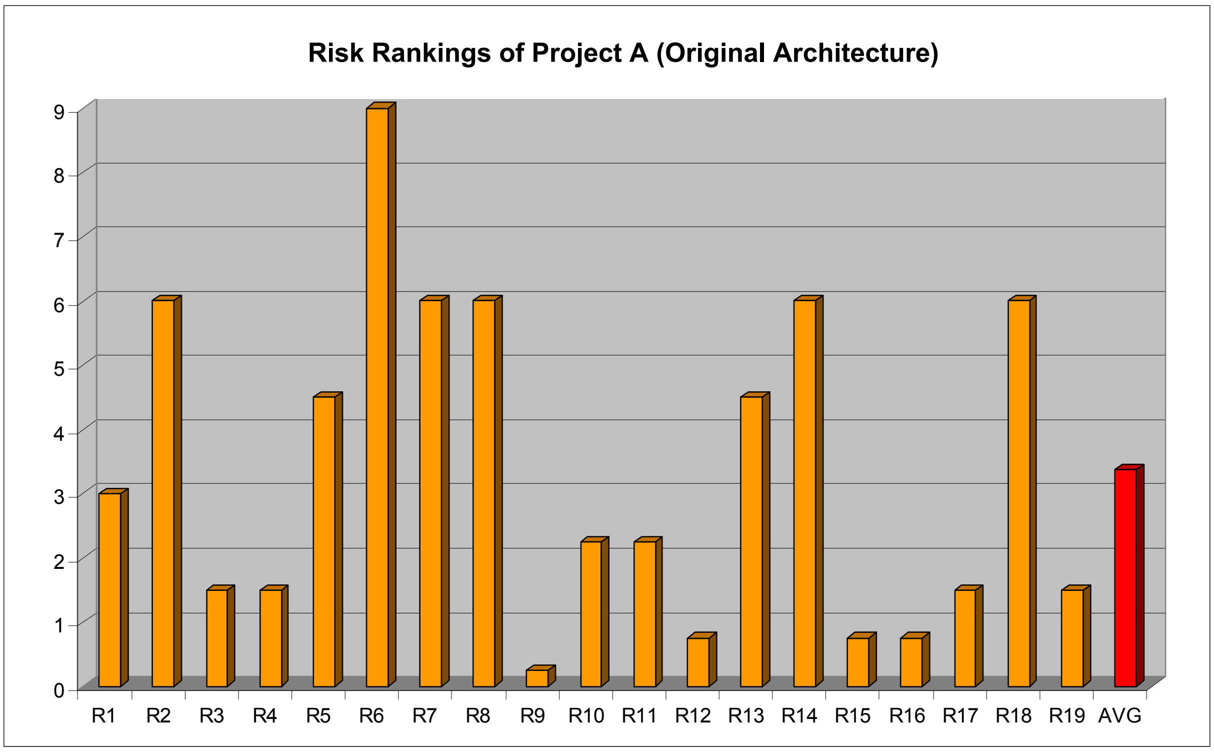


Figure 8-6: Risk Rankings of Project A (Original Architecture)

8.4.2 Step 2: Creating a Revised Architecture A*

Figure 8-7 shows the revised architecture for project A. This architecture has been created as a result of applying QUADRAD activities for addressing the risks of the original architecture. In the following we explain in detail how the original architecture has been adapted to resolve the five risks R2, R6, R7, R8, and R14 described in Table 8-10:

- **R2: Limited security for system access.**

  Architecture adaptation: The revised architecture now supports users with different privileges. This is realized in the *Personal Information Management* and *Resource Management* components (cf. Figure 8-7). User privileges depend on a specific role (e.g., car owner, car driver, service technician, child etc.). According to the role, not every user has the permission to perform the whole set of actions. Certain operations may be critical and affect system integrity (e.g., software installation) or cause costs (e.g., telephone calls or internet access). Hence, they should only be available for authorized users. Private data such as emails and addresses has also to be protected and the system should not grant access to unauthorized persons. Further, the architecture provides mechanisms to prevent unauthorized users from using particular resources (e.g., a telephone) without permission. Figure 8-8 illustrates how the associated components collaborate to resolve R2.

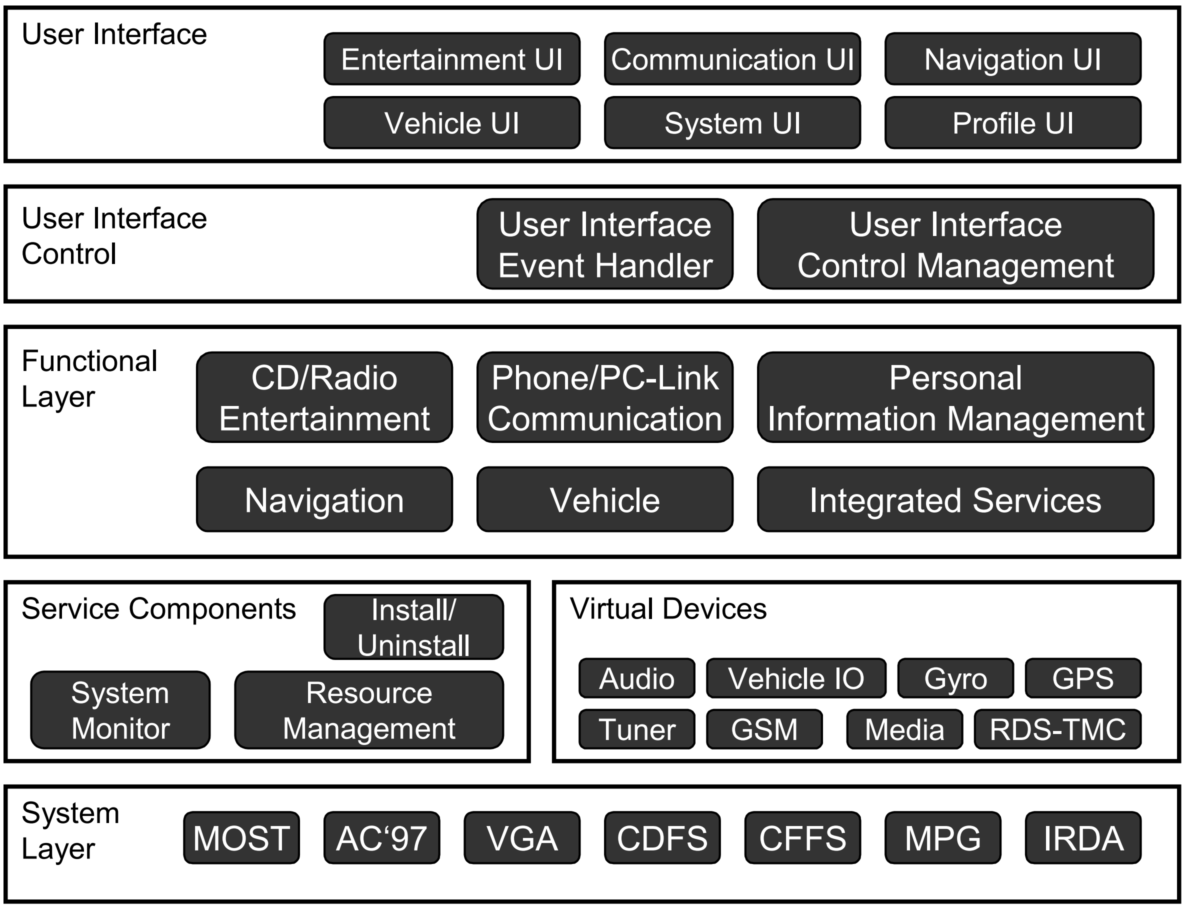


Figure 8-7: Revised Architecture of Project A

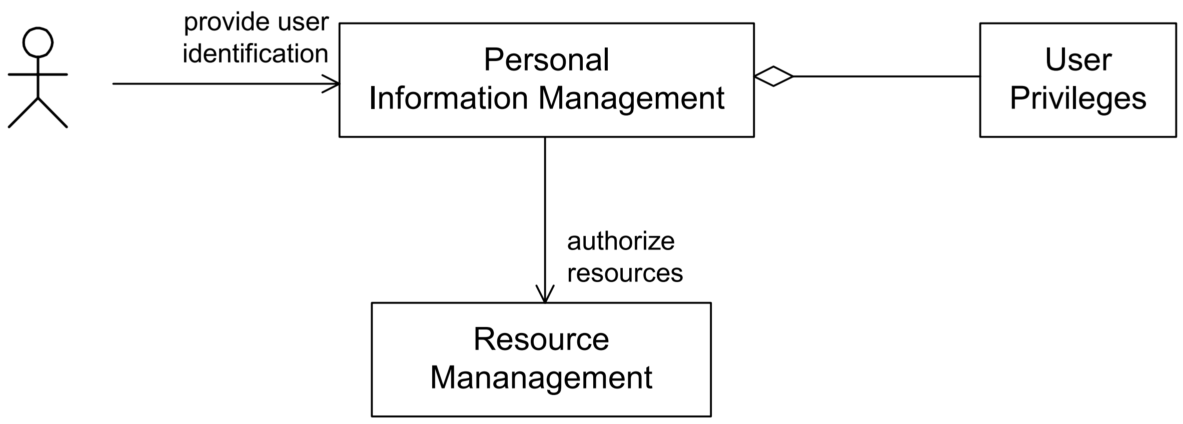


Figure 8-8: Introduction of User Privileges

- **R6: No overall system diagnosis for component-specific failures.**

  Architecture adaptation: Safety-critical software components in the revised architecture are supervised and provide quality-of-service information. Each safety-critical component now contains basic error management features. The components integrate operations to recognize, reason, report, and recover from certain errors in services they provide. They communicate with the core *System Monitor* component (cf. Figure 8-7) which includes the data storage and provides the necessary functionality to enable/disable safety-critical services, check if the required service is available, and recognize failures in the responses of a service. Any particular safety-critical component is responsible for providing quality-of- service information to the *System Monitor* over a standardized interface. The *System Monitor* then decides if a particular system service needs to be disabled. It is also capable to switch the system in a safe mode with limited capabilities gracefully. Figure 8-9 depicts the system context to resolve R6.

- **R7: Infotainment components depend on the underlying hardware devices.**

  Architecture adaptation: A *Virtual Device* layer is introduced in the revised architecture (cf. Figure 8-7). The *Virtual Device* layer provides an abstraction of the underlying hardware devices by exposing a standardized interface to the functional infotainment components. This solution allows the application functionality of the entertainment software to be independent from hardware changes or updates. The virtual devices implement the low-level access to the hardware devices. If a particular device is changed to support individual customer requirements then this change only affects the associated virtual device software component. The standardized interface guarantees that the changes in the virtual device layer do not affect functional components. Additionally, the interface allows functional components to check for the hardware capabilities supported by the devices. New and improved capabilities can thus be directly adopted in the functional components and reflected in the user interface to some extent.

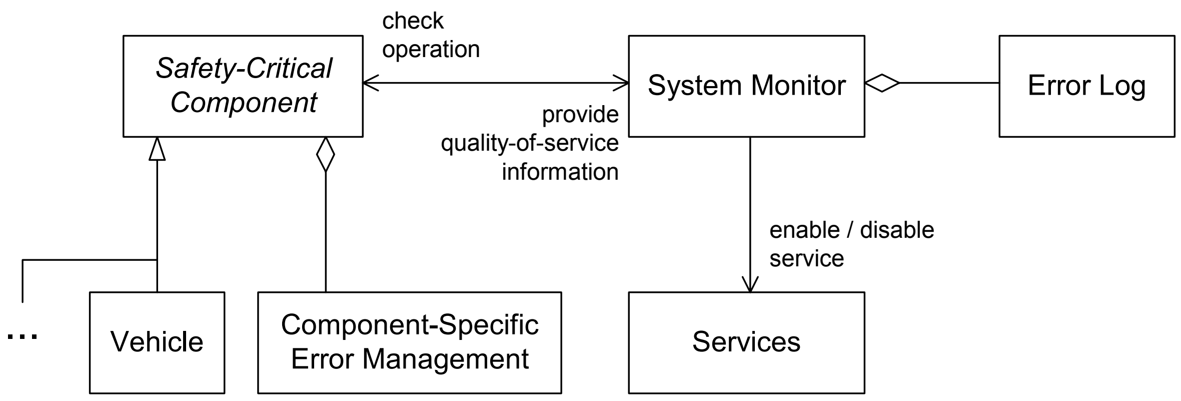

check operation
provide quality-of-service information
Safety-Critical Component
System Monitor
Error Log
enable / disable service
...
Vehicle
Component-Specific Error Management
Services

Figure 8-9: Supervision of Safety-Critical Components

- **R8: Change intensive adaptation of the look-and-feel to customer demands.**

  Architecture adaptation: The revised architecture encapsulates the user interface controls and interactions. For this purpose, a *User Interface Control* layer is introduced in the architecture (cf. Figure 8-7). This layer provides the control and interaction logic for the different User Interfaces. The interaction of the user with the system covers the tactile and visible interaction through the display and panel as well as speech interaction (e.g., command-and-control) and the instrument cluster display combined with the steering wheel buttons. For the tactile and graphical user interface, it also includes the customer-specific operating concept and associated interaction sequences. This allows an effective adaptation of the system's look-and-feel (which is very specific to the individual car manufacturers) to the underlying application functionality.

- **R14: System software cannot be updated by user.**

  Architecture adaptation: The revised architecture includes an *Install/Uninstall* component (cf. Figure 8-7) that realizes the update of system features. The *Install/Uninstall* component is responsible for evaluating the certificate provided by any new software component during installation. It manages the dependencies stated in the certificate and makes sure that the installation of the new software is compatible to the current installation (cf. Figure 8-10). In case of problems, it is capable of aborting the installation procedure and of performing a roll back to the last safe state of the system.

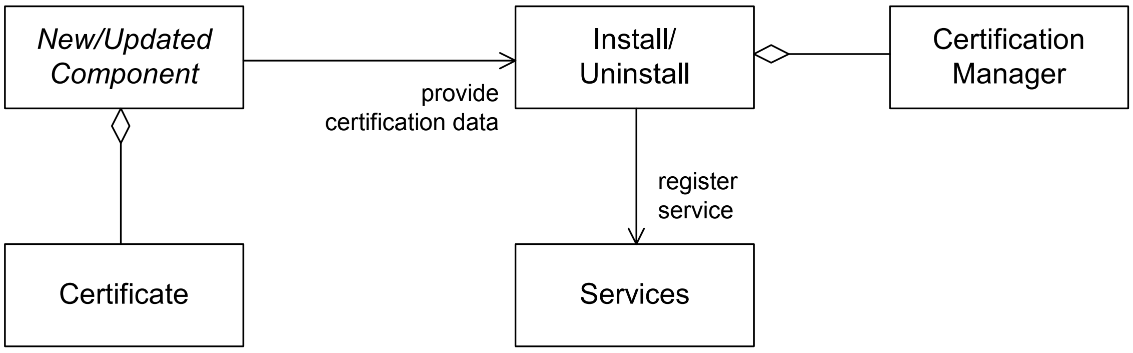


Figure 8-10: Integration of a Software Update Capability

8.4.3 Step 3: Evaluating the Revised Architecture A*

In this step the revised architecture of project A was evaluated. For this purpose, the risk mitigation strategies undertaken in the previous step were analyzed during interviews with the stakeholders. For each risk discovered during step 1, the architect explained how he had addressed the risk (see description of architecture adaptations given in Section 8.4.2). To validate the effect of the risk mitigation/resolution strategies for project A, the evaluation scenarios gathered in step 1 (cf. Table 8-8) were analyzed again.

As shown in Table 8-11, the 14 risks R1, R2, R3, R4, R6, R7, R8, R10, R11, R12, R13, R14, R17, and R19 could be resolved through an adaptation of the architecture in step 2 of the validation. Risk R5 was resolved through a change of architectural drivers. The four risks R9, R15, R16, and R18 included in the original architecture of project A could not be addressed during step 2.

In addition, a new risk R20 has been identified (see Table 8-12). This risk has its root cause in the new design decisions made in the revised architecture A*. The new risk is related to the fact that the flexible look-and-feel adaptation leads to higher memory consumption. Higher memory consumption may lead to the case that more RAM and, potentially, a more powerful micro processor needs to be provided by the system. This could lead to higher hardware costs and thus represents a risk. Detailed market and system investigations show that it is not very probable that this risk will become effective in project A (see Table 8-12). Thus, the severity of the remaining risk is non-critical. However, if the market and business changes significantly and more features are required for A, the probability of the risk may increase. Then, a further revision of architecture A* would make sense to preserve business success.

Table 8-11: Risks of Project A (Original vs. Revised Architecture)

Risks of A	Ranking Original Architecture	Architecture Revision Activity	New Risk Status	Ranking Revised Architecture
R1	3	Architecture Adaptation	Risk Resolved	0
R2	6	Architecture Adaptation	Risk Resolved	0
R3	1.5	Architecture Adaptation	Risk Resolved	0
R4	1.5	Architecture Adaptation	Risk Resolved	0
R5	4.5	Refinement & Adaptation	Risk Resolved	0
R6	9	Architecture Adaptation	Risk Resolved	0
R7	6	Architecture Adaptation	Risk Resolved	0
R8	6	Architecture Adaptation	Risk Resolved	0
R9	0.25	--	**Risk Unresolved**	0.25
R10	2.25	Architecture Adaptation	Risk Resolved	0
R11	2.25	Architecture Adaptation	Risk Resolved	0
R12	0.75	Architecture Adaptation	Risk Resolved	0
R13	4.5	Architecture Adaptation	Risk Resolved	0
R14	6	Architecture Adaptation	Risk Resolved	0
R15	0.75	--	**Risk Unresolved**	0.75
R16	0.75	--	**Risk Unresolved**	0.75
R17	1.5	Architecture Adaptation	Risk Resolved	0
R18	6	--	**Risk Unresolved**	6
R19	1.5	Architecture Adaptation	Risk Resolved	0
Total	64	--	--	7.75

Table 8-12: New Risks of Project A (Revised Architecture)

Risks of $A*$	Probability	Impact	Ranking
R20	Medium	High	**3**
Total	--	--	**3**

According to the metrics M11 and M12 given in Table 8-2, the average risk severity and architectural quality of the revised architecture of project A is calculated as follows:

Average risk severity of A*: $R_S(A*) = \frac{1}{5}\sum_{i=1}^{5}\text{Ranking}(r_{A*_i}) = \frac{1}{5}\cdot(7.75+3) = 2.15$

Architectural quality of A*: $Q(A*) = \frac{1}{R_S(A*)} = 0.47$

Figure 8-11 illustrates the risk rankings and the average risk severity of the revised architecture A*.

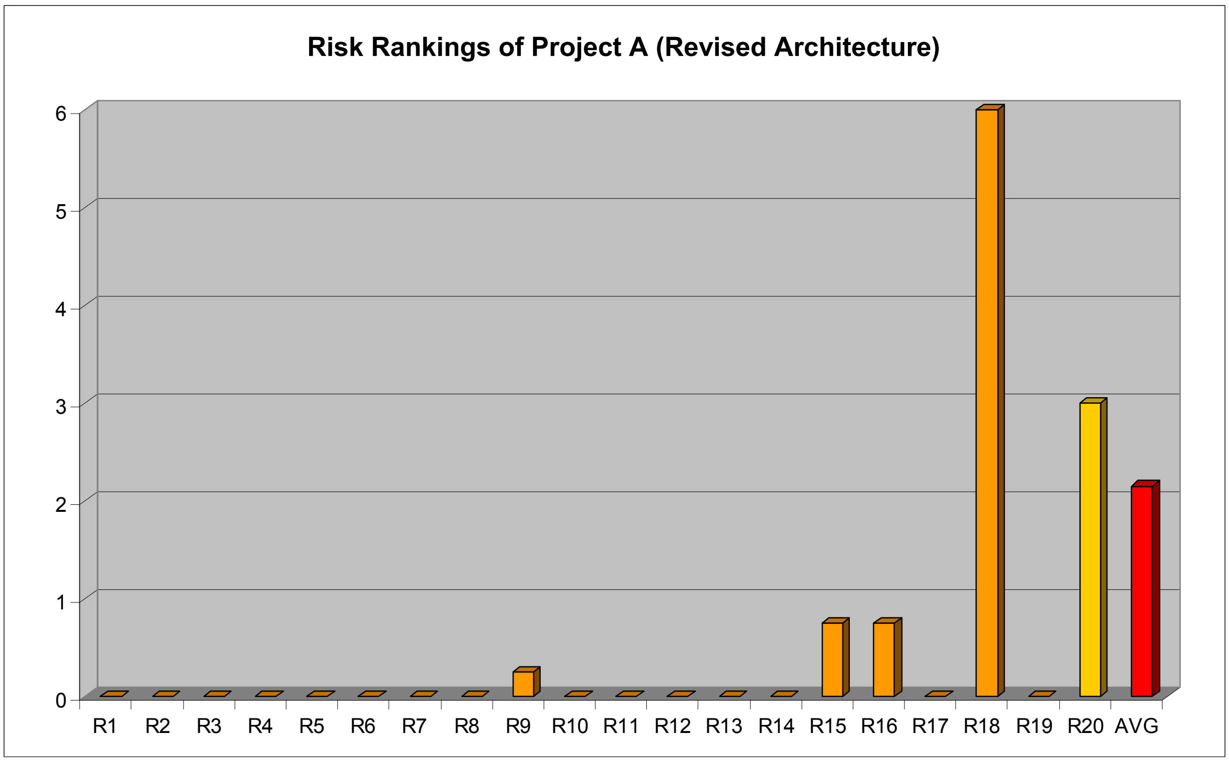


Figure 8-11: Risk Rankings of Project A (Revised Architecture)

8.4.4 Step 4: Comparison

According to the metric M13 of Table 8-2 the quality of the revised architecture A^* is *better* than that of the baseline architecture A:

$$Q(A^*) = 0.47 > 0.3 = Q(A)$$

74% of the risks in A could be reduced in A^*:

$$1 - \frac{Number\ of\ Risks(A^*)}{Number\ of\ Risks(A)} = 1 - \frac{5}{19} = 0.74$$

The quality improvement between architecture A and A^* is calculated as:

$$1 - \frac{Q(A)}{Q(A^*)} = 1 - \frac{0.3}{0.47} = 0.36$$

As a result, the architecture quality of project A could be improved by 36%. Table 8-13 summarizes the validation results.

Table 8-13: Validation Results of Project A

Case Study	**Project A (CMS)**	
	Original Architecture	Revised Architecture
Number of Risks	**19**	**5**
Average Risk Severity	**3.37 (Medium)**	**2.15 (Low)**
Architecture Quality	**0.30 (Average)**	**0.47 (Good)**
Risk Reduction	**74%**	
Architectural Quality Improvement	**36%**	

8.5 Validation in Project B

8.5.1 Step 1: Evaluating the Original Architecture B

Table 8-14 summarizes the results of evaluating the original architecture of project B. Again, the evaluation was done in a workshop with the lead architect and key stakeholders of the system by applying QUADRAD-E activities. The goal of the evaluation was to analyze how well the individual quality attributes of the system are supported by the current release of the architecture. The risks of B as well as the information on probability and impact of each risk have been captured. Evaluation details can be obtained in the next three subsections.

Table 8-14: Evaluation Results for the Original Architecture B

Case Study	**Project B**
	Automotive Assistance System (AAS)
Evaluation Goals	Quality Achievement
Number of Architectural Drivers	8
Number of Evaluation Scenarios	15
Number of Architectural Risks	13
Average Risk Severity	**3.81 (Medium)**
Architectural Quality	**0.26 (Average)**

8.5.1.1 Architecture Overview

Figure 8-12 illustrates the basic static structure of the original architecture for project B. There are three layers: the user application layer, the system layer and the Real-Time Operating System layer.

The user application layer is responsible for combining sensor data in order to provide high-level applications such as an intelligent pre-crash detection (PCD), blind-spot detection (BSD), side object detection (SOD), and parking assistance (PA). PCD provides collision object information to the restraint system. The information is used to trigger reversible and adapt thresholds for irreversible restraint actuators. BSD and SOD notify the driver of objects on the side of the vehicle and within the blind spot area of the outside mirrors, respectively. PA provides information about parking spots of sufficient size and assists maneuvering into the parking space.

The system layer realizes access to sensor devices and coordinates applications. It receives raw sensor data, pre-processes it, and provides cooked data to the application components. Moreover, the system layer provides mechanisms for error detection and recovery.

Finally, the real-time layer encapsulates the Real-Time Operating System which covers the task and memory management, handling of interrupts, and basic input/output capabilities.

8.5.1.2 Evaluation Scenarios

Table 8-15 shows the 15 scenarios used for analyzing the architecture of project B. The scenarios probe significant architecturally driving requirements of the system. As depicted in Table 8-15, safety, reliability, security, interoperability, and modifiability are important quality attributes of the system. During architecture analysis it is evaluated whether and to what extent the scenarios can be supported by the original architecture.

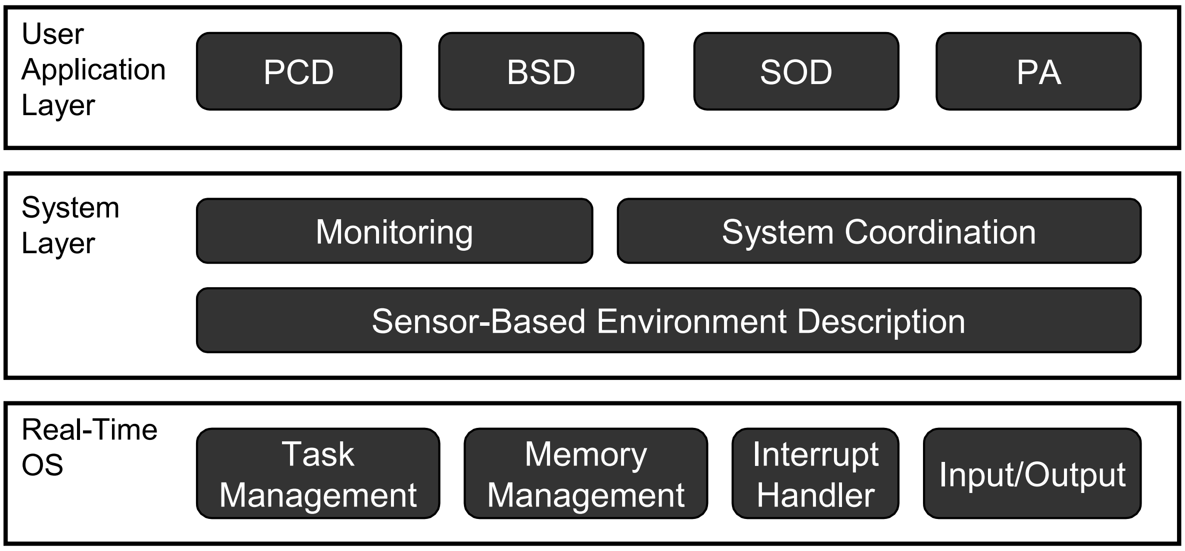


Figure 8-12: Original Architecture of Project B

Table 8-15: Evaluation Scenarios of Project B

ES#	Quality	Scenario Description
ES1	Safety	Detect snow on sensor and inform driver about this situation.
ES2	Safety	Detect malfunction of a sensor.
ES3	Safety	Detect sensor that is not correctly mounted on the system.
ES4	Reliability	Use ultrasound sensors as a second source for PA applications.
ES5	Security	Detect unauthorized modifications of the sensor software.
ES6	Functionality	Backing the car into an appropriate parking space.
ES7	Functionality	Two objects approaching from left and right shall be correctly detected by PCD.
ES8	Functionality, cost	Switch to an economic power mode to lower battery consumption.
ES9	Performance, functionality	Detect correct situation when PA and PCD applications run concurrently.
ES10	Interoperability	Integrate system in a car with CAN bus environment.
ES11	Interoperability	Integrate system in vehicle with a restraint system that supports a non-standard communication protocol.
ES12	Modifiability	Configure system with PA and PCD at EOL (end-of-line) in the production facility.
ES13	Modifiability	Update software of sensor device during system operation.
ES14	Modifiability	Integrate a new sensor in the system.
ES15	Modifiability	Replace the PA user interface with one that has animated graphics.

8.5.1.3 Architectural Risks and Risk Assessment

For project B 13 architectural risks have been identified during the architecture evaluation. Table 8-16 shows the probability, impact, and resulting ranking of the risks recorded during the risk assessment of B. Table 8-17 gives a detailed description of the risks R2, R3, R5, R10, R11, and R12.

According to the metrics M6 and M7 given in Table 8-2, the average risk severity and architectural quality of the baseline architecture of project B is calculated as follows:

Average risk severity of *B*: $$R_S(B) = \frac{1}{13}\sum_{i=1}^{13}\text{Ranking}(r_{B_i}) = \frac{1}{13}\cdot 49.5 = 3.81$$

Architectural quality of *B*: $$Q(B) = \frac{1}{R_S(B)} = 0.26$$

Figure 8-13 illustrates the risk rankings and the average risk severity of *B* in a bar chart. The average risk severity of 3.81 leads to an architectural quality of 0.26. According to Table 8-14 the original architecture of B has an average quality.

Table 8-16: Probability and Impact of the Risks in Project B

Risks of *B*	Probability	Impact	Ranking
R1	Low	Low	**0.75**
R2	Effective	High	**6**
R3	Effective	High	**6**
R4	Medium	High	**3**
R5	Effective	High	**6**
R6	Effective	Low	**3**
R7	Low	Low	**0.75**
R8	Medium	Low	**1.5**
R9	High	High	**4.5**
R10	Effective	High	**6**
R11	Effective	High	**6**
R12	Effective	Low	**3**
R13	Medium	High	**3**
Total	--	--	**49.5**

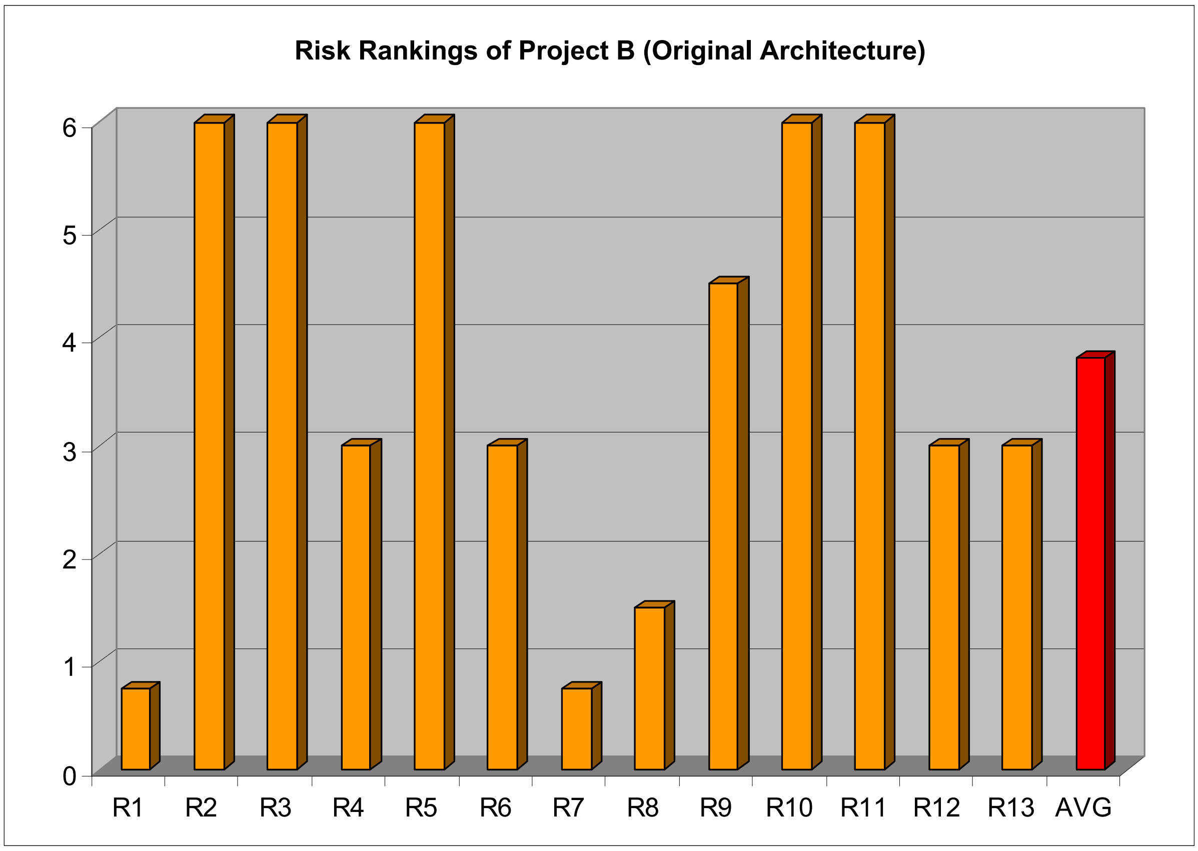


Figure 8-13: Risk Rankings of Project B (Original Architecture)

Table 8-17: Architectural Risks of Project B (Extract)

Risks of *B*	**Description**
R2	**Limited sensor data quality for particular applications.**
	Some applications of the system require high quality sensor data in order to make reliable decisions for performing a particular action. In a worst-case scenario, the sensor devices may be fully covered by dust or snow, which results in low quality sensor data. The risk is that some applications of the system may not operate in an optimal way because of fundamental errors in the sensor data.
R3	**Limited power management capabilities.**
	The current architecture does not provide enhanced power management capabilities. Most of the time the devices connected to the system are fully operational. Particular applications cannot be physically switched off. The risk is that the system will drain the battery too quickly. Low battery voltage is especially critical during system startup.
R5	**Time-critical events may not be handled in real-time.**
	The current architecture does not allow the complete control of time-critical operations. There are situations where real-time operations may be pre-empted by non-real-time tasks, which might lead to ephemerally undefined states or unhandled events. This represents a performance risk of the architecture.
R10	**Different types of sensors are not supported.**
	The raw data from the installed sensors is managed and prepared by the *Sensor-Based Environment Description* component. This component provides cooked data (e.g., object lists) to the application components for further processing. Currently, only one type of sensor is supported by the component. The risk is that not all provided applications of the system could be reliably realized by the single sensor type that is supported.
R11	**Limited software update capability for sensors and control unit.**
	The original architecture does not directly support software updates for the control unit and the installed sensor devices. Currently, software updates can only be managed by external flash applications. These applications require the unmounting of sensors from the system. This represents a risk. In particular, software updates for sensors become a time-consuming task.
R12	**Defect sensor device not detected during system start-up.**
	The *Monitoring* component of the original architecture does not support device error detection during system initialization. The risk is that a defect sensor cannot be reliably detected unless an application has subscribed for sensor data associated to the defect device. Since this is done during the initialization of the applications, the risk can be treated as non-critical.

8.5.2 Step 2: Creating a Revised Architecture B*

Figure 8-14 shows the revised architecture for project B after several iterations of QUADRAD have been performed. In the following we explain in detail how the architecture has been adapted to resolve the six risks R2, R3, R5, R10, R11, and R12 described in Table 8-17:

- **R2: Limited sensor data quality for particular applications.**

  Architecture adaptation: The revised architecture provides mechanisms to obtain the best quality sensor data possible based on a particular sensor configuration. In case one or more sensors have malfunctions, the *Environment Description* component (cf. Figure 8-14) uses related sensors, sensor fusion, and smart signal processing algorithms to improve the data quality. This improves the overall system quality significantly and leads to optimal decisions of the application components. The *Environment Description* component also improves the safety of the system. If the resulting data quality is below a specified threshold then the component takes care that the system is set to a degraded mode, which is then reported to the driver. Sensor data quality checks are, for example, done during cyclic system tests, which are performed by the *Monitoring & Self Test* component (cf. Figure 8-14). Figure 8-15 illustrates how the associated components collaborate to resolve R2.

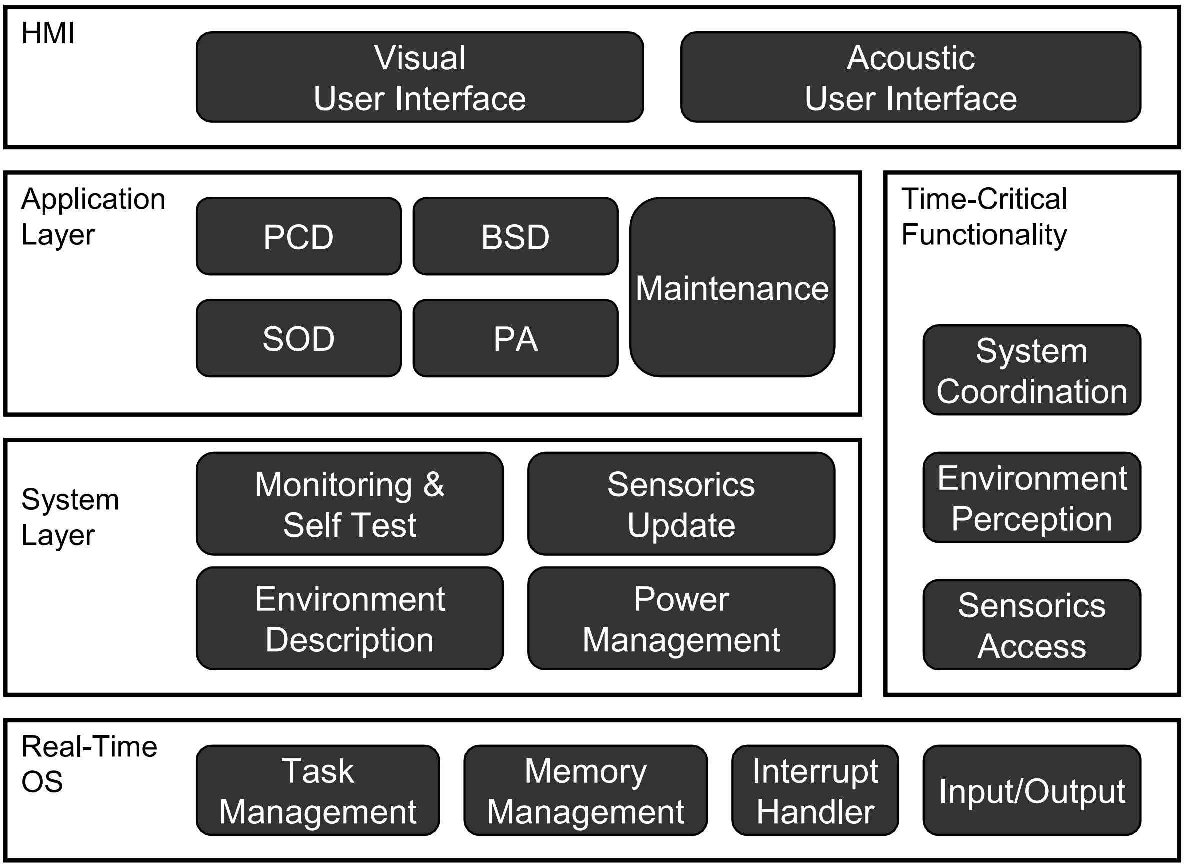


Figure 8-14: Revised Architecture of Project B

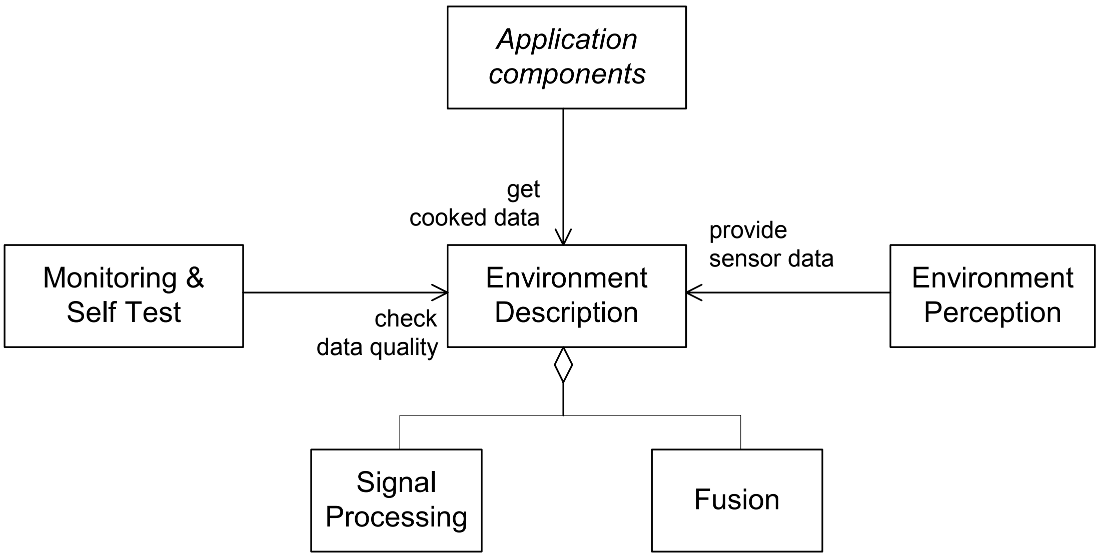


Figure 8-15: Improved Environment Description for High-Quality Sensor Data

- **R3: Limited power management capabilities.**

  Architecture adaptation: The revised architecture realizes system-wide intelligent power management. This guarantees an efficient battery usage. The *Power Management* component (cf. Figure 8-14) manages initialization, activation, and deactivation of applications. Power-controlled applications and associated devices can be set to one of three states: *off*, *standby*, or *active*. The *active* state has the subordinate states *running* and *waiting*. The state transitions are shown in Figure 8-16. The application components register for particular sensor data types. Thus, they are capable of reacting on appropriate external events.

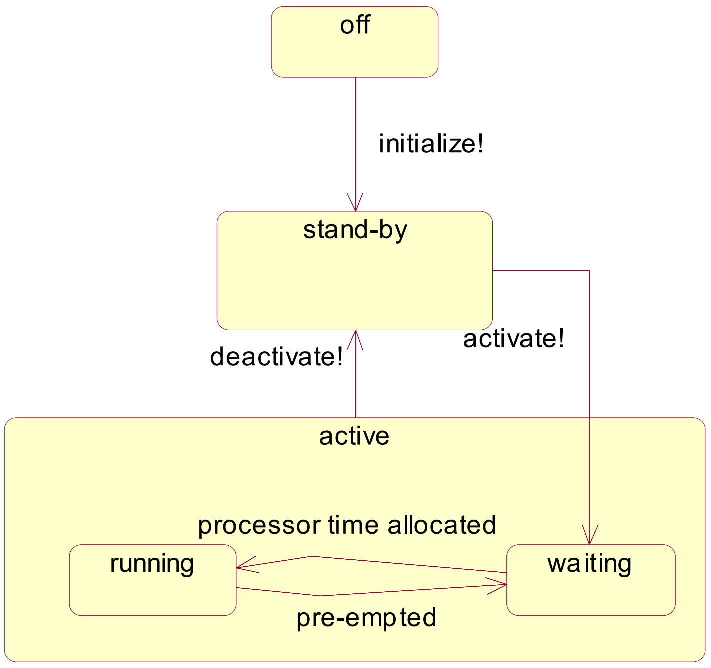


Figure 8-16: State Transitions for Power-Controlled Applications

- **R5: Time-critical events may not be handled in real-time.**

  Architectural adaptation: The revised architecture provides a slice that encapsulates time-critical functionality (cf. Figure 8-14). Time-critical functions must guarantee response times below 50 milliseconds. In the revised architecture, time-critical operations are clearly separated from other operations in the data processing chain. The components *System Coordination*, *Environment Perception*, and *Sensorics Access* (cf. Figure 8-14) include operations that have short cycle times. One example for a time-critical operation is changing the sensor mode (e.g., from closing velocity to distance mode) which is realized in the *Sensorics Access* component. An appropriate response time is guaranteed by assigning fine-grained task priorities and an adequate scheduling policy.

- **R10: Different types of sensors are not supported.**

  Requirements refinement: The original architectural driver was refined to exclude video sensors from the architecture.

  Architecture adaptation: The revised architecture includes a component *Sensorics Access* (cf. Figure 8-14) which is capable of receiving raw data from multiple sensor types such as ultrasonic and microwave sensors. *Sensorics Access* includes the low-level sensor communication functionality. This functionality has been removed and extended from the *Sensor-Based Environment Description* component of the original architecture (cf. Figure 8-12). *Sensorics Access* controls and coordinates multiple sensor devices and performs the signal processing associated with those sensors. The sensor data is further processed by the *Environment Perception* component. This component is responsible for determining the quality of a particular sensor and for managing the subscription of environment information in specific ranges around the vehicle. Figure 8-17 illustrates how the associated components collaborate to resolve R10.

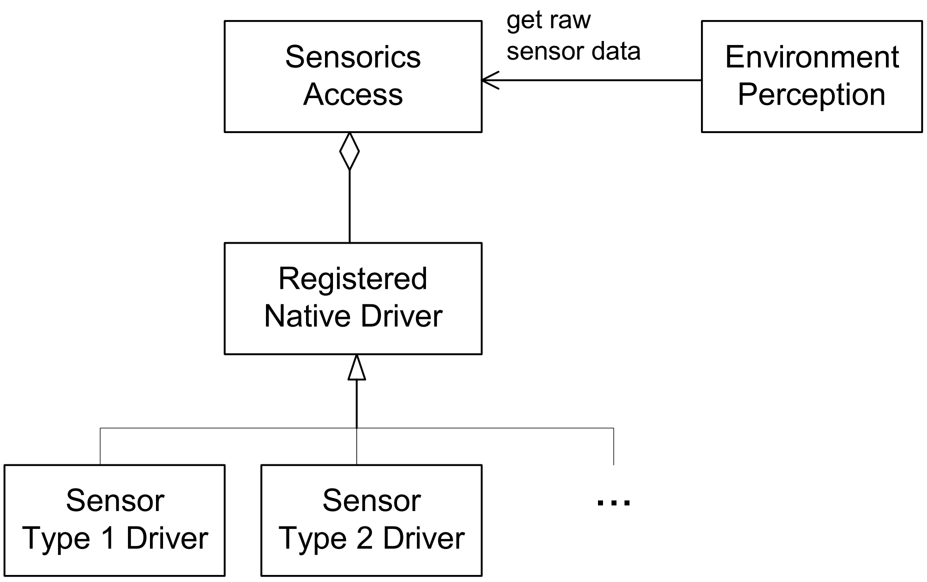


Figure 8-17: Multi Sensor Type Support

- **R11: Limited software update capability for sensors and control unit.**

  Requirements refinement: The original architectural driver has been refined to support updates of the sensor software only.

  Architecture adaptation: The revised architecture provides two components to address the risk: *Sensor Update* and *Maintenance.* The components enable and guide the installation of updated software for installed sensor devices (cf. Figure 8-14). *Sensor Update* performs the flashing of new software functionality for sensors. *Maintenance* controls the update process, validates the correctness of the software update, and supports sensor calibration. Figure 8-18 illustrates how the associated components collaborate to resolve R11.

- **R12: Defect sensor device not detected during system start-up.**

  Architectural adaptation: The revised architecture includes an improved *Monitoring & Self-Test* component (cf. Figure 8-14) which integrates watchdog functionality. This component is capable of reporting device errors during system start-up. It initiates self-tests of the installed devices and analyzes the quality of the sensor data. The quality analysis is realized by an algorithm that compares the raw data received by the sensors against standard quality profiles. Sensor malfunctions or defects can thus be detected reliably and reported to the driver instantly.

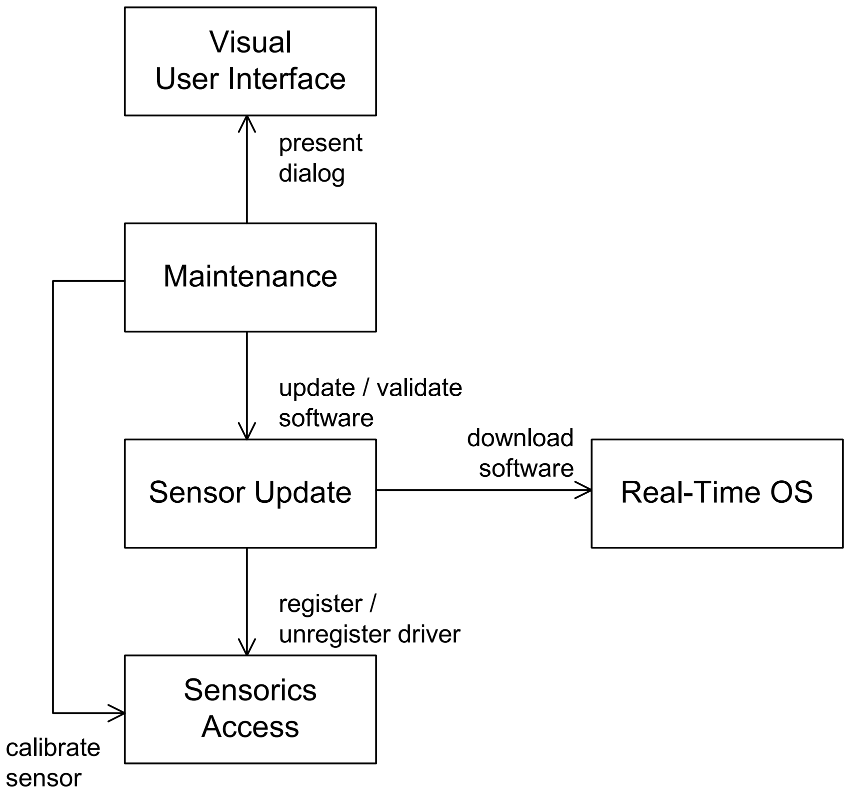


Figure 8-18: Software Update Capability for Sensor Devices

8.5.3 Step 3: Evaluating the Revised Architecture B*

Table 8-18 shows the results of evaluating the revised architecture of project B. The eight risks R2, R3, R4, R5, R6, R7, R9, and R12 could be resolved through architectural adaptation in step 2. The three risks R10, R11, and R13 have been resolved through a reconsideration of business goals and architectural drivers. Two risks (R1 and R8) could not be resolved in the revised architecture.

Table 8-18: Risks of Project B (Original vs. Revised Architecture)

Risks of B	Ranking Original Architecture	Architecture Revision Activity	New Risk Status	Ranking Revised Architecture
R1	**0.75**	--	**Risk Unresolved**	**0.75**
R2	**6**	Architecture Adaptation	Risk Resolved	**0**
R3	**6**	Architecture Adaptation	Risk Resolved	**0**
R4	**3**	Architecture Adaptation	Risk Resolved	**0**
R5	**6**	Architecture Adaptation	Risk Resolved	**0**
R6	**3**	Architecture Adaptation	Risk Resolved	**0**
R7	**0.75**	Architecture Adaptation	Risk Resolved	**0**
R8	**1.5**	--	**Risk Unresolved**	**1.5**
R9	**4.5**	Architecture Adaptation	Risk Resolved	**0**
R10	**6**	Refinement & Adaptation	Risk Resolved	**0**
R11	**6**	Refinement & Adaptation	Risk Resolved	**0**
R12	**3**	Architecture Adaptation	Risk Resolved	**0**
R13	**3**	Refinement & Adaptation	Risk Resolved	**0**
Total	**49.5**	--	--	**2.25**

According to the metrics M11 and M12 given in Table 8-2, the average risk severity and architectural quality of the revised architecture of project B is calculated as follows:

Average risk severity of *B:** $R_S(B^*) = \frac{1}{2}\sum_{i=1}^{2}\text{Ranking}(r_{B^*_i}) = \frac{1}{2}\cdot 2.25 = 1.13$

Architectural quality of *B:** $Q(B^*) = \frac{1}{R_S(B^*)} = 0.88$

Figure 8-19 illustrates the risk rankings and the average risk severity of the revised architecture *B**.

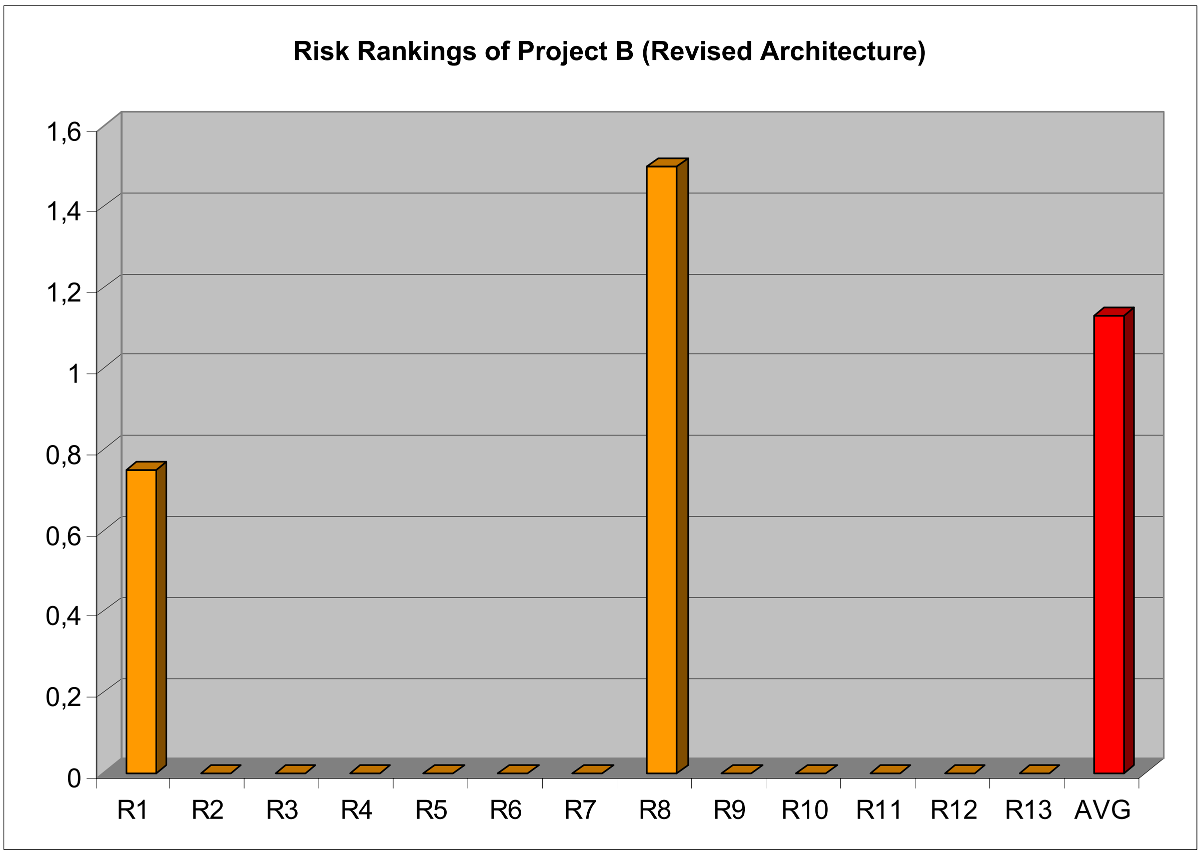


Figure 8-19: Risk Rankings of Project B (Revised Architecture)

8.5.4 Step 4: Comparison

According to the metric M13 of Table 8-2 the quality of the revised architecture $B*$ is *better* than that of the baseline architecture B:

$$Q(B^*) = 0.88 > 0.26 = Q(B)$$

85% of the risks in B could be reduced in $B*$:

$$1 - \frac{\mathit{Number\ of\ Risks}(B^*)}{\mathit{Number\ of\ Risks}(B)} = 1 - \frac{2}{13} = 0.85$$

The quality improvement between architecture B and $B*$ is calculated as:

$$1 - \frac{Q(B)}{Q(B^*)} = 1 - \frac{0.26}{0.88} = 0.7$$

As a result, the architecture quality of project B could be improved by 70%. Table 8-19 summarizes the validation results.

Table 8-19: Validation Results of Project B

Case Study	**Project B (AAS)**	
	Original Architecture	Revised Architecture
Number of Risks	**13**	**2**
Average Risk Severity	**3.81 (Medium)**	**1.13 (Low)**
Architecture Quality	**0.26 (Average)**	**0.88 (Good)**
Risk Reduction	**85%**	
Architectural Quality Improvement	**70%**	

8.6 Validation in Project C

8.6.1 Step 1: Evaluating the Original Architecture C

The evaluation of project C's original architecture was done in a workshop with the lead architect together with project stakeholders. Similarly to the validation of project A and B, we applied QUADRAD-E activities to analyze how well the individual quality attributes of the system are supported by the current release of the architecture. Again, the risks involved in C as well as the information on probability and impact of each risk have been captured. Table 8-20 summarizes the results. Evaluation details can be obtained in the next three subsections.

Table 8-20: Evaluation Results for the Original Architecture C

Case Study	**Project C**
	Architecture Exploration Tool (AET)
Evaluation Goals	Quality Achievement
Number of Architectural Drivers	8
Number of Evaluation Scenarios	22
Number of Architectural Risks	14
Average Risk Severity	**4.34 (Medium)**
Architectural Quality	**0.23 (Average)**

8.6.1.1 Architecture Overview

An overview of the original architecture for project C is given in Figure 8-20. The architecture is organized into three layers: a Presentation layer, a Business layer, and a Database Management layer. Further, a central database is connected to the system.

The Presentation layer encapsulates components that implement the graphical user interface of the system. It consists of components that realize the different dialogs for interacting with the user. The *Projects Dialog* enables the user to open a particular architecture project for investigation. The *Requirements Dialog* and *Scenarios Dialog* support the user in capturing requirements and refining them by specifying scenarios. The *Analysis Dialog* helps the system user in documenting architecture decisions and risks. Finally, the *Results Dialog* provides a summary of the architectural findings and provides a link for exporting them in plain text.

The dialog components make use of an application framework class library, the Microsoft® Foundation Classes (MFC). The *MFC Library* provides a set of classes upon which to build Windows applications. Among others, it provides classes for implementing standard user interface controls such as frames, views, menus, dialogs, and dialog controls for Windows Operating Systems (Win32 API). The dialog components use the MFC controls to create a Windows style compliant user interface.

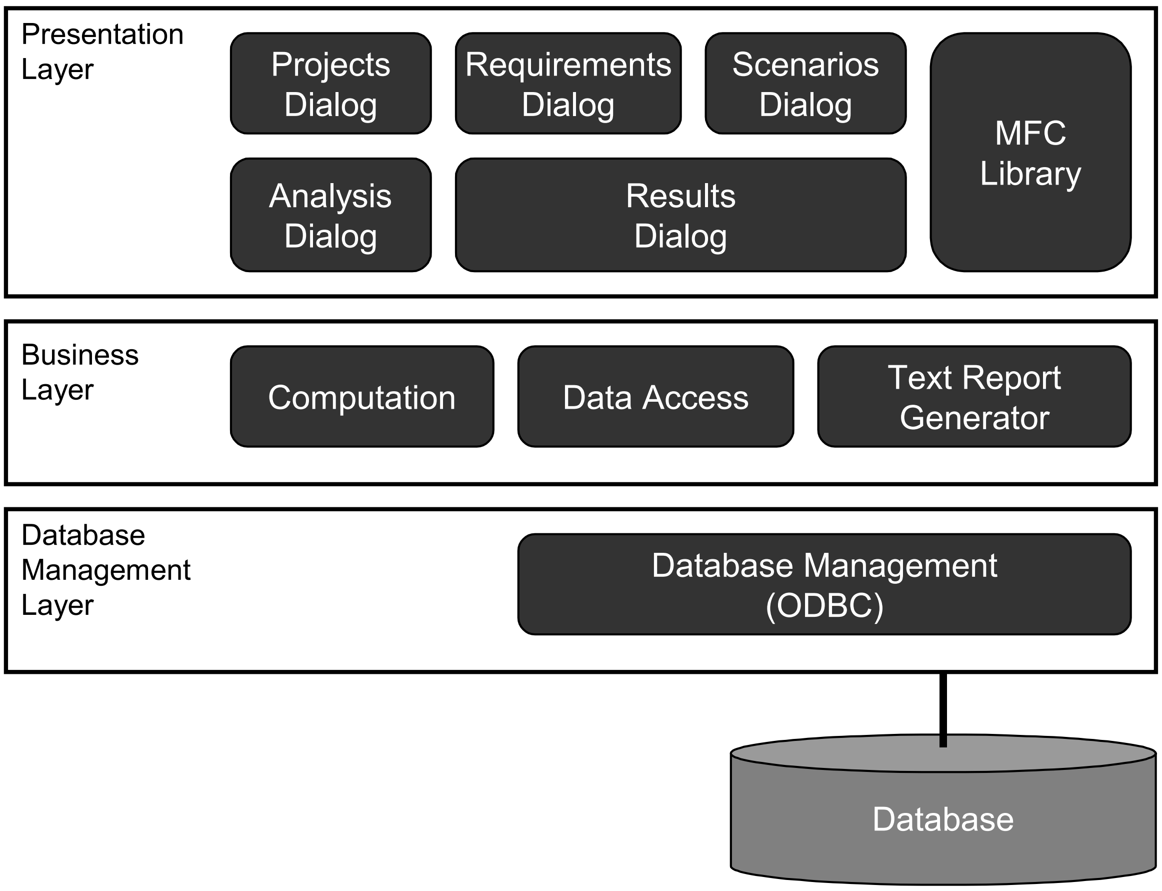


Figure 8-20: Original Architecture of Project C

The Business layer contains the application logic of the system. Requesting data from and delivering data to the central *Database Management* is realized by the *Data Access* component. The component is further responsible for converting data into the internal standard format which is used throughout the system. The *Computation* component is responsible for basic data operations such as counting and sorting. Data such as the decisions documented for an architecture or risks identified during an architecture evaluation can be exported in plain text format. This is implemented by the *Text Report Generator*.

The Database Management layer includes the central access to the Database of the system. It is realized by an ODBC (Open Database Connectivity) interface. ODBC is an established standard for data access on Windows Operating Systems. The Business Layer components, notably *Data Access*, call ODBC API functions to connect to a data source, submit SQL (Structured Query Language) statements, fetch data, and disconnect. A driver manager sits between the application system and the ODBC drivers. The driver manager decides which driver to load and manages communications as driver functions are called. The drivers implement the functions of the ODBC API for the particular database. Figure 8-21 shows how these functions interact. In this way ODBC enables access to different ODBC-compliant data sources, in different locations, using the same function calls available in the ODBC API. Project C uses a Microsoft® Access database for data storage.

Note that this architecture represents a slightly advanced version of the AET architecture presented in Figure 7-1 of Chapter 7.

8.6.1.2 Evaluation Scenarios

The AET quality requirements shown in Table 7-3 are valid for project C. Table 8-21 shows the 22 scenarios used for analyzing the architecture of project C. The scenarios probe significant architecturally driving requirements of the system. As illustrated in Table 8-21, performance, usability, reliability, and modifiability are important quality attributes of the system. During architecture analysis it is evaluated if and to what extent the scenarios can be supported by the original architecture.

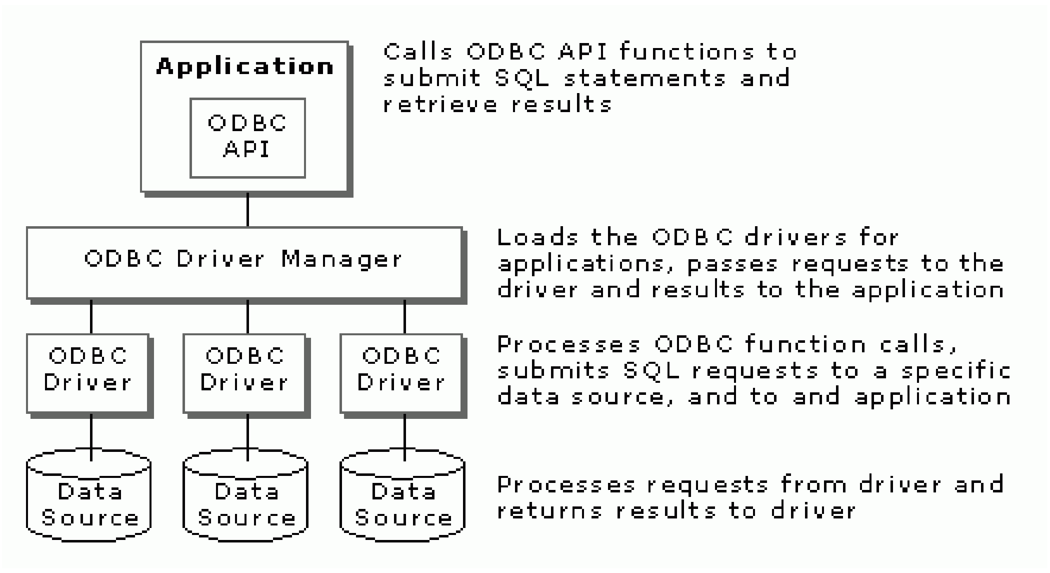

Application
ODBC API
Calls ODBC API functions to submit SQL statements and retrieve results
ODBC Driver Manager
Loads the ODBC drivers for applications, passes requests to the driver and results to the application
ODBC Driver
ODBC Driver
ODBC Driver
Processes ODBC function calls, submits SQL requests to a specific data source, and to and application
Data Source
Data Source
Data Source
Processes requests from driver and returns results to driver

Figure 8-21: Data Management in Project C

Table 8-21: Evaluation Scenarios of Project C

ES#	Quality	Scenario Description
ES1	Performance, usability	A user would like to add a new risk. The risk dialog must be displayed in less than one second.
ES2	Performance, usability	A user would like to organize risks in logically related risk themes. The list of unassigned risks must be displayed with two seconds.
ES3	Performance, usability	The QUADRAD evaluation of an architecture is finished. A slide report about the major results has to be prepared within five minutes.
ES4	Performance, usability	The QUADRAD evaluation of an architecture is finished but some data has to be adapted manually. The adapted data can easily be integrated in the generated slide report.
ES5	Performance, usability	An evaluation scenario has been changed. The updated data is displayed in each related user dialog without significant delay.
ES6	Performance, usability	Data changes in multi-user mode must be propagated to all connected clients. The clients are blocked for less than one second.
ES7	Usability	The tool supports performing a QUADRAD architecture evaluation with only three persons.
ES8	Usability	Different teams are working on defining risk themes. The teams can read data from and write data to the database.
ES9	Usability	Robustness: A risk description has been deleted by accident. The deletion can be undone.
ES10	Usability	Data export: The system can generate a report in different formats. Microsoft® Word, Microsoft® PowerPoint, and LaTeX are supported.
ES11	Usability	Data export: A customer needs the information about the explored architecture in electronic form. The relevant data can be exported in a common format.
ES12	Usability	Flexibility: The system supports applying QUADRAD activities in an order that is different from the standard procedure (e.g., a new scenario is added during the architecture evaluation).
ES13	Reliability, data consistency	A relation between an evaluation scenario and business requirement is added. The relation is reflected in the generated report.
ES14	Reliability, data consistency	An architectural decision is removed and added as a risk. Existing relationships to the decision are shown and can be deleted optionally.
ES15	Reliability, no data loss	The database server crashes. Only data from the past two minutes are lost.
ES16	Reliability, no data loss	The database server has a physical hard disk defect and crashes. The database can be reconstructed in less than ten minutes. The data is no older than two minutes.

Table 8-21: Evaluation Scenarios of Project C (Cont'd)

ES#	Quality	Scenario Description
ES17	Reliability, no data loss	The network connection is temporarily broken down. The system is working again in offline mode in less than five minutes.
ES18	Modifiability	The integration of a full-scale QUADRAD process support shall be realized within two person months.
ES19	Modifiability	The system shall be easily ported from Windows to Linux.
ES20	Modifiability	The format of the generated reports shall be easily adapted from Microsoft® Office to OpenOffice.
ES21	Modifiability	The database vendor is changed (e.g., from Access to Oracle). The new database shall work with the system in less than one week. Old databases can be converted with minimal effort.
ES22	Modifiability	Localization: The user dialogs shall be changed from English to German within one day.

8.6.1.3 Architectural Risks and Risk Assessment

For project C 14 architectural risks have been identified during the architecture evaluation. Table 8-22 shows the probability, impact, and resulting ranking of the risks recorded during the risk assessment of C. Table 8-23 gives a detailed description of the risks R2, R3, R4, R7, R12, and R13.

According to the metrics M6 and M7 given in Table 8-2, the average risk severity and architectural quality of the baseline architecture of project C is calculated as follows:

Average risk severity of *C*: $$R_S(C) = \frac{1}{14}\sum_{i=1}^{14} \text{Ranking}(r_{C_i}) = \frac{1}{14} \cdot 60.75 = 4.34$$

Architectural quality of *C*: $$Q(C) = \frac{1}{R_S(C)} = 0.23$$

Figure 8-22 illustrates the risk rankings and the average risk severity of *C* in a bar chart. The average risk severity of 4.34 leads to an architectural quality of 0.23. According to Table 8-20 the baseline architecture of C again has an average quality.

Table 8-22: Probability and Impact of the Risks in Project C

Risks of *C*	Probability	Impact	Ranking
R1	Effective	High	**6**
R2	Effective	Very High	**9**
R3	Effective	Low	**3**
R4	Effective	High	**6**
R5	Low	Low	**0.75**
R6	Low	Low	**0.75**
R7	Effective	High	**6**
R8	Medium	High	**3**
R9	High	High	**4.5**
R10	High	Low	**3**
R11	Effective	High	**6**
R12	Effective	Low	**3**
R13	High	Very High	**6.75**
R14	Effective	Low	**3**
Total	--	--	**60.75**

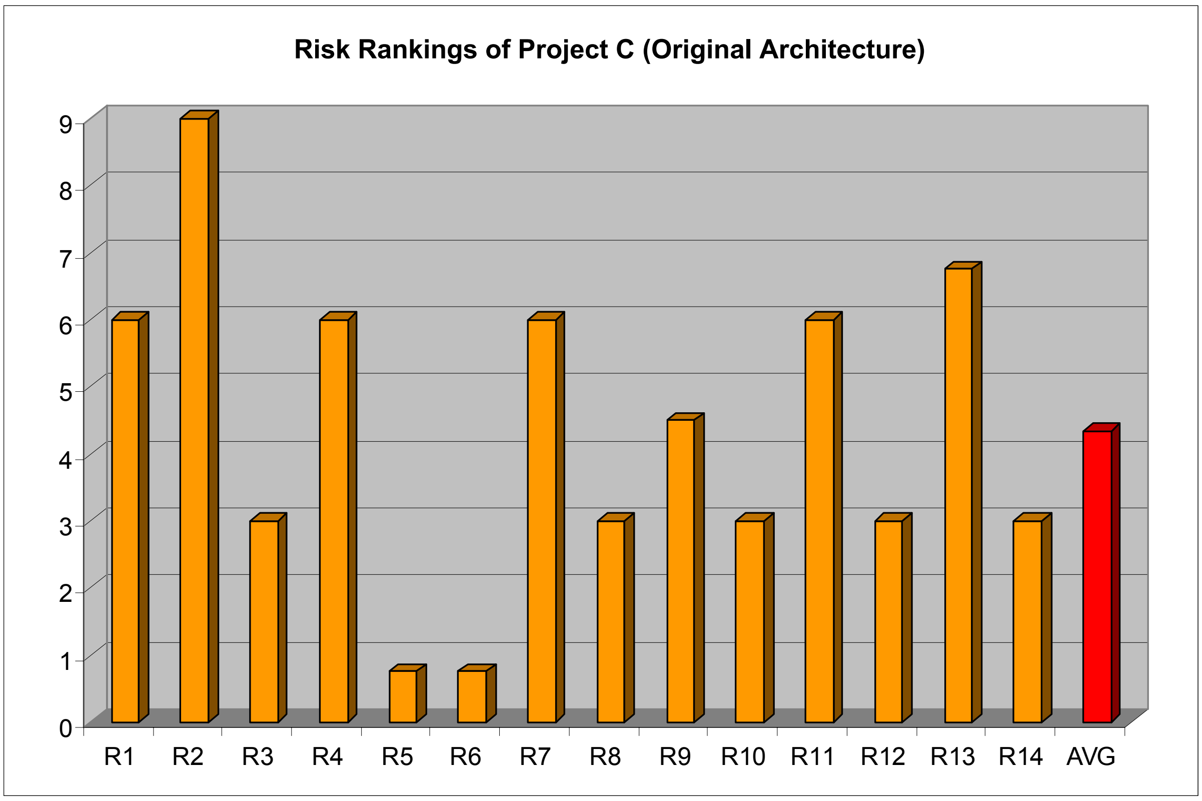


Figure 8-22: Risk Rankings of Project C (Original Architecture)

Table 8-23: Architectural Risks of Project C (Extract)

Risks of *C*	**Description**
R2	**Limited capabilities for exporting an architecture report.**
	The architecture does not support the adequate export of information stored in the tool's database. Information about a project's architecture can only be displayed in textual form on the screen. Graphics are not supported. In order to produce a written report, the architecture exploration results must be copied manually from screen dialogs. Thus, the architecture does not support an efficient report generation, which represents a major risk of the current implementation.
R3	**Database cannot be easily replaced by another vendor's product.**
	The original architecture is coupled to a specific database vendor. The risk is that the database management components that are included as COTS in the architecture do not allow full data access control. The business impact of this risk is low since the supported data access functionality is not used as a unique selling point and seems to be sufficient for the applications of project C.
R4	**Mapping of risks to affected business requirements not supported.**
	The data model of the original architecture does not support the mapping of architectural risks to affected business requirements. Risks are identified based on the evaluation of scenarios. The data model supports the documentation of risks and allows tracing the risk description back to the associated scenario. Unfortunately, linking scenarios to quality attributes and business requirements is not supported by the current data model. This is a risk since information about potentially unsupported business requirements cannot be represented.
R7	**Limited performance when database is frequently updated.**
	The original architecture does not support adequate database maintenance. Adding new and updating existing data quickly leads to a fragmented database which results in poor performance data management and higher data storage needs. The risk is that the data access becomes so slow as to make usage of the system impossible.
R12	**Calculation of statistics not supported.**
	The original architecture does not support the calculation of important statistics about the architectures stored in the tool's database. This is a risk since it is not possible to get a quick overview on the status and maturity of the architectures stored in the database. Gathering basic information about the architectures and comparing different architecture efforts becomes a time consuming task. However, the impact of this risk on the business success of the system is rather low since statistics are treated as an add-on. They can in principle be calculated manually.

Table 8–23: Architectural Risks of Project C (Cont'd)

Risks of *C*	Description
R13	**No recovery precaution for database failures.**
	The original architecture includes no mechanisms to recover the data from database failures. High data fragmentation or general system failures (e.g., Operating System failures) can result into database corruption. The risk is the total loss of important information stored in the database.

8.6.2 Step 2: Creating a Revised Architecture C*

Figure 8-23 shows the revised architecture for project C after several iterations of QUADRAD have been performed. In the following we explain in detail how the architecture has been adapted to address the six risks R2, R3, R4, R7, R12, and R13 described in Table 8-23:

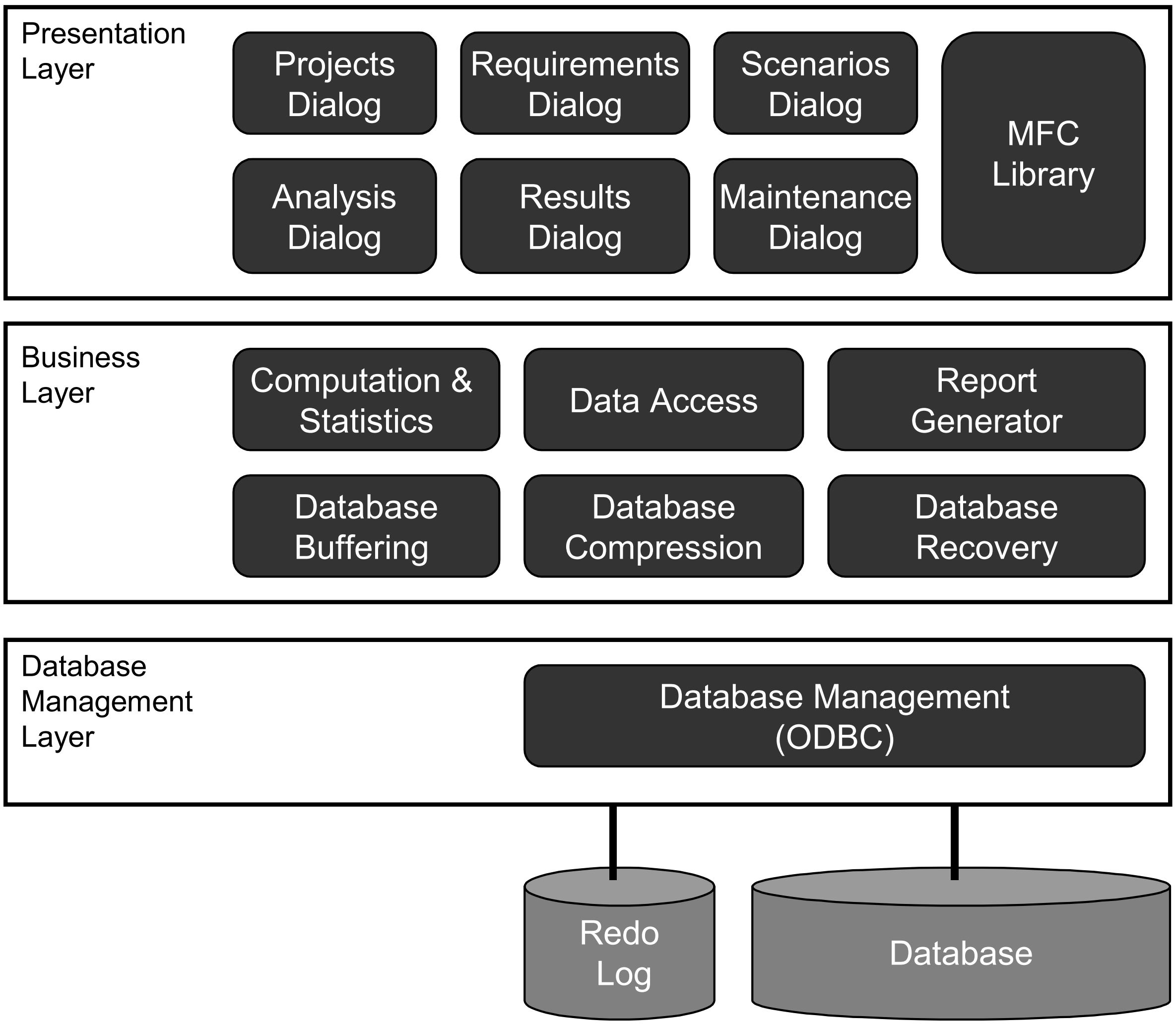


Figure 8-23: Revised Architecture of Project C

- **R2: Limited capabilities for exporting an architecture report.**

  Requirements refinement: The original requirement has been refined to include the report generation capability for Microsoft® Word and PowerPoint. The generation of LaTeX reports has been excluded from the requirement specification.

  Architecture adaptation: The revised architecture includes a completely reworked *Report Generator* component (cf. Figure 8-23). This component is capable of exporting a full architecture report in Microsoft® Word or Microsoft® PowerPoint format. The *Report Generator* uses dedicated documentation templates as an input. These templates include the basic layout and chapter outline of the final report. They also include a generic introduction for each chapter as well as a dictionary of (architecture-related) terms used in the report.

  Figure 8-24 shows the implementation of the *Report Generator* component by applying the "Abstract Factory" and "Builder" mechanisms [Gamma 1994]. The "Builder" separates the construction of a complex component from its representation, such that the same construction procedure for different representations of the component can be used. The "Abstract Factory" provides an interface for creating sets of related components. Figure 8-25 shows details of the Report Generator implementation.

  When creating a report the *Report Generator* uses the architecture information stored for a particular project in the AET database to fill in the respective chapters of the template (cf. Figure 8-26). Examples of architecture information to be included in a report are the quality attributes, scenarios used to probe the architecture, design decisions the architect used to satisfy the scenarios at the architecture level, and risks found during an architecture review. The component is capable of exporting texts, graphics, and basic formatting such as bullet lists and tables. The resulting report represents an important asset in documenting essential information of the architecture exploration. It also serves as an input to improve the architecture in a goal-oriented way by applying the required QUADRAD activities. Figure 8-27 shows a sample page of a tool-generated architecture report.

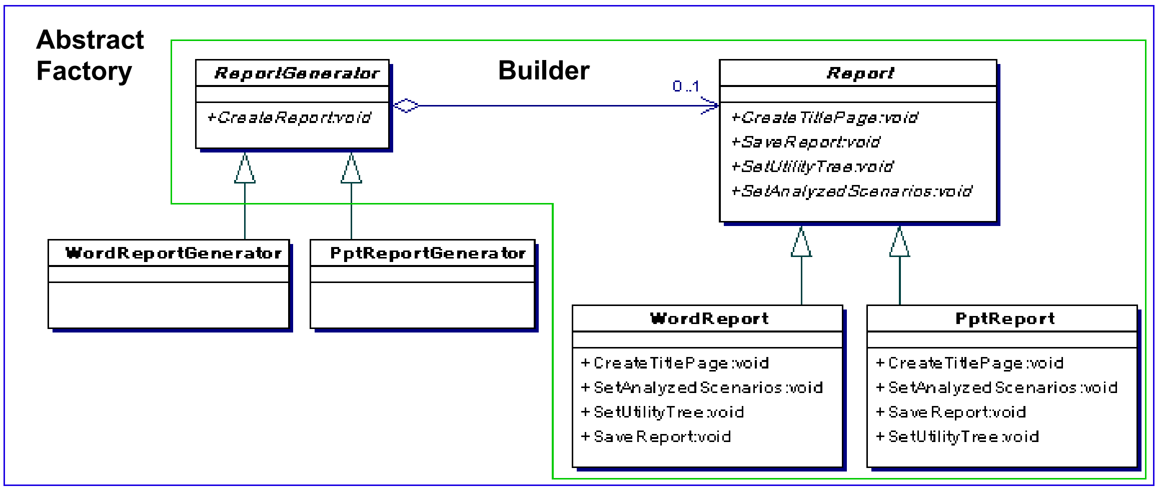


Figure 8-24: Abstract Factory and Builder Mechanisms for Report Generation

```
class ReportGenerator {
      public:
            void CreateReport() {
                  m_report->CreateTitlePage();
                  m_report->SetUtilityTree();
                  m_report->SetAnalyzedScenarios();
                  m_report->SaveReport();
            }
           ...
      protected:
            ReportGenerator();
            Report* m_report;
            ...
};

class Report {
      public:
            virtual void CreateTitlePage() = 0;
            virtual void SaveReport() = 0;
            virtual void SetUtilityTree() = 0;
            virtual void SetAnalyzedScenarios() = 0;
            ...
};
```

Figure 8-25: Report Generator Code Samples

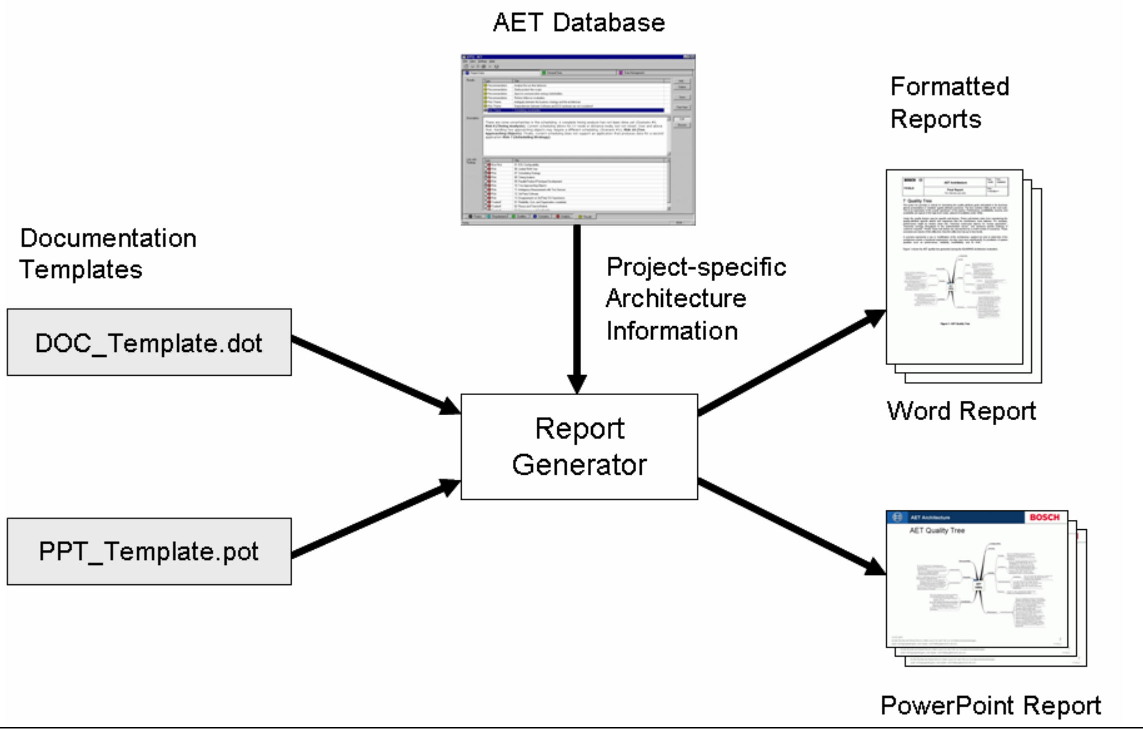
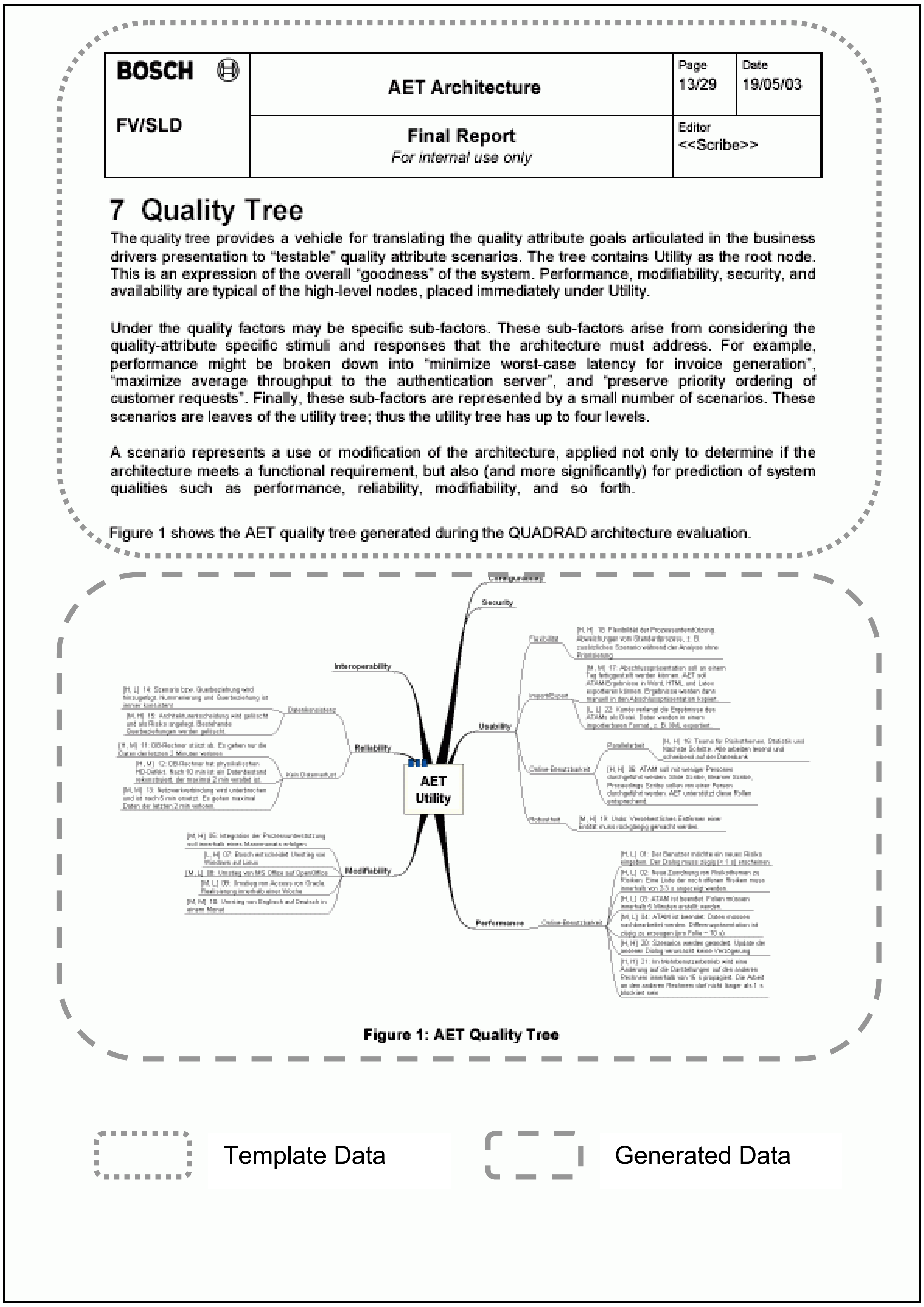


Figure 8-26: Report Generation



BOSCH FV/SLD	AET Architecture	Page 13/29	Date 19/05/03
	Final Report *For internal use only*	Editor <<Scribe>>	

7 Quality Tree

The quality tree provides a vehicle for translating the quality attribute goals articulated in the business drivers presentation to "testable" quality attribute scenarios. The tree contains Utility as the root node. This is an expression of the overall "goodness" of the system. Performance, modifiability, security, and availability are typical of the high-level nodes, placed immediately under Utility.

Under the quality factors may be specific sub-factors. These sub-factors arise from considering the quality-attribute specific stimuli and responses that the architecture must address. For example, performance might be broken down into "minimize worst-case latency for invoice generation", "maximize average throughput to the authentication server", and "preserve priority ordering of customer requests". Finally, these sub-factors are represented by a small number of scenarios. These scenarios are leaves of the utility tree; thus the utility tree has up to four levels.

A scenario represents a use or modification of the architecture, applied not only to determine if the architecture meets a functional requirement, but also (and more significantly) for prediction of system qualities such as performance, reliability, modifiability, and so forth.

Figure 1 shows the AET quality tree generated during the QUADRAD architecture evaluation.

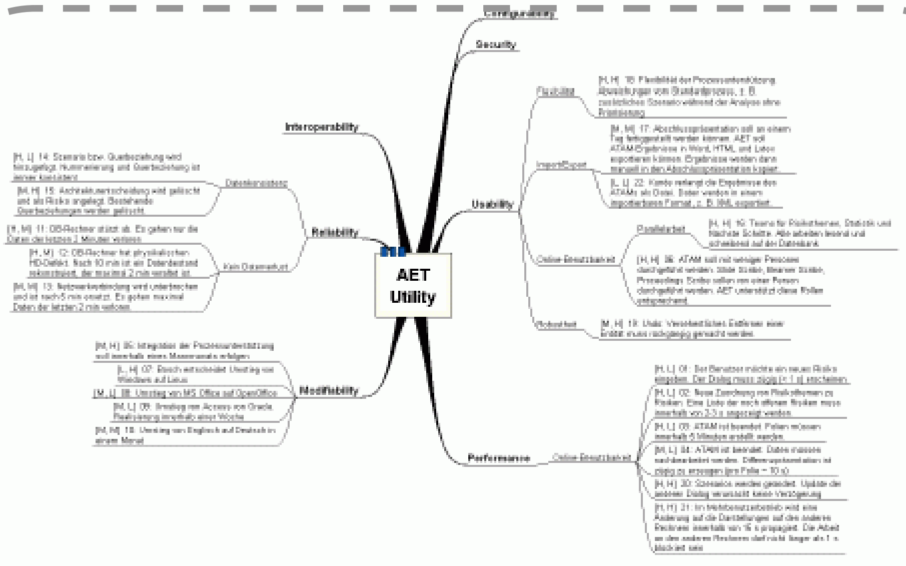


Figure 1: AET Quality Tree

Template Data Generated Data

Figure 8-27: Architecture Report (Sample Page)

- **R3: Database cannot be easily replaced by another vendor's product.**

  Architecture adaptation: The architecture has not been adapted according to this risk. The database and database management components cannot be replaced by products of another vendor without fundamental changes to the revised architecture. However, the database management capabilities have been extended to include basic maintenance functions (see architecture revisions for risks R7 and R13).

- **R4: Mapping of risks to affected business requirements not supported.**

  Architecture adaptation: The revised architecture includes a reworked relational data model of the project database, which allows tracing a risk back to its original requirement. Figure 8-28 shows the new data model. It includes the 10 entities *Projects, PriorityScales, PriorityDimension, Requirements, Mechanisms, Utilities, PriorityRankings, Analysis, Findings,* and *Results* to manage different information about a project's architecture. Each entity has a primary key for unique access and, optionally, one or more foreign keys to include data from other entities. Foreign keys are marked as "FK" in Figure 8-28. The entities are related among another, either in a 1:1, a 1:n, or a n:m relationship. For example, managing a prioritized set of quality requirements of a particular project is realized by the entities *Projects, Requirements, Utilities,* and *PriorityRankings*. Risks of a project are treated by the entity *Findings*. The entity *Results* manages different sets of logically related risks, also known as risk themes.

  The new data model allows the mapping from risks and risk themes to business requirements. Figure 8-29 shows the respective entities that need to be traversed as part of a database query in order to gather the respective information. The query can be implemented by the following SQL pseudo code:

```
SELECT   Requirements.PJ_ID AS PJID,
         Requirements.R_ID AS RID,
         Requirements.Title AS Requirement,
         Results.Title AS Result
FROM     (Utilities LEFT JOIN
            (Findings LEFT JOIN
                (Results RIGHT JOIN F_RS
                    ON Results.RS_ID = F_RS.RS_ID)
            ON Findings.F_ID = F_RS.F_ID)
         ON Utilities.U_ID = Findings.U_ID)
         RIGHT JOIN (Requirements LEFT JOIN R_U
             ON Requirements.R_ID = R_U.R_ID)
         ON Utilities.U_ID = R_U.U_ID
GROUP BY Requirements.PJ_ID,
         Requirements.R_ID,
         Requirements.Title,
         Results.Title,
         Results.Type
HAVING   (((Results.Type)="Risk Theme"))
ORDER BY Requirements.R_ID;
```

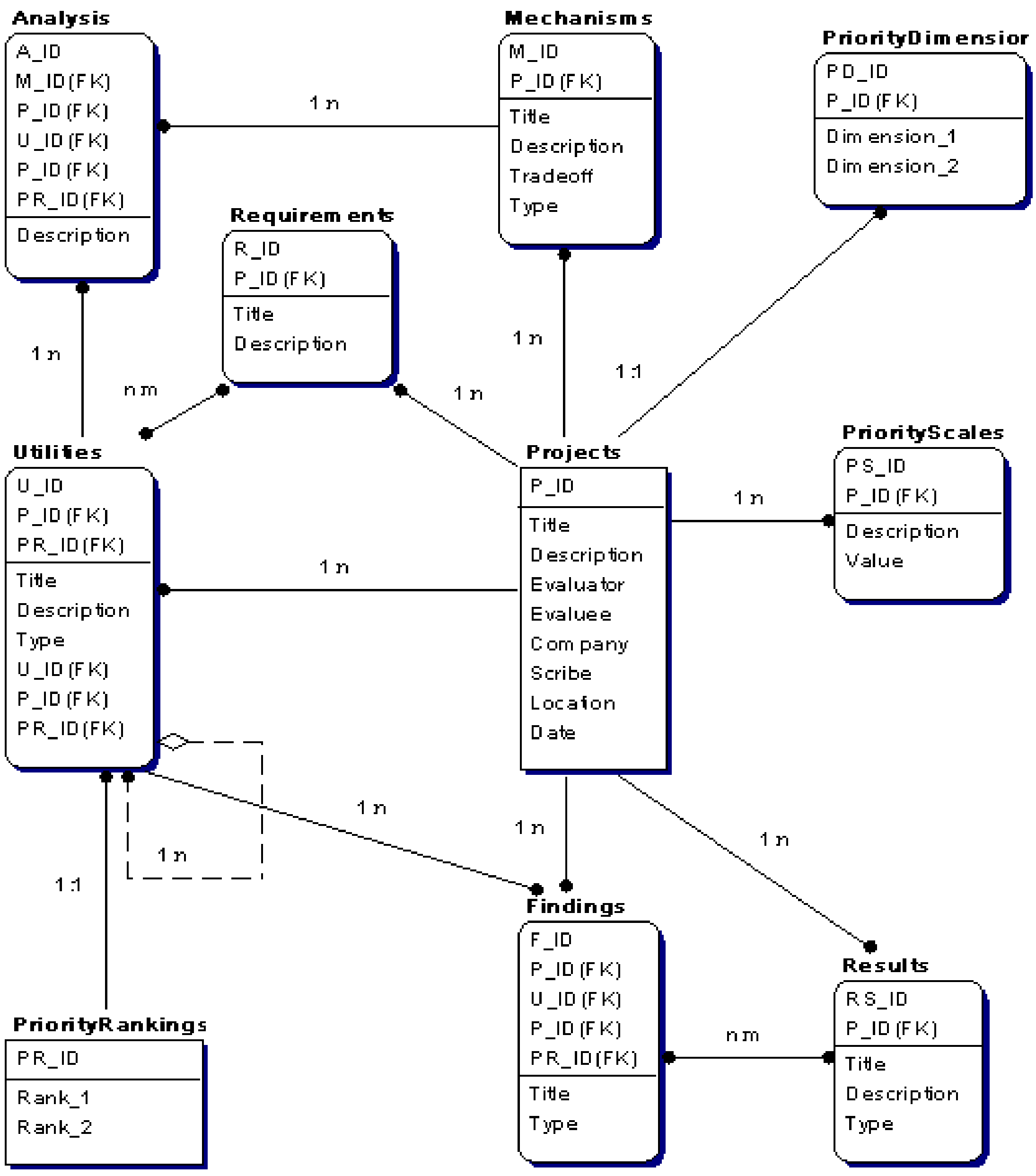


Figure 8-28: Revised Data Model

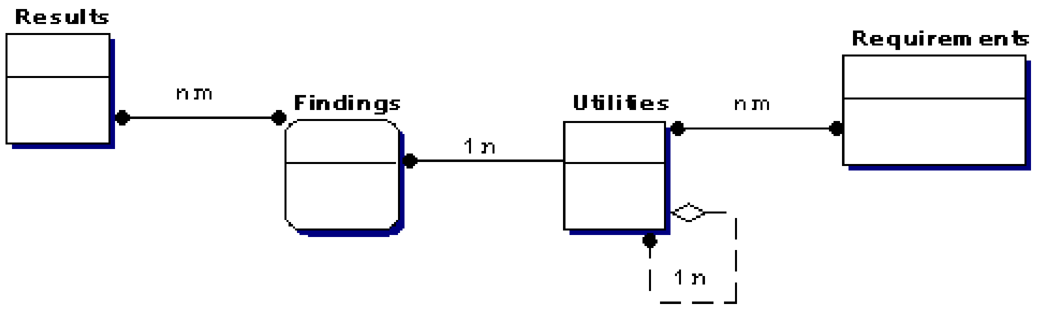


Figure 8-29: Database Query for Mapping Risks to Business Requirements

- **R7: Limited performance when database is frequently updated.**

  Architecture adaptation: The revised architecture integrates a *Database Compression* component that improves the performance of the database (cf. Figure 8-23). This component makes a copy of the database file and, if it is fragmented, rearranges how the database file is stored on disk. When completed, the compacted database has reclaimed wasted space, and is usually smaller than the original. Compacting the database ensures optimal application performance. Failures such as page corruptions and power surges are resolved. If a primary key exists in the table, the *Database Compression* component restores table records into their Primary Key order. This makes the read-ahead capabilities of the database engine much more efficient. *Database Compression* also updates the table statistics within the database that are used as optimizes queries. These statistics can become outdated as data is added, manipulated, and deleted from the various tables. Query speed will be enhanced significantly, because they are now working with data that has been rewritten to the tables in contiguous pages. Scanning sequential pages is much faster than scanning fragmented pages. Queries are also forced to recompile/optimize after each database compaction.

- **R12: Calculation of statistics not supported.**

  Architecture adaptation: The revised architecture includes a *Computation & Statistics* component (cf. Figure 8-23). This component is updated and extended based on the *Computation* component of the original architecture (cf. Figure 8-20). It now allows calculating statistics about each architecture effort stored in the database. Important statistics are, for example, the number of architecturally driving requirements, the number of risks identified during architecture evaluation, the number of reviews performed during the architecture's lifetime, and the number of change requests for requirements. This information is important for getting a quick overview of the status and maturity of the architecture.

- **R13: No recovery precaution for database failures.**

  Architecture adaptation: The revised architecture includes a *Data Recovery* component that is responsible for preventing or minimizing data loss in case the database is corrupted (cf. Figure 8-23). This component records all changes made in the database in a *Redo Log*. The *Redo Log* consists of multiple files that are stored separately from the database. The *Data Recovery* component records ongoing database changes and writes to the log file whether the changes are physically written to the database (committed) or whether they are made to the database buffers in memory (uncommitted).
  If a database failure occurs (e.g., file system failure) the first step of the recovery process is to roll forward all of the changes recorded in the redo log to the data files. After roll forward, the data blocks contain all committed changes as well as any uncommitted changes that were recorded in the redo log. The *Data Recovery* component then performs a roll back where uncommitted transactions previously applied by the rolling forward phase are undone (cf. Figure 8-30). After the roll back the database is in a consistent state. Media failures (e.g., head crash of hard disk) are managed by frequently backing up the database and log files to separate media. Note that database compression and recovery can be controlled by the user interface component *Database Maintenance Dialog* (cf. Figure 8-23).

8.6.3 Step 3: Evaluating the Revised Architecture C*

Table 8-24 shows that the ten risks R1, R4, R5, R7, R8, R9, R10, R11, R12, and R14 have been resolved through an architectural improvement of C. Two risks (R2 and R13) could be resolved by refining the architectural drivers/business goals. The two risks R3 and R6 could not be addressed during the validation. They have been omitted because the first official release (V1.0) of the system needed to be finished prior to the end of the validation. Table 8-25 shows that three additional risks have been identified during step 3. These risks are related with the new design decisions made for C*. They are not significant for the validation and thus are not described in more detail here.

According to the metrics M11 and M12 given in Table 8-2, the average risk severity and architectural quality of the revised architecture of project C is calculated as follows:

Average risk severity of *C:** $R_S(C^*) = \frac{1}{5}\sum_{i=1}^{5}\text{Ranking}(r_{C^*_i}) = \frac{1}{5}\cdot(3.75+4.75) = 1.7$

Architectural quality of *C:** $Q(C^*) = \frac{1}{R_S(C^*)} = 0.59$

Figure 8-31 illustrates the risk rankings and the average risk severity of the revised architecture *C**.

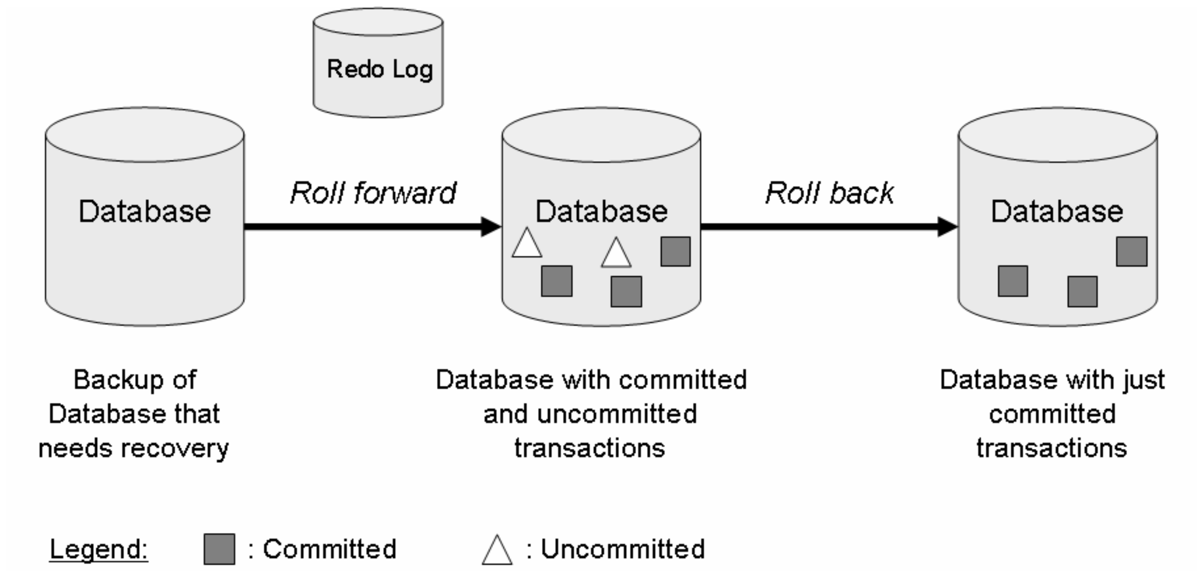

Redo Log
Database
Roll forward
Database
Roll back
Database
Backup of Database that needs recovery
Database with committed and uncommitted transactions
Database with just committed transactions
Legend:
▨ : Committed
△ : Uncommitted

Figure 8-30: Database Recovery

Table 8-24: Risks of Project C (Original vs. Revised Architecture)

Risks of C	Ranking Original Architecture	Architecture Revision Activity	New Risk Status	Ranking Revised Architecture
R1	**6**	Architecture Adaptation	Risk Resolved	**0**
R2	**9**	Refinement & Adaptation	Risk Resolved	**0**
R3	**3**	--	**Risk Unresolved**	**3**
R4	**6**	Architecture Adaptation	Risk Resolved	**0**
R5	**0.75**	Architecture Adaptation	Risk Resolved	**0**
R6	**0.75**	--	**Risk Unresolved**	**0.75**
R7	**6**	Architecture Adaptation	Risk Resolved	**0**
R8	**3**	Architecture Adaptation	Risk Resolved	**0**
R9	**4.5**	Architecture Adaptation	Risk Resolved	**0**
R10	**3**	Architecture Adaptation	Risk Resolved	**0**
R11	**6**	Architecture Adaptation	Risk Resolved	**0**
R12	**3**	Architecture Adaptation	Risk Resolved	**0**
R13	**6.75**	Refinement & Adaptation	Risk Resolved	**0**
R14	**3**	Architecture Adaptation	Risk Resolved	**0**
Total	**60.75**	--	--	**3.75**

Table 8-25: New Risks of Project C (Revised Architecture)

Risks of *C**	Probability	Impact	Ranking
R15	High	Low	**2.25**
R16	Medium	Low	**1.5**
R17	Effective	Very Low	**1**
Total	--	--	**4.75**

8.6.4 Step 4: Comparison

According to the metric M13 of Table 8-2 the quality of the revised architecture *C** is *better* than that of the baseline architecture *C*:

$$Q(C^*) = 0.59 > 0.23 = Q(C)$$

64% of the risks in *C* could be reduced in *C**:

$$1 - \frac{Number\ of\ Risks(C^*)}{Number\ of\ Risks(C)} = 1 - \frac{5}{14} = 0.64$$

The quality improvement between architecture *C* and *C** is calculated as:

$$1 - \frac{Q(C)}{Q(C^*)} = 1 - \frac{0.23}{0.59} = 0.61$$

As a result, the architecture quality of project C could be improved by 61%. Table 8-26 summarizes the validation results.

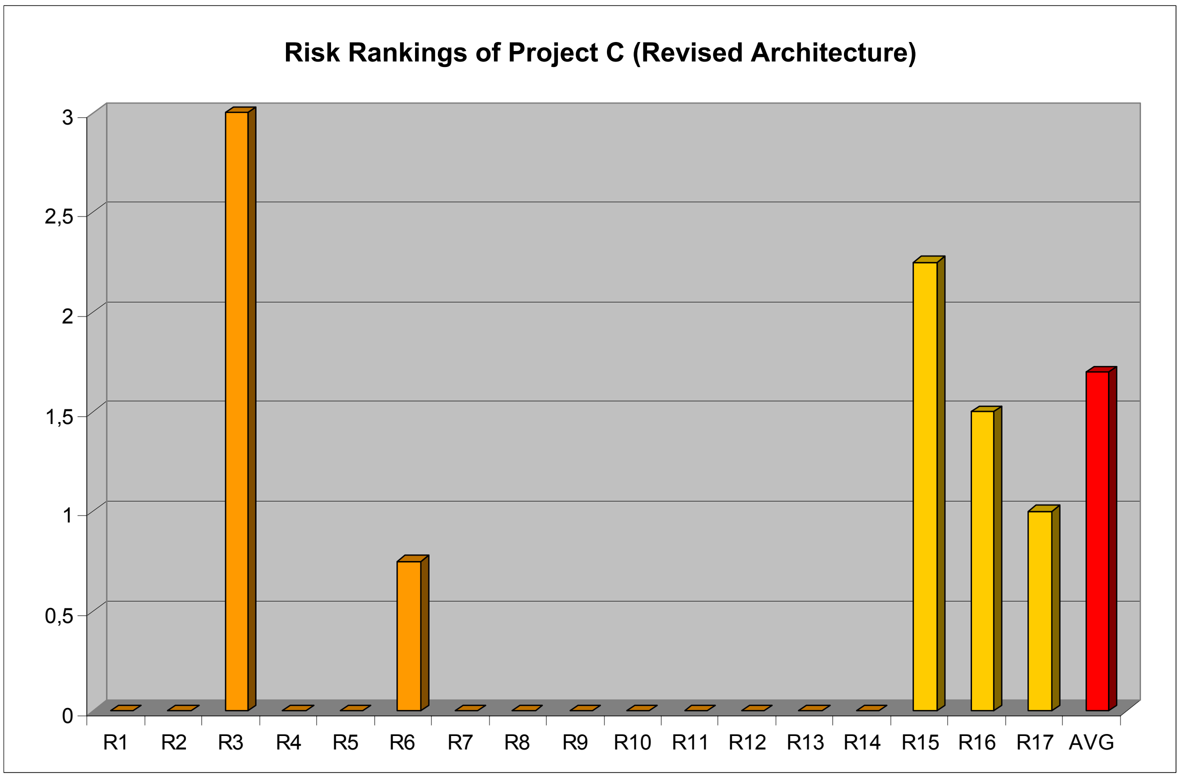


Figure 8-31: Risk Rankings of Project C (Revised Architecture)

Table 8-26: Validation Results of Project C

Case Study	**Project C (AET)**	
	Original Architecture	Revised Architecture
Number of Risks	**14**	**5**
Average Risk Severity	**4.34 (Medium)**	**1.7 (Low)**
Architecture Quality	**0.23 (Average)**	**0.59 (Good)**
Risk Reduction	**64%**	
Architectural Quality Improvement	**61%**	

8.7 Validation Results

Table 8-27 gives a summary of the validation results. For each of the three projects of the case study the number and the severity of risks could be reduced drastically. This has lead to a significant quality improvement of the three systems' architectures considered in the validation. Figure 8-32 and Figure 8-33 show a graphical summary of the results.

Table 8-27: Validation Results

Case Study	**Project A (CMS)**		**Project B (AAS)**		**Project C (AET)**	
Architecture	**A**	**A***	**B**	**B***	**C**	**C***
Number of Risks	19	5	13	2	14	5
Risk Severity	3.37	2.15	3.81	1.13	4.34	1.7
Architectural Quality	0.3	0.47	0.26	0.88	0.23	0.59
Risk Reduction	**74%**		**85%**		**64%**	
Architectural Quality Improvement	**36%**		**70%**		**61%**	

The validation results demonstrate that the hypothesis of this work has been confirmed. In Section 8.1 we stated the following validation goal to prove the hypothesis:

Goal = Estimate if the QUADRAD framework improves the architectural quality of software-intensive systems

The results clearly show that the application of QUADRAD has significantly improved the architectural quality of the three software-intensive systems that were part of the case study.

In particular, the architectural quality for project A has been improved by an average of 36%. At the same time the architectural risks have been reduced by 70%. For project B the quality of the architecture has been improved by 74%. 11 of 13 risks have been resolved for B, leading to a risk reduction of 85%. Finally, the architectural quality of project C has been improved by 61%. The architectural risks of C have been reduced by 64%.

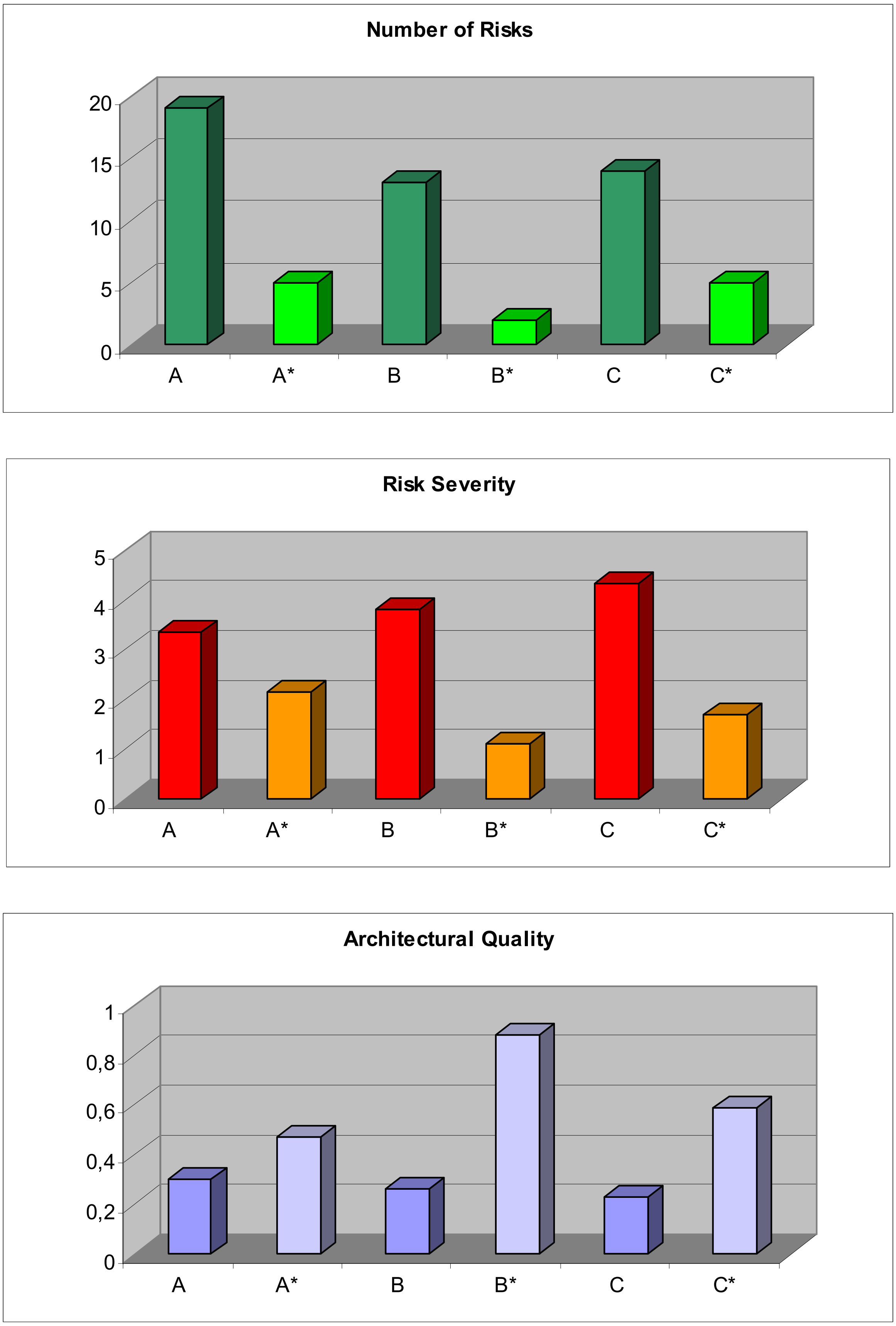
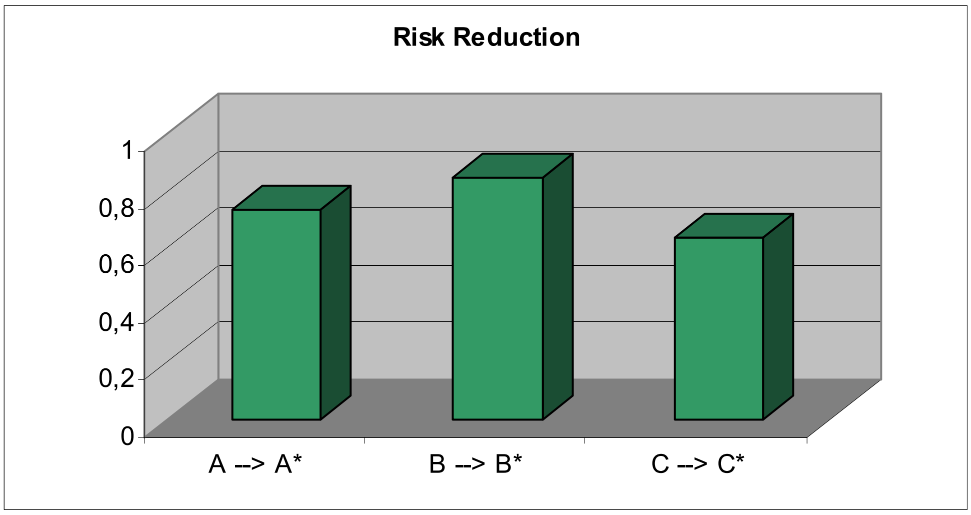


Figure 8-32: Baseline vs. Revised Architectures A, B, and C



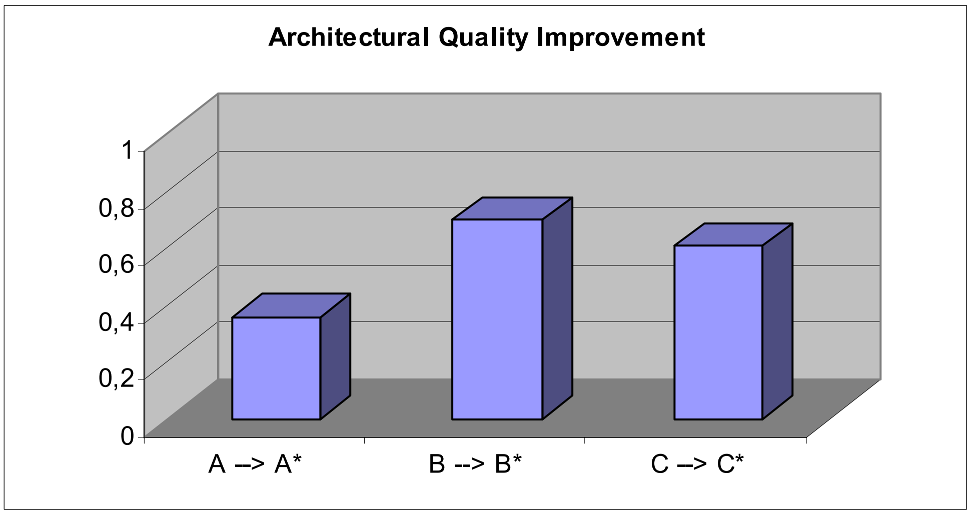


Figure 8-33: Risk Reduction and Quality Improvement Achieved with QUADRAD

As the results show, the number of risks is no adequate measure to determine whether the quality of an architecture is good or bad. Particularly, the risk reduction ratio should carefully be considered as metric when assessing the quality of an architecture. Even if 80% of the risks could be resolved due to a directed architecture revision effort this does not mean that the quality has improved by the same ratio. The results of the validation also confirm this fact. For example, the risks in A could be reduced by 70% but the quality of the architecture could "only" be improved by an average of 36%. Moreover, although the risk reduction ratio of C is smaller than that of A (64% vs. 70%), the architectural quality improvement achieved for C is better (61% vs. 36%).

This is the reason why we did not consider the number but the *severity* of risks for determining the quality of an architecture (cf. Section 8.2). Since the severity of a risk depends on the risk's probability and development impact it represents a much more practical view on how critical the design decisions in an architecture are. The more critical these decisions and the more critical decisions are included in an architecture, the higher is the risk severity of the architecture. The higher the risk severity, the more reduced is the quality of the architecture. The architectural quality is reduced because it does not achieve important requirements (e.g., safety, performance, or reliability) if severe risks have been identified.

Mitigating or resolving the risks then requires extra effort for changing the architecture. A quality architecture does not need any changes in order to achieve the requirements imposed on it.

For the three projects A, B, and C of the case study, different values for the baseline and revised risk severity and architectural quality have been calculated according to the metrics M6/M7 and M11/M12 of the GQM model (cf. Table 8-2). The calculation is based on the stakeholders' judgement of probability and impact of the risks. Judgemental errors have been avoided as far as possible as the same evaluation scenarios have been used and the same stakeholders have re-evaluated the risks after the application of QUADRAD on the baseline architectures. That there are differences in quality improvement between the projects A, B, and C mainly relies on three things:

1. *Not every risk in the three architectures could be resolved.* The risks R9, R15, R16, and R18 in project A have not been resolved (see Table 8-11). In project B, R1 and R8, in project C, R3 and R6 remain unresolved (see Table 8-18 and Table 8-24, respectively).

2. *The remaining risks in the three architectures have different severities.* The risk severity of the remaining risks of A is 7.75, while the severity of B and C is only 2.25 and 3.75, respectively (cf. Table 8-11, Table 8-18, and Table 8-24).

3. *The revision of an architecture particularly introduces new risks.* For project A one new risk (R20), for project C three new risks (R15, R16, R17) have been introduced in the architecture (see Table 8-12 and Table 8-25). The new risks have also changed the overall risk severity of projects A and C.

The fact that (repeated) application of QUADRAD does not eliminate all the risks in an architecture is quite normal. Usually, resolving every risk according to a set of evaluation scenarios is simply too expensive. The important thing is that the remaining risks have only a low severity. Then the risks are not critical for the subsequent development phases and they are not crucial for the business success of the system. A further investigation of the risks depends on the time and budget of the development effort and typically requires management decisions. A careful application of QUADRAD usually does not lead to new risks with high severity. QUADRAD includes activities for making the most appropriate architectural decisions and for identifying potential side effects between these decisions (cf. Chapter 5 and Chapter 6).

It is worth to mention that the validation metrics defined in the GQM model of Table 8-2 cannot be used to determine the risk severity and quality of an architecture in general terms. This means, that the metrics cannot be used to calculate a fixed and generally valid "quality score" for an architecture. Rather, the metrics help to determine the quality of an architecture based on given set of architectural risks. The quality is thus calculated relative to those risks. If risks change then the quality of the architecture may change, too. For each project of the validation, a particular set of evaluation scenarios has been used to identify the risks. Since these scenarios have been evaluated as most critical for the projects, there is a high probability that the analysis of these scenarios have uncovered the most critical risks. But again, if the evaluation scenarios of a project are changed, other risks may come up, which, on the other hand, may lead to a different architectural quality for that project.

However, this limitation in the metrics is not critical with respect to the validation goals of this thesis. We need to calculate the quality improvement of an architecture *relative* to a baseline architecture. Since we did not change any of the mentioned parameters (evaluation scenarios, risks, stakeholders that judge the risks) during the validation, the results reflect the architectural quality improvement observed by the case study in a consistent manner.

8.8 Lessons Learned

Some lessons have been learned during the validation of the pilot projects. In the following, the most significant issues are summarized. They provide opportunities for further improvements of guiding the process of architecture development.

Preparing the Requirements for Architecture Development

- **Requirements too abstract:** The given requirements are not well documented and thus are subject to misinterpretations, which may lead to a false evaluation of their significance for architectural development. In this case, iterate requirements specification (requirements engineering) or try brainstorming scenarios to make the essence of the requirements more concrete (see Section 2.3.3.1).
- **Requirements prioritization not possible:** Stakeholders are unable to prioritize requirements. As with the previous issue, try iterating requirements specification or brainstorm additional scenarios.
- **Architectural drivers remain undiscovered:** Stakeholders fail to discover architecturally significant requirements. In this case, try multiple passes of QUADRAD-P since the complete set of drivers is rarely identified in a first iteration. After a starting set of drivers has been implemented in the architecture, feedback can be given from detailed design and coding. This feedback often leads to the discovery of new drivers, which can then be considered in a second iteration.
- **Missing criteria for requirements prioritization and selection:** Stakeholders are not able to prioritize a requirement although it is well documented. For getting a decision on the architecture impact try to ask a question such as "how many design elements or functions would be affected by the requirement under consideration." If the number is estimated above three major elements/functions (rule of thumb) then the requirement can be ranked as architecturally significant since the effort for modifying it later seems to be high. On the other hand, for getting a decision on the business impact one can ask a question such as "is the requirement essential for the business success (e.g., to achieve cost targets or time-to-market) or does it affect a nice-to-have functionality." In the former case, the requirement would be of high importance, in the latter case of rather low importance. Capturing the reasons why a requirement has been judged as architecturally driving or not is also helpful for a better understanding of the decisions of the resulting design.
- **Changing scope:** The system scope changes significantly and/or frequently – e.g., new architecturally driving requirements come into play, others become obsolete etc. In this case, reconsider the architectural requirements and mechanisms by performing the activities "Identify architectural drivers" (cf. Section 4.2) and "Define architectural strategies and mechanisms" (cf. Section 5.2). Further, try evaluating the existing architecture for architectural risks with respect to the new requirements (cf. Chapter 6).

Modeling the Architecture

- **Decomposition flaws:** A design element may not provide the functionality needed to implement the system correctly if there is not an appropriate decomposition of the required system functionality. This risk is mitigated by a robust architectural description that includes the definition of interfaces. These interfaces represent the division of functional responsibilities. This decomposition should be checked for completeness, correctness, and consistency.
- **Inadequate specifications:** Design elements may not integrate properly if their specifications are limited to static descriptions of individual services. This risk is very likely to occur because current design element specification technologies do not have notations for expressing temporal relationships or interactions. This risk can be mitigated by supplementing standard contractual notations with techniques that specify explicitly how components that implement interacting interfaces must behave.
- **Excessive dependencies between design elements:** A design element becomes less maintainable and reusable if it has excessive dependencies on other design elements. In order to reduce design element interdependencies the responsibilities of the affected elements need to be carefully analyzed to come to a result with a clear delineation of the division of responsibilities between design elements.
- **Mechanism unknown:** Not every architectural driver can directly be linked to a corresponding mechanism in early iterations (cf. Section 5.2). Sometimes the mechanism remains ambiguous or is not known. In this case try to choose an architectural strategy that fits best to the qualities the driver addresses and derive the architectural decisions from this description.
- **Incorporation of remaining requirements:** It is explicitly not the goal of architecture development to satisfy all the given requirements but *only* those that are architecturally relevant. The requirements that are not of architectural importance are usually incorporated as black-boxes. They must be considered during detailed design or implementation. The weighting of requirements performed in the QUADRAD Preparation workflow helps to specify the set of design- and implementation-relevant requirements since they have a low importance rating by the architect.

Evaluating the Architecture

- **Missing experts / stakeholders during scenario prioritization:** This has a severe impact on the evaluation results. For a proper ranking of candidate evaluation scenarios, at least the architect (for a decision on “architectural impact”) and a business/sales representative (for decision on “business importance”) are required. Without these stakeholders a realistic elicitation and ranking of the scenarios is not feasible. A further scenario exploration makes no sense and should be omitted.
- **Gathering suitable decisions:** It is often difficult to gather the “correct” set of decisions/architectural approaches that fit to the given evaluation scenarios from the very beginning. One possibility to overcome this problem is to split the gathering process in two steps: First, elicit as many decisions in the architecture as the stakeholders are aware of and, second, relate those decisions to the evaluation scenarios.

- **Data for quantitative analysis:** Gathering supplemental data for quantitative analysis such as calculating scenario execution times (cf. Section 6.5.2) may be difficult and time-consuming. Generally, quantitative analysis should be done in later design iterations, as the architecture is more stable and more reliable information from early releases of the system are available.

8.9 Summary

In this chapter the approach and major results of the validation undertaken to prove the hypothesis of this work have been described. The validation was based on three case studies. In particular, the validation goals and the validation context have been presented. Moreover, the data that must be gathered for setting up the validation as well as the respective metrics used for interpretation have been explained. Additionally, the major validation results have been documented.

The general result of this chapter is that the hypothesis of this work could be confirmed. The number and severity of risks for each of the three pilot projects of the case study could be reduced drastically. This has lead to a significant quality improvement of the architectures of all three systems. In particular, the architectural quality for the CMS project has been improved by an average of 36%. At the same time, the number of architectural risks has been reduced by 70%. For the AAS project, the quality of the architecture has been improved by 74%. 11 of 13 risks have been resolved for AAS, leading to a risk reduction of 85%. Finally, the architectural quality of the AET project has been improved by 61%. The architectural risks have been reduced by 64%.

The results clearly show that the application of QUADRAD has significantly improved the architectural quality of the three software-intensive systems. This provides evidence to confirm the hypothesis of the thesis.

9 Summary and Outlook

9.1 The Problem Revisited

Over the past decade, the amount and complexity of software in system development has increased substantially – and it will significantly grow in future. For example, the amount of software in automobile electronics will increase by 8% per year, resulting in a €100 billion market in 2010 and reaching 13% of the total cost for a standard automobile [Mercer 2002]. The competitive pressure will rise accordingly. New features must be implemented in smaller development cycles and in a cost-effective way. A well-grounded knowledge of market needs and business-driven engineering approaches gain more relevance in order to remain competitive and to defend market positions.

Creating sustainable solutions that meet the business goals and market needs thus becomes a pivotal role in software systems engineering of an organization. Unfortunately, the increased complexity of software in the systems to be built and the failing of many organizations to adequately cope with this complexity during system development have lead to the fact that there often is a significant mismatch between the planned and the implemented system. This mismatch usually has serious consequences for the organization's financial investment, market share, and reputation.

Requirements engineering and architecture development take a central role in the overall system development life cycle and thus continue to be an area of growing importance. Errors generated during these phases are the most expensive to fix (e.g., [Pohl 1996], [Bass 2003]). Experience reports have shown that especially the transition from requirements to architecture is one of the most critical phases of development where the largest amount of fundamental mistakes of an organization's development effort are typically made (e.g., [Neumann 1995], [CHAOS 2001]). This work focuses on three research gaps that represent critical triggers to the problems of this transition:

1. There is no sufficient guidance in providing the architect with those requirements which are essential for architecture development.
2. There is no systematic support for making adequate design decisions in order to implement architecturally relevant requirements.
3. There is insufficient support for the identification and mitigation of unwanted effects of design decisions in architecture development.

The thesis aims at bridging these gaps. It is focused on mitigating serious problems associated with the gaps in the transition between requirements and architecture development.

9.2 The Research Contribution Revisited

This work has implications for the architecture development of software-intensive systems. It provides an approach that supports the creation of architectures that are better aligned to architecturally significant requirements and that include less design risks. The hypothesis is that with these results the quality of the resulting architectures can be increased, which, in turn, leads to a higher probability that these architectures permit the building of systems that match the intended business goals more accurately. In particular, this work provides the following research contributions:

The QUADRAD framework for Quality-Driven Architecture Development. This framework complements existing architecture development approaches by systematically addressing the given research gaps in the transition process from requirements to architecture. The framework includes activities, which support the development of architectures for software-intensive systems that are optimized to their essential quality requirements. The benefit of such an architecture is that it better permits building systems that meet the intended business goals and market needs of the development organization. This typically leads to higher quality systems, greater market shares, and a better reputation. The framework comprises activities for

- Identifying requirements, strategies, and mechanisms that drive architecture development;
- Making appropriate decisions that implement the essential requirements in the architecture by applying the strategies and mechanisms systematically;
- Evaluating the consequences of architectural decisions with respect to the achievement of essential quality requirements and in the light of an overall system perspective;

The Architecture Exploration Tool (AET). This research tool supports and automates essential activities of the QUADRAD framework. Particularly, it supports architecture evaluation activities and provides a systematic documentation of architecture design results. AET supports the capturing of requirements and their refinement with the help of scenarios. It further supports the prioritization of scenarios with respect to its significance for development and allows for a description of the major design decisions made to implement each scenario. AET also supports the documentation of trace information and allows traceability among quality requirements, refined scenarios, design decisions that pertain to a particular scenario, and risks associated to design decisions.

A metrics suite for evaluating architecture quality and quality improvement. The definition of the metrics follows the Goal Question Metric (GQM) approach. They rely on the assessment of the probability and impact of architectural risks and are used to analyze the effectiveness of the QUADRAD activities. The metrics are based on the fact that the higher the probability and impact of a risk the more critical it is for the development and the more likely the architecture will fail to achieve the intended requirements. In particular, the metrics suite includes measures

- to determine the severity of architectural risks,
- to estimate the associated quality of the architecture, and
- to compare the risk severity and architectural quality of two candidate architectures.

9.3 The Validation Revisited

Goals. The goal of the validation was to prove the hypothesis of this work. In order to do so, we had to test whether the QUADRAD framework improves the architectural quality of software-intensive systems. For this purpose, case studies based on three different pilot projects – a car multimedia system (CMS), an automotive assistance system (AAS), and an engineering support system (AET) – have been performed. By choosing three projects, a comparative and more representative validation for judging the quality improvement could be achieved.

Metrics. To perform the validation a set of metrics that rely on the assessment of the probability and impact of architectural risks have been defined. The metrics are based on the fact that the higher the probability and impact of a risk the more critical it is for the development and the more likely the architecture will fail to achieve the intended requirements.

Results. As a general result of the validation, the number and the severity of risks for each of the three pilot projects of the case study could be reduced drastically. This has lead to a significant quality improvement in the architecture of all three systems. In particular, the architectural quality for the CMS project has been improved by an average of 36%. At the same time, the number of architectural risks has been reduced by 70%. For the AAS project, the quality of the architecture has been improved by 74%. 11 of 13 risks have been resolved for AAS, leading to a risk reduction of 85%. Finally, the architectural quality of the AET project has been improved by 61%. The architectural risks have been reduced by 64%.

The results clearly show that the application of QUADRAD has significantly improved the architectural quality of the three software-intensive systems. The results also confirm the hypothesis of this work.

9.4 Outlook

While QUADRAD represents an effective approach to improving the quality of architectures, it also introduces opportunities for further research. The most promising topics are described next.

Quantitative Architecture Evaluation

Currently, QUADRAD uses scenario maps and qualitative analysis techniques such as attribute-specific questions in order to assess the architecture with respect to particular requirements. It also provides information and steps for estimating the execution time of performance scenarios. Further quantitative analysis approaches are beyond the scope of this work.

Advanced quantitative measurement techniques could improve the architectural evaluation capabilities of QUADRAD. They could also complement the architecture quality metrics introduced in Section 8.2, which are currently based on expert judgment of architectural risks. Two classes of metrics seem to be of special importance for further investigation: (1) metrics that are based on measuring the extent of quality attribute achievement and (2) metrics that are based on general architectural properties.

Metrics for measuring the extent of quality attribute achievement: To calculate quantitative metrics for quality attributes, additional data about the planned implementation of the architecture (e.g., use of a design framework such as Microsoft .NET®, type of programming language, amount of code generated etc.) and the environment the architecture is embedded in (e.g., CPU speed, speed of I/O devices, mean time between failures of hardware components etc.) must be gathered. Whereas the qualitative analysis used in QUADRAD shows which quality requirements cannot be achieved by the architecture under consideration, a quantitative analysis would allow more detailed QUADRAD results by estimating numbers that document to what extent the particular requirements cannot be satisfied. Musa [Musa 1999] and Smith [Smith 2002] provide promising starting points for further research with respect to detailed availability, reliability, and performance considerations.

Metrics that are based on general architectural properties: The measurement of architectural design properties is another possibility to gather quantitative data. Examples are coupling and cohesion [Heyliger 1994], fan-in/fan-out [Fenton 1991], and control flow metrics [Zhao 1998]. Most of these metrics are based on evaluating the dependency graph of an architecture. Rather than calculating the overall complexity of an architecture, the complexity of the flows in particular scenario maps could be calculated. This would support and detail the findings and assumption of a scenario analysis, especially for the attributes modifiability and maintainability.

Integration of Reconstruction Activities

An important prerequisite for revising an architecture (e.g., quality improvement or extension) is an appropriate architecture description. As discussed in Chapter 2, an architecture description should be organized into multiple views of the system, including a documentation of design elements and design decisions. Without such a description, an architecture revision that is understandable and that meets its requirements is difficult to achieve. Further, an architecture description is also critical for evaluating conformance between the architecture and its implementation in code.

Architectural reconstruction techniques and tools such as those described by O'Brian et al. [O'Brian 2002] support architectural documentation, and thus can be beneficial for QUADRAD architecture development efforts. Reconstruction information gathered from source code can, for example, be used as a second source to endorse the architect in estimating the architectural impact of requirements (QUADRAD Preparation). They can also be used to extend the QUADRAD Evaluation workflow with respect to activities for analyzing the drift between architecture and implementation.

Use of Architecture Description Languages

Architecture description languages (ADLs) represent a formal approach to describe a software architecture. Semi-formal modeling approaches based on UML are now also evolving (e.g., [MDA 2003]). The advantage of using ADLs lies in the ability to specify an architecture rigorously so that it can be analyzed in an automated manner. Many ADLs are supported by tools for doing useful analysis of architectures specified in the language (e.g., [Allen 1994], [Luckham 1995], [Garlan 1997], [Feiler 2004]).

The use of an ADL would allow the creation of architectural descriptions that could be simulated. This would especially support the scenario map analysis activities during a

QUADRAD Evaluation. With a tool-based simulation, the architect (and the review team) could gain a better insight into the capabilities actually implemented in the design elements that constitute the architecture. This would support the detection of design errors and flaws and help in validating the achievement of architecturally significant qualities. A formally specified architecture would also build a basis for automating traceability tasks.

However, converting an architecture into an ADL-based description is usually a time-consuming task. Often, there is a considerable effort for communicating the architecture to other stakeholders if no graphical representation is available. The cost and benefits of automated analysis should be carefully investigated.

Strategy and Mechanism Selection

One important activity in the QUADRAD Modeling workflow is the identification of architectural strategies and mechanisms for a set of architectural drivers (cf. Section 5.2). The strategies help to narrow the scope of appropriate solutions for the given drivers. Architectural mechanisms refine the corresponding strategies and add concrete detail to its implementation in the architecture. A starting point for further work could be the development and implementation of a database for cataloguing architectural strategies and mechanisms. Further, the database could be connected to AET (cf. Chapter 7) and appropriate recording, querying, and filtering options could be added to the tool. This would improve the supporting capabilities of AET.

The effectiveness of selecting appropriate strategies and mechanisms from a database/library depends on two issues: (1) the information provided for these entities and (2) managing the access to this information. The first issue is important for the architect to decide if the strategy or mechanism is capable of addressing the problems associated with the requirement. The second issue helps to make the decision making process more efficient. The optimization of the selection process could also be a promising starting point for further research.

10 Appendix: Architectural Patterns

This appendix provides a brief description of architectural patterns referenced in literature.

Abstract Factory [Gamma 1994] provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Abstract Manager [Liebenau 2001] focuses on the management of business objects in enterprise systems.

Acceptor-Connector [Schmidt 2000] decouples the connection and initialization of cooperating peer services in a networked system from the processing performed by the peer services after they are connected and initialized.

Activator [Stal 2000] helps to implement efficient on-demand activation and deactivation of services that are accessed by multiple clients.

Active Object [Schmidt 2000] decouples method execution from method invocation.

Adapter [Gamma 1994] converts the interface of a class into an interface expected by clients. Adapter allows classes to work together that could not do so otherwise because of incompatible interfaces.

Asynchronous Completion Token [Schmidt 2000] allows an application efficiently to demultiplex and process the responses of asynchronous operations it invokes on services.

Blackboard [Buschmann 1996] is useful for problems for which no deterministic solution strategies are known. In Blackboard several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.

Broker [Buschmann 1996] can be used to structure distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communication, such as forwarding requests, as well as for transmitting results and exceptions.

Cache Management [Grand 1998] focuses on caching objects in Java and on how to combine a cache with the Manager [Sommerlad 1997] pattern.

Cache Proxy [Buschmann 1996] implements caching inside a proxy that represents the data source from which one or multiple clients want to retrieve data.

Caching [Kircher 2004] describes how to avoid expensive re-acquisition of resources by not releasing the resources immediately after their use. The resources retain their identity, are kept in some fast-access storage, and are re-used to avoid having to acquire them again.

Client-Dispatcher-Server [Buschmann 1996] introduces an intermediate layer between clients and servers, the dispatcher component. It provides location transparency by means of a name service, and hides the details of the establishment of the communication connection between clients and servers.

Comparand [Costanza 2001] provides a means of interpreting differing objects as being the same in specific contexts. It does this by introducing an instance variable, the comparand, in each class of interest, and using it for comparison. Establishing the 'sameness' of differing objects is necessary when more than one reference refers conceptually to the same object.

Component Configurator [Schmidt 2000] allows an application to link and unlink its component implementations at run-time without having to modify, recompile, or statically relink the application.

Command Processor [Buschmann 1996] separates the request for a service from its execution. A command processor component manages requests as separate objects, schedules their execution, and provides additional services such as storing request objects for later undo.

Coordinator [Kircher 2004] describes how to maintain system consistency by coordinating the completion of tasks involving multiple participants, each of which can include a resource, a resource user and a resource provider. The pattern presents a solution such that in a task involving multiple participants, either the work done by all of the participants is completed or none are. This ensures that the system always stays in a consistent state.

Data Transfer Object [Fowler 2002] carries data, for example other objects or invocation parameters, between remote clients and servers. The encapsulation provided by this pattern reduces the number of remote operations required to transfer such data.

Deployer [Stal 2000] describes how to configure, deploy, and install software artifacts.

Disposal Method [Henney 2003] encapsulates the concrete details of object disposal by providing an explicit method for cleaning up objects, instead of abandoning the objects to be garbage collected or terminating them by deletion.

Double-Checked Locking Optimization [Schmidt 2000] reduces contention and synchronization overheads when critical sections of code must acquire locks in a thread-safe manner just once during program execution.

Eager Acquisition [Kircher 2004] describes how run-time acquisition of resources can be made predictable and fast by eagerly acquiring and initializing resources before their actual use.

Evictor [Kircher 2004] describes how and when to release resources to optimize resource management. The pattern allows different strategies to be configured to determine automatically which resources should be released, as well as when those resources should be released.

Extension Interface [Schmidt 2000] allows multiple interfaces to be exported by a component, to prevent bloating of interfaces and breaking of client code when developers extend or modify the functionality of the component.

Factory Method [Gamma 1994] defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Fixed Allocation [Noble 2000] allows memory consumption to be predicted by allocating the necessary memory at program initialization.

Flyweight [Gamma 1994] uses sharing to support large numbers of fine-grained objects efficiently.

Forwarder-Receiver [Buschmann 1996] provides transparent inter-process communication for software systems with a peer-to-peer interaction model. It introduces forwarders and receivers to decouple peers from the underlying communication mechanisms.

Half-Object Plus Protocol [Meszaros 1995] divides the responsibilities of an object into halves and assigns them to two interdependent half-objects when an object is used by two distributed clients. For efficiency reasons, each half-object implements the responsibility that is most used locally. This pattern lets the half-objects coordinate themselves via some protocol.

Half-Sync/Half-Async [Schmidt 2000] decouples asynchronous and synchronous service processing in concurrent systems, to simplify programming without unduly reducing performance. The pattern introduces two intercommunicating layers, one for asynchronous and one for synchronous service processing.

Interceptor [Schmidt 2000] allows services to be added to a framework transparently and triggered automatically when specific events occur.

Layer / Layering [Buschmann 1996] helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.

Lazy Acquisition [Kircher 2004] defers resource acquisitions to the latest possible time during system execution in order to optimize resource use.

Lazy Load [Fowler 2002] defers the loading of data from databases until it is first accessed.

Lazy Optimization [Auer 1996] optimizes the performance of a piece of software only after the design has been correctly determined.

Lazy Propagator [Feiler 1997] describes how, in a network of dependent objects, objects can determine when they are affected by the state changes of other objects, and therefore need to update their state.

Lazy State [Molin 1997] defers the initialization of the state of an object [Gamma 1994] until the state is accessed.

Leader/Followers [Schmidt 2000] that provides an efficient concurrency model where multiple threads take turns sharing a set of event sources in order to detect, demultiplex, dispatch, and process service requests that occur on the event sources.

Leasing [Kircher 2004] simplifies resource release by associating time-based leases with resources when they are acquired. The resources are automatically released when the leases expire and are not renewed.

Lookup [Kircher 2004] describes how to find and access resources, whether local or distributed, by using a lookup service as a mediating instance.

Manager [Sommerlad 1998] places functionality that applies to all objects of a class into a separate management object. This separation allows the independent variation of management functionality and its reuse for different object classes.

Master-Slave [Buschmann 1996] supports fault tolerance, parallel computation and computational accuracy. A master component distributes work to identical slave components and computes a final result from the results returned by the slaves.

Mediator [Gamma 1994] defines an object that encapsulates the way in which a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and allows their interaction to be varied independently.

Memento [Gamma 1994] encapsulates the state of an object in a separate, persistable object.

Microkernel [Buschmann 1996] applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and customer-specific parts. The microkernel also serves as a socket for plugging in these extensions and coordinating their collaboration.

Model-View-Controller [Buschmann 1996] divides an interactive application into three components. The model contains the core functionality and data, the view displays information to the user, and the controller handles user input. The view and controller together comprise the user interface. A change-propagation mechanism ensures consistency between the user interface and the model.

Monitor Object [Schmidt 2000] synchronizes concurrent method execution to ensure that only one method at a time runs within an object. It also allows an object's methods to cooperatively schedule their execution sequences.

Object Lifetime Manager [Levine 2001] is specialized for the management of singleton objects in operating systems that do not support static destructors properly, such as real-time operating systems.

Object Pool [Grand 1998] manages the reuse of objects of a type that is expensive to create, or of which only a limited number can be created.

Page Cache [Trowbridge 2003] improves response times when dynamically-generated Web pages are accessed. A page cache is associated with a Web server that uses it to store accessed pages indexed by their URLs. When the same URL is requested, the Web server queries the cache and returns the cached page instead of dynamically generating its contents again.

Partial Acquisition [Kircher 2004] describes how to optimize resource management by breaking up acquisition of a resource into multiple stages. Each stage acquires part of the resource, dependent upon system constraints such as available memory and the availability of other resources.

Passivation [Völter 2002] persists and activates memory representations of component instances to and from persistent storage.

Pipes and Filters [Buschmann 1996] provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allows you to build families of related systems.

Pooled Allocation [Noble 2000] pre-allocates a pool of memory blocks, recycling them when returned.

Pooling [Kircher 2004] describes how expensive acquisition and release of resources can be avoided by recycling resources that are no longer needed. Once the resources are recycled and pooled, they lose their identity and state.

Presentation-Abstraction-Control [Buschmann 1996] defines a structure for interactive software systems in the form of a hierarchy of cooperating agents. Every agent is responsible for a specific aspect of the application’s functionality and consists of three components: presentation, abstraction, and control. This subdivision separates the human-computer interaction aspects of the agent from its functional core and its communication with other agents.

Proactor [Schmidt 2000] allows event-driven applications to efficiently demultiplex and dispatch service requests triggered by the completion of asynchronous operations, to achieve the performance benefits of concurrency without incurring certain of its liabilities.

Proxy [Gamma 1994] provides a surrogate or placeholder for another object, to control access to it.

Proactive Resource Allocation [Cross 2002] anticipates system changes and plans necessary resource allocations ahead of time, with the goal of maintaining system performance even under changed conditions.

Publisher-Subscriber [Buschmann 1996] helps to keep the state of cooperating components synchronized. To achieve this it enables one-way propagation of changes: one publisher notifies any number of subscribers about changes to its state.

Reactor [Schmidt 2000] allows event-driven applications to demultiplex and dispatch service requests that are delivered to an application from one or more clients.

Reflection [Buschmann 1996] provides a mechanism for changing the structure and behavior of software systems dynamically. A meta level provides information about selected system properties and makes the software self-aware.

Resource Exchanger [Sane 1996] reduces a server's load when allocating and using resources by sharing common buffers that are passed between clients and servers.

Resource Lifecycle Manager [Kircher 2004] decouples the management of the lifecycle of resources from their use by introducing a separate Resource Lifecycle Manager, whose sole responsibility is to manage and maintain the resources of an application.

Singleton [Gamma 1994] ensures a class has only one instance, and provides a global point of access to it.

Slicing [Buschmann 1996] supports a relaxed layering of a system.

Sponsor-Selector [Wallingford 1997] separates three fundamentally different responsibilities: recommending a resource, selecting among resources, and using a resource.

State [Gamma 1994] allows an object to alter its behavior when its internal state changes.

Strategy [Gamma 1994] encapsulates logic, such as algorithms, into interchangeable classes that are independent of client requests.

Strategized Locking [Schmidt 2000] parameterizes synchronization mechanisms that protect a component's critical sections from concurrent access.

Thread-Local Memory Pool [Sommerlad 2002] allows a memory allocator to be created for each thread. This helps to reduce synchronization overheads, since dynamic memory allocations are performed from a thread-local pool of pre-allocated memory.

Thread-Safe Interface [Schmidt 2000] minimizes locking overhead and ensures that intra-component method calls do not incur 'self-deadlock' by trying to reacquire a lock that is held by the component already.

Thread-Specific Storage [Schmidt 2000] allows multiple threads to use one `logically global' access point to retrieve an object that is local to a thread, without incurring locking overhead on each object access.

Thread Pooling [Petriu 1997] describes how to bound the number of threads used and how to reuse unused threads.

Variable Allocation [Noble 2000] optimizes memory consumption by performing memory allocations on demand.

View Handler [Buschmann 1996] helps to manage all views that a software system provides. A view handler component allows clients to open, manipulate and dispose of views. It also coordinates dependencies between views and organizes their update.

Virtual Proxy [Gamma 1994] loads or constructs the object that the proxy represents on demand.

Whole-Part [Buschmann 1996] helps with the aggregation of components that together form a semantic unit. An aggregate component, the Whole, encapsulates its constituent components, the Parts, organizes their collaboration, and provides a common interface to its functionality. Direct access to the Parts is not possible.

Wrapper Facade [Schmidt 2000] encapsulates the functions and data provided by existing non-object-oriented APIs within more concise, robust, portable, maintainable, and cohesive object-oriented class interfaces.

Acronyms

AAS	Automotive Assistance System
ADD	Attribute-Driven Design
ADL	Architecture Description Language
AET	Architecture Exploration Tool
AHP	Analytic Hierarchy Process
ALMA	Architecture-Level Modifiability Analysis
APSM	Architecture-Level Prediction of Software Maintenance
ATAM	Architecture Tradeoff Analysis Method
AWG	Architecture Working Group
CAFÉ	Concepts to Application in System Family Engineering
CASE	Computer-Aided Software Engineering
CD	Compact Disc
CMS	Car Multimedia System
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
COTS	Components-Off-the-Shelf
CPU	Central Processing Unit
DES	Data Encryption Standard
DLL	Dynamic Link Library
ECU	Electronic Control Unit
ESAPS	Engineering Software Architectures, Processes, and Platforms for System Families
EVCS	Embedded Vehicle Control System
FAST	Family-Oriented Abstraction, Specification, and Translation
FIFO	First In, First Out
GPS	Global Positioning System
GUI	Graphical User Interface
HMI	Human-Machine Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
I/O	Input/Output

IPC	Inter Process Communication
IEEE	Institute of Electrical and Electronics Engineers
ITEA	Information Technology for European Advancement
MFC	Microsoft® Foundation Classes
MIL	Module Interconnection Language
MIPS	Million Instructions per Second
MOST	Media Oriented Systems Transport
MTBF	Mean Time Between Failures
MVC	Model View Controller
NFR	Non-Functional Requirement
ODBC	Open Database Connectivity
OMT	Object Modeling Technique
OOSE	Object-Oriented Software Engineering
QUADRAD	Quality-Driven Architecture Development
QUADRAD-E	Evaluation Workflow of →QUADRAD
QUADRAD-M	Modeling Workflow of →QUADRAD
QUADRAD-P	Preparation Workflow of →QUADRAD
RAM	Random Access Memory
RMI	Remote Method Invocation
RUP	Rational Unified Process
RDS	Radio Data System
RPC	Remote Procedure Call
RSEB	Reuse-Based Software Engineering Business
RTOS	Real-Time Operating System
SAAM	Software Architecture Analysis Method
SPE	Software Performance Engineering
SQL	Structured Query Language
SRE	Software Reliability Engineering
UCM	Use Case Navigator
UML	Unified Modeling Language
VNS	Vehicle Navigation System
VP	Variation Point
WWW	World Wide Web
XML	eXtensible Markup Language

Glossary

Abstraction	A representation in terms of presumed essentials, with a corresponding suppression of the non-essential.
Activity	Describes a unit of work in the development process that an individual playing the role represented by the →*worker* may be asked to perform.
Analysis questions	Questionnaire used during an →*architecture* evaluation to get a basic overview on important properties of the →*architecture*.
Analytic Hierarchy Process	Decision support methodology that is based on pair-wise comparisons.
Architect	The person or persons responsible for evolving and maintaining the system's →*architecture*.
Architecting	→*Architecture development*
Architectural decision	Decision made during architectural design, made from a broad-scoped or system perspective. Architectural decisions are →*strategic decisions* and have a high systemic and business impact.
Architectural description language	A formal language for describing a (software/hardware) system in terms of its architectural elements and the relationships among them.
Architectural driver	A →*requirement* that has a strong impact on the overall system (systemic impact) and that is important for achieving the business goals.
Architectural infrastructure	Consists of those →*design elements* that every application functionality requires to execute. The architectural infrastructure is essential for achieving the most important →*quality attributes* and business goals.
Architectural iteration	Evolution from one →*architectural release* to another. During an architectural iteration an →*architecture* is evolved from the current state of properties to a higher elaborated version with additional or improved properties. See also: →*iteration*.

Architectural mechanism	Describes a collection of component and connection types (and their basic responsibilities) to use for application to a design problem. Implements one or more →*architectural strategies*. See also: →*architectural style*, →*architectural pattern*.
Architectural pattern	Expresses a fundamental structural organization schema for →*software-intensive systems*. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.
Architectural quality	Measures the overall quality of an →*architecture*. Depends on the probability and impact of the →*architectural risks*.
Architectural release	The outcome of a sequence of →*architectural iterations*.
Architectural risk	A risk within the →*architecture*, potentially caused by →*design decisions* that are not technically sound or that do not support the business goals of the development effort. The loss associated with the risk could be in the form of diminished quality of the end product, increased cost, delayed completion, or failure.
Architectural strategy	Describes a general principle to solve a particular class of design problems. For example, reducing the communication overhead between software components is a general strategy for addressing performance problems.
Architectural structure	Prescribes the type of components and relationships to be used to describe the software system as well as their properties. See also: →*architectural view*.
Architectural style	A specialization of →*design element* types together with a set of constraints on how they can be used. Codifies architectural solutions with predictable properties to be applied in a certain problem context and to facilitate reuse of these solutions.
Architectural view	Describes an architecture from a particular perspective or vantage point, covering particular concerns and omitting entities that are not relevant to this perspective. Each view is based on the characteristics of the →*architectural view type*.
Architectural view model	→*Architectural view*
Architectural view type	Sets the context of an →*architectural view* (e.g., purpose of the view, component and connection types etc.).

Architecturally significant requirement	→*Requirement* that has a strong impact on the overall system (systemic impact.)
Architecture	The structure or structures of the system, which comprise software [and hardware] components, the externally visible properties of those components, and the relationships among them.
Architecture description	A collection of products to document an →*architecture*, usually organized into one or more →*architectural views* of the system, including →*design elements*, →*design decisions*, and →*variation points*.
Architecture design	The process of creating an →*architecture*. Comprises activities to make appropriate →*architectural decisions* such that the resulting architecture permits the achievement of most significant system →*requirements*.
Architecture development	→*Architecture design,* →*architecture evaluation* and →*architecture recovery.*
Architecture evaluation	The process of verifying an architecture against its (quality) requirements.
Architecture Exploration Tool	A research tool that supports an architect in documenting results and managing information of →*architecture development* within the QUADRAD framework.
Architecture reconstruction	Recovering an →*architecture description* from source or binary code of a system.
Architecture recovery	→*Architecture reconstruction*
Artifact	Input and output work products of the process; →*workers* use artifacts to perform activities, and produce artifacts in the course of performing activities.
Change	The degree to which a system, analyzed at a given time, is determined to be different from the same system analyzed at an earlier time.
Change management	The activity of controlling and tracking changes to →*artifacts*.
Change scenario	Covers typical anticipated future changes to a system. See also: →*configuration scenario*, →*evaluation scenario*, →*stress scenario*, →*usage scenario*.

Component	The principal computational element or data store that execute in a system.
Component design	The process of specifying the internal details of the →*components* of an →*architecture*. See also: →*design*, →*architecture design*.
Connector	A relationship or runtime pathway of interaction between two or more components.
Constraint	The expression of some semantic condition that must be preserved.
Dependency	A relationship between two entities (e.g., →*design elements*), in which a change to one entity will affect the other entity.
Design	The part of a development process whose primary purpose is to decide how the system will be implemented. During design, strategic and tactical decisions are made to meet the required functional and quality requirements of a system. See also: →*architecture design*, →*component design*.
Design activity	Describes a unit of work in the design process. Examples are introducing new →*design elements* into the (architecture) design, refining or changing the behavior of existing design elements, and documenting a →*variation point.*
Design constraint	Special set of →*requirements* for which a particular pre-defined solution must be taken into account. Usually results in →*design decisions* that cannot be negotiated.
Design decision	A development decision that has implications on artifacts. See also: →*architectural decision*, →*strategic design decision*, →*tactical design decision*.
Design element	→*Component* or →*connector*
Design operations	→*Design activities*
Design pattern	Provides a scheme for refining the subsystems or components of a →*software-intensive system*, or the relationships between them. It describes a commonly recurring structure of communicating components that solves a general design problem within a particular context.
Detailed design	→*Component design*

Device	A piece of hardware that has no computational resources.
Domain	An area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area.
Efficiency	Relative extent to which a resource is utilized.
Encapsulation	A means for grouping internal primitives or data and limiting external access.
Entity	A concrete or abstract thing of interest.
Evolution	The rate of →*change*.
Evaluation scenario	Used for the analysis of complex systems. See also: →*change scenario*, →*configuration scenario*, →*stress scenario*, →*usage scenario*.
Event	Some occurrence that may cause the state of a system to change.
Framework	An extensible structure for describing a set of concepts, activities, and techniques necessary for a complete product development or manufacturing process.
Functional requirement	Describe which →*functionality* has to be provided by the system.
Functionality	In the context of →*requirements engineering*: the set of functions that an end user is able to access or perceive.
Interface	A boundary across which two independent entities meet and interact or communicate with each other. The outside view of a →*design element*, which emphasizes its abstraction while hiding its internal structure.
Iteration	A distinct sequence of activities resulting in a release. See also: →*architectural iteration*.
Legacy system	An older, potentially moldy system that must be preserved for any number of economic or social reasons, yet must coexist with newly developed elements.
Methodology	A regular and systematic way of accomplishing something. The detailed, logically ordered plans or procedures followed to accomplish a task or attain a goal.

Module	An implementation unit of software that provides a coherent unit of functionality.
Process	Technically: The activation of a single thread of control.
Product line	A set of →*software-intensive systems* sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of reusable core assets in a prescribed way.
Project	An undertaking typically requiring concerted effort that is focused on developing or maintaining a specific product or products. Usually, a project has its own funding, accounting, and delivery schedule.
Quality	The degree to which a system possesses a desired combination of →*quality attributes*.
Quality attribute	Addresses a quality characteristic of a system, for example, performance, security, availability, modifiability, and portability.
Quality requirement	A →*requirement* that typically has a strong impact on the →*architecture*. A quality requirement addresses one or more →*quality attributes*.
Quality scenario	Prescribes a structure to document the →*quality requirements* in a concrete way.
Real-Time System	A system whose essential processes must meet certain critical time deadlines. A hard-real-time system must be deterministic. Missing a deadline may lead to catastrophic results.
Refinement	The process of gradually disclosing information across a series of descriptions.
Relationship	A connection between two entities (e.g., →*components*).
Requirement	A condition or capability that specifies (a portion of) what the system should do.
Requirements engineering	Emphasizes the utilization of techniques that ensure the completeness, consistency, and relevance of the system →*requirements*.

Requirements traceability	Possibility to link the →*requirements* backwards to their sources (e.g., the customer) and forward to the resulting system development →*artifacts* (e.g., →*design elements*). Requirements traceability is critical in determining the impact of potential requirements changes in a system – e.g., with respect to the change effort. See also: →*traceability*.
Resolution rule	Describes how a →*variation point* is bound.
Responsibility	Denotes the obligation of a →*design element* to provide a certain behavior. The functionality of a design element.
Reuse	Repeated use of an →*artifact*.
Risk	The possibility of suffering loss. An ongoing or impeding concern that has a significant probability of adversely affecting the success of major milestones.
Scenario	A means to specifying requirements for a →*software-intensive system* in a concrete way. A scenario is defined as a short statement describing an interaction of one of the →*stakeholders* or an interaction of another system with the system under consideration.
Scenario map	Graphical representation of the "execution path" of a →*scenario* and the associated →*design elements*.
Sensitivity point	One or more →*architectural decisions* to which a quality attribute is highly correlated.
Severity (of a risk)	Measures how high a risk is. The severity depends on the probability and impact of the risk. See also: →*architectural risk*.
Software architecture	→*Architecture*
Software product line	→*Product line*
Software-intensive system	A system with a high share of software.
Stakeholder	Any person or representative of an organization who has a vested interest in the outcome of a project or whose opinion must be accommodated. A stakeholder can, for example, be an end user, an →*architect*, a requirements analyst, a developer, or a project manager.

Strategic (design) decision	A development decision that has sweeping architectural implications. See also: *→architectural decision*, *→tactical design decision*.
Stress scenario	Covers extreme situations or changes that are expected to “stress” the system. Stress scenarios expose the limits or boundary conditions of the current design, exposing possibly implicit assumptions. See also: *→configuration scenario*, *→evaluation scenario*, *→change scenario*, *→usage scenario*.
System	A collection of software and/or hardware performing one or several tasks for a common purpose.
System architecture	*→Architecture*
Tactical (design) decision	A development decision that has local architectural implications. See also: *→architectural decision*, *→strategic design decision*.
Trace	A link between (parts of) two *→artifacts* (e.g., *→requirements* and *→design elements*).
Traceability	Supports the validation if the (architecture) design and implementation of a system satisfies the *→requirements* for that system (in the narrower sense). See also: *→requirements traceability*.
Tradeoff-point	One or more *→architectural decisions* that influences some *→quality attributes* in a positive and other quality attributes in a negative way.
Type	A predicate characterizing a collection of entities.
Unified Modeling Language	A general-purpose modeling language that can be used to describe the static structure and dynamic behavior of a system.
Usage scenario	Describes a user’s intended interaction with the completed, running system. See also: *→configuration scenario*, *→evaluation scenario*, *→change scenario*, *→stress scenario*.
Use case scenario	Describes a sequence of actions that use a piece of functionality of the system to provide the user with a valuable result. Makes functional requirements concrete.

Workflow	Represents the flow of control between →*activities* and →*artifacts*.
Work product	→*Artifact*
Worker	Represents a role played by individuals on the project, and defines how they carry out work.

Bibliography

[Abowd 1997] G. Abowd, L. Bass, P. Clements, R. Kazman, L. Northrop, A. Zamerski: "Recommended Best Industrial Practice for Software Architecture Evaluation," Technical Report CMU/SEI-96-TR-025, Software Engineering Institute, Carnegie Mellon University, January 1997.

[Alexander 1977] C. Alexander, S. Ishikawa, M. Silverstein with M. Jacobson, I. Fiksdahl-King, S. Angel: "A Pattern Language - Towns Buildings Construction," Oxford Univ. Press, 1977.

[Alexander 1979] C. Alexander: "The Timeless Way of Building," Oxford Univ. Press, 1979.

[Allen 1994] R. Allen, D. Garlan: "Beyond Definition/Use: Architectural Interconnection," *Proceedings of the ACM Interface Definition Language Workshop*, SIGPLAN Notices, August 1994.

[Ambler 2004] S. W. Ambler: "Agile Model Driven Development with UML 2," Cambridge University Press, 2004.

[Ashby 1956] W. R. Ashby: "An Introduction to Cybernetics," Methuen, New York, 1956.

[AT&T 1993] "Best Current Practices: Software Architecture Validation," Technical Report, AT&T, 1993.

[ATKearney 2001] "Software-betriebene Fahrzeugsysteme bestimmen die Zukunft des Automobils," ATKearney Global Automotive, 2001.

[Auer 1996] K. Auer: "Lazy Optimization: Patterns for Efficient Smalltalk Programming," In: J.O. Coplien, N. Kerth, and J. Vlissides (eds.): *Pattern Languages of Program Design 2*, Addison-Wesley, 1996.

[Bachmann 2000] F. Bachmann, L. Bass, G. Chastek, P. Donohoe, F. Peruzzi: "The Architecture Based Design Method," Technical Report CMU/SEI-2000-TR-001, Software Engineering Institute, Carnegie Mellon University, January 2000.

[Barbacci 1995] M. Barbacci, M. H. Klein, T. A: Longstaff, C. B. Weinstock: "Quality Attributes," Technical Report CMU/SEI-95-TR-21, Software Engineering Institute, Carnegie Mellon University, December 1995.

[Barbacci 2000] M. Barbacci, R. J. Ellison, C. B. Weinstock, W. G. Wood: "Quality Attribute Workshop Participants Handbook," Special Report CMU/SEI-2000-SR-001, Software Engineering Institute, Carnegie Mellon University, January 2000.

[Babar 2004] M. A. Babar, L. Zhu, R. Jeffery: “A Framework for Classifying and Comparing Software Architecture Evaluation Methods,” *Procedings of ASWEC-2004*, 2004.

[Basili 1992] V.R. Basili: “Software Modeling and Measurement: The Goal Question Metric Paradigm,” Computer Science Technical Report Series, CS-TR-2956 (UMIACS-TR-92-96), University of Maryland, College Park, MD, September 1992.

[Bass 1998] L. Bass, P. Clements, R. Kazman: “Software Architecture in Practice,” Addison-Wesley, 1998.

[Bass 2000] L. Bass, M. Klein, F. Bachman: “Quality Attribute Design Primitives,” Technical Report CMU/SEI-2000-TN-017, Software Engineering Institute, Carnegie Mellon University, December 2000.

[Bass 2003] L. Bass, P. Clements, R. Kazman: “Software Architecture in Practice (Second Edition),” Addison-Wesley, 2003.

[Beck 1989] K. Beck, W. Cunningham: “A Laboratory For Teaching Object-Oriented Thinking,” In: N. Meyrowitz (Ed), *Special Issue of SIGPLAN Notices*, Vol. 24, No. 10, pp. 1-6, October 1989.

[Bengtsson 1999] P. Bengtsson, J. Bosch: “Architecture-Level Prediction of Software Maintenance,” *Proceedings of 3rd EuroMicro Conference on Maintenance and Reengineering (CSMR-99)*, Los Alamitos, CA: IEEE CS Press, pp. 139-147, 1999.

[Berard 2001] E. V. Berard: “The Origins of Object-Oriented Technology,” Online Article, URL: http://www.toa.com/pub/origins_article.txt, 2001.

[Böckle 2004] G. Böckle, P. Knauber, K. Pohl, K. Schmid (ed.): “Software-Produktlinien – Methoden, Einführung und Praxis,” dpunkt-Verlag, 2004.

[Boehm 1978] B. W. Boehm, J. R. Brown, J. R. Kaspar: “Characteristics of Software Quality,” TRW Series of Software Technology, Amsterdam, North Holland, 1978.

[Boehm 1996] B. Boehm, H. In: “Identifying Quality-Requirement Conflicts,” *IEEE Software*, Vol. 13, No. 2, pp. 25-35, March 1996.

[Booch 1994] G. Booch: “Object-Oriented Analysis and Design with Applications (Second Edition),” Benjamin/Cummings Publishing Company, 1994.

[Booch 1996] G. Booch: “Object Solutions – Managing the Object-Oriented Project,” Addison-Wesley, 1996.

[Booch 1999] G. Booch, J. Rumbaugh, I. Jacobson: “The UML Modeling Language User Guide,” Addison-Wesley, 1999.

[Bosch 2000] J. Bosch: “Design and Use of Software Architectures – Adopting and Evolving a Product-Line Approach,” Addison-Wesley, 2000.

[Bowman 1999] T. Bowman, R. Holt, N. Brewster: "Linux as a Case Study: Its Extracted Software Architecture," *Proceedings of the 21st International Conference on Software Engineering*, ACM Press, 1999.

[Broekman 2003] B. Broekman, E. Notenboom: "Testing Embedded Software," Addison-Wesley, 2003.

[Brooks 1995] F. P. Brooks: "The Mythical Man-Month (20th Anniversary Edition)," Addison-Wesley, 1995.

[Buhr 1996] R. J. A. Buhr, R. S. Casselman: "Use Case Maps for Object-Oriented Systems," Prentice-Hall, USA, 1996.

[Buhr 1998] R. J. A. Buhr: "Use Case Maps as Architectural Entities for Complex Systems," *IEEE Transactions on Software Engineering*, Vol. 24, No. 12, pp. 1131-1155, December 1998.

[Buschmann 1996] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: "Pattern-Oriented Software Architecture: A System of Patterns," Wiley, 1996.

[C4ISR 1998] "Levels of Information Systems Interoperability (LISI)," C4ISR Architectures Working Group, March 1998.

[CASE 2003] "Online Case Tools Index," URL: http://www.cs.queensu.ca/Software-Engineering/tools.html, 2003.

[CHAOS 2001] "Extreme Chaos," Technical Report, The Standish Group International, Inc., 2001.

[Chastek 2002] G. J. Chastek (ed.): "Software Product Lines," *Proceedings of 2nd Software Product Line Conference (SPLC-2),* Lecture Notes in Computer Science (LNCS), Vol. 2379, Springer-Verlag, 2002.

[Chung 1995] L. Chung, B. A. Nixon, E. Yu: "Using Non-Functional Requirements to Systematically Select Among Alternatives in Architectural Design," *Proceedings of ICSE-17 Workshop on Architectures for Software Systems*, Seattle, Washington, April 24-28, 1995.

[Chung 1999] L. Chung, D. Gross, E. Yu: "Architectural Design to Meet Stakeholder Requirements," In: P. Donohoe (Ed.): *Software Architecture*, pp. 545-564, Kluwer Academic Publishing, 1999.

[Clements 2002] P. Clements, R. Kazman, M. Klein: "Evaluating Software Architectures: Methods and Case Studies," Addison-Wesley, 2002.

[Clements 2003] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford: "Documenting Software Architectures: Views and Beyond," Addison-Wesley, 2003.

[Coplien 1992] J. O. Coplien: "Advanced C++ Programming Styles and Idioms," Addison-Wesley, 1992.

[Coplien 1995] J. O. Coplien, D. C. Schmidt: “Pattern Languages of Program Design,” Addison-Wesley, 1995.

[Costanza 2001] P. Costanza, A. Haase: “The Comparand Pattern,” *Proceedings of the 6th European Conference on Pattern Languages of Programs*, Germany, July 2001.

[Cross 2002] J. K. Cross: “Proactive and Reactive Resource Reallocation, Workshop: Patterns in Distributed and Embedded Real-time Systems,” http://www.posa3.org/workshops/RealTimePatterns, OOPSLA 2002, Seattle, USA, 2002.

[Dahl 1966] O. J. Dahl, K. Nygaard: “SIMULA - an ALGOL-Based Simulation Language,” *Communications of the ACM*, Vol. 9, No. 9, pp. 671-678, September 1966.

[Dashofy 2001] E. M. Dashofy, A. v. d. Hoek, R. N. Tylor: “A Highly-Extensible, XML-Based Architecture Description Language,” *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, pp. 103-112, Amsterdam, The Netherlands, 2001.

[Davis 1990] A. M. Davis: “Software Requirements – Analysis and Specification,” Prentice Hall, 1990.

[Davis 1993] A. M. Davis: “Software Requirements: Objects, Functions, and States (Second Edition),” Prentice Hall, 1993.

[DeRemer 1976] F. DeRemer, H. H. Kron: “Programming-in-the-Large Versus Programming-in-the-Small,” *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 2, pp. 80-86, June 1976.

[Dolan 2001] T. J. Dolan: “Architecture Assessment of Information-System Families,” PhD Thesis, Technical University of Eindhoven, 2001.

[Donohoe 2000] P. Donohoe (ed.): “Software Product Lines – Experience and Research Directions,” Kluwer Academic Publishers, 2000.

[Douglas 1999] B. P. Douglas: “Doing Hard Time - Developing Real-Time Systems With UML, Objects, Frameworks, and Patterns,” Addison-Wesley, 1999.

[Feiler 1997] P. Feiler, W. Tichy: “Lazy Propagator, in Propagator: A Family of Patterns,” *Proceedings of TOOLS-23*, 1997.

[Feiler 2004] P. Feiler, D. P. Gluch, J. J. Hudak, B. A. Lewis: “Embedded Systems Architecture Analysis Using SAE AADL,” Technical Note CMU/SEI-2004-TN-05, Software Engineering Institute, Carnegie Mellon University, 2004.

[Fenton 1991] N. E. Fenton: “Software Metrics: A Rigorous Approach,” Chapman and Hall, 1991.

[Fowler 1997] M. Fowler: “Analysis Patterns: Reusable Object Models,” Addison-Wesley, 1997.

[Fowler 2002] M. Fowler: “Patterns of Enterprise Application Architecture,” Addison-Wesley, 2002.

[Gacek 1995] C. Gacek, A. Abd-Allah, B. Clark, B. Boehm: “On the Definition of Software Architecture,” Proceedings of the *First International Workshop on Architectures for Software Systems*, 1995.

[Gamma 1994] E. Gamma, R. Helm, R. Johnson, J. Vlissides: “Design Patterns – Elements of Reusable Object-Oriented Software,” Addison-Wesley, 1994.

[Gane 1979] C. Gane, T. Sarson: “Structured Systems Analysis – Tools and Techniques,” Prentice-Hall, Englewood Cliffs, NJ, 1979.

[Garlan 1993] D. Garlan, M. Shaw: “An Introduction to Software Architecture,” Advances in Software Engineering and Knowledge Engineering, Vol. I, V. Ambriola and G. Tortora (ed.), World Scientific Publishing Company, New Jersey, 1993.

[Garlan 1995] D. Garlan, D. Perry: “Introduction to the Special Issue on Software Architecture,” *IEEE Transactions on Software Engineering*, Vol. 21, No. 4, pp. 269-274, 1995.

[Garlan 1997] David Garlan, Robert T. Monroe, David Wile: “ACME: An Architecture Description Interchange Language,” *Proceedings of CASCON '97*, November 1997.

[Garland 2003] J. Garland, R. Anthony: “Large-Scale Software Architecture – A Practical Guide Using UML,” John Wiley & Sons, 2003.

[Google 2003] “Decision Support Tools (Online),” URL: http://directory.google.com/Top/Computers/Software/Databases/Data_Warehousing/Decision_Support_Tools/

[Grand 1998] M. Grand: “Patterns in Java – Volume 1,” John Wiley & Sons, Inc., 1998.

[Grimm 2003] K. Grimm: “Software Technology in an Automotive Company – Major Challenges,” *Proceedings of 25th International Conference on Software Engineering (ICSE-2003)*, pp. 498-503, May 2003.

[Harris 1995] D. Harris, H. Reubenstein, A. Yeh: “Reverse Engineering to the Architectural Level,” *Proceedings of the 17th International Conference on Software Engineering*, ACM Press, 1995.

[Hauser 1988] J. R. Hauser, D. Clausing: “The House of Quality,” *Harvard Business Review*, No. 5, pp. 63-73, May 1988.

[Henderson 1990] R. M. Henderson, K. B. Clark: “Architectural Innovation: The Reconfiguration of Existing Product Technologies and the Failure of Established Firms,” *Administrative Science Quarterly*, No. 35, 1990.

[Henney 2003] K. Henney: “Factory and Disposal Methods,” Proceedings of VikingPLoP 2003, Bergen, Norway, http://www.curbralan.com, 2003.

[Henry 1981] S. Henry, D. Kafura: “Software Structure Metrics Based on Information Flow,” *IEEE Transactions on Software Engineering*, Vol. SE-7, No. 5, pp. 510-518, September 1981.

[Heyliger 1994] G. E. Heyliger: “Coupling,” *Encyclopedia of Software Engineering*, Volume 1, pp. 220-228, Wiley 1994.

[Hofmeister 2000] C. Hofmeister, R. Nord, D. Soni: “Applied Software Architecture,” Addison-Wesley, 2000.

[Howard 2001] B. Howard, O Paridaens, B. Gamm: “Information Security: Threats and Protection Mechanisms,” *Alcatel Telecommunications Review*, 2nd Quarter, pp. 117-121, 2001.

[Hwang 1987] C. L. Hwang, M. J. Lin: “Group Decision Making under Multiple Criteria,” Springer-Verlag, Berlin, 1987.

[IEEE 1990] “IEEE Standard Glossary of Software Engineering Terminology,” IEEE Standard 610.12-1990, 1990.

[IEEE 1998] “IEEE Software Quality Metrics Methodology,” IEEE Standard 1061-1998, 1998.

[IEEE 1998b] “IEEE Recommended Practice for Software Requirements Specifications,” IEEE Standard 830-1998, 1998.

[IEEE 1999] “IEEE Recommended Practice for Architectural Description of Software-Intensive Systems,” IEEE Standard P1471, IEEE Architecture Working Group (AWG), 1999.

[ISO 1992] “Information Technology - Software Product Evaluation - Quality Characteristics and Guidelines for Their Use,” ISO 9126, International Organization for Standardization, Geneva, 1992.

[Isensee 2000] S. Isensee, K. Vredenburg: “User-Centered Design and Development,” In: P. C. Clements (ed.): *Constructing Superior Software*, Macmillan Technical Publishing, pp. 43-79, 2000.

[Issarny 1998] V. Issarny, T. Saridakis, A. Zarras: “Multi-View Description of Software Architectures,” *Third International Software Architecture Workshop*, 1998.

[Jacobson 1992] I. Jacobson: “Object-Oriented Software Engineering – A Use Case Driven Approach,” Addison-Wesley, 1992.

[Jacobson 1997] I. Jacobson, M. Griss, P. Jonsson: "Software Reuse – Architecture, Process and Organization for Business Success," Addison-Wesley, 1997.

[Jacobson 1999] I. Jacobson, G. Booch, J. Rumbaugh: "The Unified Software Development Process," Addison-Wesley, 1999.

[Jalote 1994] P. Jalote: "Fault Tolerance in Distributed Systems," Prentice Hall, 1994.

[Jazayeri 2000] M. Jazayeri, A. Ran, F. v. d. Linden: "Software Architecture for Product Families - Principles and Practice," Addison-Wesley, 2000.

[Jones 1994] C. Jones: "Assessment and Control of Software Risks," Prentice-Hall, Englewood Cliffs, NJ, 1994.

[Karlsson 1997] J. Karlsson, K. Ryan: "A Cost-Value Approach for Prioritizing Requirements," *IEEE Software*, Vol. 14, No. 5, pp. 67-74, September/October 1997.

[Kazman 1994] R. Kazman, L. Bass: "Toward Deriving Software Architectures from Quality Attributes," Technical Report CMU/SEI-94-TR-10, Software Engineering Institute, Carnegie Mellon University, August 1994.

[Kazman 1996] R. Kazman, G. Abowd, L. Bass, P. Clements: "Scenario-Based Analysis of Software Architecture," *IEEE Software*, Vol. 13, No. 6, pp. 47-56, 1996.

[Kazman 1998] R. Kazman, J. Carriere: "View Extraction and View Fusion in Architectural Understanding," *Proceedings of the 5th International Conference on Software Reuse*, Victoria, BC, Canada, June 1998.

[Kazman 1999] R. Kazman, M. Klein, P. Clements: "ATAM: Method for Architecture Evaluation," Technical Report CMU/SEI-2000-TR-004, Software Engineering Institute, Carnegie Mellon University, October 1999.

[Kazman 2000] R. Kazman, S. J. Carriere, S. G. Woods: "Toward a Discipline of Scenario-Based Architectural Engineering," *Annals of Software Engineering*, Vol. 9, pp. 5-33, 2000.

[Kazman 2001] R. Kazman, J. Asundi, M. Klein: "Quantifying the Cost and Benefits of Architectural Decisions," *Proceedings of the 23rd International Conference on Software Engineering (ICSE 23)*, Toronto, Canada, pp. 297-306, May 2001.

[Kitchenham 1996] B. Kitchenham, S. L. Pfleeger: "Software Quality: The Elusive Target," *IEEE Software*, Vol. 13, No. 1, pp. 12-21, January 1996.

[Kircher 2004] M. Kircher, P. Jain: "Pattern-Oriented Software Architecture: Patterns for Resource Management, Volume 3," John Wiley & Sons, 2004.

[Klein 1999]	M. Klein, R. Kazman: “Attribute-Based Architectural Styles,” Technical Report CMU/SEI-99-TR-022, Software Engineering Institute, Carnegie Mellon University, October 1999.
[Kruchten 1995]	P. Kruchten: “The 4+1 View Model of Architecture,” *IEEE Software*, Vol. 12, No. 6, pp. 42-50, November 1995.
[Kruchten 2000]	P. Kruchten: “The Rational Unified Process - An Introduction (Second Edition),” Addison-Wesley, 2000.
[Lanza 2003]	M. Lanza, S. Ducasse: “Polymetric Views - A Lightweight Visual Approach to Reverse Engineering,” *IEEE Transactions on Software Engineering,* Vol. 29, No. 9, pp. 782–795, September 2003.
[Laprie 1992]	J.C. Laprie (ed.): “Dependable Computing and Fault-Tolerant Systems, Volume 5, Dependability: Basic Concepts and Terminology,” Springer-Verlag, New York, 1992.
[Lassing 2002]	N. Lassing: “Architecture-Level Modifiability Analysis,” Phd Thesis, No. 2002-1, Vrije University of Amsterdam, the Netherlands, 2002.
[Leveson 1995]	N. G. Leveson: “Safeware: System Safety and Computers,” Addison-Wesley, 1995.
[Levine 2001]	D. L. Levine, C. D. Gill, D. C. Schmidt: “Object Lifetime Manager – A Complementary Pattern for Controlling Object Creation and Destruction,” In: Linda Rising (ed.), *Design Patterns in Communications Software*, Cambridge University Press, 2001.
[Liebenau 2001]	J. Liebenau: “Abstract Manager,” *Proceedings of the 8th Conference on Pattern Languages of Programs*, Allerton Park, Illinois, USA, 2001.
[Liu 1995]	J. W. S. Liu, R. Ha: “Efficient Methods of Validating Timing Constraints,” In: S.H. Son (ed.), *Advances in Real-Time Systems*, Prentice Hall, pp. 199-223, 1995.
[Loucopoulos 1995]	P. Loucopoulos, V. Karakostas: “System Requirements Engineering,” McGraw-Hill, New York, 1995.
[Luckham 1995]	D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, W. Mann: “Specification and Analysis of System Architecture Using Rapide,” *IEEE Transactions on Software Engineering*, Vol. 21, No. 4, pp. 336-355, April 1995.
[Maier 2000]	M. W. Maier, E. Rechtin: “The Art of Systems Architecting (Second Edition),” CRC Press, 2000.
[Magee 1996]	J. Magee, J. Kramer: “Dynamic Structures in Software Architectures,” Proceeeding of *ACM Sigsoft’96: Fourth Symposium on the Foundations of Software Engineering*, pp. 3-14, San Francisco, CA, October 1996.

[Malan 2002] R. Malan, D. Bredemeyer: "Software Architecture: Central Concerns, Key Decisions," Technical Report, Bredemeyer Consulting, 2002. [http://www.bredemeyer.com]

[McCall 1977] J. A. McCall, P. K. Richards, G. F. Walters: "Factors in Software Quality," RADC TR-77-369, Vol. I, US Rome Air Development Center Reports NTIS AD/A-049 014, 1977.

[McCall 1994] J. A. McCall: "Quality Factors," *Encyclopedia of Software Engineering*, Volume 2, pp. 958-969, Wiley 1994.

[McCrickard 1996] D. S. McCrickard, G. D. Abowd: "Assessing the Impact of Changes at the Architecture Level: A Case Study on Graphical Debuggers," *Proceedings of the 1996 International Conference on Software Maintenance (ICSM'96)*, IEEE Press, Los Alamitos, CA, pp. 59–67, 1996.

[MDA 2003] "Model Driven Architecture," 2003. [http://www.omg.org/mda/]

[Medvidovic 2000] N. Medvidovic, R. N. Taylor: "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Transactions on Software Engineering*, Vol. 26, No. 1, pp. 70-93, January 2000.

[Mercer 2002] "Zukunft von Elektronik und Software im Automobil," Mercer Management Consulting, 2002.

[Meszaros 1995] G. Meszaros: "Half-Object Plus Protocol," In: J.O. Coplien and D.C. Schmidt (eds.): *Pattern Languages of Program Design*, Addison-Wesley, 1995.

[Miga 1998] A. Miga: "Application of Use Case Maps to System Design with Tool Support," M. Eng. Thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 1998.

[Miga 1999] A. Miga: "Use Case Maps Navigator," May 1999. [URL: http://www.UseCaseMaps.org/UseCaseMaps/ucmnav/]

[Mindjet 2002] MindManager Business Edition 2002. [URL: http://www.mindjet.de]

[MITRE 1996] "MITRE's Architecture Quality Assessment," Technical Report, The MITRE Corporation, 1996.

[Molin 1997] P. Molin, L. Ohlsson: "The Points and Deviations Pattern Language of Fire Alarm Systems – Lazy State Pattern," In: R. C. Martin, D. Riehle, and F. Buschmann (eds.): *Pattern Languages of Program Design 3*, Addison-Wesley, 1997.

[Musa 1999] J. D. Musa: "Software Reliability Engineering," McGraw-Hill, 1999.

[Neumann 1995] P. G. Neumann: "Computer-Related Risks," Addison-Wesley, New York, 1995.

[Noble 2000] J. Noble, C. Weir: “Variable Allocation Pattern, in Small Memory Software: Patterns for Systems with Limited Memory,” Addison-Wesley, 2000.

[O’Brian 2002] L. O’Brian, C. Stoermer, C. Verhoef: “Software Architecture Reconstruction: Practice Needs and Current Approaches,” Technical Report CMU/SEI-2002-TR-024, Software Engineering Institute, Carnegie Mellon University, October 2002.

[Parnas 1971] D. L. Parnas: “Information Distribution Aspects of Design Methodology,” *Proceedings of the 1971 IFIP Congress*, pp. 339-334, 1971.

[Parnas 1972] D. L. Parnas: “On the Criteria To Be Used in Decomposing Systems into Modules,” *Communications of the ACM*, Vol. 15, No. 12, December 1972.

[Parnas 1974] D. L. Parnas: “On a ‘Buzzword’: Hierarchical Structure,” *Information Processing 74 - Proceedings of IFIP Congress 74*, J. L. Rosenfeld (ed.), August 1974.

[Perry 1992] D. E. Perry, A. L. Wolf: “Foundations for the Study of Software Architecture,” *ACM SIGSOFT Software Engineering Notes*, Vol. 17, No. 4, pp. 40-52, October 1992.

[Petriu 1997] D. Petriu, G. Somadder: “A Pattern Language For Improving the Capacity of Layered Client/Server Systems with Multi-Threaded Servers,” *Proceedings of the 2nd European Conference on Pattern Languages of Programs*, Kloster Irsee, Germany, 1997.

[Pohl 1996] K. Pohl: “Process-Centered Requirements Engineering,” RSP, Marketed by J. Wiley & Sons, Sussex, UK, 1996.

[Pohl 2005] K. Pohl, G. Böckle, F. van der Linden: “Software Product Line Engineering: Foundations, Principles, and Techniques,” Springer-Verlag, 2005.

[Pree 1995] W. Pree: “Design Patterns for Object-Oriented Software Development,” Addison-Wesley, 1995.

[Prieto-Diaz 1986] R. Prieto-Diaz, J. M. Neighbors: “Module Interconnection Languages,” *Journal of Systems and Software*, Vol. 6, No. 4, pp. 307-334, November 1986.

[Ramachandran 2002] J. Ramachandran: “Designing Security Architecture Solutions,” John Wiley, 2002.

[Randell 1995] B. Randell, J.C. Laprie, H. Kopetz, and B. Littlewood (Eds.): “Predictably Dependable Computing Systems,” ESPRIT Basic Research Series, Springer-Verlag, Berlin, 1995.

[Reynolds 1996] M. T. Reynolds: “Test and Evaluation of Complex Systems,” John Wiley and Sons, 1996.

[Rumbaugh 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: "Object-Oriented Modeling and Design," Prentice Hall, 1991.

[Sane 1996] A. Sane, R. Campbell: "Resource Exchanger," In: J. O. Coplien, N. Kerth, J. Vlissides (eds.): *Pattern Languages of Program Design 2*, Addison-Wesley, 1996.

[Schäfer 1996] H. Schäfer: "Surviving Under Time and Budget Pressure," *Proceedings of EuroSTAR 1996 Conference*, 1996.

[Schmidt 2000] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann: "Pattern-Oriented Software Architecture – Patterns for Concurrent and Distributed Objects," John Wiley & Sons, Inc., 2000.

[SEI 2004] "How Do You Define Software Architecture? (Online)," Software Engineering Institute, Carnegie Mellon University, 2004. URL: [http://www.sei.cmu.edu/architecture/definitions.html]

[Shaw 1989] M. Shaw: "Larger Scale Systems Require Higher-Level Abstractions," *Proceedings of the Fifth International Workshop on Software Specification and Design*, pp. 143-146, IEEE Computer Society Press, Los Alamitos, 1989.

[Shaw 1995] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, G. Zelesnik: "Abstractions for Software Architectures and Tools to Support Them," *IEEE Transactions on Software Engineering*, Vol. 21, No. 4, April 1995.

[Shaw 1996] M. Shaw, D. Garlan: "Software Architecture – Perspectives on an Emerging Discipline," Prentice Hall, 1996.

[Smith 1990] C. U. Smith: "Performance Engineering of Software Systems," Addidon-Wesley, 1990.

[Smith 1993] C. U. Smith, L. G. Williams: "Software Performance Engineering: A Case Study Including Performance Comparison with Design Alternatives," *IEEE Transactions on Software Engineering*, Vol. 19, No. 7, July 1993.

[Smith 1994] C. U. Smith: "Performance Engineering," *Encyclopedia of Software Engineering*, Volume 2, pp. 794-810, Wiley, 1994.

[Smith 2002] C. U. Smith, L. G. Williams: "Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software," Addison-Wesley, 2002.

[Sommerlad 1997] P. Sommerlad: "Manager Pattern," In: R.C. Martin, D. Riehle, F. Buschmann (eds.): *Pattern Languages of Program Design 3*, Addison-Wesley, 1997.

[Sommerlad 2002] P. Sommerlad: "Performance Patterns," *Proceedings of the 7th European Conference on Pattern Languages of Programs*, 2002

[Sommerville 1995] I. Sommerville: "Software Engineering (5th Edition)," Addison-Wesley, 1995.

[Spellman 2004] F. R. Spellman: "Safety Engineering: Principles and Practices (2nd Edition)," Government Institutes, 2004.

[Stal 2000] M. Stal: "Activator Pattern," http://www.stal.de/articles.html, 2000.

[Szyperski 1998] C. Szyperski: "Component Software: Beyond Object-Oriented Programming," Addison-Wesley, 1998.

[Tekinerdogan 2000] B. Tekinerdogan: "Classifying and Evaluating Architecture Design Methods," In: M. Aksit (ed.): *Software Architectures an Component Technology*, Kluwer Academic Publishers, 2000.

[Thiel 2000] S. Thiel, F. Peruzzi: "Starting a Product Line Approach for an Envisioned Market: Research and Experience in an Industrial Environment," In: [Donohoe 2000], pp. 495-512, 2000.

[Thiel 2001a] S. Thiel: "On the Definition of a Framework for an Architecting Process Supporting Product Family Development," In: F. van der Linden (ed.): *Software Product-Family Engineering*, Lecture Notes in Computer Science, LNCS 2290, Springer-Verlag, pp. 125-142, 2002.

[Thiel 2001b] S. Thiel, S. Ferber, A. Hein, T. Fischer, M. Schlick: "A Case Study in Applying a Product Line Approach for Car Periphery Supervision Systems," Technical Paper 2001-01-0025, In: *Proceedings of In-Vehicle Software 2001 (SP-1587)*, pp. 43-55, SAE 2001 World Congress, Detroit, Michigan, USA, March 5-8, 2001.

[Thiel 2001c] S. Thiel, A. Hein: "Analysis Concepts and Techniques Supporting Product Family Engineering," Deliverable BOSCH-WP1-T1.2-02, ITEA-ESAPS Project, March 2001.

[Thiel 2002a] S. Thiel, A. Hein: "Modeling and Using Product Line Variability in Automotive Systems," *IEEE Software*, Special Issue on Initiating Software Product Lines, Vol. 19, No. 4, pp. 66-72, July/August 2002.

[Thiel 2002b] S. Thiel, A. Hein: "Systematic Integration of Variability into Product Line Architectural Design," In: [Chastek 2002], pp. 130-153, 2002.

[Thiel 2003] S. Thiel, A. Hein, H. Engelhardt: "Tool Support for Scenario-Based Architecture Evaluation," Workshop *From Software Requirements to Architectures (STRAW),* 25th International Conference on Software Engineering (ICSE-25), Portland, Oregon, USA, May 2003.

[TOA 1995] "A Comparison of Object-Oriented Development Methodologies," Technical Report, The Object Agency, 1995. [URL: www.toa.com]

[Trowbridge 2003] D. Trowbridge, D. Mancini, D. Quick, G. Hohpe, J. Newkirk, D. Lavigne: "Enterprise Solution Patterns Using Microsoft .NET," Microsoft, http://www.microsoft.com/resources/practices, 2003.

[USAF 1988]	"Software Risk Abatement," Technical Report, United States Air Force, 1988.
[Van Scoy 1992]	R. L. Van Scoy: "Software Development Risk: Opportunity, Not Problem," Technical Report CMU/SEI-92-TR-30, Software Engineering Institute, September 1992.
[Vlissides 1996]	J. M. Vlissides, J. O. Coplien, N. L. Kert: "Pattern Languages of Program Design, Volume 2," Addison-Wesley, Reading, Ma., 1996.
[Völter 2002]	M. Völter, A. Schmid, E. Wolff: "Server Component Patterns – Component Infrastructures Illustrated with EJB," John Wiley & Sons, Inc., 2002.
[Wallingford 1997]	E. Wallingford: "Sponsor-Selector," In: R.C. Martin, D. Riehle, and F. Buschmann (eds.): *Pattern Languages of Program Design 3*, Addison-Wesley, 1997.
[Weber 1999]	J. Weber, T. Bertram, T. Kytölä, F. Peruzzi, S. Thiel: "Information Technology Restructures Car Electronics," Technical Paper 1999-01-0485, In: Proceedings of *Vehicle Navigation Systems and Advanced Controls (SP-1428)*, SAE International Congress and Exposition, Detroit, MI, March 1-4, 1999.
[Webster 1999]	"Merriam-Webster's Collegiate Dictionary," Merriam-Webster, 1999.
[Weidenhaupt 1998]	K. Weidenhaupt, K. Pohl, M. Jarke, P. Haumer: "Scenarios in System Development: Current Practice," *IEEE Software*, Vol. 15, No. 2, pp. 34-45, March/April 1998.
[Weiss 1999]	D. M. Weiss, C. T. R. Lai: "Software Product-Line Engineering – A Family-Based Software Development Process," Addison-Wesley, 1999.
[Wiegers 2003]	K. E. Wiegers: "Software Requirements, 2nd Edition," Microsoft Press, 2003.
[Williams 2002]	L. G. Williams, C. U. Smith: "PASA^{SM}: A Method for the Performance Assessment of Software Architectures," *Proceedings of 3rd International Workshop on Software and Performance*, Rome, Italy, 2002.
[Yourdon 1979]	E. Yourdan, L. L. Constantine: "Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design," Yourdon Press, Englewood Cliffs, NJ, 1979.
[Zachmann 1987]	J. A. Zachman: "A Framework for Information Systems Architecture," *IBM Systems Journal*, Vol. 26, No. 3, 1987.
[Zhao 1998]	J. Zhao: "On Assessing the Complexity of Software Architectures," *Proceedings of 3rd International Software Architecture Workshop*, pp.163-166, ACM SIGSOFT, ACM Press, November 1998.