

Paris Avgeriou · John Grundy
Jon G. Hall · Patricia Lago
Ivan Mistrík *Editors*

Relating Software Requirements and Architectures

Relating Software Requirements and Architectures

Paris Avgeriou • John Grundy • Jon G. Hall •
Patricia Lago • Ivan Mistrík
Editors

Relating Software Requirements and Architectures



Editors

Paris Avgeriou

Department of Mathematics and Computing
Science

University of Groningen

9747 AG Groningen

The Netherlands

paris@cs.rug.nl

Jon G. Hall

Open University

Milton Keynes MK7 6AA

United Kingdom

J.G.Hall@open.ac.uk

Ivan Mistrík

Werderstr. 45

69120 Heidelberg

Germany

i.j.mistrík@t-online.de

John Grundy

Swinburne University of Technology

Hawthorn, VIC 3122

Australia

jgrundy@swin.edu.au

Patricia Lago

Vrije Universiteit

Dept. Computer Science

De Boelelaan 1081 A

1081 HV Amsterdam

Netherlands

patricia@cs.vu.nl

ISBN 978-3-642-21000-6

e-ISBN 978-3-642-21001-3

DOI 10.1007/978-3-642-21001-3

Springer Heidelberg Dordrecht London New York

ACM Codes: D.2, K.6

Library of Congress Control Number: 2011935050

© Springer-Verlag Berlin Heidelberg 2011

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilm or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Foreword

Why have a book about the relation between requirements and architecture? Requirements provide the function for a system and the architecture provides the form and after all, to quote Louis Sullivan, “form follows function.” It turns out not to be so simple. When Louis Sullivan was talking about function, he was referring to the flow of people in a building or the properties of the spaces in the various floors. These are what we in the software engineering field would call quality attributes, not functionality. Understanding the relation between requirements and architecture is important because the requirements whether explicit or implicit do represent the function and the architecture does determine the form. If the two are disconnected, then there is a fundamental problem with the system being constructed.

The figure below gives some indication of why the problem of relating requirements and architecture is a difficult one (Fig. 1).

- There are a collection of stakeholders all of whom have their own agendas and interests. Reconciling the stakeholders is a difficult problem.
- The set of requirements are shown as a cloud because some requirements are explicitly considered in a requirements specification document and other requirements, equally or more important, remain implicit. Determining all of the requirements pertaining to a system is difficult.
- The double headed arrow between the requirements and the architecture reflects that fact that requirements are constrained by what is possible or what is easy. Not all requirements can be realized within the time and budget allotted for a project.
- The feedback from the architecture to the stakeholders means that stakeholders will be affected by what they see from various versions of the system being constructed and this will result in changes to the requirements. Accommodating those changes complicates any development process.
- The environment in which the system is being developed will change. Technology changes, the legal environment changes, the social environment changes, and the competitive environment changes. The project team must decide the extent to which the system will be able to accommodate changes in the environment.

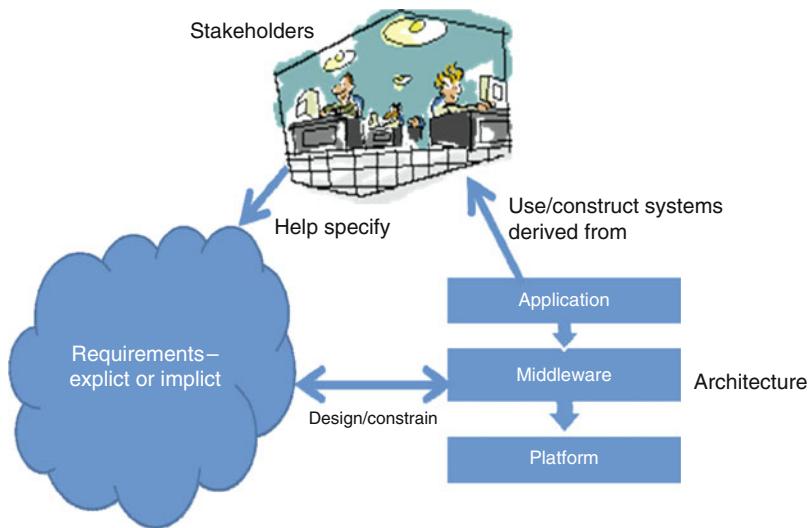


Fig. 1 A framing of the topics covered in this book

The authors in this book address the issues mentioned as well as some additional ones. But, as is characteristic of any rich problem area, they are not definitive. Some thoughts about areas that need further research are.

- Why not ask requirements engineers and architects what they see as the largest problems? This is not to say that the opinions expressed by the architects or requirements engineers would be more than an enumeration of symptoms for some deeper problem but the symptoms will provide a yardstick to measure research.
- What is the impact of scale? Architecture is most useful for large systems. To what extent are techniques for bridging the requirements/architecture gap dependent on scale? This leads to the question of how to validate any assertions. Assertions about software engineering methods or techniques are very difficult to validate because of the scale of the systems being constructed. One measure is whether a method or technique has been used by other than its author. This, at least, indicates that the method or technique is transferable and has some “face” validity.
- What aspects of the requirements/architecture gap are best spanned by tools and which by humans? Clearly tools are needed to manage the volume of requirements and to provide traceability and humans are needed to perform design work and elicit requirements from the stakeholders. But to what extent can tools support the human elements of the construction process that have to do with turning requirements into designs?
- Under what circumstances should the design/constrain arrow above point to the right and under what circumstances should it point to the left? Intuitively both directions have appeal but we ought to be able to make more precise statements

about when one does synthesis to generate a design and when one does analysis to determine the constraints imposed by the architectural decisions already made.

These are just some thoughts about the general problem. The chapters in this book provide much more detailed thoughts. The field is wide open, enjoy reading this volume and I encourage you to help contribute to bridging the requirements/architecture gap.

Len Bass
Software Engineering Institute
Pittsburgh, Pa, USA

Foreword

It must be the unique nature of software that necessitates the publication of a book such as this – in what other engineering discipline would it be necessary to argue for the intertwining of problem and solution, of requirements and architectures?

The descriptive nature of software may be one reason – software in its most basic form is a collection of descriptions, be they descriptions of computation or descriptions of the problems that the computation solves. After many years of focusing on software programs and their specifications, software engineering research began to untangle these descriptions, for example separating *what* a customer wants from *how* the software engineer delivers it. This resulted in at least two disciplines of description – requirements engineering and software architecture – each with their own representations, processes, and development tools. Recent years have seen researchers re-visit these two disciplines with a view to better understand the rich and complex relationships between them, and between the artifacts that they generate. Many of the contributions in this book address reflectively and practically these relationships, offering researchers and practicing software engineers tools to enrich and support software development in many ways: from the traditional way in which requirements – allegedly – precede design, to the pragmatic way in which existing solutions constrain what requirements can – cost-effectively – be met. And, of course, there is the messy world in between, where the customer needs change, where technology changes, and where knowledge about these evolves.

Amid these developments in software engineering, the interpretation of software has also widened – software is rarely regarded as simply a description that executes on a computer. Software permeates a wide variety of technical, socio-technical, and social systems, and while its role has never been more important, it no longer serves to solve the precise pre-defined problems that it once did. The problems that software solves often depend on what existing technology can offer, and this same technology can determine what problems can or should be solved. Indeed, existing technology may offer opportunities for solving problems that users never envisaged.

It is in this volatile and yet exciting context that this book seeks to make a particularly novel contribution. An understanding of the relationships between the problem world – populated by people and their needs – and the solution world – populated by systems and technology – is a pre-requisite for effective software engineering, and, more importantly, for delivering value to users.

The chapters in this book rightly focus on two sides of the equation – requirements and architectures – and the relationships between them. Requirements embody a range of problem world artifacts and considerations, from stakeholder goals to precise descriptions of the world that stakeholders seek to change. Similarly, architectures denote solutions – from the small executable program to the full structure and behavior of a software-intensive product line.

The ability to recognize, represent, analyze, and maintain the relationships between these worlds is, in many ways, the primary focus of this book. The editors have done an excellent job in assembling a range of contributions, rich in the breadth of their coverage of the research area, yet deep in addressing some of the fundamental research issues raised by ‘real’ world applications. And it is in this consideration of applications that their book differs from other research contributions in the area. The reason that relating requirements and architectures is an important research problem is that it is a problem that has its origins in the application world: requirements can rarely be expressed correctly or completely at the first attempt, and technical solutions play a large part in helping to articulate problems and to add value where value was hard to pre-determine. It is this ‘messy’ and changing real world that necessitates research that helps software engineers to deliver systems that satisfy, delight and even (pleasantly) surprise their customers. I am confident that, in this book, the editors have delivered a scholarly contribution that will evoke the same feelings of satisfaction, delight and surprise in its readers.

Lero (Ireland) & The Open University (UK)
April 2011

Bashar Nuseibeh

Preface

This book brings together representative views of recent research and practice in the area of relating software requirements and software architectures. We believe that all practicing requirements engineers and software architects, all researchers advancing our understanding and support for the relationship between software requirements and software architectures, and all students wishing to gain a deeper appreciation of underpinning theories, issues and practices within this domain will benefit from this book.

Introduction

Requirements Engineering and Software Architecture have existed for at least as long as Software Engineering. However, only over the past 15–20 years have they become defined as sub-disciplines in their own right. Practitioners know that eliciting and documenting good requirements is still a very challenging problem and that it forms a heady mix with the difficulties inherent in architecting modern, complex heterogeneous software systems. Although research in each area is still active, it is in their combination that understanding is most actively sought. Presenting the current state of the art is the purpose of this book.

Briefly, for requirements engineering to have taken place, a set of requirements will have been elicited from often multiple stakeholders, analyzed for incompleteness, inconsistency and incorrectness, structured so they can be effectively shared with developers. As one would expect, large, complicated systems with a wide range of stakeholders – such as large banking, telecommunications and air traffic control systems, distributed development teams – have emerging and evolving requirements which make satisfaction very challenging. However, even smaller systems that are highly novel also present challenges to requirements engineers e.g., in the rapidly changing consumer electronics domain. As requirements are typically conceptualized and expressed initially in natural language by multiple stakeholders, identifying a set of consistent and complete requirements in a (semi-) formal model

for use by developers remains very challenging. These formalized requirements then must be satisfied in any architectural solution and the imprecision in moving from informal to formal requirements complicates this.

Good software architecture is fundamental to achieving successful software systems. Initially this meant defining the high level system decomposition combined with technological solutions to be used to meet requirements previously identified and codified. More recently this has grown to encompass the representation and use of architectural knowledge throughout the software engineering lifecycle, effective reuse of architectural styles and patterns, the engineering of self-adapting and self-healing autonomic system architectures, and the evolution of complex architectures over time. Without good architectural practices, software becomes as unstable and unreliable as construction without good foundational architecture: Quality attributes are not properly managed and risks are not mitigated.

What is Relating Software Requirements to Architecture?

Software requirements and architecture are intrinsically linked. Architecture solutions need to realize both functional and non-functional requirements captured and documented. While changes to a set of requirements may impact on its realizing architecture, choices made for an architectural solution may impact on requirements e.g. in terms of revising functional or non-functional requirements that can not actually be met.

A range of challenges present themselves when working on complex and emerging software systems in relating the system's requirements and its architecture. An incomplete list includes: When a requirement changes, what is the impact on its architecture? If we change decisions made during architecting, how might these impact on the system's requirements? How do we trace requirements to architecture elements and vice versa? How do new technologies impact on the feasibility of requirements? How do new development processes impact on the relationship between requirements and architecture e.g. agile, out sourcing, crowd sourcing etc? How do we relate an item in the requirements to its realizing features in the architecture and vice-versa, especially for large systems? How do we share requirements and architectural knowledge, especially in highly distributed teams? How do we inform architecture and requirements with economic as well as functional and non-functional software characteristics? How do practitioners currently solve these problems on large, real-world projects? What emerging research directions might assist in managing the relationship between software requirements and architecture?

In this book a range of approaches are described to tackle one or more of these issues. We interpret both "requirements" and "architecture" very widely. We also interpret the relationship between requirements and architecture very widely, encompassing explicit relationships between requirements and architecture partitions and constraints, to implicit relationships between decisions made in one

tangentially impacting on the other. Some of the contributions in this book describe processes, methods and techniques for representing, eliciting or discovering these relationships. Some describe technique and tool support for the management of these relationships through the software lifecycle. Other contributions describe case studies of managing and using such relationships in practice. Still others identify state of the art approaches and propose novel solutions to currently difficult or even intractable problems.

Book Overview

We have divided this book into four parts, with a general editorial chapter providing a more detailed review of the domain of software engineering and the place of relating software requirements and architecture. We received a large number of submissions in response to our call for papers and invitations for this edited book from many leading research groups and well-known practitioners of leading collaborative software engineering techniques. After a rigorous review process 15 submissions were accepted for this publication. We begin by a review of the history and concept of software engineering itself including a brief review of the discipline's genesis, key fundamental challenges, and we define the main issues in relating these two areas of software engineering.

Part I contains three chapters addressing the issue of requirements change management in architectural design through traceability and reasoning. Part II contains five chapters presenting approaches, tools and techniques for bridging the gap between software requirements and architecture. Part III contains four chapters presenting industrial case studies and artefact management in software engineering. Part IV contains three chapters addressing various issues such as synthesizing architecture from requirements, relationship between software architecture and system requirements, and the role of middleware in architecting for non-functional requirements. We finish with a conclusions chapter identifying key contributions and outstanding areas for future research and improvement of practice. In the sections below we briefly outline the contributions in each part of this book.

Part 1 – Theoretical Underpinnings and Reviews

The three chapters in this section identify a range of themes around requirements engineering. Collectively they build a theoretic framework that will assist readers to understand both requirements, architecture and the diverse range of relationships between them. They address key themes in the domain including change management, ontological reasoning and tracing between requirements elements and architecture elements, at multiple levels of detail.

Chapter 3 proposes Change-oriented Requirements Engineering (CoRE), a method to anticipate change by separating requirements into layers that change at relatively different rates. From the most stable to the most volatile, the authors identify several layers including: patterns, functional constraints, and business policies and rules. CoRE is empirically evaluated by the authors by applying it to a large-scale software system and then studying the observed requirements change from development to maintenance. Results show that CoRE accurately anticipates the relative volatility of the requirements and can thus help manage both requirements evolution but also derivative architectural change.

Chapter 4 introduces a general-purpose ontology that the authors have developed to address the problem of co-evolving requirements and architecture descriptions of a software system. They demonstrate an implementation of semantic wiki that supports traceability between elements in a co-evolving requirements specifications and corresponding architecture design. They demonstrate their approach using a reuse scenario and a requirements change scenario.

Chapter 5 investigates homogeneous and heterogeneous requirements traceability networks. These networks are achieved by using event-based traceability and call graphs. Both of these traces are harvested during a software project. These traceability networks can be used in understanding some of the resulting architectural styles observed based on the real time state of a software project. The authors demonstrate the utility of such traceability networks to monitor initial system decisions and identify bottlenecks in an exemplar software project, a pinball machine simulator.

Part 2 – Tools and Techniques

The five chapters in this section identify a range of themes around tools and techniques. Good tool support is essential to managing complex requirements and architectures on large projects. These tools must also be underpinned by techniques that provide demonstrative added value to developers. Techniques used range from goal-directed inference, uncertainty management, problem frames, service compositions, to quality attribute refinement from business goals.

Chapter 7 presents a goal-oriented software architecting approach, where functional requirements (FRs) and non-functional requirements (NFRs) are treated as goals to be achieved. These goals are then refined and used to explore achievement alternatives. The chosen alternatives and the goal model are then used to derive a concrete architecture by applying an architectural style and architectural patterns chosen based on the NFRs. The approach has been applied in an empirical study based on the well-known 1992 London ambulance dispatch system.

Chapter 8 describes a commitment uncertainty approach in which linguistic and domain-specific indicators are used to prompt for the documentation of perceived uncertainty. The authors provide structure and advice on the development process so that engineers have a clear concept of progress that can be made to reduce

technical risk. A key contribution is in the evaluation of the technique in the engine control domain. They show that the technique is able to suggest valid design approaches and that supported flexibility does accommodate subsequent changes to requirements. The authors' aim is not to replace the process of creating a suitable architecture but to provide a framework that emphasizes constructive design actions.

Chapter 9 presents a method to systematically derive software architectures from problem descriptions. The problem descriptions are set up using Jackson's problem frame approach. They include a context diagram describing the overall problem situation and a set of problem diagrams that describe sub-problems of the overall software development problem. The different sub-problems are the instances of problem frames and these are patterns for simple software development problems. Beginning from these pattern-based problem definitions, the authors derive a software architecture in three steps: an initial architecture contains one component for each sub-problem; they then apply different architectural and design patterns and introduce coordinator and facade components; and finally the components of the intermediate architecture are re-arranged to form a layered architecture and interface and driver components added. All artefacts are expressed using UML diagrams with specifically defined UML profiles. Their tool supports checking of different semantic integrity conditions concerning the coherence of different diagrams. The authors illustrate the method by deriving an architecture for an automated teller machine.

Chapter 10 proposes a solution to the problem of having a clear link between actual applications – also referred to as service compositions – and requirements the applications are supposed to meet. Their technique also stipulates that captured requirements must properly state how an application can evolve and adapt at runtime. The solution proposed in this chapter is to extend classical goal models to provide an innovative means to represent both conventional (functional and non-conventional) requirements along with dynamic adaptation policies. To increase support to dynamism, the proposal distinguishes between crisp goals, of which satisfiability is boolean, and fuzzy goals, which can be satisfied at different degrees. Adaptation goals are used to render adaptation policies. The information provided in the goal model is then used to automatically devise the application's architecture (i.e., composition) and its adaptation capabilities. The goal model becomes a live, runtime entity whose evolution helps govern the actual adaptation of the application. The key elements of this approach are demonstrated by using a service-based news provider as an exemplar application.

Chapter 11 presents a set of canonical business goals for organizations that can be used to elicit domain-specific business goals from various stakeholders. These business goals, once elicited, are used to derive quality attribute requirements for a software system. The results are expressed in a common syntax that presents the goal, the stakeholders for whom the goal applies, and the "pedigree" of the goal. The authors present a body of knowledge about business goals and then discuss several different possible engagement methods to use knowledge to elicit business goals and their relation to software architectural requirements. They describe a new

methodology to support this approach and describe their experiences applying it with an Air Traffic Management unit of a major U.S aerospace firm.

Part 3 –Industrial Case Studies

The four chapters in this section present several industrial case studies of relating software requirements and architecture. These range from developing next-generation Consumer Electronics devices with embedded software controllers, IT security software, a travel booking system, and various large corporate systems.

Chapter 13 addresses the problems in designing consumer electronics (CE) products where architectural description is required from an early stage in development. The creation of this description is hampered by the lack of consensus on high-level architectural concepts for the CE domain and the rate at which novel features are added to products. This means that old descriptions cannot simply be reused. This chapter describes both the development of a reference architecture that addresses these problems and the process by which the requirements and architecture are refined together. The reference architecture is independent of specific functionality and is designed to be readily adopted. The architecture is informed by information mined from previous developments and organized to be reusable in different contexts. The integrity between the roles of requirements engineer and architect, mediated through the reference architecture, is described and illustrated with an example of integrating a new feature into a mobile phone.

Chapter 14 presents a view-based, model-driven approach for ensuring the compliance ICT security issues in a business process of a large European company. Compliance in service-architectures means complying with laws and regulations applying to distributed software systems. The research question of this chapter is to investigate whether the authors' model-driven, view-based approach is appropriate in the context of this domain. This example domain can easily be generalized to many other problems of requirements that are hard to specify formally, such as compliance requirements, in other business domains. To this end, the authors present lessons learned as well metrics for measuring the achieved degree of separation of concerns and reduced complexity via their approach.

Chapter 15 proposes an approach to artifact management in software engineering that uses an artifact matrix to structure the artifact space of a project along with stakeholder viewpoints and realization levels. This matrix structure provides a basis on top of which relationships between artifacts, such as consistency constraints, traceability links and model transformations, can be defined. The management of all project artifacts and their relationships supports collaboration across different roles in the development process, change management and agile development approaches. The authors' approach is configurable to facilitate adaptation to different development methods and processes. It provides a basis to develop and/or to integrate generic tools that can flexibly support different methods. In particular, it can be leveraged to improve the transition from requirements analysis to

architecture design. The development of a travel booking system is used as an exemplar application domain.

Chapter 16 illustrates a set of proven practices as well as a conceptual methods that help software engineers classify and prioritize requirements which then serve as drivers for architecture design. The author claims that all design activities follow the approach of piecemeal growth. Rather than try and react against this, they urge in supporting this explicitly in the supporting requirements and architecting processes. Similarly, a layered approach has been found to be necessary and most effective when eliciting, documenting and managing these requirements and architectures. A method supporting this evolutionary growth of requirements and architecture is presented along with experiences of applying this approach on several SIEMENS projects.

Part 4 – Emerging Issues

The three chapters in this section address some emerging issues in the domain of relating software requirements and architecture. These issues include approaches to synthesizing candidate architectures from formal requirements descriptions, explicit, bi-directional constraint of requirements and architectural elements, and middleware-based, economic-informed definition of requirements.

Chapter 18 studies the generation of candidate software architectures from requirements using genetic algorithms. Architectural styles and patterns are used as mutations to an initial architecture and several architectural heuristics as fitness tests. The input for the genetic algorithm is a rudimentary architecture representing a very basic functional decomposition of the system, obtained as a refinement from use cases. This is augmented with specific modifiability requirements in the form of allowable change scenarios. Using a fitness function tuned for desired weights of simplicity, efficiency and modifiability, the technique produces a set of candidate architectural styles and patterns that satisfy the requirements. The quality of the produced architectures has been studied empirically by comparing generated architectures with ones produced by undergraduate students.

In Chap. 19 the authors claim that the relationship of a system's requirements and its architectural design is not a simple one. Previous thought has been that the requirements drive the architecture and the architecture is designed in order to meet requirements. In contrast, their experience is that a much more dynamic relationship needs to be achieved between these key activities within the system design lifecycle. This would then allow the architecture to constrain the requirements to an achievable set of possibilities, frame the requirements by making their implications on architecture and design clearer, and inspire new requirements from the capabilities of the system's architecture. The authors describe this rich interrelationship; illustrate it with a case study drawn from their experience; and present some lessons learned that they believe will be valuable for other software architects.

Chapter 20 discusses the problem of evolving non-functional requirements, their stability implications and economic ramifications on the software architectures induced by middleware. The authors look at the role of middleware in architecting for non-functional requirements and their evolution trends. They advocate adjusting requirements elicitation and management techniques to elicit not just the current non-functional requirements, but also to assess the way in which they will develop over the lifetime of the architecture and their economic ramifications. The range of these possible future requirements may then inform the selection of distributed components technologies, and subsequently the selection of application server products. They describe an economics-driven approach, based on the real options theory, which can assist in informing the selection of middleware to induce software architectures in relation to the evolving non-functional requirements. They review its application through a case study.

Current Challenges and Future Directions

We conclude this book with a chapter drawing conclusions from the preceding 15 chapters. We note many approaches adopt a goal-oriented paradigm to bridging the gap between requirements and architecture. Similarly, many techniques and tools attempt to address the problem of traceability between requirements elements and architectural decisions. The advance of reference architectures, patterns, and successful requirements models in many domains has assisted the development of complex requirements and architectures.

We observe, as have some of the authors of chapters in this book, that the waterfall software process heritage of “requirements followed by architecture” has probably always been a degree of fallacy, and is probably more so with today’s complex and evolving heterogeneous software systems. Instead, viewing software requirements and software architecture as different viewpoints on the same problem may be a more useful future direction. Appropriate representation of architectural requirements and designs is still an area of emerging research and practice. This includes decisions, knowledge, abstractions, evolution and implications, not only technical but economic and social as well. To this end, some chapters and approaches touch on the alignment of business processes and economic drivers with technical requirements and architectural design decisions. While enterprise systems engineering has been an area of active practice and research for 30+ years, the relationship between business drivers and technical solutions is still ripe for further enhancement.

J. Grundy, I. Mistrík, J. Hall, P. Avgeriou, and P. Lago

Acknowledgements

The editors would like to sincerely thank the many authors who contributed their works to this collection. The international team of anonymous reviewers gave detailed feedback on early versions of chapters and helped us to improve both the presentation and accessibility of the work. Finally we would like to thank the Springer management and editorial teams for the opportunity to produce this unique collection of articles covering the wide range of issues in the domain of relating software requirements and architecture.

Contents

- 1 Introduction: Relating Requirements and Architectures** 1
J.G. Hall, J. Grundy, I. Mistrik, P. Lago, and P. Avgeriou

Part I Theoretical Underpinnings and Reviews

- 2 Theoretical Underpinnings and Reviews** 13
J. Grundy, P. Lago, P. Avgeriou, J. Hall, and I. Mistrik
- 3 Anticipating Change in Requirements Engineering** 17
Soo Ling Lim and Anthony Finkelstein
- 4 Traceability in the Co-evolution of Architectural Requirements and Design** 35
Antony Tang, Peng Liang, Viktor Clerc, and Hans van Vliet
- 5 Understanding Architectural Elements from Requirements Traceability Networks** 61
Inah Omoronyia, Guttorm Sindre, Stefan Biffl, and Tor Stålhane

Part II Tools and Techniques

- 6 Tools and Techniques** 87
P. Lago, P. Avgeriou, J. Grundy, J. Hall, and I. Mistrik
- 7 Goal-Oriented Software Architecting** 91
Lawrence Chung, Sam Supakkul, Nary Subramanian,
José Luis Garrido, Manuel Noguera, María V. Hurtado,
María Luisa Rodríguez, and Kawtar Benghazi

8 Product-Line Models to Address Requirements Uncertainty, Volatility and Risk	111
Zoë Stephenson, Katrina Attwood, and John McDermid	
9 Systematic Architectural Design Based on Problem Patterns	133
Christine Choppy, Denis Hatebur, and Maritta Heisel	
10 Adaptation Goals for Adaptive Service-Oriented Architectures	161
Luciano Baresi and Liliana Pasquale	
11 Business Goals and Architecture	183
Len Bass and Paul Clements	

Part III Experiences from Industrial Projects

12 Experiences from Industrial Projects	199
P. Avgeriou, J. Grundy, J. Hall, P. Lago, and I. Mistrik	
13 A Reference Architecture for Consumer Electronics Products and its Application in Requirements Engineering	203
Tim Trew, Goetz Botterweck, and Bashar Nuseibeh	
14 Using Model-Driven Views and Trace Links to Relate Requirements and Architecture: A Case Study	233
Huy Tran, Ta'id Holmes, Uwe Zdun, and Schahram Dustdar	
15 Managing Artifacts with a Viewpoint-Realization Level Matrix ...	257
Jochen M. Küster, Hagen Völzer, and Olaf Zimmermann	
16 Onions, Pyramids & Loops – From Requirements to Software Architecture	279
Michael Stal	

Part IV Emerging Issues in Relating Software Requirements and Architecture

17 Emerging Issues in Relating Software Requirements and Architecture	303
J. Grundy, P. Avgeriou, J. Hall, P. Lago, and I. Mistrik	
18 Synthesizing Architecture from Requirements: A Genetic Approach	307
Outi Räihä, Hadaytullah Kundi, Kai Koskimies, and Erkki Mäkinen	

Contents	xxiii
19 How Software Architecture can Frame, Constrain and Inspire System Requirements	333
Eoin Woods and Nick Rozanski	
20 Economics-Driven Architecting for Non Functional Requirements in the Presence of Middleware	353
Rami Bahsoon and Wolfgang Emmerich	
21 Conclusions	373
P. Avgeriou, P. Lago, J. Grundy, I. Mistrik, and J. Hall	
Editor Biographies	379
Index	383

Contributors

Katrina Attwood Department of Computer Science, University of York, York YO10 5DD, United Kingdom, katrina.attwood@cs.york.ac.uk

Paris Avgeriou Department of Mathematics and Computing Science, University of Groningen, Groningen 9747 AG, The Netherlands, paris@cs.rug.nl

Rami Bahsoon University of Birmingham, School of Computer Science, Birmingham B15 2TT, United Kingdom, r.bahsoon@cs.bham.ac.uk

Luciano Baresi Dipartimento Elettronica e Informazione, Politecnico di Milano, Milan 20133, Italy, baresi@elet.polimi.it

Len Bass Software Engineering Institute/Carnegie Mellon University, Pittsburgh, PA 15213, USA, lenbass@cmu.edu

Kawtar Benghazi Departamento de Lenguajes y Sistemas Informaticos, Universidad de Granada, Granada 18071, Spain, benghazi@ugr.es

Stefan Biffl Vienna University of Technology, Institute of Software Technology and Interactive Systems, Wien 1040, Austria, stefan.biffl@tuwien.ac.at

Goetz Botterweck Lero – the Irish Software Engineering Research Centre, University of Limerick, Limerick, Ireland, goetz.botterweck@lero.ie

Christine Choppy Université Paris 13, LIPN, CNRS UMR 7030, Villetaneuse 93430, France, Christine.Choppy@lipn.univ-paris13.fr

Lawrence Chung Department of Computer Science, University of Texas at Dallas, Dallas, TX 75083, USA, chung@utdallas.edu

Paul Clements Software Engineering Institute/Carnegie Mellon University, Pittsburgh, PA 15213, USA, clements@sei.cmu.edu

Viktor Clerc VU University Amsterdam, FEW, Computer Science, Amsterdam 1081 HV, The Netherlands, viktor@cs.vu.nl

Schahram Dustar Vienna University of Technology, Information Systems Institute, Wien 1040, Austria, dustdar@infosys.tuwien.ac.at

Wolfgang Emmerich Department of Computer Science, University College London, London WC1E 6BT, United Kingdom, W.Emmerich@cs.ucl.ac.uk

Anthony Finkelstein Department of Computer Science, University College London, London WC1E 6BT, United Kingdom, a.finkelstein@cs.ucl.ac.uk

Jose Luis Garrido Departamento de Lenguajes y, Universidad de Granada Sistemas Informaticos, Granada 18071, Spain, jgarrido@ugr.es

John Grundy Swinburne University of Technology, Hawthorn, VIC 3122, Australia, jgrundy@swin.edu.au

Hadyatullah Kundí Department of Software Systems, Tampere University of Technology, Tampere, Finland, hadaytullah@tut.fi

Jon G Hall Open University, Milton Keynes MK7 6AA, United Kingdom, J.G.Hall@open.ac.uk

Denis Hatebur Universität Duisburg-Essen, Software Engineering, Duisburg 47057, Germany, denis.hatebur@uni-duisburg-essen.de

Maritta Heisel Universität Duisburg-Essen, Software Engineering, Duisburg 47057, Germany, maritta.heisel@uni-duisburg-essen.de

Ta'id Holmes Vienna University of Technology, Information Systems Institute, Wien 1040, Austria, tholmes@infosys.tuwien.ac.at

Maria V. Hurtado Departamento de Lenguajes y Sistemas Informaticos, Universidad de Granada, Granada 18071, Spain, mhurtado@ugr.es

Kai Koskimies Department of Software Systems, Tampere University of Technology, Tampere, Finland, kai.koskimies@tut.fi

Jochen M. Küster IBM Research – Zurich, Rüschlikon 8803, Switzerland, jku@zurich.ibm.com

Patricia Lago VU University Amsterdam, FEW, Computer Science, Amsterdam 1081 HV, The Netherlands, patricia@cs.vu.nl

Peng Liang Department of Mathematics and Computing Science, University of Groningen, Groningen 9747 AG, The Netherlands, liangp@cs.rug.nl

Soo Ling Lim Department of Computer Science, University College London, London WC1E 6BT, United Kingdom, s.lim@cs.ucl.ac.uk

Erkki Mäkinen Department of Computer Science, University of Tampere, Tampere, Finland, em@cs.uta.fi

John McDermid Department of Computer Science, University of York, York YO10 5DD, United Kingdom, john.mcdermid@cs.york.ac.uk

Ivan Mistrík Independent Consultant, Heidelberg 69120, Germany, i.j.mistrik@t-online.de

Manuel Noguera Departamento de Lenguajes y Sistemas Informaticos, Universidad de Granada, Granada 18071, Spain, mnoguera@ugr.es

Bashar Nuseibeh Lero – the Irish Software Engineering Research Centre and The Open University (UK), University of Limerick, Limerick, Ireland, Bashar.Nuseibeh@lero.ie

Inah Omoronya Department of Computer and Information Science, Norwegian University of Science and Technology, Trondheim, Norway, inah.omoronya@idi.ntnu.no

Liliana Pasquale Dipartimento Elettronica e Informazione, Politecnico di Milano, Milan 20133, Italy, pasquale@elet.polimi.it

Outi Räihä Department of Software Systems, Tampere University of Technology, Tampere, Finland, outi.raihä@tut.fi

Maria Luisa Rodriguez Departamento de Lenguajes y Sistemas Informaticos, Universidad de Granada, Granada 18071, Spain, mlra@ugr.es

Nick Rozanski Software Architect for a UK Investment Bank, London, United Kingdom, nick@rozanski.org.uk

Guttorm Sindre Department of Computer and Information Science, Norwegian University of Science and Technology, Trondheim, Norway, guttorm.sindre@idi.ntnu.no

Michael Stal Siemens Corporate Technology, München, Germany and University of Groningen, Groningen, The Netherlands, michael.stal@gmail.com

Tor Stalhane Department of Computer and Information Science, Norwegian University of Science and Technology, Trondheim, Norway, tor.stalhane@idi.ntnu.no

Zoe Stephenson Department of Computer Science, University of York, York Y010 5DD, United Kingdom, zoe@cs.york.ac.uk

Nary Subramanian Department of Computer Science, University of Texas at Tyler, Tyler, USA, Nary_Subramanian@UTTyler.edu

Sam Supakkul Department of Computer Science, University of Texas at Dallas, Dallas, USA, ssupakkul@ieee.org

Antony Tang Swinburne University of Technology, Hawthorn, VIC, 3122, Australia, ATang@groupwise.swin.edu.au

Huy Tran Vienna University of Technology, Information Systems Institute, Wien 1040, Austria, htran@infosys.tuwien.ac.at

Tim Trew Independent Embedded Software Architect and Software Specialist, Horley, Surrey, RH6 7BX, United Kingdom, tiptrew@theiet.org, tim.trew@btinternet.com

Hans van Vliet VU University Amsterdam, FEW, Computer Science, Amsterdam 1081 HV, The Netherlands, hans@cs.vu.nl

Hagen Völzer IBM Research – Zurich, Rüschlikon 8803, Switzerland, hvo@zurich.ibm.com

Eoin Woods Artechra, Hemel Hempstead, Hertfordshire, United Kingdom, eoin.woods@artechra.com

Uwe Zdun Faculty of Computer Science, Vienna University of Technology, Wien 1090, Austria, zdun@infosys.tuwien.ac.at

Olaf Zimmermann IBM Research – Zurich, Rüschlikon 8803, Switzerland, olz@zurich.ibm.com

Chapter 1

Introduction: Relating Requirements and Architectures

J.G. Hall, J. Grundy, I. Mistrik, P. Lago, and P. Avgeriou

This book describes current understanding and use of the relationship between software requirements and software architectures.

Requirements and architectures have risen to be preeminent as the basis of modern software; their relationship one to the other is the basis of modern software development process.

Their case was not always clear cut, and neither was their preeminence guaranteed. Indeed, many tools and techniques have held the spotlight in software development at various times, and we will discuss some of them in this chapter. Now, however, it is clear that requirements and architectures are more than just software development fashion: used together they have helped software developers build the largest, most complex and flexible systems that exist today. They are trusted, in the right hands, as the basis of efficiency, effectiveness and value creation in most industries, in business and in the public sector. Many fast-moving areas, such as manufacturing, chemical and electronic engineering, and finance owe much of their success to their use of software!

It has been a virtuous spiral with the relationship between requirements and architectures driving progress.

In the light of this, this chapter reflects on two strands to which the relationship in question has made a defining contribution.

The first strand considers software developers' management of growing code complexity, a topic that lead to requirements and architectures, as well as motivating the need to understand their relationship. Specifically, we look at systems whose context of operation is well-known, bounded and *stable*. Examples are taken from the familiar class of embedded systems – for instance, smart phone operating systems, mechanical and chemical plant controllers, and cockpit and engine management systems. As will be clear, embedded systems are by no means unsophisticated, nor are the needs that such systems should fulfill simple: some, such as air traffic control, contain complex hardware and exist within complex socio-technical settings. Indeed, embedded system complexity has grown with the confidence of the software developer in the complexity of the problem that they can solve. For the purposes of this chapter, however, their

defining characteristic is that the context of the system is more or less stable so that change that necessitates redevelopment is the exception rather than the norm. The relationship between requirements and architectures for such systems is well-known – it is recounted in Sect. 1.1 – and the uses, efficiency and effectiveness of it is the topic of much fruitful research reported elsewhere in this book.

The second strand deals with systems for which a changing context is the norm, and that are enabled by the flexibility – some might say the agility – of the software development process that links requirements and architectures. It describes how developers work within a volatile context – the domain of application – and with the volatile requirements to be addressed within that domain.¹ Software systems typical of this class are those which underpin business processes in support of the enterprise (whence our examples), with problem volatility being simply the volatility of the business context and its needs. The relationship between requirements and architectures in the face of volatility is less well explored and we look at the source of some deficiencies in Sect. 1.2. New thinking, techniques and tools for improving the efficiency and effectiveness of the treatment of volatility is the topic of other leading-edge research reported elsewhere in this book.

Complexity and volatility are now known to be key developmental risk indicators for software development (see, for instance, [1]). And it is the successful treatment of complexity and volatility that places software requirements and software architectures as preeminent in linking problem and solution domains: as they are the key in the management of developmental risk, so most mainstream current practice is based on their relationship.

This chapter is organised as follows: after this introduction, we consider the rise of complexity in systems, providing an historic perspective that explains the current importance of the relationship between requirements and architectures. We consider next the rise of volatility in the light of the widely held critique of software project failure. The last section makes two observations, and from them motivates the continuing and future importance of the relationship between software requirements and architectures.

1.1 The Rise of Complexity and Volatility

The themes of this chapter are the developmental management of complexity and of volatility. Figure 1.1 (with Table 1.1) classifies various example systems according to their characteristics. In the figure, software complexity (measured by the proxy of lines of code²) is related to context volatility.

¹We note that context and requirements may change independently; as they define the *problem* that should be solved, we refer to problem volatility as indicative of these changes.

²The precise details of this relationship between complexity and lines of code have been subject to many debates over the years; we do not replay them here.

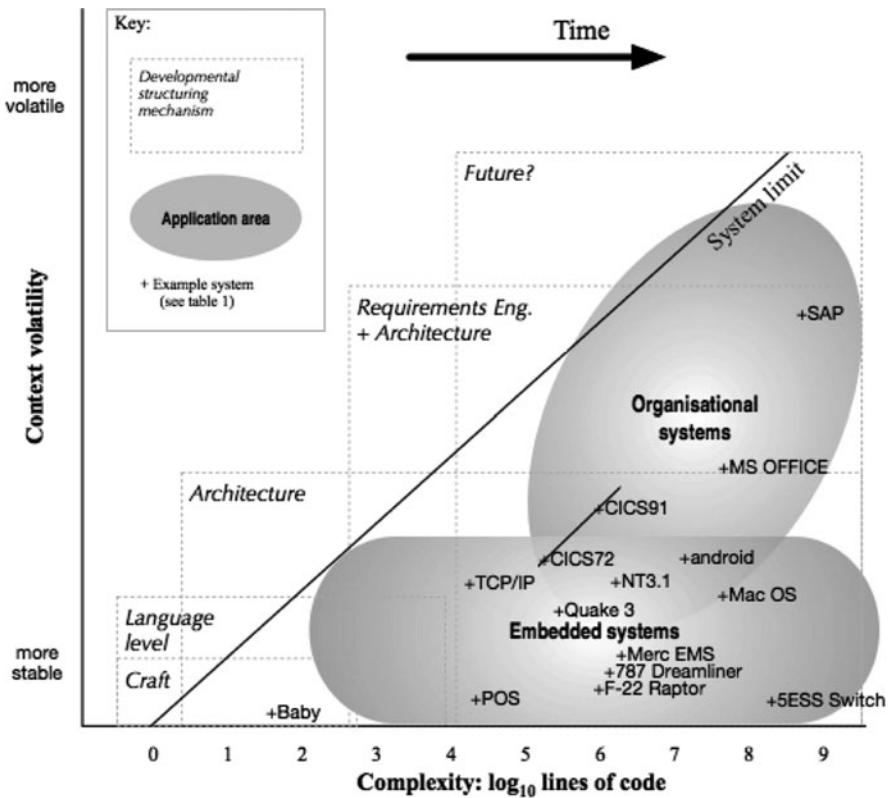


Fig. 1.1 Relating of requirements and architectures, the rise of complexity an volatility

The line labelled ‘System limit’ in the figure provides a bound based on the observed laws on the growth of computing power. The most relevant (and most widely known) is Moore’s observation that, essentially, computing power doubles every 2 years [2].

Also overlaid on the figure are various ‘Developmental structuring mechanisms.’ These are referred to in the text, with the figure serving to delimit the approximate scope of their applicability.

1.1.1 The Rise of Complexity

Manchester University housed the first stored program computer, the Small-Scale Experimental Machine (SSEM in Fig. 1.1) – aka Baby – which ran its first program on 21 June 1948. Baby’s first run was on a 17-line program³ that calculated:

³See [3] for an intriguing reconstruction of that first program, together with a transcript of the notes accompanying its design.

Table 1.1 Details of the systems referred to in Fig. 1

Fig. 1 Ref	Description	Source: (last access: 27 March 2011)
Baby	First program run on the Manchester small-scale experimental machine, aka Baby	[3]
TCP/IP POS	The transfer control protocol/internet protocol software stack Java point of sale open source system Quake III Arena software, a multiplayer computer game released December 2, 1999	www.sliedshare.net/ericvh/hare www.filagraru.com/apps/point_of_sale_java_source_code ftp://ftp.idsoftware.com/1dstuff/source/quake3-1.32b-source.zip
F-22 Raptor	□ 787	
Dreamliner	A comparison of avionics' and car management system.	spectrum.ieee.org/green-tech/advanced-cars/this-car-runs-on-code
□ Mercedes CMS		
5ESS NT 3.1	The 5ESS switch is a class 5 telephone electronic switching system sold by Alcatel-Lucent Windows NT 3.1 operating system	en.wikipedia.org/wiki/5ESS_switch en.wikipedia.org/wiki/Source_lines_of_code www.gnbatron.com/blog/2010/05/23/how-many-lines-of-code-does-it-take-to-create-the-android-os/
Android	Android: the open source handheld operating system	blogs.msdn.com/b/macmojo/archive/2006/11/03/it-s-all-in-the-numbers.aspx
MS OFFICE	Microsoft Office software suite, the core of many organisations information strategy	jshurwitz.wordpress.com/2007/04/30/how-does-sap-turn-250-million-lines-of-code-into-modular-services/
SAP	SAP enterprise resource planning software	en.wikipedia.org/wiki/Source_lines_of_code
MAC OS	www.sap.com Apple Macintosh operating systems, version 10.4	
CICS72-91	IBM CICS (Customer information control system). The line represents its beginnings in 1968 until 1991, when complexity had grown to 800,000 loc.	www-01.ibm.com/software/http/cics/35/

[...] the highest factor of an integer. We selected this problem to program because it used all seven instructions [in the SSEM instruction set]. It was of no mathematical significance, but everybody understood it.

Tom Kilburn [4]

This trivial problem was, arguably, the beginning of the modern information age. At that time, a simple software process was sufficient: the program above was simply crafted by inspection of the problem and debugged until it was correct.

As higher problem complexity was encountered, code complexity increased. One response to the need to manage higher code complexity was to provide richer and richer structures within the programming languages in which to implement it. High-level programming languages were developed to provide greater abstraction, often with concepts from the problem domain allowing more complex code to service more complex problems. There was a steady trend in sophistication leading to the ‘fifth-generation languages’ (5GLs) and ‘domain-specific languages.’

However, in 1992, Perry and Wolfe [5] brought ‘elements,’ ‘forms’ and ‘rationale’ together and so placed the new topic of ‘software architecture’ at the heart of code design. For, with them, Perry and Wolfe captured the repeated code and thought structures used to code and to justify code. Shaw and Garlan’s book [6] laid the groundwork for the development of software architecture as a discipline. From our vantage point and by their lasting popularity and utility, architectures (in their various guises) are the correct abstraction for software: they are programming language independent and serve to structure large systems⁴ [7].

With architectures came a growing confidence in the extent to which complexity could be handled and this led to a need for techniques and tools for managing complex (if stable) problem domains.

Even though the term software engineering had been defined in 1968, coincident with the rise of architectures was a growing realisation of the importance of an engineering approach to software. The traditional engineering disciplines had already been a rich source of ideas. The successful construction and manufacturing process model has, for instance, inspired the early software process model, today known as the Waterfall model, as illustrated in Fig. 1.2.

With architecture at heart of program design, software requirements began to be explored through software requirements *engineering* (for instance, [9]) which managed problem complexity through phases of elicitation, modelling, analysis, validation and verification. As required by the waterfall, software requirements engineering led from system requirements to software requirements specifications on which architectural analysis could be performed.

Things did not go smoothly, however; already in 1970, Royce [8] had suggested an amended model based on his extensive experience of the development of embedded software ‘[...] for spacecraft mission planning, commanding and post-flight analysis’ which had found it wanting. In particular, Royce saw the need to

⁴Their rise has also been coincident with a waning in the popularity of high-level programming languages; always wary of *cum hoc ergo propter hoc*.

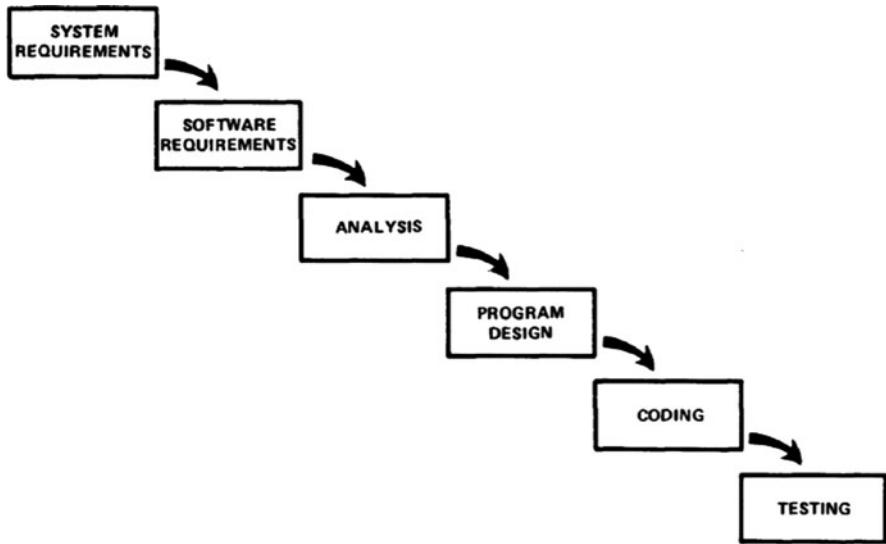


Fig. 1.2 The traditional ‘Waterfall Model’ for software: a successful development process from construction and manufacturing (Adapted from [8])

iterate between software requirements and program design (and between program design and testing): for software, the early stages of development could not always be completed before commencement of the later stages.

It was Royce’s enhanced model, illustrated in Fig. 1.3, that inspired a generation of software processes. And it is this model, and its many derivatives, that characterise the massively successful relationship between requirements and architectures, at least for embedded systems.

1.1.2 *The Rise of Volatility*

The waterfall model made one other assumption that is not true of software, which is that a stable problem exists from which to begin development. It is the lifting of this assumption that we explore in this section.

Of course, there were many early business uses of computers: a very early one started in 1951 when the Lyons⁵ Electronic Office (LEO, [10]) was used for order control and payroll administration. These early applications typically involved scheduling, reporting and number crunching and, although they provided many technical challenges to be solved that were complex for that time, what amounts to

⁵J. Lyons and Co. A UK food manufacturer in the mid 20th century.

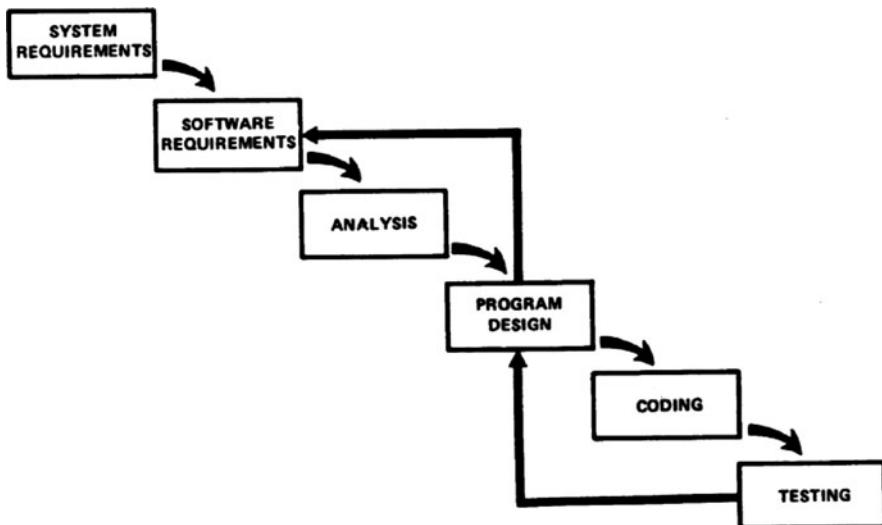


Fig. 1.3 Royce's updated waterfall for embedded system development (Adapted from [8])

first mover advantage [11] meant Lyons enjoyed a stable environment in which to explore their business use of computers.

The wish for more sophisticated business computing use meant developers dealing with the volatility of the business world. Any connection between the informal organisational world and the (essentially) formal world of the computer [12] must link *two* volatile targets: the world of enterprise and the world of technology. As the move from real-world to formal world is unavoidable [12], development must both initially cross and, as the needs exist in the real-world, subsequently recross the boundary as development continues.

Due to volatility, the resulting relationship between requirements and architectures is an order of magnitude more complex than for stable systems, and the treatment of it offered by traditional techniques, such as those derived from Royce's processes, are increasingly seen as inadequate. Indeed, software development in the face of such volatility is now recognised as one of the greatest challenges it faces.

Unfortunately, only the magnitude of the challenge is understood fully: a recently published study of 214 of large IT Projects⁶ [13] concluded that 23.8% were cancelled before delivery, with 42% (of those that ran to completion) overrunning in time and/or cost. According to the study, the total cost across the European Union of Information Systems failure was estimated to be €142 billion in 2004.

⁶Modally, the projects were in the €10–20 million range.

With the best minds on the problem of developing software in volatile contexts, exploration is in progress; indeed, a sizeable proportion of this book describes state of the art thinking for this problem.

1.2 The Future of Requirements and Architectures

It is, of course, possible to imagine that the changes that are wrought to software processes in response to volatile contexts will make redundant both requirements and architectures, and so knowledge of their relationship. It may, for instance, be that the need to treat volatility will lead back to high-level language development. There are already some small trends that lead away from requirements and architectures to self-structuring or multi-agent systems that, for instance, learn what is needed and then supply it autonomously. Who knows from whence the breakthrough will come!

There are, however, a number of observations of the systems we have considered in this chapter that lead us, with some certainty, to the conclusion that requirements and architectures will remain topics of importance for many, many years, and that their relationship will remain the basis of mainstream, especially organisational, software development.

Firstly, even if change is the norm, the scope of change in organisations is not the *whole* organisational system. For instance, there are general organising principles – the need for user to access a system, the need for the business processes conducted by the organisation to be represented, the need for organisational data to be captured and stored – which will remain for the foreseeable future and unless organisations change from their current form to be unrecognisable. As such, their representation in software will always be amenable to an architectural treatment much like as today.

Secondly, most organisations lack embedded software expertise sufficient to meet their own recognised software needs so that software development will, typically, always be outsourced to specialist development organisations. As the primary or driving need exists within the organisation, there is no alternative to accessing the stakeholder within the commissioning organisation for understanding their needs to develop the software intensive solution. As such, the need to capture and process requirements between organisations will always be necessary.

One happy conclusion is that the relationship between requirements and architectures is here to stay, and that the work of this book and that like it will grow in relevance!

Acknowledgments The first author wishes to thank Lucia Rapanotti for her careful reading and detailed critique of this chapter.

References

1. Ferreira S, Collofello J, Shunk D, Mackulak G (2009) Understanding the effects of requirements volatility in software engineering by using analytical modeling and software process simulation. *J Syst Softw* 82(10):1568–1577, SI: YAU
2. Schaller R (1997) Moore's law: past, present and future. *IEEE Spectrum* 34(6):52–59
3. Tootill G (1998) The original original program. Resurrection: the computer conservation society, no. 20. ISSN 0958–7403
4. Enticknap N (1998) Computing's golden jubilee. Resurrection: the computer conservation society, no. 20. ISSN 0958–7403
5. Perry DE, Wolf AL (1992) Foundations for the study of software architecture. *ACM SIGSOFT Software Eng Notes* 17(4):40–52
6. Shaw M, Garlan D (1996) Software architecture: perspectives on an emerging discipline. Prentice Hall, Englewood Cliffs
7. Bass L, Clements P, Kazman R (2003) Software architecture in practice, 2nd edn. Addison-Wesley Longman, Boston
8. Royce W (1970) Managing the development of large software systems. Proc IEEE WESCON 26:1–9
9. Cheng B, Atlee J (2007) Research directions in requirements engineering. In: Proceedings of FoSE 2007: future of software engineering. Washington, DC, pp 285–303
10. Bird P (1994) LEO: the first business computer. Hasler, Workingham
11. Grant RM (2003) Cases in contemporary strategy analysis. Blackwell, USA/UK/Australia/Germany. ISBN 1405111801
12. Turski WM (1986) And no philosophers' stone, either. *Information processing* 86:1077–1080
13. McManus J, Wood-Harper T (2008) A study in project failure. Tech rep, Chartered Institute of IT. Last accessed 27 March 2011

Part I

Theoretical Underpinnings and Reviews

Chapter 2

Theoretical Underpinnings and Reviews

J. Grundy, P. Lago, P. Avgeriou, J. Hall, and I. Mistrík

Requirements are fundamental to any engineered system. They capture the key stakeholder functional needs, constraints on the operation of the system, and often form a basis for contracting, testing and acceptance [1, 2]. Architecture captures the structuring of software solutions, incorporating not just functional properties of a system but design rationale, multi-layer abstractions and architectural knowledge [3, 4]. One can not exist without the other. Requirements need to be realized in a software system, described in essence by appropriate software architectures. Architecture must deliver on specified functional and non-functional requirements in order for the software system to be at all useful.

A major challenge to requirements engineers and architects has been keeping requirements and architecture consistent under change [5, 6]. Traditionally when requirements change, architecture (and implementation derived from architecture) must be updated. Likewise changes to the system architecture result in refining or constraining the current requirements set. However, more recently the bi-directional interaction between requirements and architecture changes; new software development processes such as agile, open source and outsourcing; new paradigms such as service-orientation and self-organizing systems; and a need to constrain requirements to what is feasible by e.g. rapidly emerging hardware technologies such as smartphones and embedded sensor devices, has complicated this change management process.

A further challenge to developers has been more effective traceability and knowledge management in software architecture [7–9]. It has become recognized that architectural knowledge management, including relationships to requirements, is very complicated and challenging, particularly in large organizations and in single-organisation multi-site, outsourcing and open source projects. Traceability between documentation and source code has been a research topic for many years. Tracing between requirements, architecture and code has become a key area to advance management of the relationship between requirements and architecture. Unfortunately recovering high value traceability links is not at all straightforward. Different stakeholders usually require different support and management for both traceability support and knowledge management.

A major research and practice trend has been to recognize that requirements and architecture co-evolve in most systems. To this end, not only traceability needs to be supported but recovery of architectures and requirements from legacy systems [10]. Separating architectural concerns and knowledge has been implicitly practiced for many years in these activities, needing to be supported by more explicit approaches [11].

The three chapters in this part of the book identify a range of fundamental themes around requirements engineering and software architecture. They help to build a theoretic framework that will assist readers to understand both software requirements engineering, software architecture and the diverse range of relationships between the two. They address key themes across the domain of this book including the need for appropriate change management processes, supporting ontological reasoning about the meaning of requirements and architectural elements, and the need to be able to trace between requirements elements and architecture elements, at varying levels of detail.

Chapter 3, co-authored by Soo Ling Lim and Anthony Finkelstein, proposes Change-oriented Requirements Engineering (CoRE). This is a new method to anticipate change by separating requirements into layers that change at relatively different rates. From the most stable to the most volatile, the authors identify several layers that need to be accommodated. Some of these layers include patterns, functional constraints, and business policies and rules. CoRE is empirically evaluated by applying it to a large-scale software system and then studying the observed requirements changes from development to maintenance. The results of this evaluation show that their approach accurately anticipates the relative volatility of the requirements. It can thus help developers to manage both requirements evolution but also derivative architectural changes.

Chapter 4, co-authored by Antony Tang, Peng Liang, Viktor Clerc and Hans van Vliet, introduces a general-purpose ontology to address the problem of co-evolving requirements and architecture descriptions of a software system. The authors developed this ontology to address the issue of knowledge management in complex software architecture domains, including supporting requirements capture and relating requirements elements and architectural abstractions. They demonstrate an implementation of a semantic wiki that supports traceability between elements in a co-evolving requirements specifications and a corresponding architecture design. They demonstrate their approach using a reuse scenario and a requirements change scenario.

Chapter 5, co-authored by Inah Omoronyia, Guttorm Sindre, Stefan Biffl and Tor Stålhane, investigates homogeneous and heterogeneous requirements traceability networks. The authors synthesize these networks from a combination of event-based traceability and call graphs. Both of these traces are harvested during the running of a software project using appropriate tool support. These traceability networks are used to help understand some of the architectural styles being observed during work on a software project. The authors demonstrate the utility of their traceability networks to monitor initial system decisions and identify bottlenecks in an example project.

References

1. Nuseibeh B, Easterbrook S (2000) Requirements engineering: a roadmap. In: Proceedings of the conference on the future of software engineering (ICSE). ACM, New York, pp 35–46
2. Berenbach B, Paulish DJ, Kazmeier J, Daniel P, Rudorfer A (2009) Software systems requirements engineering: in practice. McGraw-Hill Osborne Media, New York
3. Garlan D, Perry DE (1995) Introduction to the special issue on software architecture. *IEEE T Software Eng* 21(4):269–274
4. Kruchten P, Lago P, van Vliet H (2006) Building up and reasoning about architectural knowledge. In: Hofmeister C (ed) QoSA-quality of software architecture. Springer, Vasteras, pp 43–58
5. Harker SDP, Eason KD, Dobson JE (1993) The change and evolution of requirements as a challenge to the practice of software engineering. In: Proceedings of the IEEE international symposium on requirements engineering. San Diego, pp 266–272
6. Stark GE, Oman P, Skillicorn A, Ameele A (1999) An examination of the effects of requirements changes on software maintenance releases. *J Softw Maint-Res Pr* 11(5):293–309
7. Gotel OCZ, Finkelstein ACW (1994) An analysis of the requirements traceability problem. In: IEEE international symposium on requirements engineering (RE). Colorado Springs, pp 94–101
8. Lago P, Muccini H, van Vliet H (2009) A scoped approach to traceability management. *J Syst Softw* 82(1):168–182
9. Hayes JH, Dekhtyar A, Sundaram SK (2006) Advancing candidate link generation for requirements tracing: the study of methods. *IEEE Trans Software Eng* 32(1):4–19
10. Mendonça N, Kramer J (2001) An approach for recovering distributed system architectures. *Autom Softw Eng* 8(3–4):311–354
11. Jakobac V, Medvidovic N, Egyed A (2005) Separating architectural concerns to ease program understanding. *SIGSOFT Softw Eng Notes* 30(4):1–5

Chapter 3

Anticipating Change in Requirements Engineering

Soo Ling Lim and Anthony Finkelstein

Abstract Requirements change is inevitable in the development and maintenance of software systems. One way to reduce the adverse impact of change is by anticipating change during requirements elicitation, so that software architecture components that are affected by the change are loosely coupled with the rest of the system. This chapter proposes Change-oriented Requirements Engineering (CoRE), a method to anticipate change by separating requirements into layers that change at relatively different rates. From the most stable to the most volatile, the layers are: patterns, functional constraints, non-functional constraints, and business policies and rules. CoRE is empirically evaluated by applying it to a large-scale software system, and then studying the requirements change from development to maintenance. Results show that CoRE accurately anticipates the relative volatility of the requirements.

3.1 Introduction

Requirements change is inevitable in the development and maintenance of software systems. As the environment and stakeholders' needs continuously change, so must the system, in order to continue meeting its intended purpose. Studies show that requirements change accounts for a large part of the rework in development, in some cases up to 85% [4, 16, 34]. As such, it is one of the top causes of project failure [30], and often regarded as one of the most chronic problems in software development [14].

One way to reduce the impact of change is to anticipate change during requirements elicitation. Identifying volatile requirements from the start enables the design of the system such that architectural components that realise the requirements are loosely coupled with the rest of the system [29, 33]. Then, software changes to accommodate the requirements change are easier to implement, reducing the amount of rework.

Despite the centrality of change management in requirements engineering, the area of change management lacks research [33]. Existing requirements engineering methods regard change anticipation as a separate activity *after* documentation [29].

Without the notion of future changes, the documentation mixes stable and volatile requirements, as the existing method section will show. Existing change anticipation approaches are guidelines that rely on domain experts and experienced requirements engineers [29], who may be absent in some projects. Finally, existing literature is almost entirely qualitative: there is no empirical study on the accuracy of these guidelines in real projects.

To address these problems, this chapter proposes Change-oriented Requirements Engineering (CoRE), an expert independent method to anticipate requirements change. CoRE separates requirements into layers that change at relatively different rates *during* requirements documentation. This informs architecture to separate components that realise volatile requirements from components that realise stable requirements. By doing so, software design and implementation prepares for change, thus minimising the disruptive effect of changing requirements to the architecture.

CoRE is empirically evaluated on its accuracy in anticipating requirements change, by first applying it to the access control system project at University College London, and then studying the number of requirements changes in each layer and the rate of change over a period of 3.5 years, from development to maintenance. This study is one of the first empirical studies of requirements change over a system's lifecycle. The results show that CoRE accurately anticipates the relative volatility of the requirements.

The rest of the chapter is organised as follows. Section 3.2 reviews existing methods in requirements elicitation and change anticipation. Section 3.3 introduces the idea behind CoRE. Section 3.4 describes CoRE and Sect. 3.5 evaluates it on a real software project. Section 3.6 discusses the limitations of the study before concluding.

3.2 Existing Methods

3.2.1 Requirements Elicitation

In requirements elicitation, model-based techniques, such as *use case* and *goal modelling*, use a specific model to structure their requirements, which often mixes stable and volatile requirements.

Use case is one of the common practices for capturing the required behaviour of a system [7, 13]. It models requirements as a sequence of interactions between the system and the stakeholders or other systems, in relation to a particular goal. As such, a use case can contain both stable and volatile requirements. An example use case for an access control system is illustrated in Table 3.1. In the use case, “displaying staff member details” in Step 1 is more stable than “sending photos to the staff system” in Step 4, because the verification and retrieval of person information is central to all access control systems, but providing photos to another system is specific to that particular access control system.

Table 3.1 Use case: issue cards to staff

Step	Action description
1.	The card issuer validates the staff member's identity and enters the identity into the system.
2.	The system displays the staff member's details.
3.	The card issuer captures the staff member's photo.
4.	The system generates the access card and sends the photo to the staff system.

Goal modelling (e.g., KAOS [32] and GBRAM [1]) captures the intent of the system as goals, which are incrementally refined into a goal-subgoal structure. High-level goals are, in general, more stable than lower-level ones [33]. Nevertheless, goals at the same level can have different volatility. For example, the goal “to maintain authorised access” can be refined into two subgoals: “to verify cardholder access rights” and “to match cardholder appearance with digital photo.” The second subgoal is more volatile than the first as it is a capability required only in some access control systems.

3.2.2 Requirements Documentation

When it comes to documenting requirements, most projects follow standard requirements templates. For example, the *Volere Requirement Specification Template* by Robertson and Robertson [24] organises requirements into functional and non-functional requirements, design constraints, and project constraints, drivers, and issues. The example access control system has the functional requirement “the access control system shall update person records on an hourly basis.” Within this requirement, recording cardholder information is more stable than the frequency of updates, which can be changed from an hour to 5 min when the organisation requires its data to be more up-to-date.

A standard requirements template is the *IEEE Recommended Practice for Software Requirements Specification* [11]. The template provides various options (e.g., system mode, user class, object, feature) to organise requirements for different types of systems. For example, the system mode option is for systems that behave differently depending on mode of operation (e.g., training, normal, and emergency), and the user class option is for systems that provide different functions to different user classes. Nevertheless, none of these options organise requirements by their volatility, which is useful for systems with volatile requirements, such as software systems in the business domain [14].

3.2.3 Requirements Change Management

In requirements change management, one of the earliest approaches by Harker et al. [10] classifies requirements into *enduring requirements* and *volatile requirements*.

Volatile requirements include *mutable requirements*, *emergent requirements*, *consequential requirements*, *adaptive requirements*, and *migration requirements*. Harker et al.'s classification is adopted in later work by Sommerville, Kotonya, and Sawyer [15, 28, 29]. Although the classification clearly separates stable requirements from volatile ones, the relative volatility among the types of volatile requirements is unknown. For example, *mutable requirements* change following the environment in which the system is operating and *emergent requirements* emerge as the system is designed and implemented. Without further guidelines, it is unclear whether a *mutable requirement* is more volatile than an *emergent requirement*.

Later work provides guidelines to anticipate requirements change. Guidelines from van Lamsweerde include: (1) stable features can be found in any contraction, extension, and variation of the system; (2) assumptions and technology constraints are more volatile than high-level goals; (3) non-functional constraints are more volatile than functional requirements; and (4) requirements that come from decisions among multiple options are more volatile [33]. Guidelines from Sommerville and Sawyer [29] are: (1) identify requirements that set out desirable or essential properties of the system because many different parts of the system may be affected by the change; and (2) maintain a list of the most volatile requirements, and if possible, predict likely changes to these requirements. The caveat with these guidelines is that they require experienced requirements engineers or domain experts to identify requirements that are likely to be volatile, and still, errors can occur [29].

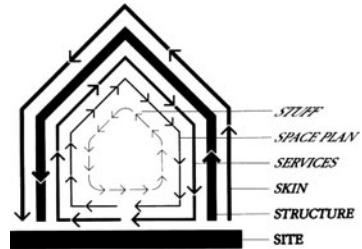
To summarise, existing requirements elicitation methods lack the ability to anticipate change. In addition, existing requirements templates do not separate requirements by their volatility, and existing change management approaches are expert dependent. In contrast, in the CoRE method proposed in this chapter, requirements anticipation is part of the requirements modelling process and independent of the person doing the analysis.

3.3 The Shearing Layers

CoRE adopts the concept of shearing layers from building architecture. This concept was created by British architect Frank Duffy who refers to buildings as composed of several layers of change [5]. The layers, from the most stable to most volatile, are site, structure, skin, services, space plan, and “stuff” or furniture (Fig. 3.1). For example, services (the wiring, plumbing, and heating) evolve faster than skin (the exterior surface), which evolves faster than structure (the foundation). The concept was elaborated by Brand [5], who observed that buildings that are more adaptable to change allow the “slippage” of layers, such that faster layers are not obstructed by slower ones. The concept is simple: designers avoid building furniture into the walls because they expect tenants to move and change furniture frequently. They also avoid solving a 5-min problem with a 50-year solution, and vice versa.

The shearing layer concept is based on the work of ecologists [22] and systems theorists [26] that some processes in nature operate in different timescales and as a

Fig. 3.1 The shearing layers of architecture [5]



result there is little or no exchange of energy or mass or information between them. The concept has already been adopted in various areas in software engineering. In software architecture, Foote and Yoder [9], and Mens and Galal [20] factored artefacts that change at similar rates together. In human computer interaction, Papantoniou et al. [23] proposed using the shearing layers to support evolving design. In information systems design, Simmonds and Ing [27] proposed using rate of change as the primary criteria for the separation of concerns.

Similar to the elements of a building, some requirements are more likely to change; others are more likely to remain the same over time. The idea behind CoRE is to separate requirements into shearing layers, with a clear demarcation between parts that should change at different rates.

3.4 Change-oriented Requirements Engineering (CoRE)

3.4.1 *The Shearing Layers of Requirements*

CoRE separates requirements into four layers of different volatility and cause of change. From the most stable to the most volatile, the layers are: patterns, functional constraints, non-functional constraints, and business policies and rules (Fig. 3.2). Knowledge about patterns and functional constraints can help design and implement the system such that non-functional constraints, business policies and rules can be changed without affecting the rest of the system.

3.4.1.1 Patterns

A pattern is the largest combined essential functionality in any variation of a software component that achieves the same goal. As such, they remain unchanged over time unless the goal is no longer needed. For example, the goal of an inventory system is to maintain a stock of items for sale. It can be achieved by the Inventory

Fig. 3.2 The shearing layers of requirements. The layers with more arrows are more volatile

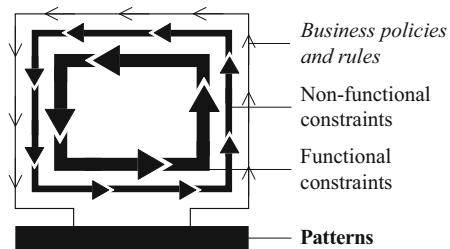


Fig. 3.3 Example patterns:

- (a) Inventory pattern
- (b) Person pattern [2]

a	b
Inventory <ul style="list-style-type: none"> Add inventory entry Remove inventory entry Get inventory entry Find inventory entry Get product types Make reservation Cancel reservation Get reservations Find reservation 	Person <ul style="list-style-type: none"> Get identifier Get person name Get addresses Get gender Get date of birth (optional) Get other names (optional) Get ethnicity (optional) Get body metrics (optional)

pattern illustrated in Fig. 3.3 (a) with functionalities such as making reservations, adding and finding products. These functionalities have existed long before software systems and are likely to remain unchanged. Different patterns can be found in different domains, e.g., patterns in the medical domain revolve around patients, doctors, patient records [28] and patterns in the business domain revolve around products, customers, inventory [2]. In the business domain, Arlow and Neustadt [2] developed a set of patterns which they named enterprise archetypes as the patterns are universal and pervasive in enterprises¹. Their catalogue of patterns consists of various business related pattern, including the ones in Fig. 3.3.

3.4.1.2 Functional Constraints

Patterns allow freedom for different instantiations of software components achieving the same goals. In contrast, functional constraints are specific requirements on the behaviour of the system that limit the acceptable instantiations. These constraints are needed to support the stakeholders in their tasks, hence remain unchanged unless the stakeholders change their way of working. For example, an access control system's main goal is to provide access control. The pattern assigned to this goal is the **PartyAuthentication** archetype that represents an agreed and trusted

¹From this point on, the word “archetype” is used when referring specifically to the patterns by Arlow and Neustadt [2].

way to confirm that a party is who they say they are [2]. A functional constraint on the achievement of this goal is that the system must display the digital photo of the cardholder when the card is scanned, in order to allow security guards to do visual checks.

3.4.1.3 Non-Functional Constraints

A non-functional constraint is a restriction on the quality characteristics of the software component, such as its usability, and reliability [6]. For example, the ISO/IEC Software Product Quality standard [12] identifies non-functional constraints as a set of characteristics (e.g., reliability) with sub-characteristics (e.g., maturity, fault tolerance) and their measurable criteria (e.g., mean time between failures). Changes in non-functional constraints are independent of the functionality of the system and occur when the component can no longer meet increasing quality expectation. For example, in an access control system, a person's information has to be up-to-date within an hour of modification. The constraint remains unchanged until the system can no longer support the increasing student load, and a faster service is needed.

3.4.1.4 Business Policies and Rules

A business policy is an instruction that provides broad governance or guidance to the enterprise [3, 21]. A business rule is a specific implementation of the business policies [3, 21]. Policies and rules are an essential source of requirements specific to the enterprise the system operates in [3, 25]. They are the most volatile [3], as they are related to how the enterprise decides to react to changes in the environment [21]. For example, a university deploying the access control system may have the business policy: *access rights for students should correspond to their course duration*. The business rule based on the policy is: *a student's access rights should expire 6 months after their expected course end date*. For better security, the expiration date can be shortened from 6 months to 3 months after the students' course end dates.

3.4.2 CoRE Method

CoRE is based on goal modelling methods [8, 35]. To separate requirements into the shearing layers, CoRE applies five steps to the goals elicited from stakeholders (Fig. 3.4). The access control system example for a university is used as a running example to demonstrate the method.

Step 1: Assign patterns to goals. CoRE starts by checking if there is a pattern for each goal. A pattern can be assigned to a goal if and only if the operation(s) in the pattern is capable of achieving the goal. There are two ways for this to happen. First,

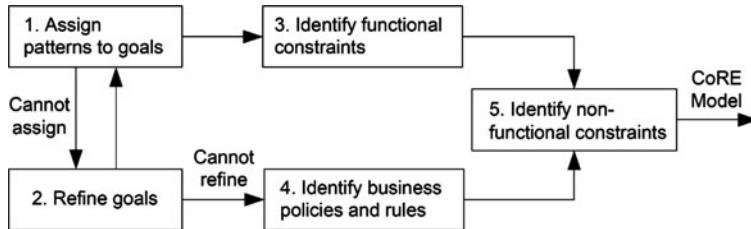


Fig. 3.4 The five steps in CoRE

the goal is a direct match to a functionality in the pattern. For example, the goal of searching for a person by name can be directly mapped to the functionality *to find a person by ID or name* in the `PartyManager` archetype [2]. Second, the goal can be refined into subgoals that form a subset of the operations in the pattern. For example, the goal *to manage people information centrally* can be refined into subgoals such as *to add or delete a person*, and *to find a person by ID or name*. These subgoals are a subset of the operations in the `PartyManager` archetype. If no patterns can be assigned to the goal, proceed to Step 2 with the goal. Otherwise, proceed to Step 3.

Step 2: Refine goals. This step refines high-level goals into subgoals and repeats Step 1 for each subgoal. To refine a goal, the KAOIS goal refinement strategy [8] is used where a goal is refined if achieving a subgoal and possibly other subgoals is among the alternative ways of achieving the goal. For a complete refinement, the subgoals must meet two conditions: (1) they must be distinct and disjoint; and (2) together they must reach the target condition in the parent goal. For example, the goal *to control access to university buildings and resources* is refined into three subgoals: *to maintain up-to-date and accurate person information*, *assign access rights to staff, students, and visitors*, and *verify the identity of a person requesting access*. If these three subgoals are met, then their parent goal is met.

As the refinement aims towards mapping the subgoals to archetypes, the patterns are used to guide the refinement. For example, the leaf goal² *to maintain up-to-date and accurate person information* is partly met by the `PartyManager` archetype that manages a collection of people. Hence, the leaf goal is refined into two subgoals: *to manage people information centrally*, and *to automate entries and updates of person information*. A goal cannot be refined if there are no patterns for its subgoals even if it is refined. For example, the goal *to assign access rights to staff, students, and visitors* has no matching patterns as access rights are business specific. In that case, proceed to Step 4.

Step 3: Identify functional constraints. For each pattern that is assigned to a goal, this step identifies functional constraints on the achievement of the goal. This involves asking users of the system about the tasks they depend on the system

²A leaf goal is a goal without subgoals.

to carry out, also known as task dependency in i* [35]. These tasks should be significant enough to warrant attention. For example, one of the security guard's task is to compare the cardholders' appearance with their digital photos as they scan their cards. This feature constrains acceptable implementations of the **PartyAuthentication** archetype to those that enable visual checks.

Step 4: Identify business policies and rules. The goals that cannot be further refined are assigned to business policies and rules. This involves searching for policies and rules in the organisation that support the achievement of the goal [21]. For example, the goal *to assign access rights to staff, students, and visitors* is supported by UCL access policies for these user categories. These policies form the basis for access rules that specify the buildings and access times for each of these user categories and their subcategories. For example, undergraduates and postgraduates have different access rights to university resources.

Step 5: Identify non-functional constraints. The final step involves identifying non-functional constraints for all the goals. If a goal is annotated with a non-functional constraint, all its subgoals are also subjected to the same constraint. As such, to avoid annotating a goal and its subgoal with the same constraint, higher-level goals are considered first. For example, the access control system takes people data from other UCL systems, such as the student system and human resource system. As such, for the goal *to maintain up-to-date and accurate person information*, these systems impose data compatibility constraints on the access control system.

The output of the CoRE method is a list of requirements that are separated into the four shearing layers. A visual representation of its output is illustrated in Fig. 3.5. This representation is adopted from the KAOS [8] and i* methods [35]. For example, the goal refinement link means that the three subgoals should together achieve the parent goal, and the means-end link means that the element (functional constraint, non-functional constraint, or pattern) is a means to achieve the goal.

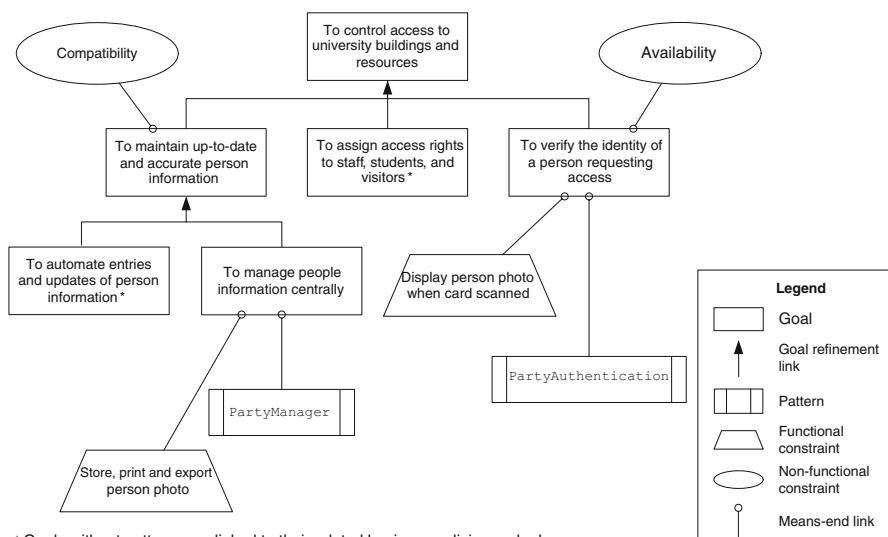


Fig. 3.5 Partial CoRE output for the university access control system

3.5 Evaluation

CoRE's goal is to separate requirements into layers that change at different rates. The evaluation asks if CoRE can be used to separate requirements into the shearing layers, and if it accurately anticipates the volatility of each shearing layer. The access control system project in University College London is used as a case study to evaluate CoRE. First, CoRE is used to model the initial requirements for the project. Then, the requirements change in the system is recorded over a period of 3.5 years, from the development of the system to the current date after the system is deployed. The result is used to find out the volatility for each layer and when changes in each layer occur in the system lifecycle.

3.5.1 *The RALIC Project*

RALIC (**R**eplacement **A**ccess, **L**ibrary and **I**D **C**ard) was the access control system project at University College London (UCL). RALIC was initiated to replace the existing access control systems at UCL, and consolidate the new system with identification, library access and borrowing. RALIC was a combination of development and customisation of an off-the-shelf system. The objectives of RALIC included replacing existing access card readers, printing reliable access cards, managing cardholder information, providing access control, and automating the provision and suspension of access and library borrowing rights.

RALIC was selected as the case study to evaluate CoRE for the following reasons. First, the stakeholders and project documentation were accessible as the system was developed, deployed, and maintained at UCL. Second, RALIC was a well-documented project: the initial requirements and subsequent changes were well-documented. Third, the system development spanned over 2 years and the system has been deployed for more than 2 years, providing sufficient time to study change during development as well as maintenance. Finally, RALIC was a large-scale project with many stakeholders [19] in a changing environment, providing sufficient data in terms of requirements and their changes to validate CoRE.

3.5.2 *Applying CoRE to RALIC*

The CoRE method was used to separate the requirements for the RALIC project into the shearing layers. The initial requirements model was built using the requirements documentation signed off by the client as the baseline. The initial requirements model for RALIC consists of 26 elements from the CoRE layers: 3 patterns, 5 functional constraints, 4 non-functional constraints, 5 business policies, and 9 business rules.

To study requirements change, modifications to the requirements documentation after the baseline are considered as a change. There are three types of change:

- *Addition*: a requirement is introduced after sign-off.
- *Deletion*: an existing requirement is removed.
- *Modification*: an existing requirement is modified due to changes in stakeholder needs. Corrections, clarifications, and improvements to the documentation are not considered as changes.

As RALIC was an extremely well-documented project, requirements change can be studied retrospectively. During the project, the team members met fortnightly to update their progress and make decisions. All discussions were documented in detail in the meeting minutes as illustrated in Fig. 3.6, by an external project support to increase the objectiveness of the documentation. As such, studying the minutes provided an in-depth understanding of the project, its progress, requirements changes, and their rationale.

RALIC used a semi-formal change management process. During development, minor changes were directly reflected in the documentation. Changes requiring further discussions were raised in team meetings and reported to the project board. Major changes required board approval. Meeting discussion about changes and their outcome (accepted, postponed, or rejected) were documented in the minutes. During maintenance, team meetings ceased. The maintenance team recorded change requests in a workplan and as action items in a change management tool.

The following procedure was used to record changes. All project documentation related to requirements, such as specifications, workplans, team and board meeting minutes, were studied, as changes may be dispersed in different locations. Care was taken not to consider the same changes more than once. Repeated documentation of

<p><u>Update 30/11/05</u></p> <p>Round the table discussions (@ team mtg 30/11/05) resulted in the following further agreements (revised demo card layouts appended);</p> <ol style="list-style-type: none"> 1. Portico Hologram (or similar) to be located on the front of the card (bottom LH corner). 2. "SN" to replace "SRN" - & "SN" and "UPI" to be stacked above "Expires". 3. Photos to be the same size on all cards. <p>Revised demo card design to be submitted for Board approval (Board to clarify/confirm the need to print "Departmental names").</p>	<p><u>Update 13/12/05</u></p> <p>The card design has been completed to the satisfaction of the project team and was approved by the Board at Board meeting dated 9/12/05 – fine tuning & decision on hologram remain o/s.</p> <p>(Note: The Board further clarified the requirement for "Departmental names" to be printed on the cards – the preferred format being, "Department; Departmental name" - & to make use of the "wrap text" facility to enable printing on more than one line, as appropriate).</p>
<p><u>Update 11/01/06</u></p> <p>Discussed "Departmental Names" - agreed to</p> <ol style="list-style-type: none"> i) review the purpose of printing "Departmental name" on the card (MD to discuss with NK from a security perspective). ii) locate the definitive "departmental name" list or establish/agree on a suitable list . <p>MRP to discuss ii) with N. A. [REDACTED] & other stakeholders.</p>	<p><u>Update 26/01/06</u></p> <p>MRP & NK would prefer "Departmental name" to be printed on card.</p> <p>Round the table discussions concluded that i) checks need to be made to ensure source data is reliable & consistent ii) "Departmental name" information printed on card will be derived from HR (i.e. Resource Link). MRP to discuss ii) with N. A. [REDACTED] & other stakeholders.</p> <p>The card design "fine tuning & decision on hologram" remain o/s – this is delayed until demo cards can be printed & compared.</p>

Fig. 3.6 An excerpt of RALIC's meeting minutes on card design. Names have been anonymised for reasons of privacy

the same changes occurred because changes discussed in team meetings can be subsequently reported in board meetings, reflected in functional specification, cascaded into technical specification and finally into workplans. Interviews were also conducted with the project team to understand the project context, clarify uncertainties or ambiguities in the documentation, and verify the findings.

Some statements extracted from the documentation belong to more than one CoRE layer. For example, the statement “for identification and access control using a single combined card” consists of two patterns (i.e., Person and PartyAuthentication) and a functional constraint (i.e., combined card). In such cases, the statements are split into their respective CoRE layers.

Although the difference between pattern, functional constraint, and non-functional constraint is clear cut, policies and rules can sometimes be difficult to distinguish. This is because high-level policies can be composed of several lower-level policies [3, 21]. For example, the statement “Access Systems and Library staff shall require leaver reports to identify people who will be leaving on a particular day” is a policy rather than a rule, because it describes the purpose of the leaver reports but not how the reports should be generated. Sometimes, a statement can consist of both rules and policies. For example, “HR has changed the staff organisation structure; changes were made from level 60 to 65.” Interviews with the stakeholders revealed that UCL has structured the departments for two faculties from a two tier to a three tier hierarchy. This is a UCL policy change, which has affected the specific rule for RALIC, which is to display department titles from level 65 of the hierarchy onwards.

Each change was recorded by the date it was approved, a description, the type of change, and its CoRE layer (or N/A if it does not belong to any layer). Table 3.2 illustrates the change records, where the type of change is abbreviated as A for addition, M for modification, and D for deletion, and the CoRE layers are abbreviated as P for pattern, FC for functional constraint, NFC for non-functional constraint, BP for business policies, BR for business rules, and N/A if it does not belong to any layer. There were a total of 97 changes and all requirements can be exclusively classified into one of the four layers.

3.5.3 Layer Volatility

To evaluate if CoRE accurately anticipates the volatility of each shearing layer, the change records for RALIC (Table 3.2) is used to calculate each layer’s volatility. The volatility of a layer is the total number of requirements changes divided by the initial number of requirements in the layer. The volatility ratio formula from Stark et al. [31] is used Eq (3.1).

$$\text{volatility} = \frac{\text{Added} + \text{Deleted} + \text{Modified}}{\text{Total}}, \quad (3.1)$$

Table 3.2 Partial change records

Date	Description	Type	Layer
6 Oct 05	The frequency of data import from other systems is 1 h (changed from 2 h).	M	NFC
6 Oct 05	The access rights for students expire 3 months after their expected course end date (changed from 6 months).	M	BR
18 Oct 05	End date from the staff and visitor systems and student status from the student system is used to determine whether a person is an active cardholder.	A	BR
16 Nov 05	Expired cards must be periodically deleted.	A	BP
30 Nov 05	Access card printer should be able to print security logos within the protective coating.	A	FC
8 May 06	The highest level department name at departmental level 60 should be printed on the card.	A	BR
17 Jan 07	The frequency of data import from other systems is 2 min (changed from 5 min).	M	NFC
2 Apr 07	Replace existing library access control system that uses barcode with the new access control system.	D	BP
1 Jul 08	Programme code and name, route code and name, and faculty name from the student system is used to determined their access on the basis of courses.	A	BR
15 Aug 08	The highest level department name at departmental level 65 should be printed on the card (changed from 60).	M	BR
1 Jan 09	Introduce access control policies to the Malet Place Engineering Building.	A	BP

where *Added* is the number of added requirements, *Deleted* is the number of deleted requirements, *Modified* is the number of modified requirements, and *Total* is the total number of initial requirements for the system. Volatility is greater than 1 when there are more changes than initial requirements.

Using Eq. 3.1 to calculate the volatility for each layer enables the comparison of their relative volatility. As expected, patterns are the most stable, with no changes over 3.5 years. This is followed by functional constraints with a volatility ratio of 0.6, non-functional constraints with a volatility ratio of 2.0, and business policies and rules with a volatility ratio of 6.4. The volatility ratio between each layer is also significantly different, showing a clear boundary between the layers. Business policies and business rules have similar volatility when considered separately: policies have a volatility ratio of 6.40 and rules 6.44.

3.5.4 Timing of Change

The volatility ratio indicates the overall volatility of a layer. To understand *when* the changes occur, the number of quarterly changes for each layer is plotted over the duration of the requirements change study, as illustrated in Fig. 3.7.

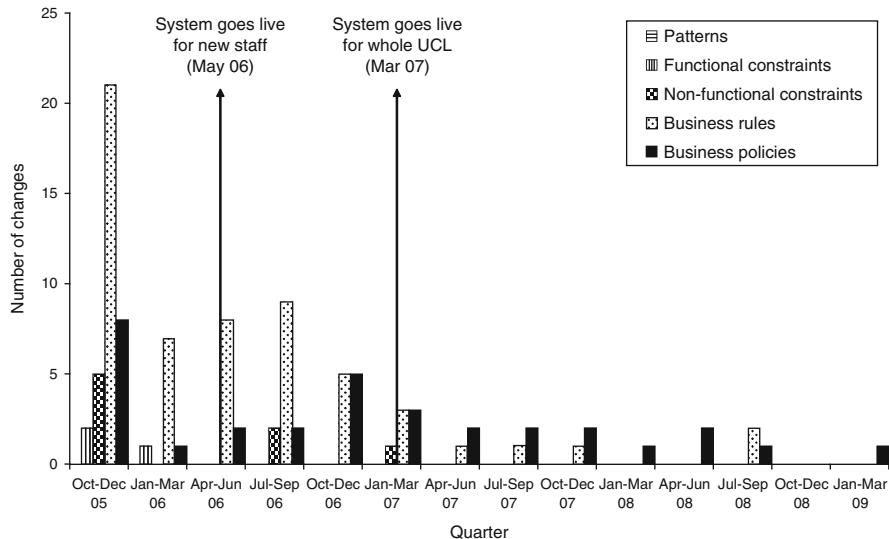


Fig. 3.7 Quarterly requirements changes for each layer

The quarter Oct–Dec 05 has the highest number of changes for functional constraints, non-functional constraints, business policies and rules because the requirements elicitation and documentation were still in progress. The project board had signed off the high-level requirements, but the details surrounding access rights and card processing were still under progress. Many of the changes were due to better understanding of the project and to improve the completeness of the requirements.

Consistent with the existing literature (e.g., [7]), missing requirements surfaced from change requests after the system was deployed. The system went live first for new staff in May 06 and then for the whole of UCL in March 07. Each time it went live, the number of requirements change increased in the following quarters.

A rise in policy change in quarters Oct–Dec 05 and Jan–Mar 07 was followed by a rise in rule change in the following quarters, because business rules are based on business policies. As more than one rule can be based on the same policy, the number of changes in rules is naturally higher than that of policies. Nevertheless, after the system went live, policy changes did not always cascade into rule changes. For example, application of the access control policy to new buildings required only the reapplication of existing rules.

Interestingly, the quarterly changes for business rules resemble an inverse exponential function, as the number of changes was initially large but rapidly decreased. In contrast, the quarterly changes for business policies shows signs of continuous change into the future. Rules suffered from a high number of changes to start with, as the various UCL divisions were still formulating and modifying the rules for the new access control system. After the system went live, the changes reduced to one per quarter for three quarters, and none thereafter. One exception is

in quarter Jul–Sep 08, where UCL faculty restructuring had caused the business processes to change, which affected the rules. Nevertheless, these changes were due to the environment of the system rather than missing requirements.

3.5.5 *Implications*

CoRE produces requirements models that are adequate without unnecessary details because leaf goals are either mapped to archetypes, which are the essence of the system, or to business policies and rules, ensuring that business specific requirements are supported by business reasons. CoRE does not rely on domain experts because the archetypes capture requirements that are pervasive in the domain. The requirements models are complete and pertinent because all the requirements in RALIC can be classified into the four CoRE layers. Also, RALIC stakeholders could readily provide feedback on CoRE models (e.g., Fig. 3.5), showing that the model is easy to understand.

As CoRE is based on goal modelling, it inherits their multi-level, open and evolvable, and traceable features. CoRE captures the system at different levels of abstraction and precision to enable stepwise elaboration and validation. The AND/OR refinements enables the documentation and consideration of alternative options. As CoRE separates requirements based on their relative volatility, most changes occur in business policies and rules. The rationale of a requirement is traceable by traversing up the goal tree. The source of a requirement can be traced to the stakeholder who defined the goal leading to the requirement.

Finally, CoRE externalises volatile business policies and rules. As such, the system can be designed such that software architecture components that implement these volatile requirements are loosely coupled with the rest of the system. For example, in service-oriented architecture, these components can be implemented such that changes in business policies are reflected in the system as configuration changes, and changes in business rules are reflected as changes in service composition [18]. This minimises the disruptive effect of changing requirements on the architecture.

3.6 Future Work

The study is based on a single project, hence there must be some caution in generalising the results to other projects, organisations, and domains. Also, the study assumed that all requirements and all changes are documented. Future work should evaluate CoRE on projects from different domains, and in a forward looking manner, i.e., anticipate the change and see if it happens. As RALIC is a business system, enterprise archetype patterns were used. Future work should investigate the use of software patterns in other domains, such as manufacturing or medical

domains. Finally, future work should also investigate the extent of CoRE's support for requirements engineers who are less experienced.

The requirements changes that CoRE anticipates are limited to those caused by the business environment and stakeholder needs. But requirements changes can be influenced by other factors. For example, some requirements may be more volatile than others because they cost less to change. In addition, CoRE does not consider changes due to corrections, improvements, adaptations or uncertainties. Future work should consider a richer model that accounts for these possibilities, as well as provide guidance for managing volatile requirements.

CoRE anticipates change at the level of a shearing layer. But among the elements in the same layer, it is unclear which is more volatile. For example, using CoRE, business rules are more volatile than functional constraints, but it is unclear which rules are more likely to change. Future work should explore a predictive model that can anticipate individual requirements change and the timing of the change. This could be done by learning from various attributes for each requirement such as the number of discussions about the requirement, the stakeholders involved in the discussion and their influence in the project, and the importance of the requirement to the stakeholders. Much of these data for RALIC have been gathered in previous work [17].

3.7 Conclusion

This chapter has described CoRE, a novel expert independent method that classifies requirements into layers that change at different rates. The empirical results show that CoRE accurately anticipates the volatility of each layer. From the most stable to the most volatile, the layers are patterns, functional constraints, non-functional constraints, and business policies and rules.

CoRE is a simple but reliable method to anticipate change. CoRE has been used in the Software Database project³ to build a UCL wide software inventory system. Feedback from the project team revealed that CoRE helped the team better structure their requirements, and gave them an insight of requirements that were likely to change. As a result, their design and implementation prepared for future changes, thus minimising the disruptive effect of changing requirements to their architecture.

Acknowledgments The authors would like to thank members of the Estates and Facilities Division and Information Services Division at University College London for the RALIC project documentation, their discussions and feedback on the requirements models, as well as Peter Bentley, Fuyuki Ishikawa, Emmanuel Letier, and Eric Platon for their feedback on the work.

³<http://www.ucl.ac.uk/isd/community/projects/azlist-projects>

References

1. Anton AI (1996) Goal-based requirements analysis. In: Proceedings of the 2nd international conference on requirements engineering. Colorado Springs, pp 136–144
2. Arlow J, Neustadt I (2003) Enterprise patterns and MDA: building better software with archetype patterns and UML. Addison-Wesley, Boston
3. Berenbach B, Paulish DJ, Kazmeier J, Daniel P, Rudorfer A (2009) Software systems requirements engineering: in practice. McGraw-Hill Osborne Media, New York
4. Boehm BW (1981) Software engineering economics. Prentice Hall, Englewood Cliffs
5. Brand S (1995) How buildings learn: what happens after they're built. Penguin Books, New York
6. Chung L, Nixon BA, Yu E, Mylopoulos J (1999) Non-functional requirements in software engineering. Springer, Berlin
7. Cockburn A (2002) Writing effective use cases. Addison-Wesley, Boston
8. Dardenne A, van Lamsweerde A, Fickas S (1993) Goal-directed requirements acquisition. *Sci Comput Programming* 20(1–2):3–50
9. Foote B, Yoder J (2000) Big ball of mud. *Pattern Lang Program Des* 4(99):654–692
10. Harker SDP, Eason KD, Dobson JE (1993) The change and evolution of requirements as a challenge to the practice of software engineering. In: Proceedings of the IEEE international symposium on requirements engineering, San Diego, pp 266–272
11. IEEE Computer Society (2000) IEEE recommended practice for software requirements specifications, Los Alamitos
12. International Organization for Standardization (2001) Software engineering – product quality, ISO/IEC TR 9126, Part 1–4
13. Jacobson I (1995) The use-case construct in object-oriented software engineering. In: Scenario-based design: envisioning work and technology in system development. Wiley, New York, pp 309–336
14. Jones C (1996) Strategies for managing requirements creep. *Computer* 29(6):92–94
15. Kotonya G, Sommerville I (1998) Requirements engineering. Wiley, West Sussex
16. Leffingwell D (1997) Calculating the return on investment from more effective requirements management. *Am Program* 10(4):1316
17. Lim SL (2010) Social networks and collaborative filtering for large-scale requirements elicitation. PhD Thesis, University of New South Wales. Available at http://www.cs.ucl.ac.uk/staff/S.Lim/phd/thesis_soolinglim.pdf, accessed on 14 December 2010
18. Lim SL, Ishikawa F, Platon E, Cox K (2008) Towards agile service-oriented business systems: a directive-oriented pattern analysis approach. In: Proceedings of the 2008 IEEE international conference on services computing, Honolulu, Hawaii, vol 2. pp 231–238
19. Lim SL, Quercia D, Finkelstein A (2010) StakeNet: using social networks to analyse the stakeholders of large-scale software projects. In: Proceedings of the 32nd international conference on software engineering, vol 1. New York, pp 295–304
20. Mens T, Galal G (2002) 4th workshop on object-oriented architectural evolution. Springer, Berlin, pp 150–164
21. Object Management Group (2006) Business motivation model (BMM) specification. Technical Report dtc/060803
22. O'Neill RV, DeAngelis DL, Waide JB, Allen TFH (1986) A hierarchical concept of ecosystems. Princeton University Press, Princeton
23. Papantonio B, Nathanael D, Marmaras N (2003) Moving target: designing for evolving practice. In: HCI international. MIT Press, Cambridge
24. Robertson S, Robertson J (2006) Mastering the requirements process. Addison-Wesley, Reading
25. Ross RG (2003) Principles of the business rule approach. Addison-Wesley, Boston
26. Salthe SN (1993) Development and evolution: complexity and change in biology. MIT Press, Cambridge, MA

27. Simmonds I, Ing D (2000) A shearing layers approach to information systems development, IBM Research Report RC21694. Technical report, IBM
28. Sommerville I (2004) Software engineering, 7th edn. Addison-Wesley, Boston
29. Sommerville I, Sawyer P (1997) Requirements engineering: a good practice guide. Wiley, Chichester
30. Standish Group (1994) The CHAOS report
31. Stark GE, Oman P, Skillicorn A, Ameele A (1999) An examination of the effects of requirements changes on software maintenance releases. *J Softw Maint Res Pr* 11(5):293–309
32. van Lamsweerde A (2001) Goal-oriented requirements engineering: a guided tour. In: Proceedings of the 5th IEEE international symposium on requirements engineering. Toronto, pp 249–262
33. van Lamsweerde A (2009) Requirements engineering: from system goals to UML models to software specifications. Wiley, Chichester
34. Wiegers K (2003) Software requirements, 2nd edn. Microsoft Press, Redmond
35. Yu ESK (1997) Towards modelling and reasoning support for early-phase requirements engineering. In: Proceedings of the 3rd IEEE international symposium on requirements engineering. Annapolis, pp 226–235

Chapter 4

Traceability in the Co-evolution of Architectural Requirements and Design

Antony Tang, Peng Liang, Viktor Clerc, and Hans van Vliet

Abstract Requirements and architectural design specifications can be conflicting and inconsistent, especially during the design period when requirements and architectural design are co-evolving. One reason is that stakeholders do not have up-to-date knowledge of each other's work to fully understand potential conflicts and inconsistencies. Specifications are often documented in a natural language, which also makes it difficult for tracing related information automatically. In this chapter, we introduce a general-purpose ontology that we have developed to address this problem. We demonstrate an implementation of semantic wiki that supports traceability of co-evolving requirements specifications and architecture design.

4.1 Introduction

Let us begin by considering a typical software architecting scenario:

A team of business analysts and users work on a new software system in an organization. The business analysts and users document the business goals, use-case scenarios, system and data requirements in a requirements document. The team of software and system architects studies this document, which is in a draft version, and they start to create some designs. The architects realize that more information from the stakeholders is required, and they must validate the usability requirements with the operators to ensure they understand the efficiency requirements of the user interface; they also realize that they must understand the data retention and storage requirements from the business managers; finally, they have to analyze the performance requirements of the system. They find that the performance of retrieving data is slow and that hinders the data entry task. They have to discuss and resolve this issue together with the business analysts who represent the business operation unit. In the meantime, the business analysts have decided to add new functionalities to the system ...

In this scenario, many people are involved in the development of the system, and the knowledge used in the development is discovered incrementally over time. Common phenomena such as this occur every day in software development. Three problematic situations often arise that lead to knowledge communication issues in software design.

The first problematic situation is that *knowledge is distributed*. System development always involves a multitude of stakeholders and each stakeholder possesses only partial knowledge about some aspects of a system. In this case, business users only know *what* they want, but they do not know *how* to make it work, and vice versa for the architects. In general, requirements are specified by many stakeholders such as end-users, business managers, management teams, and technology specialists. Architecture designs, in turn, are specified by architects, application software designers, database specialists, networking specialists, security specialists, and so on. As a result, the requirements and architectural design specifications that are created by different stakeholders are often conflicting and inconsistent.

Secondly, *information is imperfect*. Not all information about requirements and architecture design is explicitly documented and retrievable. The requirements and architecture design are for the most part recorded in specifications but some knowledge will remain only in the heads of those who are deeply involved in the software development project. The vast number of requirements and design entities in large-scale systems can potentially hide requirements and design conflicts. These conflicts can remain undetected until the relevant design concerns are considered in certain views and with certain scenarios. Additionally, not all relationships between the design entities and the requirements statements are captured sufficiently in the specifications to allow stakeholders to detect potential conflicts.

Thirdly, *requirements and architecture design can co-evolve over time*. Requirements and insight into how these requirements may be implemented evolve over time through exploration, negotiation, and decision-making by many people. In the scenario given at the beginning of this chapter, architects understand the performance constraints in data retrieval that the business users have no knowledge of. Because of the performance constraint, compromises in the design and requirements will have to be made. Sometimes, requirement decisions that have profound impact on the architecture design can be made before the start of the design activities. In this way, requirements documents can be signed off before architecture design commences. However, agreeing on these important requirement decisions is not always possible.

Owing to these issues, it is obvious that the development of requirements specifications and the architectural design specifications would overlap in time, implying that these specifications can co-evolve simultaneously. In order to allow stakeholders to communicate the potential impacts and conflicts between requirements and the architectural design during their co-evolution, different stakeholders must be able to trace between requirements and design to assess the viability of the solution during this process.

Traceability between requirements and design has been studied previously [1–4]. These methods use static trace links to trace different types of requirements, design, and code objects. They employ different ways to construct traces. However, these methods suffer from two issues: (a) the need to laboriously establish the trace links and maintain them as a system evolves; (b) they do not support on-going design activities. An improvement to these methods is to provide dynamic tracing at different levels of design abstraction. An example of this dynamism is a scoped

approach to the traceability of product line and product levels [5]. However, this approach is not suitable for general purpose traceability of requirements to architecture design.

In this research, we investigate how requirements and design relationships can become traceable when requirements and design objects are both incomplete and evolving simultaneously, and the static trace links used by conventional traceability methods are insufficient and out-of-date. Our work provides a general ontological model to support the traceability of co-evolving architectural requirements and design. Based on this ontology, we have applied semantic wikis to support traceability and reasoning in requirements development and architecture design.

This remaining of this chapter is organized as follows. Section 4.2 describes the issues on current traceability management from requirements to architecture design. Section 4.3 presents the traceability use cases for co-evolving architecture requirements and design with a metamodel that supports this traceability. Section 4.4 introduces the implementation of Software Engineering Wiki (SE-Wiki), a prototype tool that supports the dynamic traceability with an underlying ontology based on the traceability metamodel. Section 4.5 presents three concrete examples of using SE-Wiki to perform the traceability use cases. We conclude this chapter in Section 4.6.

4.2 Issues in Finding the Right Information

Requirements traceability is the ability to describe and follow the life of requirements [1]. Ideally, such traceability would enable architects and designers to find all relevant requirements and design concerns for a particular aspect of software and system design, and it would enable users and business analysts to find out how requirements are satisfied. A survey of a number of systems by Ramesh and Jarke [2] indicates that requirements, design, and implementation ought to be traceable to ensure continued alignment between stakeholder requirements and various outputs of the system development process. The IEEE standards recommend that requirements should be allocated, or traced, to software and hardware items [6, 7].

On the other hand, [1] distinguishes two types of traceability: *pre-requirements specification* and *post-requirements specification*. The difference between these two traceability types lies in *when* requirements are specified in a document. With the emergence of agile software development and the use of architecture frameworks, the process of requirements specification and design becomes more iterative. As a result, the boundary between pre- and post-requirement traceability is harder to define because of the evolving nature of requirements specification activity.

In this section, we examine the knowledge that is required to be traced, the challenges of using conventional requirements traceability methods that are based on static information, and compare that with an environment where information

changes rapidly and the capabilities to trace such dynamic requirements information must improve.

4.2.1 Architectural Knowledge Management and Traceability

Architectural knowledge is the integrated representation of the software architecture of a software-intensive system (or a family of systems), the architectural design decisions, and the external context/environment. For facilitating better design decision-making, architects require “just-in-time” knowledge [8]. Just-in-time knowledge refers to the right architectural knowledge, provided to the right person, at any given point in time.

Architectural knowledge should capture not just the outcomes of a design but also the major architectural decisions that led to it [9]. Capturing the architectural decisions facilitates a better decision-making process in shorter time, saving rework and improving the quality of the architecture [10, 11]. Hence, it is important to not only trace to the resulting architecture design, but also to the decisions, including their rationale, that led to that design.

Sound management of architectural knowledge can help in providing just-in-time knowledge by building upon two important knowledge management strategies [12]. *Personalisation* implies providing knowledge that urges the knowledge workers to interact with each other, by making known who possesses certain knowledge. *Codification*, on the other hand, focuses on identifying, eliciting, and storing the knowledge in e.g., repositories.

A hybrid strategy that uses both personalisation and codification aspects can be beneficial to sound architectural knowledge management, especially in the iterative process of architecting. When tracing back and forth between requirements and architecture, architects need specific support with adequate information relevant for addressing the design issues at hand. Hence, the proposed traceability method using semantic wikis is aligned with the current knowledge management strategy.

4.2.2 Requirements and Architecture Design Traceability

During the development life cycle, architects and designers typically use specifications of business requirements, functional requirements, and architecture design. Traceability across these artifacts is typically established as a static relationship between entities. An example would be to cross-reference requirement *R13.4* which is realized by module *M_comm()*. It is argued by [3] that relating these pieces of information helps the designers to maintain the system effectively and accurately, and it can lead to better quality assurance, change management, and software maintenance. There are different ways in which such traceability between requirements and architecture design can be

achieved. Firstly, use a traceability matrix to associate requirements to design entities in a document [13]. This is typically implemented as a table or a spreadsheet. Traceability is achieved by finding the labels in a matrix and looking up the relevant sections of the documents. Secondly, use a graphical tool in which requirements and design entities are represented as nodes and the relationships between them as arcs. Traceability is achieved by traversing the graph. Examples of such a system are provided by [2, 14]. Thirdly, use some keyword- and metadata-based requirements management tools. The metadata contains relationships such as *requirement X* is realized by *component Y*. The user would, through the tool, access the traceable components. Examples of such systems are DOORS [15], RequisitePro [16], and [17]. Fourthly, automatically generate trace relationships through supporting information such as source code [4], or requirements documents [18, 19].

Traceability is needed not only for maintenance purpose when all the designs are complete and the system has been deployed; static traceability methods can work well under this circumstance. Traceability is also needed when a system design is in progress, and the relationships between requirements and design entities are still fluid. The following scenarios are typical examples:

- When multiple stakeholders make changes to the requirements and the architecture design simultaneously during development
- Stakeholders are working from different locations and they cannot communicate proposed changes and ideas to the relevant parties instantly
- Requirements decisions or architectural decisions may have mutual impact on each other, even conflict with each other, but these impacts are not obvious when the two parties do not relate them

Under these circumstances, static traceability methods would fail because it is difficult to establish comprehensive traceability links in a documentation-based environment. In real-life, potential issues such as these are discussed and resolved in reviews and meetings. Such a solution requires good communication and management practice for it to work. A solution was proposed to use events to notify subscribers who are interested in changes to specific requirements [20]. This, however, would not serve for situations in which many new requirements and designs are being created.

In order to address this issue, this chapter outlines the use of a query-based traceability method to allow architects and requirements engineers to find relevant information in documents. This method applies a software engineering ontology to requirements and architecture design documentation.

4.2.3 Applying Semantic Wikis in Software Engineering

Software development is from one perspective a social collaborative activity. It involves stakeholders (e.g., customers, requirements engineers, architects,

programmers) closely working together and communicating to elicit requirements and to create the design and the resulting software product. This collaboration becomes more challenging when an increasing number of projects are conducted in geographically distributed environments – Global Software Development (GSD) becoming a norm. In this context, many CSCW (Computer Supported Collaborative Work) methods and related tools have been applied in software engineering to promote communication and collaboration in software development [21], but the steep learning-curve and the lack of openness of these methods and tools inhibit their application in industrial projects.

Semantic wikis combine wiki properties, such as ease of use, open collaboration, and linking, with Semantic Web technologies, such as structured content, knowledge models in the form of ontologies, and reasoning support based on formal ontologies with reasoning rules [22, 23]. As such, a semantic wiki intends to extend wiki flexibility by allowing for reasoning with structured data: semantic annotations to that data correspond to an ontology that defines certain properties. Once these semantic annotations are created, they are then available for extended queries and reasoning [22]. The combination of these features provides an integrated solution to support social collaboration and traceability management in software development. From one perspective, semantic wikis can facilitate social collaboration and communication in software development. Normal wikis have been used by the software industry to maintain and share knowledge in software development (e.g., source code, documentation, project work plans, bug reports, and so on) [24], requirements engineering [25], and architecture design [26]. With the semantic support of an underlying ontology and semantic annotations, semantic wikis can actively support users in understanding and further communicating the knowledge encoded in a wiki page by – for example – appropriately visualizing semantically represented project plans, requirements, architecture design, and the links between them [22]. From the other perspective, the underlying ontologies that support semantic wikis are composed of the concepts from software engineering and the problem domains, and the relationships between these concepts can be formally specified by the RDF [27] and OWL [28] ontology languages. This ontology representation helps users to search for semantic annotations encoded in the semantic wikis through concept relationships and constraints, and provides reasoning facilities to support dynamic traceability in software development.

Semantic wikis have been applied to different areas of software engineering, mostly in research environments. One application focuses on combining documents from Java code, and to model and markup wiki documents to create a set of consistent documents [29]. Ontobrowse was implemented for the documentation of architecture design [30]. Softwiki Ontology for Requirements Engineering (SWORE) is an ontology that supports requirements elicitation [31]. So far, we know of no ontological model that supports the traceability between requirements and architectural design.

4.3 What Needs to be Traced and Why?

4.3.1 Architectural Design Traceability

Many requirements traceability methods implicitly assume that a final set of requirements specifications exists from which traceability can be performed. Some methods require users to specify the traces manually [32], whilst others automatically or semi-automatically recover trace links from specifications [3, 17]. The assumption that a definitive set of unchanging documents exists does not always hold because tracing is also required when requirements and architecture design are being developed. This is a time when requirements and architecture design co-evolve. Architectural design activities can clarify non-functional requirements and trade-offs can compromise business requirements. During this time, a set of final specifications are not ready but traceability between related items can help architects find their ways.

Traceability between requirements and architecture design is generally based on the requirements and design specifications, but the other types of documented knowledge should also be traceable to the architecture design. This knowledge often defines the context of a system, e.g., technology standards that need to be observed in a design or the interface requirements of an external system.

In discussing the support for the traceability of group activities, [1] noted that *Concurrent work is often difficult to coordinate, so the richness of information can be lost*. There are some issues with supporting concurrent updates. Firstly, the information upon which a trace link is based has changed. For example, the requirement statement has changed. The trace link will need to be investigated and may be updated because information that is linked through it may be irrelevant or incorrect. It is laborious and therefore error prone to keep trace links up to date as requirements and designs change. Secondly, many decision makers exist and many parts of the requirements and designs can be changed simultaneously. In this situation, not all relevant information can be communicated to the right person at the right time. For instance, a business user adding a requirement to the system may not know that this change has a performance impact on the architecture design, thus she/he may not be aware that such a decision requires an architectural design assessment. In this case, some hints from an intelligent tracing system could help to highlight this need.

4.3.2 Traceability Use Cases in Co-evolving Architectural Requirements and Design

In order to develop traceability techniques to support requirements-architecture design co-evolution, we have developed a set of traceability use cases. These use

cases show examples of typical activities of architects that require support by a reasoning framework (see Sect. 4.1). The use cases are described following a technique introduced in [33] providing a scenario, problem and solution description, and a detailed description of the scenario.

Scenario 1 – Software Reuse An architect wants to check if existing software can be reused to implement a new functional requirement, and the new functionality is similar to the existing functionality.

Problem The architect needs to understand the viability of reusing software to satisfy existing and new functional and quality requirements.

Solution The architect first finds all the architecture components that realize the existing functional requirements which are similar to the new functional requirement. Then, the architect can trace the existing architecture components to determine what quality requirements may be affected, and whether the existing software is supporting the new requirement.

Scenario description

1. The architect thinks that the existing software can support a new functional requirement which is similar to existing functional requirements.
2. The architect selects the existing functional requirements and identifies all the software components that are used to realize them.
3. For each software component found, the architect identifies the related architectural structure and the quality requirements.
4. The architect assesses if the existing quality requirements are compatible with the quality requirements of the new functional requirement.
5. If so, the architect decides to reuse the components to implement the new functional requirement.

Scenario 2 – Changing Requirement An architect wants to update the architecture design because of a changing functional requirement.

Problem The architect needs to understand the original requirements and the original architecture design in order to cater for the change.

Solution The architect first finds all existing requirements that are related to the changing requirement. Then the architect identifies the decisions behind the original design. The architect can assess how the changing requirement would affect related existing requirements and the original design.

Scenario description

1. The architect identifies all the related artifacts (e.g., related requirements, architectural design decisions, and design outcomes) concerning the changing requirement.
2. The architect evaluates the appropriateness of the changing requirement with related existing requirements.
3. The architect extracts previous architectural design decisions and rationale for the changing requirement.
4. The architect identifies new design issues that are related to the changing requirement.

5. The architect proposes one or more alternative options to address these new issues.
6. The architect evaluates and selects one architectural design decision from alternative options. One of the evaluation criteria is that the selected decision should not violate existing architectural design decisions and it should satisfy the changing requirement.
7. The architect evaluates whether the new architectural design outcome can still satisfy those non-functional requirements related to the changing functional requirement.

Scenario 3 – Design Impact Evaluation An architect wants to evaluate the impact a changing requirement may have on the architecture design across versions of this requirement.

Problem The architect needs to understand and assess how the changing requirement impacts the architecture design.

Solution The architect finds all the components that are used to implement the changing requirement in different versions, and evaluates the impact of the changing requirement to the architecture design.

Scenario description

1. The architect extracts all the components that realize or satisfy the changing requirement in different versions, functional or non-functional.
2. The architect finds all the interrelated requirements in the same version and the components that implement them.
3. The architect evaluates how the changes between different versions of the requirement impact on the architecture design, and can also recover the decision made for addressing the changing requirement.

In order to support these traceability scenarios, a dynamic traceability approach is needed. This approach would require the traceability relationships to remain up-to-date with evolving documentation, especially when the stakeholders work with different documents and some stakeholders do not know what others are doing. In summary, the following traceability functions need to be provided for such an approach to work effectively:

- Support the update of trace links when specification evolves – this function requires that as documents are updated, known concepts from the ontology are used automatically to index the keywords in the updated documents, thereby providing an up-to-date relationship trace information.
- Support flexible definition of trace relationships – the traceability relationships should not be fixed when the system is implemented. The application domain and its vocabulary can change and the ways designers choose to trace information may also change. Thus the trace relationships should be flexible to accommodate such changes without requiring all previously defined relationships to be manually updated.
- Support traceability based on conceptual relationships – certain concepts have hierarchical relationships. For instance, performance is a quality requirement,

response time and throughput are requirements that concretize a performance requirement. A user may wish to enquire about the quality requirements of a system, the performance requirements, or, even more specifically, the response time of a particular function.

- Concurrent use by requirements engineers and architects – business architects, requirements engineers, data architects, and software architects typically work on their respective areas concurrently. They, for instance, need to find the latest requirements that affect their design, then make some design decisions and document them. As they do, their decisions in turn may impact the others who are also in the process of designing. The concurrent nature of software development requires that this knowledge and its traces are up-to-date.

4.3.3 Traceability Metamodel

The Traceability metamodel for Co-evolving Architectural Requirements and Design (T-CARD) is based on the IBIS notations (Issue, Position, Argument, and Decision) [34] to represent design argumentation. This metamodel is constructed to satisfy the traceability use cases identified earlier. The concepts and the relationships of T-CARD are presented in UML notation, grouped into the problem space and the solution space, as shown in Fig. 4.1. It consists of the following concepts:

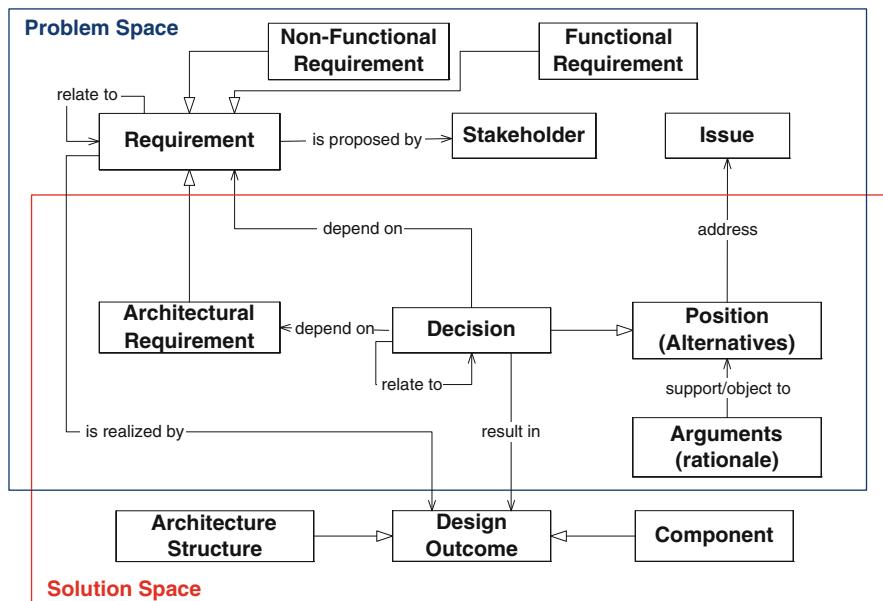


Fig. 4.1 Traceability metamodel for co-evolving architectural requirements and design

Stakeholder: refers to anyone who has direct or indirect interest in the system. A *Requirement* normally is proposed by a specific *Stakeholder*, which is the original source of requirements.

Requirement: represents any requirement statements proposed by a specific *Stakeholder*, and a *Requirement* can relate to other *Requirements*. There are generally two types of requirements: *Functional Requirements* and *Non-Functional Requirements*, and a *Requirement* is realized by a set of *Design Outcomes*. Note that the general relationship relate to between *Requirements* can be detailed further according to the use case scenarios supported.

Architectural Requirement: is a kind of *Requirement*, and *Architectural Requirements* are those requirements that impact the architecture design. An *Architecture Requirement* can also relate to other *Architectural Requirements*, and the relate to relationship is inherited from its superclass *Requirement*.

Issue: represents a specific problem to be addressed by alternative solutions (*Positions*). It is often stated as a question, e.g., what does the data transport layer consist of?

Position: is an alternative solution proposed to address an *Issue*. Normally one or more potential alternative solutions are proposed, and one of them is to be selected as a *Decision*.

Argument: represents the pros and cons argument that either support or object to a *Position*.

Decision: is a kind of *Position* that is selected from available *Positions* depending on certain *Requirements* (including *Architectural Requirements*), and a *Decision* can also relate to other *Decisions* [35]. For instance, a *Decision* may select some products that constrain how the application software can be implemented.

Design Outcome: represents an architecture design artifact that is resulted from an architecture design *Decision*.

Component and **Architecture Structure:** represent two types of *Design Outcomes*, that an *Architecture Structure* can be some form of layers, interconnected modules etc.; individual *Components* are the basic building blocks of the system.

The concepts in this metamodel can be classified according to the Problem and Solution Space in system development. The Problem and Solution Space overlap: *Architectural Requirement* and *Decision*, for example, belong to both spaces.

4.4 Using Semantic Wikis to Support Dynamic Traceability

The metamodel depicted in Fig. 4.1 shows the conceptual model and the relationships between the key entities in the Problem and Solution Space. This conceptual model, or metamodel, requires an ontological interpretation to define the semantics of the concepts it represents. In this section, we describe the ontology of our model to support the use cases of co-evolving architectural requirements and design.

An ontology defines a common vocabulary for those who need to share information in a given domain. It provides machine-interpretable definitions of basic concepts in that domain and the relations among them [36]. In software development, architects and designers often do not use consistent terminology. Many terms can refer to the same concept, i.e., *synonyms*, or the same term is used for different concepts, i.e., *homonyms*. In searching through software specifications, these inconsistencies can cause a low recall rate and low precision rate, respectively [30].

An ontology provides a means to explicitly define and relate the use of software and application domain related terms such as design and requirements concepts. The general knowledge about an application domain can be distinguished from the specific knowledge of its software implementation. For instance, system throughput is a general concept about quality requirements and that is measurable; it can be represented in a sub-class in the hierarchy of quality requirements class. In an application system, say a bank teller system, its throughput is a specific instance of a performance measure. Using an ontology that contains a definition for these relationships, this enables effective searching and analysis of knowledge that are embedded in software documents.

Ontology defines concepts in terms of classes. A class can have subclasses. For instance, the *throughput* class is a subclass of *efficiency*, meaning that throughput is a kind of performance measure. A throughput class can have instances that relate to what is happening in the real-world. Some examples from an application system are: *the application can process 500 transactions per second or an operator can process one deposit every 10 s*.

A class can be related to another class through some defined relationships. For instance, a bank teller system *satisfies* a defined throughput rate. In this case, *satisfies* is a property of the bank teller system. The property *satisfies* links a specific requirement to a specific throughput.

4.4.1 A Traceability Ontology for Co-evolving Architectural Requirements and Design

An ontology requires careful analysis and planning. If an ontology is designed for a single software application, then it may not be flexible and general enough to support other systems. To support general traceability of requirements and architecture design specifications, we define an ontology using the requirements and architecture metamodel (Fig. 4.1). The ontology (Fig. 4.2) is represented in a UML diagram that depicts the class hierarchy and the relationships between the classes. The dotted line represents the relationships between classes. The relationships are defined in terms of the properties within a class.

In this model, there are five key concepts, represented by five groups of classes. These concepts are commonly documented in requirements and architecture design

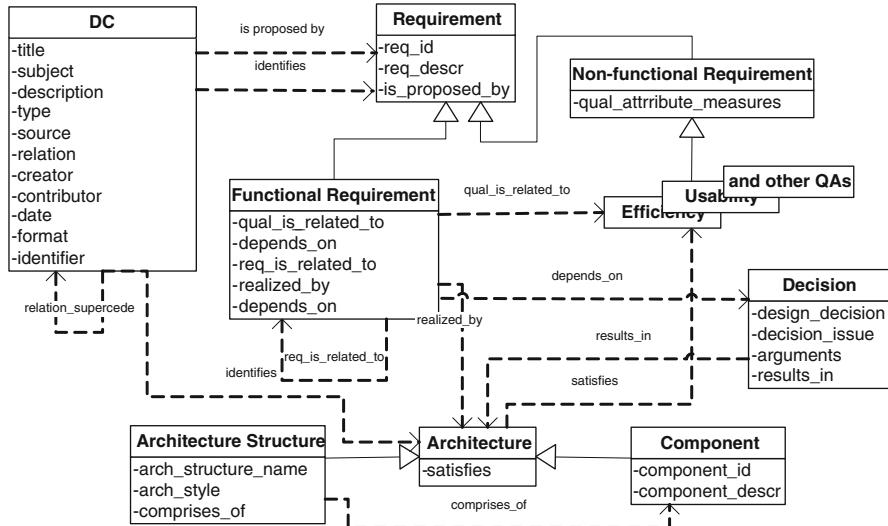


Fig. 4.2 Ontology for traceability between requirements and architecture

specifications, and the ontology is designed to represent these key concepts in software specifications:

- *DC* is a concept about the information of a document or a record. Dublin Core Metadata Initiative (DCMI) is an open organization engaged in the development of interoperable metadata standards that support a broad range of purposes and business models [37]. We make use of the concept defined in *dc:record* to identify the documents that are created for requirements and architecture purpose. In particular, we make use of the elements defined in the *DC* concept to support traceability of requirements and design across multiple versions of a single document. For example, a *DC* instance can identify the creator, the version, and the contributors of a requirement.
- *Requirement* is a concept that represents all the requirements of a system, including functional and non-functional requirements. A requirement has a unique identification and a description. These elements are implemented as properties (sometimes also referred to as slots) of the *Requirement* class. The properties of the *Requirement* class are inherited by all its subclasses. A requirement has an identifier and a description, so both functional and non-functional requirements have these properties as well. For example, an instance of a functional requirement would be a sub-class of *Requirement*. It would have a *req_id* of R1.1.3; a *req_descr* of *Change User Access*; it can be *realized_by* a component called *DefineAccessRight*. A user of the semantic wiki can ask the system for all requirements, and both functional and non-functional requirements would be retrieved.
- *Non-functional Requirement* represents all the quality requirements that must be satisfied by a system. Its subclasses such as efficiency and usability represent

different types of non-functional requirements. Non-functional requirements are sometimes measurable, e.g., throughputs. So, we use a property called `qual_attribute_measures` to capture this information for all measurable quality attributes.

- *Decision* represents the decisions that have been made. It has properties that capture the issues, arguments of a decision. For instance, the arguments for choosing an architecture design can be captured and linked to the design.
- *Architecture* represents the design outcomes of a decision, and the architecture realizes all requirements, both functional and non-functional. *Architecture* has two subclasses, *Architecture Structure* and *Component*. *Architecture Structure* represents the architecture styles that are used in an architecture design, such as multi-tier, web-based etc., whereas *Component* represents the individual building blocks that are used in an architecture. For instance, the ontology can capture the instances of a web-server architecture style and use the `comprise_of` property to link to *components* that generate dynamic HTML pages from a database application component.

Figure 4.2 depicts two class relationships: (a) class hierarchy represents an `is-a` relationship. So efficiency `is-a` non-functional requirement, and therefore it `is-a` requirement also; (b) a relationship between two disjoint classes is implemented through the property of a class. An example is that a requirement is proposed by a stakeholder. A stakeholder is represented in the ontology as a `dc:contributor`. In this case, both the DC record and the requirement are two disjointed classes linked together by the property field `is_proposed_by` in *Requirement* class. All the important relationships in this ontology are described below:

- A DC record identifies a document, be it a requirements document or an architecture design. This identification makes use of the standard elements provided by the DC metamodel. The amount of information that is contained in a document, whether it is one or a set of requirements, is up to the user. The key elements are: (a) the `title` and `subject` identify a requirement or a design; (b) the `source` identifies the version of a requirement; (c) the `relation` identifies if the document supercedes another document; (d) the `identifier` is the URI of the semantic wiki page. (e) the `contributor` identifies the stakeholders who contribute to the requirement or the design.
- Functional Requirement depends_on a decision. If a decision or a rationale of a design decision has been documented, then the requirement can be explained by the documented decision.
- Functional Requirement qual_is_related_to non-functional requirements. Often a requirements specification explicitly defines what quality is required by a system. In such cases, traceability can be provided if this relationship is captured in the ontology.
- Decision results_in an architecture. When business analysts and architects capture a decision, the outcome or the architecture design of a decision, including its rationale, can be traced to the decision. When `results_in` relationship is used in combination with the `depends_on` relationship, architects can query

what components are used to realize a specific requirement and why, for instance.

- Functional Requirement `is_realized_by` an architecture design. Designers, programmers, and testers often need to know the implementation relationships. If a decision has been documented and included in the ontology, then this relationship can be inferred from the original requirement. However, design decisions are often omitted, and so the implied realization link between requirements and design outcomes becomes unavailable. In order to circumvent this issue, we choose to establish a direct relationship between requirements and architecture.
- Architecture Design `satisfies` some non-functional requirements. This relationship shows that an architecture design can satisfy the non-functional requirements.

Together these relationships mark and annotate the texts in requirements and architecture specifications, providing the semantic meaning to enable architects and analysts to query and trace these documents in a meaningful way. Each trace link is an instance of the ontology relationships. Traceability is implemented by a semantic wiki implementation that supports querying or traversing.

4.4.2 SE-Wiki Implementation

In this section, we describe a semantic wiki implementation for Software Engineering, called SE-Wiki, which is implemented based on Semantic MediaWiki (SMW) [38]. We present how the ontology described in Sect. 4.4.1 is implemented with other semantic features in SE-Wiki. SMW is one of the prototype implementations of semantic wikis. There are two reasons for selecting SMW as the basis of SE-Wiki: (1) SMW implements most of semantic functions, including ontology definition and import, semantic annotation and traceability, and semantic query etc., which provide fundamental capabilities to perform the use cases presented in Sect. 4.3.2; and (2) SMW is a semantic extension of MediaWiki¹, which is the most popular wiki implementation on the Web, e.g., used by Wikipedia². The popularity and maturity of MediaWiki will make SE-Wiki easily adoptable by industry.

The SE-Wiki uses and extends the capability of SMW by applying the semantic features in the software engineering domain, from documentation, issue tracing, reuse, and collaboration to traceability management. In this chapter, we focus on the traceability management for the co-evolution of architectural requirements and design, combined with the ontology that supports dynamic traceability between

¹<http://www.mediawiki.org/>

²<http://www.wikipedia.org/>

Table 4.1 Ontology definition in SMW

Ontology construct	SMWConstruct	Example in SMW
Class	Category	<code>[[Category:Requirement]]</code>
Class property	Property	<code>[[req id::FR-001]]</code>
Class relationship	Property that links to the instance of other Class	<code>[[is proposed by::Stakeholder A]]</code> In the editing box of <i>Category:Functional Requirement</i> , specify <code>[[Category:Requirement]]</code>
SubClassOf	Category subcategorization	

architectural requirements and design. The implementation details of SE-Wiki are presented below.

Ontology support: as mentioned before, a semantic wiki is a wiki that has an underlying ontology that is used to describe the wiki pages or data within pages in the wiki. The ontology model elaborated in Sect. 4.4.1 is composed of four basic constructs, which can be defined in SMW as shown in Table 4.1. For example, `[[Category:Requirement]]` defines the class *Requirement*.

Semantic annotation: SMW only supports semantic annotation of wiki pages without supporting semantic annotation of data within wiki pages. This means that each semantic annotation in SMW is represented as a wiki page that belongs to a certain concept in the ontology model. In SE-Wiki, it is quite easy to annotate a requirement or architecture design artifact by adding text `[[Category:Concept Name]]` in the editing box of the wiki page based on the ontology defined or imported.

Semantic traceability refers to the semantic tracing between semantic annotations. In common wikis implementation, traceability is established by links between wiki pages without specific meaning of these links, while in semantic wikis, the semantics of these links are specified and distinguished by formal concept relationships in an ontology, which is beneficial to our purpose. For example, *Functional Requirement 001* is *proposed_by Stakeholder A*. The *Functional Requirement 001* and *Stakeholder A* are semantic annotations that belong to concept *Functional Requirement* and *Stakeholder* respectively. The concept relationship *is_proposed_by* between *Functional Requirement* and *Stakeholder* is used to trace semantically the relationship between the two annotations. In SE-Wiki, a semantic tracing can be established by an instance of Property in SMW between two wiki pages (i.e., semantic annotations), e.g., for above example, we can add text `[[is proposed by::Stakeholder A]]` in the editing box of *Functional Requirement 001* to create the semantic tracing.

Semantic query is used to query semantically the data (i.e., semantic annotations recorded in SE-Wiki) with semantic query languages, e.g., SPARQL [39] or a special query language supported by SMW. The capability of semantic queries is supported by the underlying ontology of the SE-Wiki, for example, *show all the Functional Requirements proposed by Stakeholder A*. Two methods for semantic query are provided in SE-Wiki: semantic search and in-line query. Semantic search provides a simple query interface, and user can input queries and

get the query results interactively. For example, query input `[[Category:Functional Requirement]][[is proposed by::Stakeholder A]]` will return all the functional requirements proposed by *Stakeholder A*. Semantic search is applicable to temporary queries that vary from time to time. In-line query refers to the query expression that is embedded in a wiki page in order to dynamically include query results into pages. Consider this in-line query: `ask: [[Category:Requirement]][[is proposed by::Stakeholder A]] / ?is proposed by`. It asks for all the requirements proposed by *Stakeholder A*. In-line query is more appropriate in supporting dynamic traceability between software artifacts, e.g., when a functional requirement proposed by *Stakeholder A* is removed from a requirements specification, the requirements list in the wiki page of *Stakeholder A* will be updated automatically and dynamically.

Example uses of these semantic features supported in SE-Wiki for the traceability use cases are further described in the next section.

4.5 Examples of Using SE-Wiki

In this section, we present several examples of applying SE-Wiki for performing the use cases presented in Sect. 4.3.2. We show how the semantic features in SE-Wiki can be used to support the co-evolution of architectural requirements and design. We draw these examples from the NIHR (National Institute for Health Research of United Kingdom) Portal Project [40]. The system aims to provide a single gateway to access information about health research and manage the life-cycles of research projects for the broad community of NIHR stakeholders, including e.g., researchers, managers, and complete research networks. We apply the SE-Wiki to the requirements and design specifications from this project. Then we demonstrate the use cases that we have defined to show SE-Wiki support for the traceability in co-evolving architectural requirements and design.

As presented in Sect. 4.4.2, some basic semantic functions are provided by SE-Wiki, including:

Ontology support: the underlying ontology concepts and the semantic relationships between concepts are defined in SMW.

Semantic annotation is used to annotate a requirement or architecture design artifact documented in a wiki page with a concept (i.e., *Category* in SMW).

Semantic traceability is supported by semantic tracing which is established between semantic annotations, and semantic traces will follow the semantic relationships defined at the ontology level.

Semantic query: the semantic annotations of requirements or architecture design artifacts allow SE-Wiki to query the annotations semantically by simple or complex queries. Queries can be entered manually through the query interface or embedded as in-line query in the wiki pages.

With the support of these basic semantic functions, we demonstrate how the use cases presented in Sect. 4.3.2 can be achieved with the examples from the NIHR

Portal project. In order to implement the use cases, all the relevant requirements and architecture specifications must be semantically annotated based on the traceability ontology specified in Sect. 4.4.1, e.g., in a sample requirement statement: *Student would like to download course slides from course website.*, *Student* is annotated as an instance of concept *Stakeholder*, *would like to* is annotated as an instance of concept relationship *is_proposed_by*, and *download course slides from course website* is annotated as an instance of concept *Requirement*.

These semantic annotations are performed by business analysts and architects as they document the specifications. The main difference between this method and some other requirements traceability methods is that individual requirement and design are semantically annotated, and their traceability is enabled by reasoning with the ontology concepts.

4.5.1 Scenario 1 Software Reuse

Description: An architect wants to check if existing software can be reused to implement a new functional requirement, which is similar to existing functional requirements that have been implemented (see Sect. 4.3.2).

Example: A new functional requirement *Track Usage: The Portal tool should be able to track usage of resources by all users* is proposed by the *Portal Manager*. The architect thinks that this new functional requirement is similar to an existing functional requirement: i.e., *Change User Access: The Portal tool should be able to change user's access rights to resources*³. The architect wants to check if the existing software (i.e., design outcomes/architecture) that is used to implement the requirement *Change User Access* can be reused to implement the new requirement *Track Usage*, especially with regards to the quality requirements.

Since the requirements and architecture specifications are already semantically annotated in SE-Wiki, semantic query can be employed to query the direct and indirect tracing relationships from an instance of *Functional Requirement* (i.e., the existing functional requirement *Change User Access*) to all the concerned *Design Outcomes* that realize this functional requirement, and all the *Non-Functional Requirements* that the *Design Outcomes* can satisfy. The snapshot of this scenario through semantic query is shown in Fig. 4.3. The top part of this figure is the editing box for semantic query input, and the lower part shows the query results.

As described in the example, the architect first extracts all the *Design Outcomes* that are used to realize the existing functional requirement *Change User Access*, and then queries all the *Non-Functional Requirements* that are satisfied by these *Design Outcomes*, in order to evaluate whether these *Design Outcomes* can be

³Resources in NIHR Portal project refer to all the information maintained by the Portal, e.g., sources of funding for different types of research.

Semantic search

The screenshot shows a semantic query interface with two main input fields:

- Query**: Contains the query `[[Category:Design Outcome]] [[realizes::Change User Access]]`. Below it, the result **Functional Requirement** is shown.
- Additional data to display (add one property name per line)**: Contains the query `?satisfies [[Category:Non-Functional Requirement]]`. Below it, the results **Integration Requirement** and **Interoperability Requirement** are shown.

Below the queries are buttons for **Find results**, **Hide query**, **Show embed code**, and **Querying help**.

At the top right, there are navigation links: **Previous**, **Results 1–2**, **Next**, and a dropdown for page size: **(20 | 50 | 100 | 250 | 500)**.

Below the results, there are two sections with checkboxes:

- Design Outcome** (checkbox checked): **REST Structure** and **SOA Structure**. The **REST Structure** item is highlighted with a red box.
- Satisfies Non-Functional Requirement** (checkbox checked): **Integration Requirement** and **Interoperability Requirement**. The **Integration Requirement** item is highlighted with a red box.

Fig. 4.3 Scenario 1 through semantic query interface in SE-Wiki

reused or not for the implementation of the new functional requirement *Track Usage*. This query is composed of two parts: the query input in the upper left of Fig. 4.3 `[[Category:Design Outcome]] [[realizes::Change User Access]]` extracts all the *Design Outcomes* that realize *Change User Access* requirement, i.e., *REST Structure* and *SOA Structure*, which are directly related with *Change User Access* requirement; the query input in the upper right `?satisfies [[Category:Non-Functional Requirement]]` returns all the *Non-Functional Requirements*, i.e., *Integration Requirement* and *Interoperability Requirement*, which are indirectly related with *Change User Access* requirement through the *Design Outcomes*.

With all the *Non-Functional Requirements* and their associated *Design Outcomes* related to *Change User Access* requirement, which are all shown in one wiki page, the architect can have a whole view of the implementation context of the new functional requirement *Track Usage*, and assess the compatibility of these *Non-Functional Requirements* with the *Non-Functional Requirements* related to the new functional requirement. With this information, the architect will decide whether or not to reuse these *Design Outcomes* for the implementation of the new functional requirement *Track Usage*.

When new *Design Outcomes* are added to realize a requirement, in this case the requirement *Change User Access*, the semantic query will return the latest results (i.e., updated *Design Outcomes* realizing *Change User Access*). This allows SE-Wiki to support dynamic changes to requirements and architecture design which normal wikis cannot achieve with static trace links.

Under the current ontology definition, other possible software reuse scenarios can be supported by SE-Wiki, some of them are:

- Find all components that support a particular kind of quality requirements, and satisfy some quality requirements thresholds.
- Find all components that are influenced by two specific quality requirements simultaneously.

- Find the architecture design and all the components within it that support an application sub-system.
- Trace all components that are influenced by a design decision to assess if the components are reusable when the decision changes.

4.5.2 Scenario 2 Changing Requirement

Description: An architect wants to update an architecture design according to a changing requirement (see Sect. 4.3.2).

Example: A functional requirement *Change User Access: The Portal tool should be able to change user's access rights to resources.* is changed into *Change User Access: The Portal tool should only allow System Administrator to change user's access rights to resources.* Accordingly, the design based on this requirement should be updated as well. To achieve this, the architect should make sure that this changing requirement has no conflict with related existing requirements, and understand the context of this requirement before updating the design. The architect first extracts all the related artifacts concerning this changing requirement by navigating to the wiki page of this requirement in SE-Wiki, which records all the artifacts (e.g., requirements, architectural design decisions, and design outcomes) related to this requirement as shown in Fig. 4.4.

In this wiki page, the architect can easily evaluate those related artifacts concerning the changing requirement by navigating to their wiki pages. For example, the changing requirement *Change User Access* is related to the requirement *Track Usage: The Portal tool should be able to track usage of resources by all users.* There are two types of traces shown in this page: outgoing and incoming traces, which are both supported by the concept relationships defined in underlying ontology. Outgoing traces are recorded by *property*, e.g., *requirement ID*, *is proposed by*, etc. These outgoing traces show how this requirement relates to other artifacts, in a one-to-one or often one-to-many relationships. Incoming traces are shown in this page by in-line queries, which is another kind of semantic query feature provided by SE-Wiki as presented in Sect. 4.4.2. There are three in-line queries to show the incoming traces in Fig. 4.4, for example, the first incoming trace *Decision: Portal Personalization depends_on Change User Access* is created by in-line query `ask: [[Category:Decision]] [[depend on::Change User Access]] / ? depend on`. These incoming traces show how other artifacts relate to this requirement. The advantage of incoming traces generated by in-line queries is that the results of the in-line query shown in the wiki page will be updated dynamically according to the query results at run-time, which is most beneficial to evaluate and synchronize requirements and architecture design when both of them co-evolve simultaneously by different stakeholders, for example, when a new *Design Outcome* is made to realize the changing requirement *Change User Access*, then the incoming traces about the *Design Outcomes* that realize *Change User Access* will be updated automatically in this wiki page.

Change User Access

property

requirement description The Portal tool should be able to change user's access rights to resources.

requirement ID FR-006

is proposed by Network Manager Stakeholder

requirement is related to Track Usage Functional Requirement

Decision Depend on

Portal Personalization Change User Access

Non-Functional

Requirement

Qual is related to

Integration Requirement

Track Usage

Interoperability Requirement

Change User Access

Track Usage

Change User Access

Design Outcome Realizes

Personal Web Page

Change User Access

REST Structure

Change User Access

SOA Structure

Change User Access

Category: Functional Requirement

Fig. 4.4 Scenario 2 through in-line semantic query in SE-Wiki

In Scenario 2, the architect evaluates and finds that related requirement *Track Usage* is not affected by the change of requirement *Change User Access*. But the architect finds an issue *Access Control by Identity* caused by the changing requirement. To address this issue, a design option *Identity Management: Provide an identity management infrastructure in portal personalization management* is selected by the architect and documented as a *Decision*. A design outcome *Identity Management Component* is designed to realize the changing requirement *Change User Access*. All these updates on related artifacts are recorded in this requirement wiki page through incoming and outgoing traces as shown in Fig. 4.5.

With the information found in this page, the architect can further evaluate whether the newly-added decision *Identity Management* is compatible with other existing *Designs*, e.g., *Portal Personalization*, and whether the updated *Design Outcomes* still satisfy those related *Non-Functional Requirements*, e.g., *Integration Requirement*. The *Decisions* and *Design Outcomes* may change accordingly based on these further evaluations.

A number of other use cases that are similar to the changing requirement can also be supported by SE-Wiki:

Change User Access

<i>requirement description</i>	The Portal tool should only allow System Administrator to change user's access rights to resources.
<i>requirement ID</i>	FR-006
<i>is proposed by</i>	Network Manager
<i>requirement is related to</i>	Track Usage
<input checked="" type="checkbox"/> added Decision	<input checked="" type="checkbox"/> Depend on
Identity Management	Change User Access
Portal Personalization	Change User Access
<input checked="" type="checkbox"/> Qual is related to	
Integration Requirement	Track Usage Change User Access
Interoperability Requirement	Track Usage Change User Access
<input checked="" type="checkbox"/> added Component	<input checked="" type="checkbox"/> Realizes
Identity Management Component	Change User Access
Personal Web Page	Change User Access
REST Structure	Change User Access
SOA Structure	Change User Access
<hr/>	
Category: Functional Requirement	

Fig. 4.5 Updated results of scenario 2 through In-line semantic query in SE-Wiki

- Find all functional requirements that may be affected by the changing requirement.
- Find all non-functional requirements that have quality impacts on a functional requirement.
- Find all functional requirements that would be influenced by a change in the non-functional characteristic of a system, e.g., performance degradation.

4.5.3 Scenario 3 Design Impact Evaluation

Description: Requirements are frequently changed from one software version to the next, and an architect tries to evaluate and identify the impacts of the changing requirements on architecture design, so that requirements and architecture design are consistent.

Example: The requirement *Change User Access* is updated in the next version, i.e., *Version 1: The Portal tool should be able to change user's access rights to resources*, and *Version 2: The Portal tool should only allow System Administrator to change user's access rights to resources*. The architect extracts different versions of the requirement with the same *requirement ID* using a semantic query

(e.g., *[[Category:Requirement]][[is identified by::DC 001]]*), in which *DC 001* is the DC element to identify the version of a requirement. The architect finds the components for implementing the requirements by clicking the wiki page of the requirement in different versions. The architect then finds the other components for implementing related requirements through reasoning support (e.g., iteratively traverse all the related requirements), which is based on the reasoning rules and relationships defined on ontology. According to the information, the architect can identify the changes to the architecture design in two sequential versions of the requirement. From that she/he can evaluate the change impacts to the architecture design. A comparison of the wiki pages of requirements across two versions (left side is a latest version of the requirement *Change User Access*, and right side is a previous version of *Change User Access*, which is superseded by the latest version) is shown in Fig. 4.6. The requirement changes between versions with changed decisions and design (circled in Fig. 4.6) will be further evaluated for design impact analysis.

A number of other use cases that employ the reasoning framework can also be performed by SE-Wiki:

Change User Access

requirement description The Portal tool should only allow System Administrator to change user's access rights to resources.

requirement ID FR-006

is proposed by Network Manager

requirement is related to Track Usage

is identified by DC 002

<input checked="" type="checkbox"/> added Decision	<input checked="" type="checkbox"/> Depend on
Identity Management	Change User Access
Portal Personalization	Change User Access
	Change User Access 001

<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> Qual is related to
Integration Requirement	Track Usage Change User Access Change User Access 001
Interoperability Requirement	Track Usage Change User Access Change User Access 001

<input checked="" type="checkbox"/> added Component	<input checked="" type="checkbox"/> Realizes
Identity Management Component	Change User Access
Personal Web Page	Change User Access Change User Access 001
REST Structure	Change User Access Change User Access 001
SOA Structure	Change User Access Change User Access 001

Category: Functional Requirement

Change User Access 001

requirement description The Portal tool should be able to change user's access rights to resources.

requirement ID FR-006

is proposed by Network Manager

requirement is related to Track Usage

is identified by DC 001

<input checked="" type="checkbox"/> Depend on	<input checked="" type="checkbox"/> Qual is related to
Portal Personalization	Change User Access Change User Access 001

<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> Realizes
Integration Requirement	Track Usage Change User Access Change User Access 001
Interoperability Requirement	Track Usage Change User Access Change User Access 001

<input checked="" type="checkbox"/> Realizes	
Personal Web Page	Change User Access Change User Access 001
REST Structure	Change User Access Change User Access 001
SOA Structure	Change User Access Change User Access 001

Category: Functional Requirement

Fig. 4.6 Scenario 3 through comparison in SE-Wiki

- An architect wants to get a list of open quality requirements for which architectural decisions are needed.
- An architect wants to evaluate and detect the soundness of the software artifacts, e.g., a design decision is wanted when an architecture is used to realize a functional requirement.
- An architect can identify the architecture design components that have been changed from the previous software version.
- Analysts or architects can find the latest changes to a requirement or a design of interest.
- Analysts or architects can find changes that have been made by certain people or within a certain period of time.

4.6 Conclusions

Large-scale software development involves many people/stakeholders who develop requirements and architectural design. Often, these people are dispersed geographically, and the decisions that they make on the requirements and design evolve over time. This situation has created a knowledge communication issue that can cause conflicts and inconsistencies in requirements and design. Traceability methods based on static trace links cannot address this problem because the stakeholders often do not know what has been changed, let alone creating those trace links. Moreover, specifications and communications such as emails and meeting minutes are mostly documented in a natural language, making the search of related information difficult.

We solve this problem by providing a new method that makes use of semantic wiki technologies. We propose a general-purpose ontology that can be used to capture the relationships between requirements and architectural design. These relationships are derived from the use cases that we have identified. Semantic MediaWiki has been used to implement SE-Wiki. SE-Wiki supports a traceability metamodel and implements traceability use cases using a traceability ontology. Furthermore, SE-Wiki supports semantic annotation and traceability, and the annotated semantic wiki pages provide an information base for constructing semantic queries. This approach allows business analysts and designers to find up-to-date and relevant information in an environment of co-evolving requirements and designs.

Acknowledgments This research has been partially sponsored by the Dutch “Regeling Kenniswerkers”, project KWR09164, Stephenson: Architecture knowledge sharing practices in software product lines for print systems, the Natural Science Foundation of China (NSFC) under Grant No. 60950110352, STAND: Semantic-enabled collaboration Towards Analysis, Negotiation and Documentation on distributed requirements engineering, and NSFC under Grant No.60903034, QuASAK: Quality Assurance in Software architecting process using Architectural Knowledge.

References

1. Gotel OCZ, Finkelstein ACW (1994) An analysis of the requirements traceability problem. In: IEEE International Symposium on Requirements Engineering (RE), 94–101
2. Ramesh B, Jarke M (2001) Towards reference models for requirements traceability. *IEEE Trans Software Eng* 27(1):58–93
3. Spanoudakis G, Zisman A, Perez-Minana E, Krause P (2004) Rule-based generation of requirements traceability relations. *J Syst Softw* 72(2):105–127
4. Egyed A, Grumbacher P (2005) Supporting software understanding with automated requirements traceability. *Int J Software Engineer Knowledge Engineer* 15(5):783–810
5. Lago P, Muccini H, van Vliet H (2009) A scoped approach to traceability management. *J Syst Softw* 82(1):168–182
6. IEEE (1996) IEEE/EIA Standard – Industry Implementation of ISO/IEC 12207:1995, Information Technology – Software life cycle processes (IEEE/EIA Std 12207.0–1996)
7. IEEE (1997) IEEE/EIA Guide – Industry Implementation of ISO/IEC 12207:1995, Standard for Information Technology – Software life cycle processes – Life cycle data (IEEE/EIA Std 12207.1–1997)
8. Farenhorst R, Izaks R, Lago P, van Vliet H (2008) A just-intime architectural knowledge sharing portal. In: 7th Working IEEE/IFIP Conference on Software Architecture (WICSA), 125–134
9. Bass L, Clements P, Kazman R (2003) Software architecture in practice, 2nd edn. Addison Wesley, Boston
10. Ali-Babar M, de Boer RC, Dingsøyr T, Farenhorst R (2007) Architectural knowledge management strategies: approaches in research and industry. In: 2nd Workshop on SHAring and Reusing architectural Knowledge – Architecture, Rationale, and Design Intent (SHARK/ADI)
11. Rus I, Lindvall M (2002) Knowledge management in software engineering. *IEEE Softw* 19(3): 26–38
12. Hansen MT, Nohria N, Tierney T (1999) What's your strategy for managing knowledge? *Harv Bus Rev* 77(2):106–116
13. Robertson S, Robertson J (1999) Mastering the requirements process. Addison-Wesley, Harlow
14. Tang A, Jin Y, Han J (2007) A rationale-based architecture model for design traceability and reasoning. *J Syst Softw* 80(6):918–934
15. IBM (2010) Rational DOORS – A requirements management tool for systems and advanced IT applications. <http://www-01.ibm.com/software/awdtools/doors/>, accessed on 2010-3-20
16. IBM (2004) Rational RequisitePro - A requirements management tool. <http://www-01.ibm.com/software/awdtools/reqpro/>, accessed on 2010-3-20
17. Hayes JH, Dekhtyar A, Osborne J (2003) Improving Requirements Tracing via Information Retrieval. In: 11th IEEE International Conference on Requirements Engineering (RE), 138–147
18. Assawamekin N, Sunetnanta T, Pluemptiwiriyawej C (2009) Mupret: an ontology-driven traceability tool for multiperspective requirements artifacts. In: ACIS-ICIS, . 943–948
19. Hayes JH, Dekhtyar A, Sundaram SK (2006) Advancing candidate link generation for requirements tracing: the study of methods. *IEEE Trans Software Eng* 32(1):4–19
20. Cleland-Huang J, Chang CK, Christensen M (2003) Event-based traceability for managing evolutionary change. *IEEE Trans Software Eng* 29(9):796–810
21. Mistrk I, Grundy J, Hoek A, Whitehead J (2010) Collaborative software engineering. Springer, Berlin
22. Schaffert S, Bry F, Baumeister J, Kiesel M (2008) Semantic wikis. *IEEE Softw* 25(4):8–11
23. Liang P, Avgeriou P, Clerc V (2009) Requirements reasoning for distributed requirements analysis using semantic wiki. In: 4th IEEE International Conference on Global Software Engineering (ICGSE), 388–393
24. Louridas P (2006) Using wikis in software development. *IEEE Softw* 23(2):88–91

25. Hoenderboom B, Liang P (2009) A survey of semantic wikis for requirements engineering. SEARCH <http://www.cs.rug.nl/search/uploads/Publications/hoenderboom2009ssw.pdf>, accessed on 2010-3-20
26. Bachmann F, Merson P (2005) Experience using the web-based tool wiki for architecture documentation. Technical Note CMU, SEI-2005-TN-041
27. Lassila O, Swick R (1999) Resource Description Framework (RDF) Model and Syntax. <http://www.w3.org/TR/WD-rdfsyntax>, accessed on 2010-3-20
28. McGuinness D, van Harmelen F (2004) OWL web ontology language overview. W3C recommendation 10:2004–03
29. Aguiar A, David G (2005) Wikiwiki weaving heterogeneous software artifacts. In: international symposium on wikis, WikiSym, pp 67–74
30. Geisser M, Happel HJ, Hildenbrand T, Korthaus A, Seedorf S (2008) New applications for wikis in software engineering. In: PRIMIUM 145–160
31. Riechert T, Lohmann S (2007) Mapping cognitive models to social semantic spaces-collaborative development of project ontologies. In: 1st Conference on Social Semantic Web (CSSW) 91–98
32. Domges R, Pohl K (1998) Adapting traceability environments to project-specific needs. Commun ACM 41(12):54–62
33. Lago P, Farenhorst R, Avgeriou P, Boer R, Clerc V, Jansen A, van Vliet H (2010) The GRIFFIN collaborative virtual community for architectural knowledge management. In: Mistrik I, Grundy J, van der Hoek A, Whitehead J (eds) Collaborative software engineering. Springer, Berlin
34. Kunz W, Rittel H (1970) Issues as elements of information systems. Center for Planning and Development Research, University of California, Berkeley
35. Kruchten P (2004) An ontology of architectural design decisions in software-intensive systems. In: 2nd Groningen Workshop on Software Variability Management (SVM)
36. Noy N, McGuinness D (2001) Ontology development 101: a guide to creating your first ontology
37. Powell A, Nilsson M, Naeve A, Johnston P (2007) Dublin Core Metadata Initiative – Abstract Model. <http://dublincore.org/documents/abstract-model>, accessed on 2010-3-20
38. Krotzsch M, Vrandecic D, Volkel M (2006) Semantic Mediawiki. In: 5th International Semantic Web Conference (ISWC), 935–942
39. Prud'Hommeaux E, Seaborne A (2006) SPARQL Query Language for RDF. W3C working draft 20
40. NIHR (2006) NIHR Information Systems Portal User Requirements Specification. <http://www.nihr.ac.uk/files/pdfs/NIHR4.2%20Portal%20URS002%20v3.pdf>, accessed on 2010-3-20

Chapter 5

Understanding Architectural Elements from Requirements Traceability Networks

Inah Omoronyia, Guttorm Sindre, Stefan Biffl, and Tor Stålhane

Abstract The benefits of requirements traceability to understand architectural representations are still hard to achieve. This is because architectural knowledge usually remains implicit in the heads of the architects, except the architecture design itself. The aim of this research is to make architectural knowledge more explicit by mining homogenous and heterogeneous requirements traceability networks. This chapter investigates such networks achieved by event-based traceability and call graphs. Both traces are harvested during a software project. An evaluation study suggests the potential of this approach. Traceability networks can be used in understanding some of the resulting architectural styles based on the real time state of a software project. We also demonstrate the use of traceability networks to monitor initial system decisions and identify bottlenecks in a software project.

5.1 Introduction

In spite of substantial research progress in the areas of requirements engineering and software architectures, little attention has been paid to how to bridge the gap between the two [1]. It is essential to know how to transition from requirements to architecture and vice versa, and to understand the impact of architectural design on existing and evolving software requirements. This research focuses on investigating how requirement traceability approaches can be used to bridge this gap between software requirements and architectural representations.

When software requirements evolve, appropriate traceability mechanisms can provide an understanding and better management of the linking between requirements and associated artefacts during evolving project cycles [2]. Evolution of software requirements suggests a similar evolution in architecture because changes to software requirements normally imply updates to the different components used to achieve the system. Such component updates can trigger a change in structure of the system and the relationship of updated components with other components of the system.

Thus, the main research question is how software requirements evolution impacts on the underlying architecture of the system. This question will be addressed by investigating how traceability relations between software requirements and different components in a system reveal its architectural implications. Turner et al. [3] describe a requirement feature as “a coherent and identifiable bundle of system functionality that helps characterize the system from the user perspective.” We envisage a scenario where decisions are previously taken on the desired architecture to be used in implementing a specified feature in the system. We subsequently harvest homogenous and heterogeneous requirements traceability networks. Such traceability networks can also represent semantic graphs from which the actual architectural representation of the system can be inferred. The aim then is to compare and validate the desired architecture against the real-time inferred system architecture used to implement a desired user feature.

In the remaining part of this chapter, Sect. 5.2 first provides the background on requirements traceability and software architectures and discusses the architectural information needs of different stakeholders. Section 5.3 presents the automated requirements traceability mechanism that is used to realize our traceability networks. A system architectural inference mechanism based on extracted requirements traceability networks is explained. Section 5.4 presents an evaluation of our approach based on an implemented prototype. Section 5.5 presents related work and subsequently our conclusion and further work in Sect. 5.6.

5.2 Requirements Traceability and Software Architectures

5.2.1 *Inferring Architectural Rationale from Traceability Networks*

Software architecture seeks to represent the structure of the system to be developed. The structure is defined by components, their properties and inter-relationships [4]. As pointed out by Bass et al. [5], a project’s software architecture acts as a blueprint, serves as a vehicle for communication between stakeholders and contains a manifest of earliest design decisions. An architecture is the first artifact that can be analyzed to determine how well the quality attributes of an ongoing project are being achieved. In line with other chapters in this book, architectural characterisations range from the actual architecture of the system to its inferred or intended architecture. Such architectures do not exist in isolation, rather they are influenced by external factors such as the system requirement features and quality goals from the customer and developing organization, task breakdown structure and developer task assignment, lifecycle etc. Architectural rationale is thus a means to understand design architectures by considering the external factors that has influenced their realisation. Architectural rationale is realised as stakeholders endeavour to satisfy their architectural information needs by asking questions that have architectural

implications (see example of questions in Sect. 5.2.2). Architectural rationale is essential to access if a desired architectural plan for achieving a specified system's requirement is being realized in the tangible real-time representation of the system.

There are different viewpoints on traceability, but mostly aimed at addressing the same research problem of enhancing conformance and understanding during software development processes. Palmer [6] claims "traceability gives essential assistance in understanding the relationships that exist within and across software requirements, design and implementation." Requirements traceability enables the harmonization between the stakeholder's requirements and the artifacts produced along the software development process. Alternatively, requirements traceability is aimed at identifying and utilizing relationships between system requirements and other artefacts produced during a software project's lifecycle [7]. Typically, such artefacts include external documents, code segments, hardware components and associated stakeholders. Traceability facilitates software understanding, accountability, and validation & verification processes. These benefits of traceability have particularly been realized between explicit software artifacts, such as homogenous relationships between instances of requirement and heterogeneous relationships between requirements and code artefacts [8]. Relationships have varying degrees of relevance depending on the stakeholder involved.

The benefits of requirements traceability to software architectural representations are still little explored. This is because architectural knowledge which consists of architecture design, design decisions, assumptions and context, usually remains implicit in the minds of the architects, except the delivered architecture design itself [9].

Figure 5.1 represents our approach to investigating architectural representations from harvested traceability links. This scenario assumes that at the early phase of

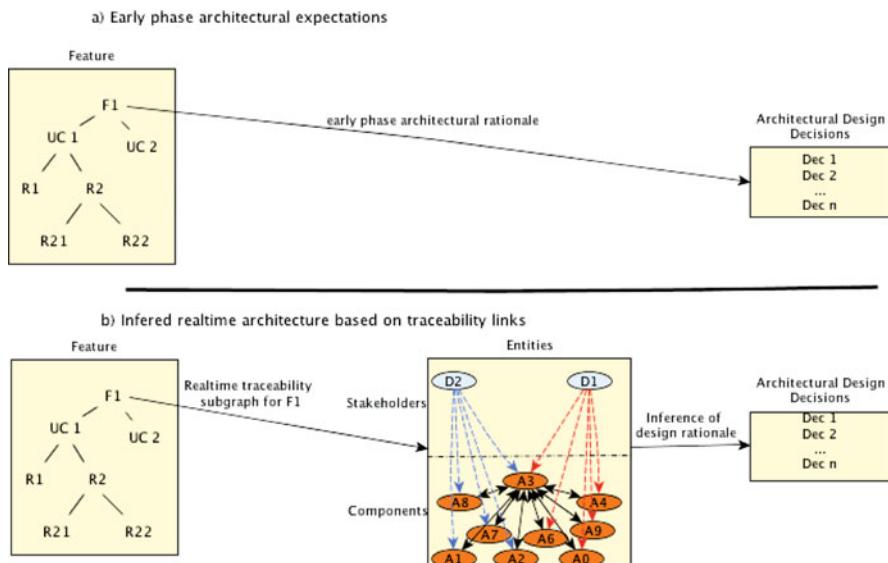


Fig. 5.1 Inferring architectural rationale from traceability links

the project, architectural decisions are made to implement specific features of the system. Such features can further generate a set of use cases and more concrete requirements that achieve the use case (Fig. 5.1a). Subsequently, different stakeholders use a set of components to achieve a specified system feature as shown in Fig. 5.1b. Thus, the main concern here is deriving some real time architectural insight based on the trace links generated between system features or use cases, components and stakeholders.

It is not straightforward to achieve architectural insight based on tangible representation of the system and harvested traceability links since different components of a system can be associated with multiple desired features of the system and in most cases worked on by different stakeholders from varying perspectives, to achieve different tasks or features. Hence, traces between system components, features and stakeholders will result in a complex web from which the core challenge is to infer an earlier guiding design rationale. The aim of this research is to reveal architectural rationale from such a real-time traceability viewpoint. This is achieved by mining homogenous and heterogeneous requirements traceability networks. In this chapter, we focus on a subset of possible architectural insights classified either as explicit or implicit. Explicit insight can be directly inferred from generated traceability networks, e.g., the architecture style revealed by real-time links between project entities. Implicit architectural insight is the additional information that can be assumed based on the interaction between the different project entities, e.g., development model, feature and requirements breakdown structure, decomposition and allocation of responsibility, assignment to processes and threads, etc.

5.2.2 Stakeholder Needs for Architectural Information

Palmer's [6] viewpoint of requirements traceability suggests that the relationships between different project entities can allow architects to show compliance with the requirements and help early identification of requirements not satisfied by the initial architecture. Hence, some information needs must be satisfied by the traceability links. It is expected that a stakeholder's needs will vary depending on role (e.g., architect/programmer/tester) and task (e.g., initial development/maintenance). As an example, consider software developer D who needs to make some changes to a software product due to a requirements change, e.g., the customers have expressed a wish for altering a certain use case. Unless the task is trivial, there are a number of questions that D might ask, for example:

- Which code artefacts (e.g., classes) are involved in implementing the use case and how do they affect the architecture of the system, e.g., a predefined architectural style or the system feature and requirements breakdown structure?
- If the class C is modified to fulfil the requirements change, what other use cases or system features might also be affected by this modification? And what

adaptation is required in the initial architecture, (e.g., the assignment of class components to processes and threads)?

- Who were the stakeholders involved in writing the use case, or the class C and other classes that are relevant for the use case? How has the desired change affected the decomposition and allocation of responsibility in the initial software architecture?

We refer to these requests as architectural information needs, defined as the traceability links between entity instances, system features or use cases, stakeholders and code artefacts. Such information is essential to understand the architecture. Ideally the traceability links required to answer such questions might have been explicitly captured during the project, e.g., which developer contributes to which artefacts, and which artefacts are related to each other? But this rarely happens – at best such traceability information is incomplete and outdated because many developers find it too time-consuming to update it.

In the remaining part of this research, we investigate traceability links harvested automatically by event based tracing and the use of call graphs. We then evaluate a number of architectural representations inferred from the harvested traceability links.

5.3 Deriving Requirements Traceability Networks for Inferring Architectural Representations

In this section, we present an automated method for harvesting a traceability network based on the scenario below:

Scenario. Bill, Amy, and Ruben are members of a team developing an online cinema ticketing system, TickX. There are two front-end use cases required: Purchase Tickets and Browse Movies. Additional use cases for system administrators are not discussed here. A number of code artefacts are being developed to realise TickX, including Ticket.java, Customer.java, Account.java, Booking.java, Movie.java, MovieCatalog.java, and Cinema.java.

While Amy and Bill have been collaborating to implement the Purchase Tickets use case, Ruben has been responsible for the Browse Movies use case. The following interaction trails were observed:

- While Amy was collaborating on Purchase Tickets she created and updated the Account.java and Customer.java code artefacts. She viewed and updated Booking.java a number of times. She also viewed MovieCatalog.java and Cinema.java.
- In the initial phase of Bill's collaboration on the Purchase Tickets use case, he viewed Account.java and MovieCatalog.java. Then he created and updated Ticket.java and Booking.java.

- Ruben’s implementation of the Browse Movies use case involved the creation and further updating of MovieCatalog.java, Cinema.java, and Movie.java. Ruben also viewed Ticket.java a number of times.

Traceability links can be homogenous, e.g., a code component being related to another code component, or heterogeneous, e.g., a relationship between a developer and code component, or a use case and component. The detailed interaction event trails is as shown in Fig. 5.2. Any selected time-point corresponds to at least one event associated with a use case, a developer, and a code artefact. For instance, at time-point 1, a create event associated with Account.java was executed by Amy while working on the Purchase Tickets use case. Similarly, time-point 7 has two events: Ruben updated Cinema.java (absolute update delta 50 [magnitude of the update based on character difference]) while working on Browse Movies, and Bill viewed Account.java as he worked on Purchase Tickets.

In this scenario, the Purchase Tickets use case is associated with Bill, Amy and a number of code artefacts. Also, MovieCatalog.java is associated with the three developers as well as the two use cases. On the whole, within such a rather small and seemingly uncomplicated scenario involving only two use cases, three developers and eight code artefacts, 27 different traceability links can be identified. To make sense of such number of dependencies, they must be ranked for relevance. For instance, the relevance of traceability links between a use case and developer is dependent on the number of interaction events generated over time by the developer in achieving the use case. A relevance measure of trace links between two entities is non-symmetric. This is because the relevance measure is firstly dependent on the number of other entity instances a selected entity can be traced to, and secondly the amount of interaction events generated as a result of each trace link.

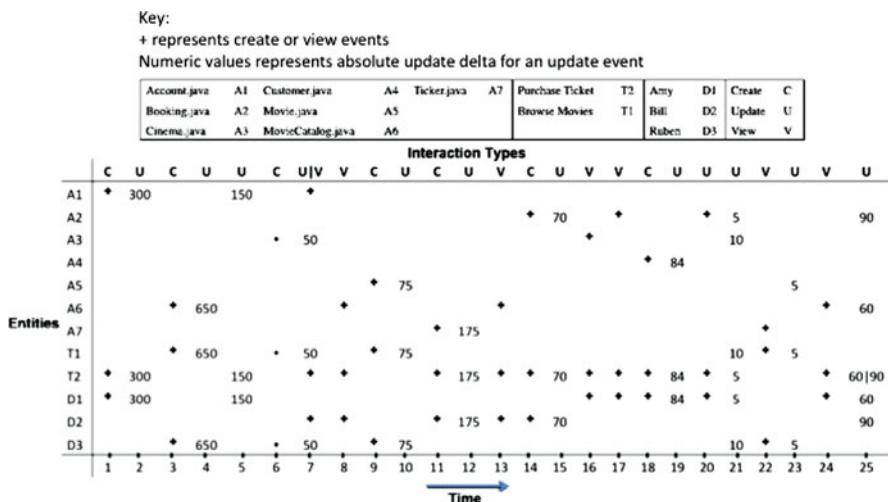


Fig. 5.2 Monitored interaction trails used to achieve TickX across 25 time-points

This demonstrated scenario poses a number of questions. Firstly, what are the possible automated methods for harvesting a traceability network? Secondly, how can the system's architectural representations be revealed by these networks? In addressing the first question, this research investigates an event based mechanism for retrieving interaction events for the subsequent generation of traceability networks. This involves capturing navigation and change events that are generated by the developers. The advantage of this approach is the opportunity to automatically harvest real-time trace links. The event based approach also provides a basis for inferring real-time architectural representations. Some ideas for such an approach has been presented in earlier work [10]. In this section, we present an event based linear mechanism to generate traceability links and rank their relevance. Since event based approaches are sometimes prone to generating false positives, we also use call graphs to validate the event based networks.

5.3.1 Event-Based Mechanism for Capturing Trace Links

In a previous paper [2], we proposed an automated event-based process for harvesting requirement traceability links relating code artefacts and developers to use cases. Trace links were formed by monitoring events initiated by a developer working in the context of a use case on a code artefact. The relative importance, or relevance, of a code artefact or developer to a selected use case was based on the type and frequency of developer actions, e.g., create, edit or view; and on the entity's *sphere of influence* in the system, i.e., how many other entities they are associated with .

This chapter explores how the harvested requirement traceability links can be used to generate a complete traceability network for a software development project. Furthermore, it is investigated how the relative importance of trace links can be used to provide insight into the centrality of each developer, use case and code artefact to the software project as a whole. Centrality is a structural attribute of nodes within a network and provides insight into the importance, influence and prominence of that particular node. Based on the centrality of entities in traceability networks, we investigate architectural rationale that can be inferred.

The event based mechanism uses events generated within a development tool and the sphere of influence of project entities to derive requirements trace networks. Rather than monitoring the entire space of interactions that can occur, we focus on a core set of event types that influence the changing state of a software project – create, update and view. Associated with an ‘update’ is the update delta – the absolute difference in the number of characters changed or added to the code artefact before and after the event. A ‘view’ event indirectly affects the state of artefacts, possibly enhancing the understanding of a developer in order to update the same artefact or other artefact instances.

During collaboration different work contexts – associations between use case (system features), developer and artefact entities – are formed. These work contexts

are constantly changing in response to events, and entities may participate in several work contexts. Figure 5.3 shows example work contexts for Amy, Purchase Tickets, and MovieCatalog.java. In Fig. 5.3a, Amy is the entity that forms the perspective of the work context graph while the Purchase Tickets use case and the classes MovieCatalog, Account, Customer, Booking and Cinema are all the entities relevant to Amy's work context.

Weights are assigned to each interaction event type as shown in Table 5.1. The weights were derived from the study of CVS records in real development projects [11], and are in line with related work by Fritz et al. [12] that emphasized the importance of the creator of code artefacts. In addition, studies conducted by Zou and Godfrey [2] suggested the need to distinguish between random and relevant view events. Thus, viewing is weighted relatively lightly compared to creates and updates (weighted by the size of the update in terms of the absolute number of characters changed). Typically, changing one line of code is much less significant compared to rewriting an entire module.

This research assumes that the size of an entity's work context is proportional to its relative influence in the collaboration space. A use case implemented by several developers and artefacts is considered to hold more information about the state of a project than a use case associated with only a small number of developers and artefacts. This is captured by the concept of sphere of influence (SOI).

SOI is a general concept used to capture both geographic and semantic groupings, and provides a well-defined boundary for interactions [13]. SOI indicates the region over which an entity exerts some kind of relevance and is defined by its work context. The *SOI ratio* is used to represent the relative influence an entity has on the collaboration space. The SOI ratio of an entity is defined as the

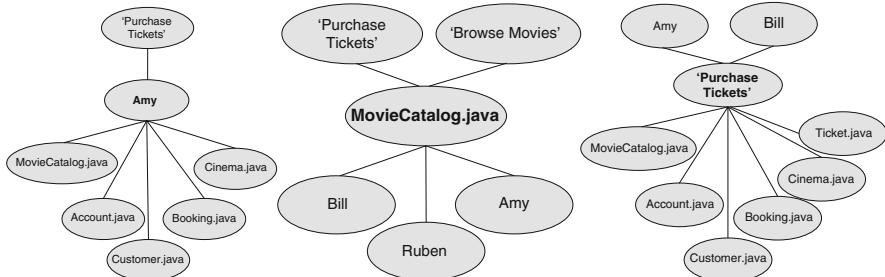


Fig. 5.3 Work context graphs

Table 5.1 Interaction type weightings

Interaction type	View	Update	Create
Weighting factor	0.001	$0.0001 * \Delta$	0.01

Δ , absolute update delta

number of unique entity instances directly associated with it divided by the number of unique entity instances in the whole collaboration space. For the motivating example the SOI ratio of Amy is 6/9 (entities in Amy's work context/total number of entities – two use cases and seven classes).

The concepts of interaction events combined with SOI ratio forms the basis for deriving trace networks with semantic insight on centrality of involved entity instances. Figure 5.3 shows three directed graphs. In general, a graph G has a set of nodes $E = \{e_1, e_2, \dots, e_n\}$ and a set of arcs $L = \{l_1, l_2, \dots, l_m\}$ which are ordered pairs of distinct entities $l_k = \langle e_i, e_j \rangle$. The arc $\langle e_i, e_j \rangle$ is directed from e_i to e_j . Thus, $\langle e_i, e_j \rangle \neq \langle e_j, e_i \rangle$. In our usage, the graphs are three-partite since their entities E can be partitioned into three subsets E_c, E_d and E_a (use cases, developers and code artefacts). All arcs connect entities in different subsets.

The weight attribute of each arc is specified by the accumulative linear combination of weights gained as a result of events associated with that arc and the sphere of influence of the entity that forms the perspective of work context. More formally, the cumulative weight x associated with an arc $\langle e_i, e_j \rangle$ in response to an event is given by (5.1), where t is the type of event (possible values shown in table 1), s the SOI ratio of e_i , and n the total number of interactions associated with the arc $\langle e_i, e_j \rangle$. Thus, the weight attributed for the arc $\langle e_i, e_j \rangle$ after n interactions is based upon its previous value plus the value of the last interaction multiplied by the SOI ratio of e_i .

$$x_{(n)\langle e_i, e_j \rangle} = x_{(n-1)\langle e_i, e_j \rangle} + t_{(n)\langle e_i, e_j \rangle} s_{(n)e_i} \quad (5.1)$$

As a further illustration of how to use events generated while developing TickX as shown in Fig. 5.2, time-point 8 represents a view event carried out by Bill while working on the Purchase Tickets use case using MovieCatalog.java. Subsequent to this, time-points 1, 2, 5 and 7 are other events carried out by Amy and Bill using Account.java within the work context of Purchase Tickets. Thus, the SOI of Purchase Tickets at time-point 8 is 0.67 (four artefacts and developers in the Purchase Tickets work context divided by six artefacts and developers in total). The weight of the arc tracing MovieCatalog.java to the work context of Purchase Tickets at time-point 8 is 0.0007. The next event involving the use of MovieCatalog.java within Purchase Tickets is represented in time-point 13, and the weight gained as a result of this event is 0.0006 and the cumulative weight is 0.0013. By the end of the trail in time-point 25 the relation from MovieCatalog.java to Purchase Tickets work context has obtained a cumulative weight value of 0.0069.

The total number of context graphs in a software project depends on the unique number of use cases, code artefacts and developers in the project. A use case may be related to a number of code artefacts and developers, and vice versa. This produces a complex network combining the results of different work context graphs. A typical example of such a network for TickX project is shown in Fig. 5.4.

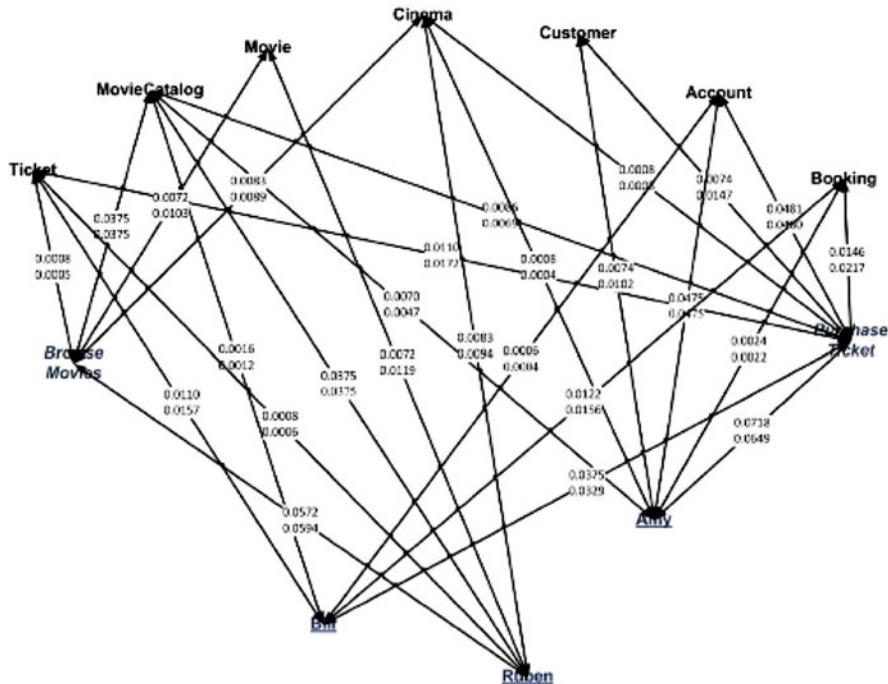


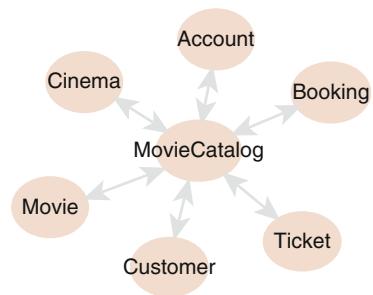
Fig. 5.4 Traceability network for TickX

5.3.2 Capturing Trace Links Between Components Via Call Graphs

The event-based approach is suitable for capturing heterogeneous links between stakeholders, use cases/features and associated code components. From an architectural viewpoint, there is also need to capture direct links between different components. Here, we investigate the use of *call graphs* to achieve homogenous traceability between software components. Call graphs are directed graphs that represent calling or message passing relationships between components of a system. From a software engineering perspective, call graphs are either dynamically generated (at execution time) or statically generated (at compile time). The core focus of this work is on the use of static call graphs generated by message passing between code components. Figure 5.5 is an example of a call graph for TickX. Figure 5.5 shows that MovieCatalog is a central component through which other components pass or receive messages.

On the whole, a requirements traceability network is a merge of homogenous and heterogeneous traceability links. Thus, a requirements traceability network is a graph of system components, use cases/desired system features and stakeholders of the system. An example of such a traceability network is shown in Fig. 5.8.

Fig. 5.5 Graphical representation of a call graph between different TickX components



5.3.3 Centrality of Entities in Traceability Networks

In network analysis, centrality indices are normally used to convey the intuitive feeling that in most networks some vertices or edges are more central than others [14, 15]. A centrality index which suits the requirements traceability networks definition is the Markov centrality, which can be applied to directed and weighted graphs. To obtain the centrality of entities in this research, the weighted requirements traceability network shown in Fig. 5.4 is viewed as a Markov chain. White and Smyth [16] described a Markov chain as a single ‘token’ traversing a graph in a stochastic manner for an infinitely long time, and the next node (state) that the token moves to is a stochastic function of the properties of the current node. They also interpreted the fraction of time (sojourn time) that the token spends at any single node as being proportional to an estimate of the global importance or centrality of the node relative to all other nodes in the graph. From the viewpoint of this research, a Markov chain enables the characterisation of a token moving from a developer to a selected use case as an indication of the relative importance of the use case instance to the developer. Similarly, a token moving from a use case instance to a code artefact indicates the importance of the artefact instance in achieving the use case.

Centrality is calculated by deriving a transition matrix from the weighted requirements traceability network, assuming that the likelihood of a token traversal between two nodes is proportional to the weight associated with the arc linking the nodes. The weights in a traceability network are then converted to transition probability weights by normalising the weights on arcs associating entities with a work context to one. Thus, transition probability is dependent on each arc weight value and the total number of entities within a work context. Figure 5.6 gives the transition matrix for TickX. The transition probability of a token from Ticket to Browse Movies use case is 0.0339 while the reverse probability is 0.0044. Each of the rows in the transition matrix sums to one. The algorithm and computational processes for the derivation of transition matrix and the subsequent centrality of entities was carried out using the Java network/graph framework (JUNG) [17].

Figure 5.8 shows a graph for TickX where the size of each entity is proportional to its Markov centrality. This figure shows the relatively higher centrality that MovieCatalog.java has achieved in the collaboration space.

	Amy	Bill	Ruben	Purchase ticket	Browse movies	Movie	MovieCatalog	Booking	Cinema	Account	Ticket	Customer
Amy	0	0	0	0.5000	0	0	0.0359	0.0167	0.0023	0.3658	0	0.0787
Bill	0	0	0	0.5000	0	0	0.0178	0.2374	0	0.0061	0.2387	0
Ruben	0	0	0	0	0.5000	0.1005	0.3154	0	0.0794	0	0.0047	0
Purchase ticket	0.3284	0.1716	0	0	0	0	0.0315	0.0993	0.0036	0.2196	0.0786	0.0673
Browse movies	0	0	0.5000	0	0	0.0897	0.3281	0	0.0773	0	0.0044	0
Movie	0	0	0.5000	0	0.5000	0	0	0	0	0	0	0
MovieCatalog	0.0759	0.0174	0.4067	0.0933	0.4067	0	0	0	0	0	0	0
Booking	0.0822	0.4178	0	0.5000	0	0	0	0	0	0	0	0
Cinema	0.0440	0	0.4560	0.0440	0.4560	0	0	0	0	0	0	0
Account	0.4938	0.0062	0	0.5000	0	0	0	0	0	0	0	0
Ticket	0	0.4661	0.0339	0.4661	0.0339	0	0	0	0	0	0	0
Customer	0.5000	0	0	0.5000	0	0	0	0	0	0	0	0

Fig. 5.6 Transition matrix for TickX requirements traceability network

5.3.4 Model Implementation

The implementation of a prototype envisages a scenario where the requirement analysts can specify the use cases or features in a shared collaboration space. These use cases can be updated or removed over the life time of the project and new ones can be added. Developers are then able to select any use case they are interested in implementing. Finally, the traceability model is achieved as the use case selected is automatically traced to every update, create and view event that the developer carried out on code artefacts while implementing that use case.

The requirements traceability approach in this chapter has been implemented as a client server architecture, where the Eclipse IDE for each developer is a client and the model processing logic and storage of event data is performed on the server. The client server approach models a shared collaboration space. The client monitors view, update and create events executed within Eclipse. When a network connection exists, event data are offloaded to the server. While there is no connection (or a slow connection) the client will temporarily store event data locally and perform local model processing logic to give the developer a partial view of current trace links and their relative centrality – offline mode. The architecture is distributed across client and server ends, and consists of four core layers: the model, event, messaging and Rich Client Platform (RCP). The client end of each layer is plugged into the Eclipse platform while the server end resides on an Apache Tomcat web application server.

The model layer is the main event processing unit in the architecture. This layer is responsible for the formation of entity work contexts and their related SOI ratios, and also generates the centrality values for entities associated with monitored trace links. The model layer also generates a call graph by parsing the abstract syntax tree representing a java component in Eclipse IDE. The event layer is responsible for capturing and archiving interaction event sequences generated within a software

project. The log.event component is the clearing centre and data warehouse of all events generated by the project collaborators. The messaging layer carries out asynchronous processing of request/response messages from the server. The offline emulator component emulates the server end functions of the model and event layers while a developer is generating interaction events in the offline mode. Finally, the RCP layer resides only on the client end, and provides the minimal set of components required to build a rich client application in Eclipse.

Figure 5.7 shows a snapshot of an Eclipse view of the visualisation.rpc component. System developers can open, activate and deactivate their use cases of interest by using the popup menu labelled 7 in Fig. 5.7. All events generated by the developer are traced to the work context of an activated use case. The RCP layer is also responsible for generating visualisations of requirements traceability networks of developers, artefacts and use cases. A system developer using the button labelled 3 in Fig. 5.7 triggers the generation of the traceability network shown in Fig. 5.8. The size of each node corresponds to its centrality in the traceability network. A selected node in the network can be moved around within the visual interface to enhance clarity of trace relations for increasingly complex trace networks.

The workflow requires that each time a developer wants to carry out a coding activity, they log in and activate an existing use case located in the central repository or create a new one. For each client workstation, only one use case can be active at a selected time, working on another use case requires that the developer activates the new use case which automatically deactivates the previous one. Similarly, the active code artefact is the current artefact being viewed, updated or created. Switching to another artefact automatically deactivates the previous artefact. This workflow enables cross cutting relations amongst artefacts, developers and use cases since, over their lifetime, and as they are used to achieve different aspects of a project, each can be associated with any number of other instances.

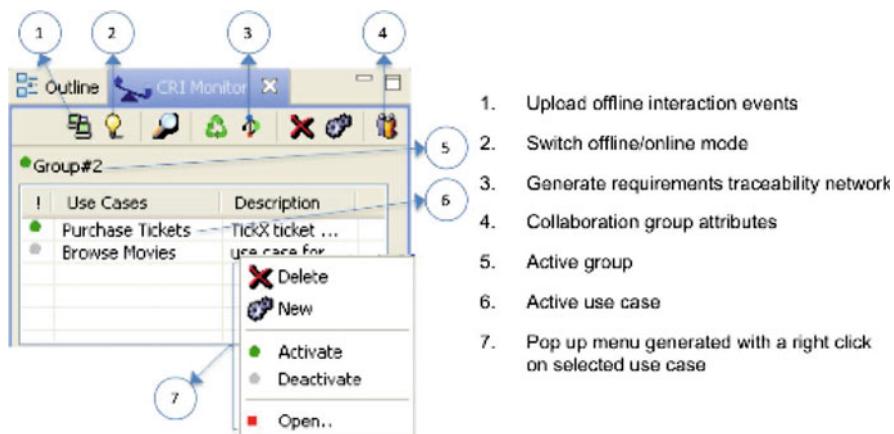


Fig. 5.7 Snapshot of eclipse view of visualization components

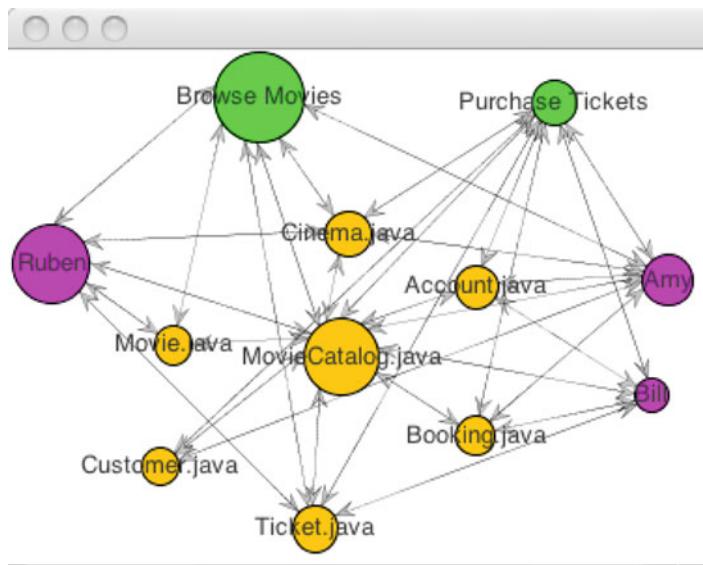


Fig. 5.8 Trace graph for TickX

As events generated by the developer are traced to the work context of an active use case and artefact on the server, the centrality value of each entity instance involved in the traceability network is recalculated.

5.4 Preliminary Study: Inferring Architectural Representations from Traceability Networks

In this section, we aim to provide possible avenues to addressing the questions on architectural information needs discussed in Sect. 5.2.2. We achieve this by discussing insights from the use of traceability networks to infer architectural representations. The discussion is based on a repository of event based requirements traceability networks and call graphs generated during a six weeks study involving ten software engineering students. The students were in the third year of their Masters/Honours programme. All participants had at least 2.5 years of object-oriented development experience using Java. All were participating in project developing ‘Gizmoball’ – an editor and simulator for a pinball table – working in groups of three [11]. During the study, use cases and system features were modelled and tagged with meaningful short form descriptions or acronyms that were easy to understand by the collaborators. Furthermore, to minimize intrusion and closely mimic real collaboration scenarios, use cases and system features were defined by developers and subsequently used as a basis for tasks assignment. At the end of the 6 weeks, structured interviews were conducted with

eight of the participants (the two remaining participants were unavoidably absent). The interviews were personalised based on the use cases/system features and code artefacts that the participant had worked on. All data were anonymized for analysis and presentation. Feedback from participants suggested that the tool captured between 60–90% of the interaction events carried out over the study period. The remaining part of this section first presents how traceability networks are used to provide insight on architectural styles, then how they help validate initial system decision and identify potentially overloaded components, critical bottlenecks and information centres with ensuing architectural implications.

5.4.1 Understanding Architectural Style

Our expectation is that layouts of architectural styles are unfolded and realised with the accumulation of trace events generated by stakeholders. Thus, if traceability networks harvested from events associated with the achievement of system features and desired requirements is realised, then it is also possible to infer the architectural style used to realize the specified feature or system requirement.

Insights were obtained from our initial study on the inference of architectural styles from event based traceability networks. Figure 5.9 demonstrates a traceability network for the feature ‘File Demo’ in Gizmoball (a feature requirement that users should be able to load gizmos from file). The figure shows the different code artefacts that the developer ‘Tony’ used to realize the desired feature and the trace

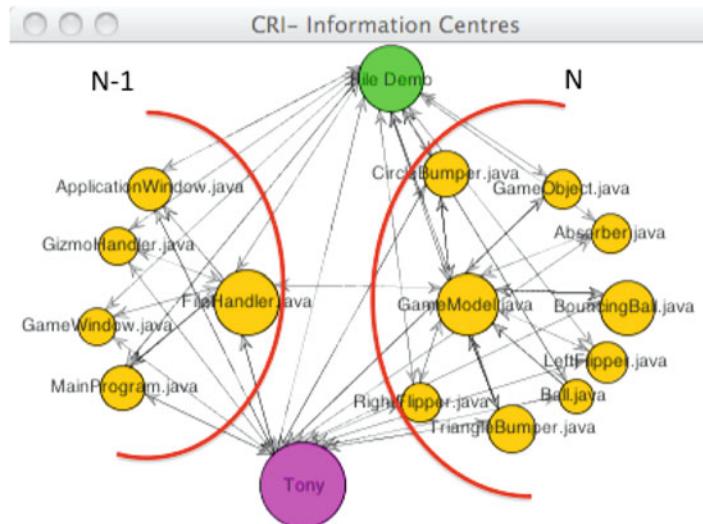


Fig. 5.9 Revealed architectural styles associated in the achievement of gizmoball feature – ‘File Demo’

links between the artefacts. A visual arrangement and repositioning of the artefacts in the traceability network reveals that a 2-tier architectural style is being used by Tony to achieve ‘File Demo.’ Furthermore, each of the tiers reveals a possible blackboard approach. This example demonstrates how different architectural styles can be combined to achieve a specified system feature.

It is important to note that we do not claim here that the discovery and combination of architectural styles is trivial. While some styles such as n-tier or batch sequential are more easily recognised from visualisation of traceability networks, other styles such as the blackboard requires more investigation. Also, call graphs in non trivial cases does not provide the information needed to infer styles. An example is in cases where communication between the clients of a blackboard and the blackboard could be via data sharing, middleware, or network communication. Secondly, the traceability network in Fig. 5.9 demonstrates the pivotal role displayed by the artefacts FileHandler and GameModel in realising the architectural style associated with File Demo. The two artefacts are responsible for the linking of the two different blackboard styles to reveal a 2-tier architectural style. This becomes obvious due to our use of different node sizes based on centrality, thus demonstrating the advantage of this visualization.. Furthermore, for every new link amongst artefacts that is subsequently introduced by collaborators to the network, the trace network reveals corresponding adaptation that is required in the initial architectural rationale for the associated feature of the system.

This study also reveals that traceability networks for non-trivial projects can be overwhelming with hundreds or thousands of components. The implied architectural style used to achieve ‘File Demo’ was revealed by a simple manual visual rearrangement of existing nodes in the network. To give support for bigger projects, further work is needed to focus on the automatic machine learning of architectural styles based on a given traceability network.

5.4.2 Monitoring Initial System Decision and Identifying Critical Pointers

One of the important lessons learned from the repository of event-based traceability networks during the 6 weeks study, is related to information that can be derived from an entity’s centrality measures. An entity’s centrality is useful in revealing a number of latent properties in the trace relation between requirements, code components and the underlying system/software architecture. For instance, a high centrality measure for a developer may suggest that they are working with many parts of the system. Such high centrality for developers can further suggest that the components and system features they are working on are crucial to achieving the system and hence are central to the development process.

The study showed that stakeholders built a perception of their expected centrality measures for entities in the trace network. These expectations are envisaged

based on previous decisions made on achieving the system. Such expectations are then used to monitor the state of the system. An example is the traceability network shown in Fig. 5.10 and involving collaboration between Greg, Boris and Blair to achieve Gizmoball. Two project stages were identified – ‘From Demo to Final’ (Translate game demo to final mode), and *JUnit Tests* (generate test cases for each gizmo object). Forty five artefacts were identified as being used to achieve these use cases. While the major responsibility of achieving ‘From Demo to Final’ was assigned to Boris, the responsibility for *JUnit Tests* was mainly assigned to Blair. A snippet from Boris demonstrating insight he obtained while navigating the traceability network generated as a result of their collaboration (Fig. 5.10) is shown below:

Boris: ... If we have done ‘JUnit Test’ how come it only relates to *Gizmo.java*, *Square.java* and *GizmoModel.java*...? Because I know that it should be looking at virtually all of the code.there is more work to be done in ‘JUnit Tests’

This feedback suggests that Boris was expecting *JUnit Tests* to have a higher centrality in the network. He also expected the use case to be related to more code artefacts. This is because they had decided to use test-driven development and as such needed every code artefact to be assigned a test case. While they had agreed and documented their decision on test driven development in their previous group meeting, the traceability network of the current state of the project rather suggested that there was still much work to be done to achieve their agreed objective.

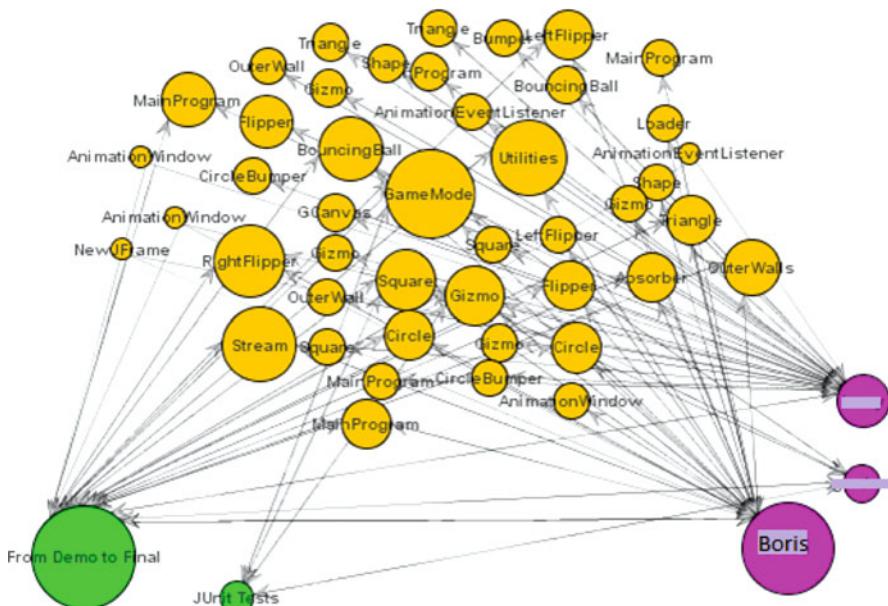


Fig. 5.10 Requirements traceability network with 44 code artefacts

If most developers tend to be associated with a high and equal measure of centrality, then it might imply a shared code ownership development model such as extreme programming. In this case, the architectural design rationale associated with extreme programming practices can then be assumed. This scenario is demonstrated in the case where the centralities of Tony, Alex and Luke in relation to the use case ‘Build Mode’ were closely similar. Transcripts from the interview session confirmed that the three collaborators all worked together in an interchanging pair fashion to realise the ‘Build Mode’ feature of Gizmoball.

The initial study also showed that the requirements traceability network helped to reveal issues that developers would easily have overlooked. For instance, interview transcripts from the collaboration between Luke, Alex and Tony to achieve the Gizmoball project suggested that they used the graph to visualize where the bigger challenges in the system were. The centrality of nodes in traceability networks were also used by the group to get a grasp of which use case or system feature had changed more considerably recently or over the lifetime of the project. Finally, it can be expected that if a requirements use case or system feature has a high centrality relative to other use cases, then this can indicate its importance to the development process. On the other hand it might indicate poor architectural design and use case definition/allocation practice -for instance, the use case has not been broken down enough or the architecture has not been well segmented. Figure 5.11 demonstrates an example of poor segmentation and allocation of components to system features. The feature ‘User Interface’ clearly attained a higher centrality measure relative to other system features. Further insight on the artefacts associated with the identified system feature revealed that it was associated with components necessary for realizing build mode (configuration of gizmos) and play mode (running of gizmos), which are the two main interfaces through which a user can interact with the gizmoball game. This suggests that the ‘User Interface’ feature could more appropriately be further decomposed into two other system features.

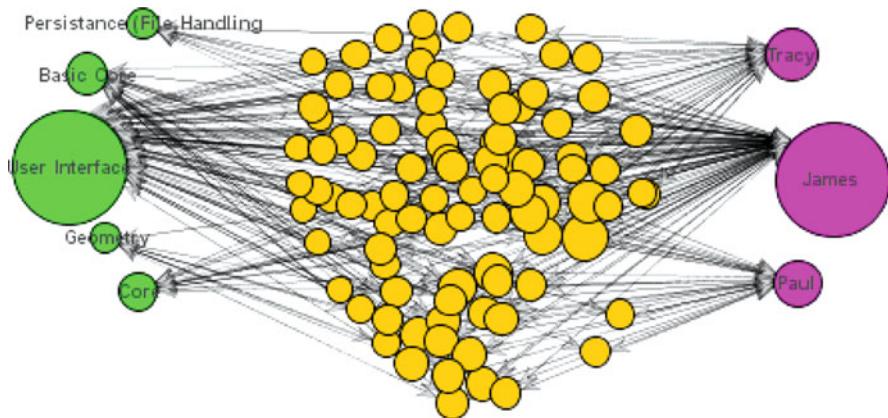


Fig. 5.11 Requirements traceability network involving 92 code artefacts, five system features and three developers

5.4.3 *Lessons Learned and Limitations of Traceability Approach*

An advantage of our approach is that requirements traceability links are automatically harvested and constantly updated to reflect the current state of the project. Furthermore, entities that are more likely to hold greater information about the project are emphasized by their larger centrality values. The use of call graphs is essential to harvesting homogenous traceability links between software components. The focus of this work has been on the use of abstract syntax tree representing a software component to generate its static call graphs. Homogenous traceability links were harvested for top-level static function callers.

A challenge is that the traceability network becomes increasingly cluttered as the number of entities increases. Thus, while a selected entity from a traceability network could be moved around within the implementation interface for visual clarity, this was a difficult process for complex networks. To help overcome this drawback, a Fisheye visualisation based on centrality has been implemented. Fisheye view has been shown to be an efficient mechanism to enhance clarity for complex visualisations with increasing number of nodes [18]. Another challenge related to scalability is the performance overhead that arises with increasing volume of captured developer interaction events. Finally, the use of interaction patterns to make inference on system decision and identifying critical pointers is based only on the small set of participants in the study. Thus, there is need for more empirical data in subsequent studies.

An implied workflow constraint, based on the implementation of the traceability model, is that systems analysts and developers explicitly need to be working within the context of a selected system feature. This is achieved by activating the desired features or use cases within the development tool. Insight obtained from the initial study suggests that such workflow constraint can sometimes be difficult to achieve, especially when developers have strict project schedules. Feedback from our study shows that the explicit activation of a use case during development work is sometimes not a primary concern of the participant, and he/she might forget to formally carry out the use case activation processes within Eclipse IDE. Also coding on a real project would not necessarily be for a specific use case, but “utility” code needed by other modules such as generic data access or manipulation routines.

5.5 Related Work

There are some methods and guidance available that help in the development and tracing system requirements into an architecture satisfying those requirements. The work presented by Grünbacher et al. [19, 20] on CBSP (Connector, Bus, System, Property) focuses on reconciling requirements and system architectures. Grünbacher et al.’s approach has been applied to the EasyWinWin requirements negotiation

technique and the C2 architectural models. The approach taken in our work differs from Grünbacher et al. as our focus is rather on the use of requirements traceability approach to help collaborating developers understand the architectural implications of each action they perform.

A closely related work is that presented on architectural design recovery by Jakobac et al. [21–23]. The main motivation for their work is based on the frequent deviation of developers from the original architecture causing architectural erosion – *a phenomenon in which the initial architecture of an application is (arbitrarily) modified to the point where its key properties no longer hold*. The approach assumes that a given system’s implementation is available, while the architecturally relevant information either does not exist, is incomplete, or is unreliable. Jakobac et al. then used source code analysis techniques for architectural recovery from the systems implementation. Finally, architectural styles were then leveraged to identify and reconcile any mismatch between existing and recovered architectural models. A distinction of our work from Jakobac et al. approach is the associations of requirement use cases or desired system features to the subsequent tangible architectural style used to realize the feature or use case. Furthermore, our traceability links are harvested real time as the system is being realized. Harvested traces are subsequently used to provide developers with information about the revealed architecture based on the work that is currently carried out. We provide pointers to potential bottlenecks and information centres that exist as a result of an initial architectural rationale.

There are a number of other reverse engineering approaches by which the architectures of software systems can be recovered. For instance, the IBIS and Compendium originating from the work of Werner and Rittel [24], presents the capability to facilitate the management of architectural arguments. Mendonca and Kramer [25] presented an exploratory reverse engineering approach called X-ray to aid programmers in recovering architectural runtime information from a distributed system’s existing software artifacts. Also, Guo et al. [26] used static analysis to recover software architectures. Guo et al.’s. approach extracted software architecture based on program slicing and parameter analysis and dependencies between the objects based on relation partition algebra. However, these approaches do not directly focus on how such extracted architectures are related to stakeholders’ requirements of the system. Again, there are different approaches to harvesting traceability networks. This research has focused on an event based approach for automated harvesting of heterogeneous relations, and call graph to retrieve homogenous trace links between components achieving the system. Other automated mechanisms for harvesting traceability networks include the use of information retrieval mechanisms and scenario driven approach. Traceability networks generated from information retrieval techniques are based on the similarity of terms used in expressing requirements and design artefacts [27–29]. The scenario-driven approach is accomplished by observing the runtime behaviour of test scenarios. Observed behaviour is then translated into a graph structure to indicate commonalities among entities associated with the behaviour [30].

Mader et al. [24] proposed an approach for the automated update of existing traceability relations during the evolution and refinement of UML analysis and design models. The approach observes elementary changes applied to UML models, recognises the broader development activities and triggers the automated update of impacted traceability relations. The elementary change events on model elements include *add*, *delete* and *modify*. The broader development activity is also recognised using a set of rules which helps in associating an elementary change as constituent parts of intentional development activity. The key similarity between the approach in this research and Mader et al.'s approach is the focus on maintaining up-to-date post-requirement traceability relations. In addition, our approach provides a perception of the centrality of traced entities.

5.6 Conclusion and Further Work

This chapter was motivated by the potential of requirements traceability to understanding architectural representations, responding to some typical architectural information needs during a software project lifecycle. It has presented a technique for the automatic harvesting of traceability networks for inferring architectural rationale. Our technique is based on the use of event-based mechanisms to capture heterogeneous trace links, while call graphs are used to generate homogenous traceability links between components. The heterogeneous and homogenous trace links were then combined to form a unified traceability network of system components, use cases/desired system features and stakeholders (developers) of the system. The advantage of our approach is that the relative potential and architectural implications of each node in the traceability network can then be determined.

An evaluation using a prototype tool implementation has demonstrated the usefulness of our approach. Using event data captured from a student-based project carried out over 6 weeks, we demonstrated how traceability networks are used to provide insight on architectural styles. We also detail how the participants in our study used the traceability tool to understand the architectural implications of the different interaction events carried out during their project. Such architectural implications included impact of executed events on initial system decision and also identifying bottlenecks and information centres in the software project.

The focus of further work is twofold. First, we aim to investigate the accuracy of centrality values. This involves understanding the effect various tasks (e.g. maintenance, debugging, refactoring or simply forward engineering) on centrality of entities. Second, for non-trivial projects, traceability networks can be overwhelmingly complex. Thus, we aim to focus on enhancing the process of inferring architectural rationale, offering a machine learning approach to supplement manual analysis. We also plan to find ways to gain better insight from the complex traceability networks resulting from non-trivial projects.

References

1. Galster M, Eberlein A, Moussavi M (2006) Transition from requirements to architecture: a review and future perspective
2. Omoronyia I et al (2009) Use case to source code traceability: the developer navigation view point
3. Turner CR, Fuggetta A, Lavazza L, Wolf AL (1999) A conceptual basis for feature engineering. *J Syst Softw* 49(1):3–15
4. Eden AH, Kazman R (2003) Architecture, design, implementation. ICSE, Portland
5. Bass L, Clements P, Kazman R (2003) Software architecture in practice, 2nd edn. Addison Wesley, Reading
6. Palmer JD (1997) Traceability. In: Thayer RH, Dorfman M (eds) Software requirements engineering. IEEE Computer Society Press, Los Alamitos, pp 364–374
7. Ramesh B, Jarke M (2001) Toward reference models for requirements traceability. *IEEE Trans Software Eng* 27(1):58–93
8. Egyed A (2003) A scenario-driven approach to trace dependency analysis. *IEEE Trans Software Eng* 29(2):116–132
9. Kruchten P, Lago P, van Vliet H (2006) Building up and reasoning about architectural knowledge. In: Hofmeister C (ed) QoSA-Quality of software architecture. Springer, Vasteras, pp 43–58
10. Omoronyia I, Ferguson J, Roper M, Wood M (2009) Using developer activity data to enhance awareness during collaborative software development. *Comput Supported Coop Work* 18(5–6 December 2009):509–558
11. Omoronyia I (2008) Enhancing awareness during distributed software development. Ph.D. Dissertation, University of Strathclyde, Glasgow, Scotland
12. Fritz T, Murphy GC, Hill E (2007) “Does a programmer’s activity indicate knowledge of code?” in ESEC/SIGSOFT FSE 341–350
13. Gutwin C, Greenberg S, Roseman M (1996) Workspace awareness in real-time distributed groupware: framework, widgets, and evaluation. In: BCS HCI, London, UK, pp 281–298
14. Brandes U, Erlebach T (2007) Network analysis -methodological foundations – introduction, ser. Lecture notes in computer science. vol 3418. Springer-Verlag, Berlin (2005)
15. Latora V, Marchiori M (2007) A measure of centrality based on network efficiency. *New J Phys* 9:188
16. White S, Smyth P (2003) Algorithms for estimating relative importance in networks. In: Getoor L, Senator TE, Domingos P, Faloutsos C (eds) KDD. ACM, Washington, pp 266–275
17. Java universal network/graph framework. [Online] Available: <http://jung.sourceforge.net>
18. Hornbaek K, Hertzum M (2007) Untangling the usability of fisheye menus. *Acm Transactions On Computer-Human Interaction* 14: 2
19. Grünbacher P, Egyed A, Medvidovic N (2001) Reconciling software requirements and architectures: the CBSP approach. Fifth IEEE international symposium on requirements engineering (RE’01)
20. Grünbacher P, Egyed A, Medvidovic N (2000) Dimensions of concerns in requirements negotiation and architecture modelling
21. Jakobac V, Medvidovic N, Egyed A (2005) Separating architectural concerns to ease program understanding. *SIGSOFT Softw Eng Notes* 30(4):1
22. Jakobac V, Egyed A, Medvidovic N (2004) ARTISAn: an approach and tool for improving software system understanding via interactive, tailorable source code analysis, TR USC-CSE-2004-513. USC, USA
23. Medvidovic N, Egyed A, Gruenbacher P (2003) Stemming architectural erosion by coupling architectural discovery and recovery
24. Mader P, Gotel O, Philippow I (2008) Enabling automated traceability maintenance by recognizing development activities applied to models., pp 49–58

25. Mendonça N, Kramer J (2001) An approach for recovering distributed system architectures. *Automated Softw Eng* 8(3–4):311–354
26. Guo J, Liao Y, Pamula R (2006) Static analysis based software architecture recovery, computational science and its applications – ICCSA 2006
27. Antoniol G et al (2002) Recovering traceability links between code and documentation. *Softw Eng IEEE Trans* 28(10):970–983
28. Oliveto R (2008) Traceability management meets information retrieval methods: strengths and limitations. In: Proceedings of the 2008 12th European conference on software maintenance and reengineering. IEEE Computer Society, Athens, Greece
29. Lormans M, van Deursen A (2005) Reconstructing requirements coverage views from design and test using traceability recovery via LSI. In: Proceedings of the 3rd international workshop on traceability in emerging forms of software engineering, ACM, Long Beach
30. Egyed A (2006) Tailoring software traceability to value-based needs. In: Stefan Biffl AA, Boehm B, Erdogmus H, Grünbacher P (eds) Value-based software engineering, Egyed A., Springer-Verlag, pp 287–308

Part II

Tools and Techniques

Chapter 6

Tools and Techniques

P. Lago, P. Avgeriou, J. Grundy, J. Hall, and I. Mistrík

In software engineering, tools and techniques are essential for many purposes. They can provide guidance to follow a certain software development process or a selected software lifecycle model. They can support various stakeholders in validating the compliance of the development results against quality criteria spanning from technical non-functional requirements to business/organizational strategies. Finally, tools and techniques may help various types of stakeholders in codifying and retrieving the knowledge necessary for decision making throughout their development journey, hence providing reasonable confidence that the resulting software systems will execute correctly, fulfill customer requirements, and cost-effectively accommodate future changes.

Tools and techniques supporting various activities in requirements engineering [1, 2] and software architecting [3, 4] have been devised in both industry and academia. Looking at the state of the art and practice we draw two observations. First, independently from the software lifecycle phase they cover, most existing tools and techniques concentrate on delivering a satisfactory solution, this being a requirements specification (the result of engineering requirements), an architectural model (the result of architecture design), or the implemented software system (the result of coding). Each of us can certainly think of tens of examples of development environments, case tools or stand-alone applications that fall in this category. It is also true that recent research and development efforts have been and are being dedicated to supporting the reasoning process and decision-making leading to such solutions. Again, also in this case we can identify a number of tools aimed at supporting various stakeholders in the knowledge they need as input (or produce as output) to come up with the satisfactory solution mentioned above [5]. Unfortunately, we see a gap not yet filled, which is in techniques and tools supporting the actual decision-making process as such [6]. For instance, modeling notations focus on modeling the resulting software system (the chosen solution) at different levels of abstraction. They are insufficient to help practitioners to make logical decisions (choices based on logical and sound reasoning) because developers can be subject to cognitive biases [7], e.g. making decisions driven by available expertise instead of optimally solving the problem at hand. In other words, as emphasized in [7],

we need to understand how stakeholders like analysts, architects, and developers reason about software problems, how they make decisions and, even more importantly, how they should be trained and supported in logical thinking and better decision making. Understanding and supporting sound decision-making is an important topic for the research community to treat. To deliver tools and techniques that will bridge the gap between how requirements are transformed into architecture in a sound way, and how architecture feeds back requirements changes, researchers and practitioners will need to work together, leverage best practices and reasoning skills that, so far, have remained only in the heads of experts [8].

Our second observation is that many tools pose their main focus on either requirements engineering or architecting – rarely both at the same time. This further motivates the present book and highlights the timeliness of the contributions included in this part.

The chapters in this part of the book address the problem of bridging requirements and architecture. This can be addressed from two main viewpoints: either by linking requirements artifacts and architecture artifacts in an explicit manner, or by evaluating the links between the two worlds, or both. While Chaps. 7, 10, and 11 mainly take the first viewpoint, Chaps. 8 and 9 take both. Further, all chapters present innovative techniques, with Chap. 9 including tool support, too.

Chapter 7 by Lawrence Chung, Sam Supakkul, Nary Subramanian, José Luis Garrido, Manuel Noguera, María V. Hurtado, María Luisa Rodríguez, and Kawtar Benghazi presents a technique (called GOSA, Goal-Oriented Software Architecting) deriving architecture models from goal models. To this end, derivation rules guide developers in this derivation process, which includes how decisions can be mapped on architecture by applying suitable architectural styles and patterns.

Chapter 8 by Zoë Stephenson, Katrina Attwood and John McDermid propose a technique aiding designers to progress in the design while dealing with requirement uncertainties and risk management. Indicators are used to suggest possible risk areas, potential uncertainties in the requirements to be realized, and finally identify options for a lower-risk architecture solution.

Similar to Chap. 7, also Chap. 9 by Christine Choppy, Denis Hatebur and Maritta Heisel addresses the problem of deriving architectures from requirements. However, the work presented in Chap. 9 takes a broader perspective by using context models and problem frames to initiate and guide the derivation process. The presented method supports both UML-based model-driven derivation and automated validation of predefined validity conditions. The associated tool is based on a UML profile and has been implemented on the Eclipse platform.

Chapter 10 by Luciano Baresi and Liliana Pasquale focuses on the problem of controlling dynamic adaptation requirements (characterizing service-oriented systems) next to traditional functional and non-functional requirements. The presented approach uses adaptation rules to govern runtime service adaptation to changing requirements. This introduces an innovative way to look at adaptation as the instrument to bridge the gap between requirements and architecture, for the specific domain of SOA.

Last but not least, Chap. 11 by Len Bass and Paul Clements delves into business goals as well-known source of requirements but seldom captured in an explicit manner. As a consequence, system architectures end up being misaligned with such business goals, potentially leading to IT project failures. The proposed technique includes a reference classification of business goals as tool to drive elicitation, and a method to help architects in eliciting business goals, relevant associated quality requirements and architecture drivers.

References

1. Nuseibeh B, Easterbrook S (2000) Requirements engineering: a roadmap. In: Proceedings of the conference on the future of software engineering (ICSE). ACM, New York, pp 35–46
2. Zave P, Jackson M (1997) Four dark corners of requirements engineering. ACM T Softw Eng Meth 6(1):1–30
3. Garlan D, Perry DE (1995) Introduction to the special issue on software architecture. IEEE T Software Eng 21(4):269–274
4. Shaw M, Clements P (2006) The golden age of software architecture. IEEE Softw 23(2):31–39
5. Tang A, Avgeriou P, Jansen A, Capilla R, Babar MA (2010) A comparative study of architecture knowledge management tools. J Syst Softw 83:352–370, Elsevier
6. Tang A, Lago P (2010) Notes on design reasoning techniques (V1.4), TR SUTICT-TR2010.01. Swinburne University of Technology, Melbourne
7. Tang A (2011) Software designers, are you biased? International workshop on SHAring and Reusing architectural knowledge (SHARK), ICSE Companion. May 21–28, 2011, Hawaii, USA, p 8
8. Rus I, Lindvall M (2002) Guest editors' introduction: knowledge management in software engineering. IEEE Softw 19(3):26–38

Chapter 7

Goal-Oriented Software Architecting

Lawrence Chung, Sam Supakkul, Nary Subramanian, José Luis Garrido, Manuel Noguera, María V. Hurtado, María Luisa Rodríguez, and Kawtar Benghazi

Abstract Designing software architectures to meet both functional and non-functional requirements (FRs and NFRs) is difficult as it oftentimes relies on the skill and experience of the architect, and the resulting architectures are rarely derived directly from the requirements models. As a result, the relationships between the two artifacts are not explicitly captured, making it difficult to reason more precisely whether an architecture indeed meets its requirements, and if yes, why. This chapter presents a goal-oriented software architecting approach, where FRs and NFRs are treated as goals to be achieved, which are refined and used to explore achievement alternatives. The chosen alternatives and the goal model are then used to derive, using the provided mapping rules, a logical architecture, which is further mapped to a final concrete architecture by applying an architectural style and architectural patterns chosen based on the NFRs. The approach has been applied in an empirical study based on the 1992 London ambulance dispatch system.

7.1 Introduction

Software architecture (SA) defines the structure and organization by which system components interact [22] to meet both functional and non-functional requirements (FRs and NFRs). However, multiple architectures can be defined for the same FRs, each with different quality attributes. As a result, software architecting is often driven by quality or NFRs [15].

However, designing software architectures to meet both FRs and NFRs is not trivial. A number of architectural design methods have been introduced to provide guidelines for the three general architectural design activities [15]: (1) architectural analysis, an activity that identifies architectural significant requirements (ASRs); (2) architectural synthesis, an activity that designs or obtains one or more candidate architectures; and (3) architectural evaluation, an activity that evaluates and selects an architecture that best meets the ASRs.

These methods provide broad coverage of architectural design, many of which intentionally leave the modeling details to the architect on how to transition from requirements models to architectural models [1, 3]. As a result, software architecting often relies on the architect’s skill, experience, and the collaboration with other stakeholders, to arrive at one or more candidate architectures, a task that is difficult, especially for those unfamiliar with the application domain. Additionally, the resulting architecture oftentimes lacks explicit traceability with the requirements models, making it difficult to reason precisely whether an architecture indeed meets its FRs and NFRs, and if yes, why.

To alleviate this difficulty, this chapter presents a goal-oriented software architecting (GOSA) approach to systematically designing architectures from FRs and NFRs. In this approach, FRs and NFRs are treated as goals to be achieved, which are refined and used to explore achievement alternatives. The chosen alternatives and the goal model are used to derive a logical architecture using the provided mapping rules. The logical architecture is further mapped to a concrete architecture by applying an architectural style and architectural patterns chosen based on the tradeoffs concerning system NFRs. The goal model produced during this process captures the relationships between the requirements and the resulting SA, as well as the design rationale.

This approach has been applied in an empirical study based on the 1992 London ambulance dispatch system. The results showed that the approach could be useful for the architectural synthesis activity, particularly for the information systems domain.

The rest of the chapter is structured as follows. Section 7.2 presents the GOSA approach, followed by an empirical study report in Sect. 7.3. Section 7.4 discusses related work, which is followed by a summary and future direction.

7.2 The Goal-Oriented Software Architecting (GOSA) Approach

The goal-oriented software architecting (GOSA) approach is at high-level a three-step process. A goal-oriented requirements analysis is first performed to explore and capture design rationale to arrive at desirable means to achieve FRs and NFRs. The results are then used to derive a software architecture in two steps: deriving a logical architecture from FRs, then deriving a concrete architecture from the logical architecture and NFRs.

The process for the GOSA approach is shown in Fig. 7.1 for simplicity as a top-down process, but in practice it is an iterative and interleaving process where any previous step may be repeated in any order as necessary.

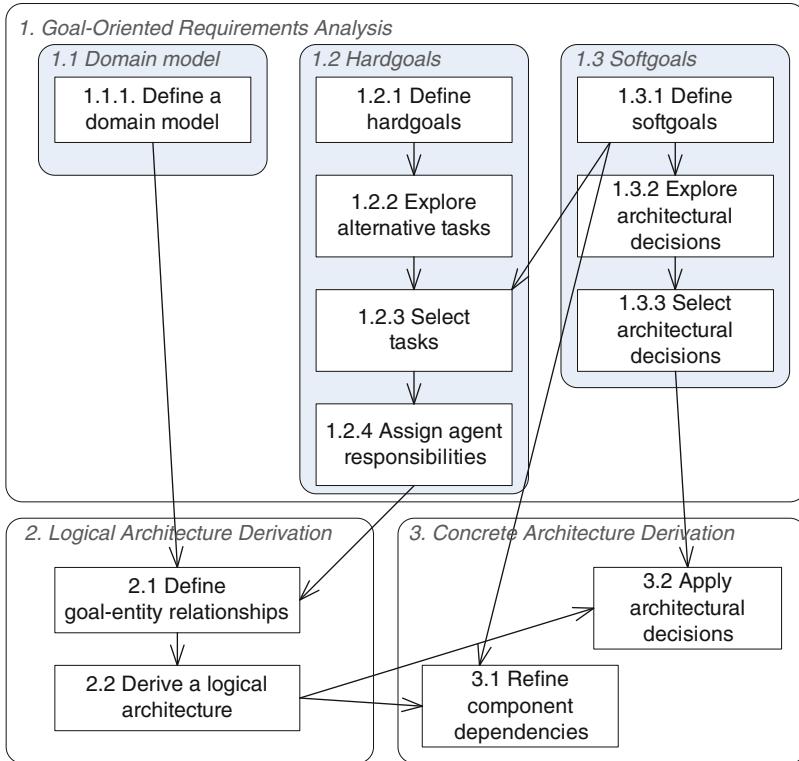


Fig. 7.1 The goal-oriented software architecting process

7.2.1 Step 1: Goal-Oriented Requirements Analysis

In this step, FRs are captured as hardgoals and NFRs as softgoals to be achieved. FRs are treated as hardgoals as they generally have clear-cut achievement criteria. For instance, a hardgoal of having an ambulance dispatched is satisfied when an ambulance has been dispatched, an absolute proposition. On the other hands, NFRs are treated as softgoals in this approach since many of them have less clear-cut definition and achievement criteria. For instance, it is difficult to precisely define security without using other NFR terms, such as confidentiality and integrity, which in turn will have to be defined. It is also difficult to determine concrete criteria for security since it may be unacceptable to define certain number compromises per a period of time while on the other hand it is impossible to guarantee that a system will never be compromised. Therefore, oftentimes, a softgoal can only be *satisfied*, referring to the notion of “good enough” satisfaction [6].

To achieve the goals, alternatives are explored, evaluated, and selected based on tradeoff analyses in consideration of NFRs that are originally posted and those identified as side-effects [6, 25, 33]. Each selected means is then assigned to be fulfilled by an external agent or the system under development [11].

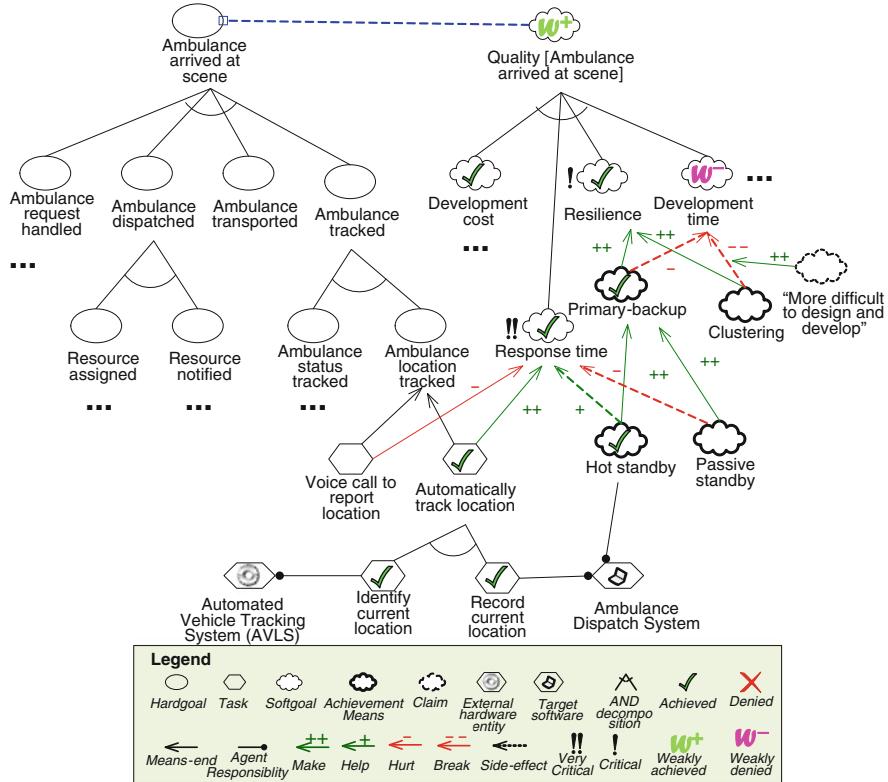


Fig. 7.2 A partial goal model for an ambulance dispatch system

Using an ambulance dispatch system as an example, Fig. 7.2 shows a partial goal model that may be produced during the goal-oriented requirements analysis step. Parts of the model are omitted for brevity where denoted by “...”.

On the left hand side of Fig. 7.2, corresponding to step 1.2.1 in Fig. 7.1, Ambulance arrived at scene is the ultimate hardgoal of the system, it is refined using an AND-decomposition to four sub-goals, including Ambulance tracked, a goal that is further AND-decomposed to Ambulance status tracked and Ambulance location tracked sub-goals. Each sub-goal can be in turn further decomposed as necessary until the goal is sufficiently refined and agreeable to the stakeholders. Corresponding to step 1.2.2, each leaf-goal is then used as a goal to explore alternative tasks needed to operationalize the goal. For instance, Ambulance location tracked goal may be achieved by either Voice call to report the location or Automatically track the location alternative tasks. Corresponding to step 1.2.3, the latter alternative is chosen for its more favorable contribution towards Response time softgoal, as its Make/++ contribution towards the goal is preferred to the Hurt/- contribution of the former alternative.

Consider the right hand side of the goal model for a moment. Corresponding to step 1.3.1, Quality [Ambulance arrived at scene] softgoal represents the overall desirable quality in the context of the FRs as represented by the root hardgoal.

Softgoals may also be defined in the context of other FRs representations, such as use cases [29]. Similar to hardgoals, each softgoal may be AND- or OR-decomposed to sub-softgoals. In this example, the root softgoal is AND-decomposed to sub-softgoals, including Response time and Resilience.

Each leaf softgoal is then used as a criterion for exploring and choosing among alternatives (step 1.3.1 and 1.3.2). For example, Response time is a criterion for choosing between Voice call to report the location and Automatically track the location tasks. Similarly, Resilience is used as a goal to explore and select between Primary-backup and Clustering architectural alternatives. Primary-backup alternative is chosen, as part of step 1.3.3, for a better side-effect towards Development time as its Hurt/- contribution is preferred to the Break/- contribution of Clustering alternative. The selected Primary-backup alternative is subsequently used as a goal to explore two more specific architectural alternatives: Hot standby and Passive standby alternatives [12], where the former is chosen for a better side-effect towards Response time.

Once achievement means have been selected, they are analyzed for impacts on the softgoals. For example, Hot standby alternative is labeled as Satisficed (denoted by a check mark) to represent the selection. The label is then propagated to a Satisficed label on the Primary-backup softgoal over a Make/++ contribution, and subsequently a Satisficed label on the Resilience softgoal. If all sub-goals of the Quality root softgoal were Satisficed, the root's label would be derived to be Satisficed over the AND-decomposition. But in this case, Development time sub-goal is Weakly Denied by the propagation of Primary-backup's Satisficed label over a Hurt/- contribution. The architect must decide whether having Development time weakly denied is acceptable. If it is, he or she may manually label the Quality softgoal as Satisficed or Weakly Satisficed as appropriate; otherwise, the architect may select different existing alternatives or explore additional alternatives until the root softgoal is deemed sufficiently satisfied.

Each selected means is then assigned to be fulfilled by an agent in the environment or the system under development [11] (step 1.2.4.). For example, Identify current location task is assigned to AVLS external hardware, while Record current location task and Hot standby architectural decision are assigned to be fulfilled by Ambulance Dispatch System, the system under development.

In addition to goal identification and refinement, domain entities are also discovered during the requirements analysis [11]. For instance, in corresponding to step 1.1.1, the notion of ambulance is discovered when Ambulance arrived at scene goal is defined. These entities are captured, refined, and related in a domain model, such as that shown in Fig. 7.3.

7.2.2 Step 2: Logical Architecture Derivation

Components in a software architecture are generally defined using either object-oriented or function-oriented approach. For instance, “incident_mgr” and

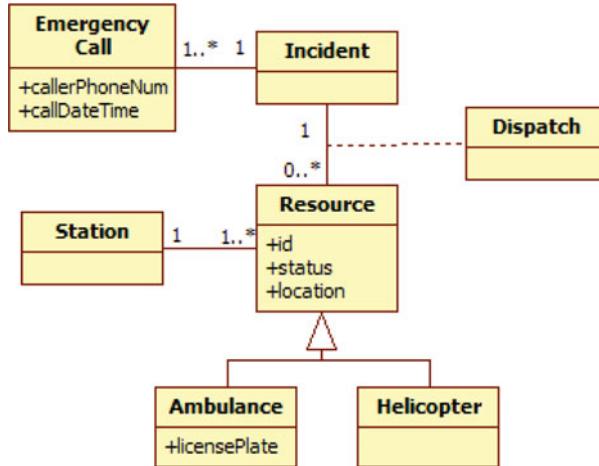


Fig. 7.3 A domain model for an ambulance dispatch system

“dispatch_mgr” [23] are examples of object-oriented components where the component names reflect key domain entities, while “Alphabetizer” and “Circular Shift” [28] are examples of function-oriented components where the component names reflect key functions.

In the GOSA approach, we take the object-oriented approach to derive components in the logical architecture. A logical architecture defines architectural components and their inter-dependencies that are derived from hardgoals representing the FRs and their relationships with domain entities.

Since the components would be too fined grain if each domain entity is assigned a component, only significant entities are assigned a component. A domain entity is considered significant if it is used or changed as a result of goal fulfillment, which is determined from the relationship between the entity and hardgoals.

7.2.2.1 Step 2.1: Define Hardgoal-Entity Relationships

A relationship between a hardgoal and a domain entity is defined in terms of the role that the goal plays in relation to the entity. Two roles are determined using the following rules:

R1: Producer Goal. A hardgoal is considered a producer goal of a domain entity if the fulfillment of the goal necessitates changes to the domain entity. This relationship is represented by a uni-directional line from the producer goal towards the domain entity.

R2: Consumer Goal. A hardgoal is considered a consumer goal of a domain entity if the fulfillment of the goal necessitates use of information from the domain

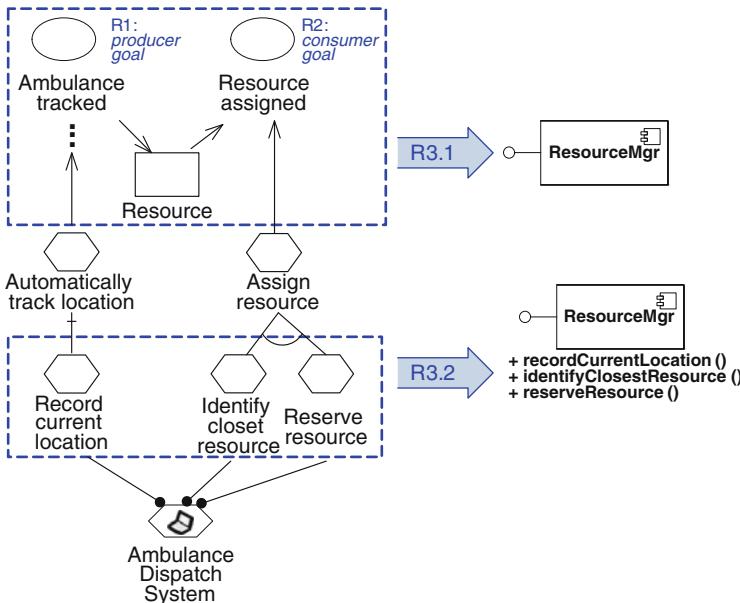


Fig. 7.4 Examples of rule R1, R2, and R3 applications

entity. This relationship is represented by a uni-directional line from the domain entity towards the consumer goal.

Examples of the goal-entity relationships are depicted in Fig. 7.4 where Ambulance location tracked is a producer goal and Resource assigned a consumer goal of Resource entity.

7.2.2.2 Step 2.2: Derive a Logical Architecture

This step defines architectural components and their inter-dependencies using the goal model and goal-entity relationships. Two kinds of components are derived: process and interface components. A process component provides services related to the corresponding entity, while an interface component handles the communication between the system and an external entity. The services provided by a component are represented by component operations. Process components and their operations are determined by the following rules:

R3: Process Component Derivation.

R3.1: Define a process component using nomenclature "`< domain-entity > Mgr`" for each domain entity that is associated with a producer and a consumer hardgoals.

R3.2: For each derived process component, define an operation for each selected task that is assigned to the system under development.

Figure 7.4 shows examples of rule R1, R2, and R3 applications. Here, ResourceMgr process component is derived because Resource entity is associated with both Ambulance tracked as a producer goal and Resource assigned as a consumer goal. Three operations, including recordCurrentLocation, identifyClosestResource, and reserveResource operations, are derived from the three tasks assigned to Ambulance Dispatch System, the system under development.

The rules for determining interface components and their operations are:

R4: Interface Component Derivation.

R4.1: Define an interface component using nomenclature “< external-agent > Interface” for each external agent that is assigned to carry out one or more tasks.

R4.2: For each derived interface component, define an operation for each task that is assigned to the agent.

Figure 7.5 shows an application of rule R4, where AVLSInterface component is defined to handle the communication with the Automated Vehicle Location System (AVLS) external hardware. The component has an operation, identifyCurrentLocation, which is derived from the only task assigned to the AVLS external agent.

Given the two kinds of components, two kinds of inter-component inter-dependencies can be derived: process-process and process-interface component dependencies. Process-process component dependencies are determined using the following rules:

R5: Process-Process Component Dependency Derivation. Define a component dependency between process component A and process component B where A is assigned to be the depender and B the dependee of the relationship if there is a hardgoal that is both a producer goal of A’s entity and B’s entity.

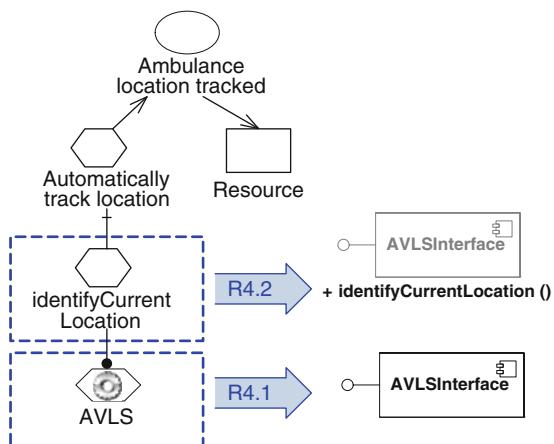


Fig. 7.5 An example of rule R4 application

Fig. 7.6 An example of rule R5 application

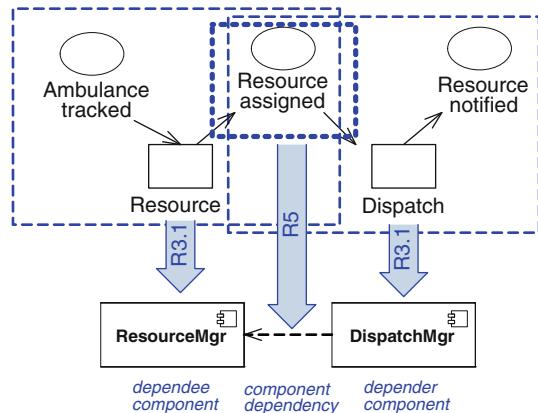


Figure 7.6 shows an application of rule R5 where ResourceMgr and DispatchMgr are derived using R3.1, and the dependency between them is derived using R5 because Resource assigned is both a producer goal of Dispatch entity and a consumer goal of Resource entity.

The rules for determining process-interface component dependencies are:

R6: Process-Interface Component Dependency Derivation. Define a component dependency between process component A and interface component B if a task of the producer goal or the consumer goal related to A is assigned to an external agent being communicated via component B.

R6.1: Process component A is assigned to be the depender of the dependency if the goal of the assigned task is a producer goal.

R6.2: Process component A is assigned to be the dependee of the dependency if the goal of the task is a consumer goal.

Figure 7.7 shows an application of rule R6. Here, a dependency is defined between ResourceMgr process component and AVLSInterface interface component as Identify current location, a task of Ambulance tracked goal is assigned to AVLS agent (rule R6). For this dependency, ResourceMgr is considered the depender since Ambulance location tracked is a producer goal of the ResourceMgr component (rule R6.1).

7.2.3 Step 3: Concrete Architecture Derivation

In this step, the logical architecture is refined using an architectural style to map component dependencies to concrete connectors and using architectural patterns to realize the architectural decisions made during the goal-oriented requirements analysis. The notions of architectural styles and patterns are often used

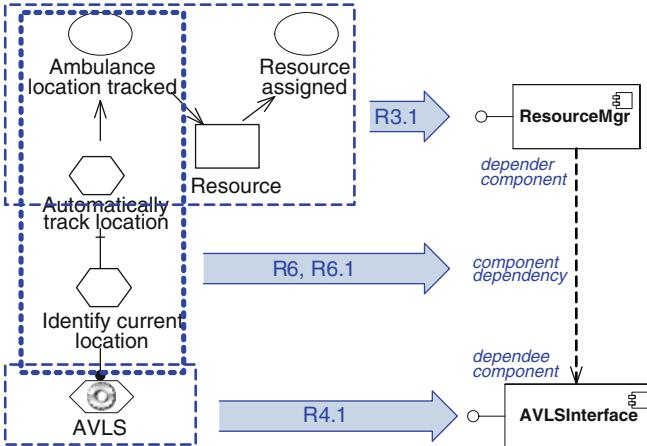


Fig. 7.7 An example of rule R6 application

interchangeably [17] to achieve NFRs [13]. But in this approach, they are used for two subtle different purposes: a style is used to affect how all components interact to achieve the desirable quality attributes [28] while a pattern is used to affect other aspects of the architecture in a less predominant manner, for instance to achieve NFRs such as concurrency and persistency [4].

7.2.3.1 Step 3.1: Refine Component Dependencies

In this step, each component dependency in the logical architecture is refined using an architectural style that is chosen from a qualitative goal-oriented tradeoff analysis. Figure 7.8 shows a goal-oriented tradeoff analysis to select among several styles, including Abstract Data Type, Shared Data, Implicit Invocation, and Pipe & Filter, in consideration of Comprehensibility, Modifiability, Performance, and Reusability softgoals [8].

The selection is made by evaluating each alternative in turn to determine whether its overall impacts on the softgoals are acceptable. An impact on a softgoal is determined by applying the label propagation procedure [7]. For example, Abstract Data Type is labeled “satisficed” (denoted by a checkmark) to represent the selection. The label is propagated to “Weakly-Satisficed” (W+) label on Time performance softgoal over the Help/+ contribution, while the same label is propagated over a Break/- – contribution to “denied” label (denoted by a cross) on Extensibility [Function]. In this example, Abstract Data Type provides the most acceptable compromises based on the weakly satisficed labels (denoted by W+) on more critical softgoals (Time performance, Modifiability [Data rep] and Modifiability [Function]), and the weakly denied labels on less critical softgoals (Modifiability [Process], Extensibility [Function], Space performance).

The selected architectural style is then applied to each component dependency in the logical architecture. Figure 7.9 shows the application of abstract data type and

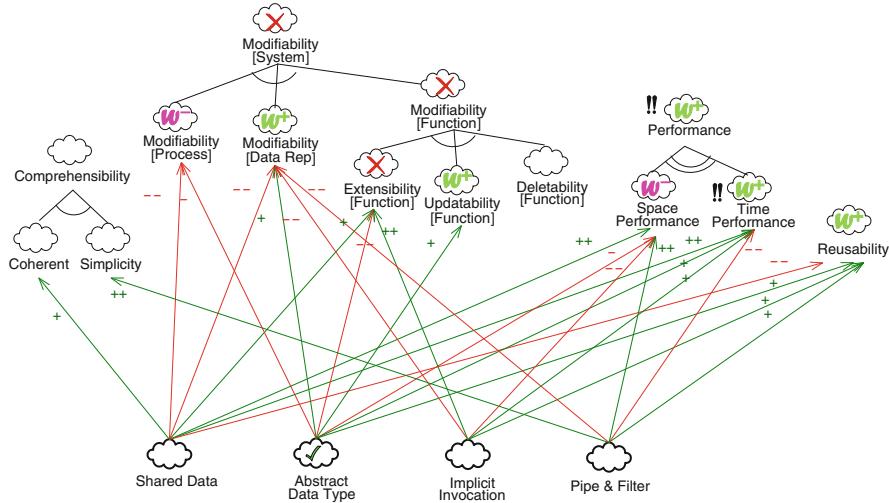


Fig. 7.8 A goal-oriented tradeoff analysis for selecting a desirable architectural style

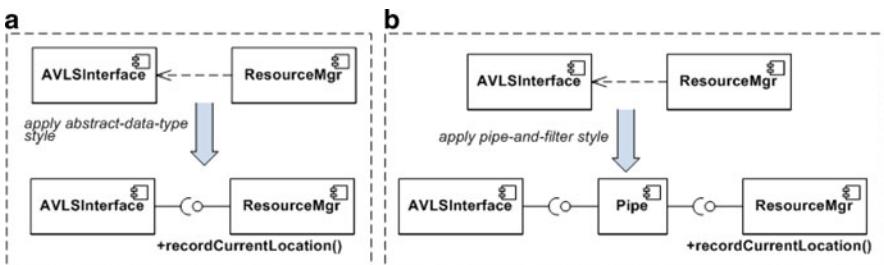


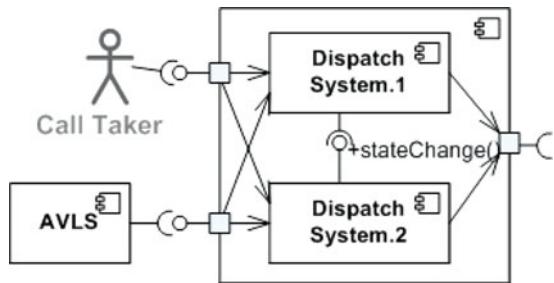
Fig. 7.9 Applying the abstract data type style (a) and the pipe & filter style (b) using UML

pipe & filter architectural styles, using adapted representations of architectural styles in UML [17].

7.2.3.2 Step 3.2: Apply Architectural Decisions

The architectural decisions made during the goal-oriented analysis are applied to the logical architecture in this step. For instance, the decision to use hot-standby backup to achieve Resilience softgoal in Fig. 7.2 may be applied by using a primary-backup architectural pattern. Figure 7.10 shows the resulting system architecture where two copies of the system (DispatchSystem.1 and DispatchSystem.2) are operational where both simultaneously receive inputs from external entities (CallTaker and ALVS). If the primary copy fails, the hot-standby copy would immediately take over as the new primary. The failed copy would then be repaired and brought back to operation and

Fig. 7.10 The system architecture after a hot-standby pattern has been applied



marked as the new hot-standby copy. If multiple patterns are considered, they may be chosen using a goal-oriented tradeoff analysis similar to that shown in Fig. 7.8.

7.3 An Empirical Study: The London Ambulance Dispatch System

The goal-oriented software architecting (GOSA) approach has been applied to the 1992 London ambulance dispatch system [26] in an empirical study, a controlled experiment performed by the authors using publicly available case reports (e.g. [26] and [20]), with the following objectives: (1) to study how the approach can be used to systematically design from requirements a reasonable candidate software architecture, and (2) to study how the approach helps capture and maintain relationships between requirements and the resulting software architecture as well as the design rationale.

7.3.1 Description of the Case

The failure of the 1992 London Ambulance Service's computer-aided dispatch system arguably caused several deaths soon after its deployment, from failing to deliver emergency care in time, including an 11-year old girl dying from a kidney condition after waiting for an ambulance for 53 min and a man dying from a heart attack after waiting for 2 h [10].

The primary FRs of the system were to support the ambulance dispatch service, including call handling, resource identification and dispatch, resource mobilization, and resource management and tracking [26]. More importantly, the primary driver for the computer-aided dispatch system was to comply with a new response time regulation that required an ambulance to be dispatched in 3 min after an emergency call is received, and arrived at the scene in 11 min, for a total of 14 min. In addition, the system was also required to meet throughput, ease-of-use, resilience, flexibility, development time and cost NFRs [26].

7.4 Results

Figure 7.11 shows the final software architecture in UML, which consists of 10 components and 10 connectors. The components are four process components whose name ending with “Mgr” suffix, and six interface components whose name ending with “Interface” suffix. The connectors are represented by 10 coupled UML “provided” and “required” interfaces: three process-to-process component connectors (those among RequestMgr, IncidentMgr, and DispatchMgr) and seven process-to-interface component connectors (those connecting CallTakerInterface, ResourceAllocatorInterface, AVLSInterface, MapServerInterface, StationPrinterInterface, and MDTInterface). The relationships between the requirements and resulting architecture are the relationships among the elements in the goal model (hardgoals, softgoals, and domain entities), the logical architecture (components, component dependencies), the concrete architecture (components and connectors), and the rules used, as described in Sect. 7.2.

7.4.1 Threats to Validity

Based on the case study research [19], threats to validity of the empirical study are discussed in this section in terms of construct validity, internal validity, external validity, and experimental reliability.

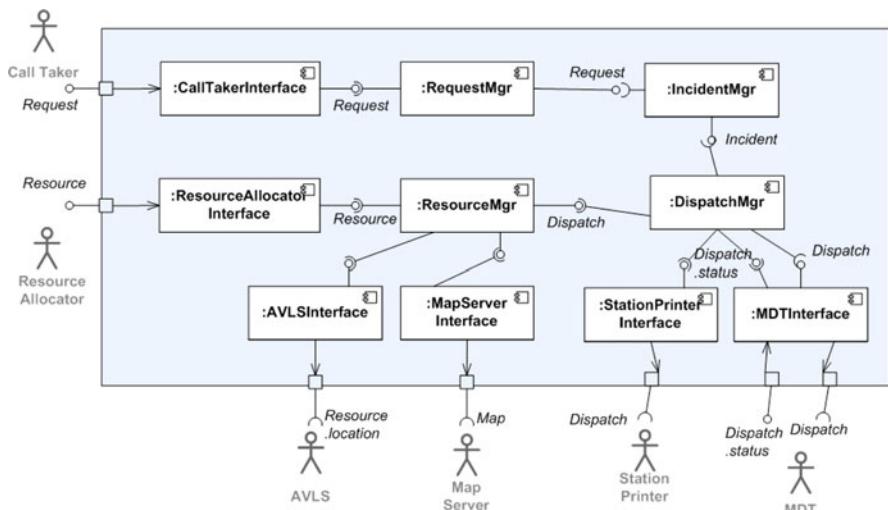


Fig. 7.11 The final software architecture for the empirical study

7.4.1.1 Construct Validity

Construct validity refers to the establishment of correct operational measures for the concepts being studied. We determined the reasonableness of a resulting architecture by comparing its overall structure with an independently designed architecture using the following criteria: (1) semantically compatible components, (2) semantically compatible connectors. The independent architecture was used as an experimental control to represent the status quo [19]. For this study, we used an architecture designed by the ACME research group for the same London ambulance dispatch system [23], where a visual representation of the architecture [34] is shown in Fig. 7.12.

A comparison between the two architectures shows several similarities. For criterion i, both architectures appear to use the object-oriented approach for defining components. Both also have similar process and interface components. For instance, CallMgr, IncidentMgr, ResourceMgr, and DispatchMgr in the GOSA architecture appear to be semantically similar to call_entry, incident_mngr, resource_mngr, and dispatcher components in the ACME architecture. Additionally, MapServer-Interface appears to be the same interface component as the map_server component. For criterion ii, both consist of several similar component connectors, including the connector between CallMgr and IncidentMgr, and that between ResourceMgr and DispatchMgr, which exist in both the GOSA architecture and the ACME architecture.

7.4.1.2 Internal Validity

Internal validity or the causal relationships between the input and output of the GOSA process are illustrated by the artifacts produced from applying the GOSA

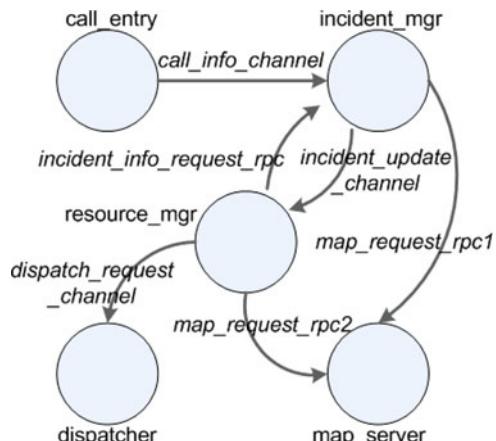


Fig. 7.12 An independently designed control architecture for the London ambulance system

process. The process and artifacts shown in Sect. 7.2 illustrate the step-by-step transition of goals to alternatives, to a logical architecture, and finally to a concrete architecture through model refinements and rule applications. Through this detailed process, each element in the architecture is traceable back to the source FRs and NFRs, along with the design rationales.

7.4.1.3 External Validity

External validity or the domains to which the study's findings can be generalized are suggested by the application domain in the study. The London ambulance dispatch system may be considered a typical information system where the system interacts with the users and manipulates business objects. To help support this observation, we have also applied the GOSA approach to two other information systems: an airfare pricing system and an catering pricing system. These studies also produced architectures with similar architectural characteristics. However, the results from those supplemental studies were inconclusive due to the lack of control architectures as they were proprietary software systems. For other application domains, such as environmental control application domains where the system controls the environment based on pre-defined rules, this study neither confirms nor invalidates the applicability of the GOSA approach.

7.4.1.4 Experimental Reliability

With respect to the experimental reliability or the ability to repeat the study, the main factor affecting the reliability appears to be the subject who carries out the study. Any subject applying the GOSA should produce architectures with similar characteristics. However, the goal-oriented analysis skill of the subject and his or her subjectiveness in making tradeoff analyses may affect the resulting architectures. The possible variations are due to the subjectiveness in interpreting textual subject matter literature since text is generally imprecise by nature and is oftentimes ambiguous, incomplete, or conflicting. As a result, it is difficult for different subjects to arrive at the same interpretation. Similarly, NFRs by nature can be subjective and oftentimes conflicting. Different subjects may arrive at different tradeoff decisions. Such unreliability elevates the need for explicit and (semi) formal representation of requirements, for instance using a goal-oriented approach, so that the interpretation and decision making are more transparent and agreeable among the stakeholders. To help achieve reliable and repeatable goal-oriented analysis, patterns of goals, alternatives, and their tradeoffs may be used for similar or applicable domains. [30].

7.5 Related Work and Discussion

As a goal-oriented method, the GOSA approach adopts and integrates three well known goal-oriented methods in a complementary manner: the *i** Framework [33] for analyzing FRs, the NFR Framework [7] for analyzing NFRs, and KAOS [11] for representing agent responsibility assignments. The GOSA approach further extends the three underlying methods beyond the goal-oriented requirements engineering to goal-oriented software architecting.

Deriving software architectures from system goals has been proposed by an extension of the KAOS method [32] and the Preskiptor process [5], the extended KAOS being the most closely related to our approach. At high-level, both the extended KAOS and our GOSA approaches capture requirements as goals to be achieved, which are then used to derive a logical architecture and subsequently a concrete architecture. However, the goal model in the extended KAOS approach is more formal and is mainly concerned with functional goals while the goal model in the GOSA method deals with both functional and non-functional goals. Non-functional goals are used in the GOSA method for explicit qualitative tradeoff analysis. Furthermore, the GOSA method derives process and interface components, and their operations from the goal model.

A software architecture is generally defined with a number of architectural constituents, including components, connectors, ports, and roles. In the GOSA approach, components and connectors are derived from the goal model and the goal-entity relationships using the mapping rules, architectural style and pattern applications. Since component ports and roles are not generally determined directly by requirements, they are therefore defined in the patterns and are not derived from the requirements.

A number of industrial methods have been used to support various architectural design activities: architectural analysis, synthesis, and evaluation activities. Examples of methods that support the architectural synthesis include the Attribute-Driven Design (ADD) method [1], the RUP’s 4 + 1 Views [21], the Siemens’s Four-Views (S4V) [16], the BAPO/CAFCR method [15], the Architectural Separation of Concerns (ASC) [27], and The Open Group Architecture Framework (TOGAF) [31] methods. Many of these methods provide high-level notation-neutral processes with common features such as scenario-driven and view-based architectural design. Our GOSA approach shares a number of features. For instance, similar to the ADD method, the GOSA approach explores and evaluates different architectural tactics and uses styles and patterns during the architectural design. However, the alternatives are represented and reasoned about more formally using goal-oriented modeling and analysis.

For the architectural evaluation activity, many methods use scenarios to provide context for the architecture evaluation. These methods include Scenario-Based Architecture Analysis Method (SAAM) [18], the Architecture Trade-Off Analysis Method (ATAM) method [9], and the Scenario-Based Architecture Reengineering (SBAR) method [2]. In these approaches, NFRs are generally defined by metrics to

be achieved by the architecture. For example, performance is defined in terms of response time in seconds and availability in terms of mean time to repair/recovery (MTTR) in minutes [1]. This quantitative-based evaluation can be considered a product-oriented approach where the achievement is determined by measuring an executable artifact, such as a prototype, a simulation model, or a completed system, against pre-determined metrics. As a goal-oriented approach, the GOSA approach, on the other hand, is considered a process-oriented approach [24] where early decisions are evaluated against softgoals using qualitative reasoning. The two approaches can be used in a complementary manner where architectural decisions may be qualitatively evaluated early during the development process, which are then confirmed by quantitative measurements of an executable artifact that realizes the selected decisions [14].

In addition to designing the architecture of individual systems, the GOSA approach can potentially be used for designing the architecture of systems-of-systems. For instance, the GOSA approach may be used to design the architecture of the top-most level system where its components represent other systems involved. The approach is then recursively used to design the architecture of each sub-system and their respective sub-systems or components.

7.6 Conclusion

This chapter presents a goal-oriented software architecting (GOSA) approach to designing software architectures from requirements models, which also captures the relationships between the two artifacts as well as the design rationale used during the architecting process. In this process, FRs and NFRs are represented as hardgoals and softgoals respectively. They are refined and used to explore and select alternative tasks and architectural decisions. The goal model and the goal-entity relationships are then used to derive a logical architecture, consisting of components and inter-component dependencies. The logical architecture is then further mapped to a concrete software architecture and a system architecture by applying an architectural style and architectural patterns chosen from the tradeoffs concerning system NFRs. The approach has been applied in an empirical study that showed that the approach could be useful for the information systems domain. Future works include independently conducted case studies to confirm the findings from our controlled empirical study, and to validate whether the approach is also applicable for other application domains and systems-of-systems situations. Additionally, a tool support is needed to confirm and further study the strengths and weaknesses of the approach.

Acknowledgments The authors would like to thank the anonymous reviewers for their valuable comments that greatly helped improve the presentation of this chapter.

References

1. Bass L, Clements P, Kazman R (2003) Software architecture in practice. Addison-Wesley Professional, New York
2. Bengtsson P, Bosch J (1998) Scenario-based software architecture reengineering. In: Proceedings 5th International Conference on Software Reuse, IEEE, pp 308–317
3. Blevins TJ, Spencer J, Waskiewicz F (2004) TOGAF ADM and MDA. The Open Group and OMG. <http://www.opengroup.org/bookstore/catalog/w052.htm>
4. Bosch J, Molin P (2002) Software architecture design: evaluation and transformation. In: Proceedings IEEE Conference and Workshop on Engineering of Computer-Based Systems (ECBS'99), IEEE, pp 4–10, http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=755855
5. Brandozzi M, Perry D (2003) From goal oriented requirements to architectural prescriptions: the preskriptor process. In: Proceedings from Software Requirements to Architectures Workshop (STRAW'03)
6. Chung L, Leite JCSP (2009) On non-functional requirements in software engineering. In: Borgida AT, Chaudhri VK, Giorgini P, Yu ES (eds) Conceptual modeling: foundations and applications. Springer, Berlin, pp 363–379
7. Chung L, Nixon BA, Yu E, Mylopoulos J (2000) Non-Functional requirements in software engineering. Kluwer Academic Publishers, Boston
8. Chung L, Nixon BA, Yu E (1995) Using non-functional requirements to systematically select among alternatives in architectural design. In: Proceedings 1st International Workshop on Architectures for Software Systems, Seattle, pp 31–43
9. Clements P, Kazman R, Klein M (2002) Evaluating software architectures: methods and case studies. Addison-Wesley, Reading
10. Dalcher D (1999) Disaster in London: the LAS case study. In: Proceedings 6th Annual IEEE International Conference and Workshop on Engineering of Computer Based System (ECBS), pp 41–52
11. Dardenne A, van Lamsweerde A, Fickas S (1993) Goal-directed requirements acquisition. *Sci Comput Program* 20(1–2):3–50
12. Feiler P, Gluch D, Hudak J, Lewis B (2005) Pattern-based analysis of an embedded real-time system architecture. *Arch Description Lang* 176:51–65
13. Harrison N, Avgeriou P (2007) Leveraging architecture patterns to satisfy quality attributes. *Software Architecture* 4758:263–270
14. Hill T, Supakkul S, Chung L (2009) Confirming and reconfirming architectural decisions on scalability: a goal-driven simulation approach. In: Proceedings 8th International Workshop on System/Software Architectures, Springer, pp 327–336
15. Hofmeister C, Kruchten P, Nord RL, Obbink H, Ran A, America P (2007) A general model of software architecture design derived from five industrial approaches. *J Syst Softw* 80(1): 106–126
16. Hofmeister C, Nord R, Soni D (2000) Applied software architecture. Addison-Wesley Professional, Reading
17. Ivers J, Clements P, Garlan D, Nord R, Schmerl B, Silva JR (2004) Documenting component and connector views with UML 2.0. Technical report, Software Engineering Institute, CMU
18. Kazman R, Bass L, Webb M, Abowd G (1994) SAAM A method for analyzing the properties of software architectures. In: Proceedings 16th International Conference on Software engineering, 81–90
19. Kitchenham B, Pickard L, Peege SL (2002) Case studies for method and tool evaluation. *Softw IEEE* 12(4):52–62
20. Kramer J, Wolf A (1996) Successions of the 8th international workshop on software specification and design. *ACM SIGSOFT Softw Eng Notes* 21(5):21–35
21. Kruchten P (1995) The 4 + 1 view model of architecture. *IEEE Softw* 12(6):42
22. Kruchten P, Obbink H, Stafford J (2006) The past, present, and future for software architecture. *Softw IEEE* 23(2):22–30

23. Monroe B, Garland D, Wile D (1996) ACME BNF and examples. In Microsoft Component-Based Software Development Workshop
24. Mylopoulos J, Chung L, Nixon BA (1992) Representing and using nonfunctional requirements: a process-oriented approach. *IEEE Trans Software Eng* 18(6):483–497
25. Mylopoulos J (2006) Goal-oriented requirements engineering, part II. Keynote. In: 14th IEEE International Requirements Engineering Conference
26. Page D, Williams P, Boyd D (1993) Report of the inquiry into the London Ambulance Service. South West Thames Regional Health Authority, vol. 25
27. Ran A (2000) ARES conceptual framework for software architecture. In: Jazayeri M, Ran A, van der Linden F (eds) *Software architecture for product families principles and practice*. Addison-Wesley, Boston
28. Shaw M, Garlan D (1996) *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Upper Saddle River
29. Supakkul S, Chung L (2005) A UML profile for goal-oriented and use case-driven representation of NFRs and FRs. In: Proceedings 3rd International Conference. on Software Engineering Research, Management & Applications (SERA05)
30. Supakkul S, Hill T, Chung L, Tun TT, Leite JCSP (2010) An NFR pattern approach to dealing with NFRs. In: 18th IEEE International Requirements Engineering Conference, pp 179–188
31. The Open Group (2003) The Open Group Architecture Framework v8.1. <http://www.opengroup.org/architecture/togaf8-doc/arch>, accessed on Dec. 19, 2003
32. van Lamsweerde A (2003) From system goals to software architecture. *Formal Methods for Software Architectures* 2804:25–43
33. Yu E, Mylopoulos J (1994) Understanding why in software process modelling, analysis, and design. In: Proceedings 16th International Conference on Software Engineering, pp 159–168
34. Zhao J (1997) Using Dependence Analysis to Support Software Architecture Understanding. In: Li M (ed) *New Technologies on Computer Software*, International Academic Publishers, pp 135–142

Chapter 8

Product-Line Models to Address Requirements Uncertainty, Volatility and Risk

Zoë Stephenson, Katrina Attwood, and John McDermid

Abstract Requirements uncertainty refers to changes that occur to requirements during the development of software. In complex projects, this leads to task uncertainty, with engineers either under- or over-engineering the design. We present a proposed commitment uncertainty approach in which linguistic and domain-specific indicators are used to prompt for the documentation of perceived uncertainty. We provide structure and advice on the development process so that engineers have a clear concept of progress that can be made at reduced technical risk. Our contribution is in the evaluation of a proposed technique of this form in the engine control domain, showing that the technique is able to suggest design approaches and that the suggested flexibility does accommodate subsequent changes to the requirements. The aim is not to replace the process of creating a suitable architecture, but instead to provide a framework that emphasises constructive design action.

8.1 Introduction

In a conventional development process, a requirements writer creates an expression of his needs, to be read by a requirements reader. The requirements reader then creates a system to meet that need [1]. A particular problem faced by developers is the gradual squeezing of implementation time as deadlines become tighter and requirements are not fully agreed and validated until late in the development programme. Our experience with product-line modelling and architecture suggests that an uncertain requirement may be treated as a miniature product line, albeit one that varies over time rather than between different products. In the ideal case, this will derive flexibility requirements that help to accommodate subsequent fluctuation in the requirements, allowing the engineer to commit to the design even though the requirements are still uncertain. In this chapter, we take inspiration from uncertainty analysis, product-line engineering and risk management to synthesise

an approach that provides insight into the flexibility needs of implementations driven by uncertain requirements.

Our approach, which is described in detail in Sect. 8.2 below, has two stages. Firstly, we use a combination of lightweight linguistic indicators and domain expertise to identify potential uncertainties in natural-language requirements, either in terms of the technical content of a requirement or of its precise meaning. We then use these uncertainties as the basis of a process of restatement of the original requirement as one or more ‘shadow’ requirements, which take account of changes which might arise in the interpretation of the original as the development proceeds. Each ‘shadow requirement’ is tagged with an indication as to the likelihood of its being the actual intended requirement. System design can then proceed with a clearer understanding of the risk of particular implementation choices. We refer to the approach as ‘commitment uncertainty’, to reflect the trade-off between requirements *uncertainty* and the need for (a degree of) design *commitment* at an early stage in the development process.

Our claim is that our particular choice of techniques provides up-front information about flexibility needs such that the resulting implementation is better able to cope with the eventual agreed requirements. Our contribution includes details of the choice of techniques, their specialisation and evaluation of the effectiveness of the resulting approach.

The remainder of the introduction provides a review of the areas of literature that inspire this work. Then, in the following section, we describe our approach to commitment uncertainty, followed by an evaluation of the approach in two separate studies.

8.1.1 Product-Line Engineering

Product-line engineering enables the provision of flexibility constrained to a particular scope [2]. The principal processes of product-line engineering are domain engineering, in which the desired product variation is modelled and supported with reusable artefacts and processes; and application engineering, in which the predefined processes configure and assemble the predefined artefacts to create a particular product [3]. The ability to rapidly create products within the predefined scope offsets the up-front cost of domain engineering, but it relies on a high degree of commonality between products [4] to reduce the size and complexity of the product repository.

Several of the technologies specific to product-line engineering are useful when dealing with uncertainty. Feature models [5] provide a view of the configuration space of the product line, documenting the scope of available products and controlling the configuration and build process. These models present a simple selection/dependency tree view of the underlying product concepts [6]. Topics in feature modelling include staged configuration [7] in which a process is built around partially-configured feature models that represent subsets of the available

products, and feature model semantics [8] in which the underlying propositional structure of the feature model is examined. The former is useful in uncertainty analysis as it provides a way for the design to track the gradual evolution of changes to the requirements; the latter is problematic because it generally normalises the propositional selection structure into a canonical form. Such a normalised model could make it difficult to precisely identify and manage variation points that exist because of uncertainty.

Domain-specific languages [9] often complement feature models; while a feature tree is good for simple dependencies and mutual exclusions, a domain-specific language is better able to cope with multiple parameters with many possible values. Domain-specific languages are typically used along with automated code generation and assembly of predefined artefacts [10]. Given suitable experience and tool support for small-scale domain-specific languages, it may be feasible to use such approaches in making a commitment to a design for uncertain requirements.

In addition to these techniques, some more general architectural strategies are often used with product-line engineering, and would be suitable candidates for design decisions for uncertain requirements. These include explicitly-defined abstract interfaces that constrain the interactions of a component; decoupling of components that relate to different concerns; and provision of component parameters that specialise and customise components to fit the surrounding context [11].

8.1.2 Requirements Uncertainty and Risk

Requirements uncertainty is considered here to be the phenomenon in which the requirement as stated is believed by the requirements reader not to be the requirement that is intended by the requirements writer; that once the system is delivered, the requirements writer will detect the discrepancy and complain about the mismatch [12]. In dealing with requirements uncertainty, approaches take into account both organisational and linguistic concerns.

Uncertainty is considered to be present throughout a project, and presents a risk both to that project and to the organisation as a whole [13]. To deal with uncertainty from an organisational perspective, it must be identified, managed and controlled (see e.g. Saarinen and Vepsäläinen [14] for a more complete overview of risk management in this area) in tandem with other technical and programme risk activities. More pragmatically, in a study of U.S. military projects, Aldaijy [15] identified a strong link between requirements uncertainty and task uncertainty. That is, when faced with the prospect that the requirements may change, engineers often lack clarity on what tasks to perform.

Techniques to deal with uncertainty vary widely in their nature and scope. A significant body of work is aimed at linguistic techniques, to control language and identify problems in ambiguous requirements for feedback to the requirements writer [16, 17]. From this literature, we recognise the important trade-off between the “weight” of the language analysis and the effort involved in obtaining useful

information, as evidenced by the use of lightweight techniques that only use a shallow parse of the requirements [18] or targeted techniques that ignore ambiguities that are easy to detect in favour of highlighting those that are difficult to work with [19].

Once uncertainty is detected, it should at a minimum be recorded, e.g. as a probability distribution [20]. While uncertainty remains, there is an increased possibility of the requirements being inconsistent; this should be respected and maintained throughout the lifecycle and only forced to be resolved when necessary [21]. A further step is to use the information in negotiating for clearer requirements with the requirements writer, a strategy that is coloured by the project's conceptualisation of the requirements writer (e.g. as explained by Moynihan [22]).

8.2 The Commitment Uncertainty Process

Our approach to requirements uncertainty assumes that it is possible to explicitly analyse requirements and context for potential future changes and provide insight into the design phase so that the design accommodates those changes. In this respect, the approach is identical to a conventional product-line engineering approach (albeit with temporal variation rather than population variation) and is similar in its structure to other requirements-uncertainty approaches that aim to influence the design directly rather than waiting to negotiate further with the requirements writer. For example, Finkelstein and Bush report [23] on an uncertainty approach that considers scenarios in different versions of future reality as a basis for stability assessment of goal-based requirements representations. Within this structure of suggesting derived requirements to control flexibility, we use a classification of requirements language issues that is broadly similar to the checklist decomposition proposed by Kamsties and Paech [17]. In addition to classifying the problem in the requirement, we also classify the situation of the requirements writer that might have led to the uncertainty, building on the insights found in Moynihan's requirements-uncertainty analysis of IS project managers [22]. Finally, we introduce one important terminological distinction: in addition to requirements uncertainty, which is associated with the problems in communicating the requirement, we also refer to requirements volatility, the change that could occur to the requirement.

8.2.1 Process Overview

The commitment uncertainty process is shown in Fig. 8.1. In sub-process p_1 we examine requirements for indicators of uncertainty, taking into account both the requirement and its assumed development context. The techniques employed in sub-process p_1 are described in detail in Sects. 8.2.2 and 8.2.3 below. In p_2 we

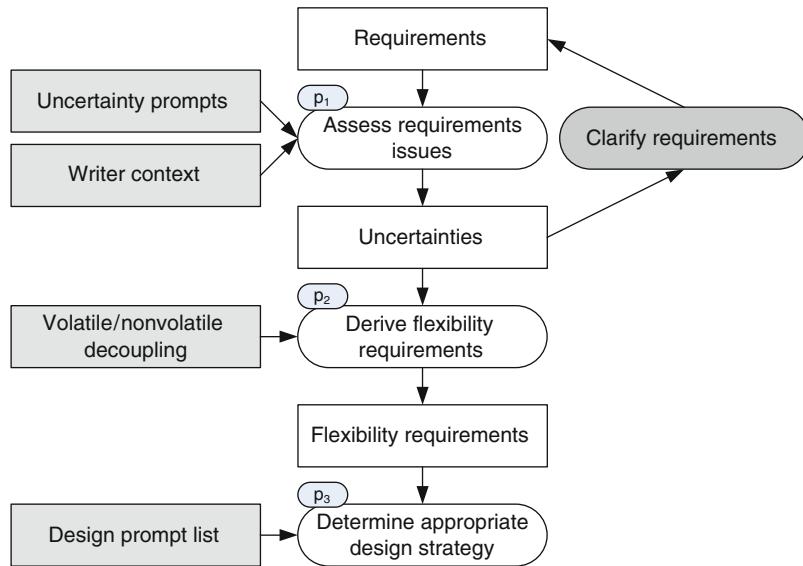


Fig. 8.1 Overview of the commitment uncertainty process. Specific materials on the *left-hand side* are explained in this chapter. The clarification process on the *right-hand side* is outside of the scope of this chapter

create flexibility requirements by factoring the uncertainty specification into volatile and non-volatile parts. This sub-process is detailed in Sect. 8.2.4. Finally, in Sect. 8.2.5, we describe sub-process p_3 , in which we suggest possible implementation strategies for the requirements based on a predefined prompt list for the scope of the product in which the volatility lies.

8.2.2 Requirements Uncertainty Prompts

The checklist for requirements issues is as shown in Table 8.1. The list contains issues that are related to the linguistic structure of the requirement as well as issues that relate to the technical content of the requirement. In this analysis technique, we recommend an explicit record of uncertainty, linking to relevant supporting information, to enable effective impact analysis. This is similar to the use of domain models and traceability in product-line engineering, and is essential to effective uncertainty risk management.

In practice, a single requirement may include many forms of uncertainty, which may interact with one another. Rather than trying to classify all such interactions, the analysis prompts the reader into thinking about the requirement from different standpoints. In some situations, no particular reason for the uncertainty will emerge.

Table 8.1 Uncertainty indicator checklist

Area	Uncertainty	Form
Incompleteness	Unfinished requirement	A part of the requirement has not yet been written. There could be a trailing unfinished clause or an ellipsis. There will usually be little to indicate how to fill the gap.
	Placeholders	A placeholder is used for part of the requirement. This is typically some metasyntactic expression – perhaps “TBD” or “[upper limit]”. In some organisations, placeholders may be given as information paragraphs or marked-up notes.
	Missing counterpart	In many cases, requirements come as a set. For example, there may be a startup requirement for each mode of a system. Even with little domain knowledge, it should be apparent when part of the set is missing.
	Under-specification	An underspecified requirement constrains the implementation to some extent, but leaves options open. In some cases, this is a careful abstraction to avoid over-constraining the implementation. In others, the requirement is simply not concrete enough.
	Terminology	Some terminology in the requirement is not well-defined; it needs qualification, better definition or replacement.
Ambiguity	Syntactic structure	The sentence has a structure that can be read in more than one plausible way. This areas is well-studied in linguistic approaches to requirements ambiguity [18, 24]
	Incorrectness	There is some detail in the requirement that is demonstrably incorrect, through a scenario or some logical inference.
Commitment	Overspecification	The requirement includes more detail than necessary, giving awkward or infeasible constraints.
	Misplaced requirement	The positioning of the requirement (section heading, informative context) conflicts with its content.
	Mislabeled domain information	The statement is presented as a requirement but it contains only definitions, therefore technically not requiring any action.
Mislabelling		

A pragmatic trade-off must be made between the effort of explaining the uncertainty and the costs saved by performing the analysis.

As an example, consider this sample requirement:

110. The system shall provide an indication of the IDG oil temperature status to the aircraft via ARINC.

This requirement suffers from (at least) two different uncertainties. Firstly, the phrase “an indication of the IDG oil temperature status” contains words (“indication”, “status”) that are either redundant (meaning ‘provide the IDG oil temperature’) or poorly-defined (meaning to get the status of the oil temperature, and then transmit an indication of the status).

In all likelihood, the real requirement is the following:

110a. The system shall provide its measurement of the IDG oil temperature to the aircraft via ARINC.

The second issue is that the requirement does not directly identify where to find out information about the message format or value encoding. The link could be written into the requirement:

110b. The system shall provide its measurement of the IDG oil temperature to the aircraft via ARINC according to protocol P100-IDG.

It is more likely that the link would be provided through a tool-specific traceability mechanism to an interface control document.

8.2.3 Identifying Requirements Writer Context

We recognise that there is also value in trying to understand the context within which a requirement is written; given a set of such requirements, it may be possible to infer details of the context and hence suggest specific actions to take to address uncertainty. Table 8.2 explains possible reasons for problems appearing in requirements. Rather than capturing every requirements issue, we instead present possible issues for the engineer to consider when deciding on how much information to feed back to the requirements writer and when. We make no claim at this stage that the table is complete; however, it covers a number of different aspects that might not ordinarily be considered, and on that basis we feel it should be considered at least potentially useful.

In practice, it will rarely be possible to obtain a credible picture of the requirements writer context. Nevertheless, it can be useful to consider the possible context to at least try to understand and accommodate delays in the requirements. Explicitly recording assumptions about the writer also facilitates useful discussion among different engineers, particularly if there are actually multiple issues behind the problems in a particular requirement. Finally, the overall benefit of this identification step is that it gives clear tasks for the engineer, reducing the so-called “task uncertainty” and improving the ability to make useful progress against the requirements.

8.2.4 Recording and Validating Uncertainties

To record the uncertainty in a way that is useful to all interested stakeholders, we advocate a multi-stage recording process shown in Fig. 8.2. First, the original requirement and uncertainty analysis are identified. Then, the requirement is restated by identifying volatile parts and presenting a miniature product line view of the requirement. The exact process here is one of engineering judgement based on the form of the uncertainty and the reason behind it; the interested reader is referred to a more comprehensive work on product lines (e.g. Bosch [3] or Weiss and Lai [9]) for further elaboration.

Table 8.2 Writer context checklist

Reason	Details
Novelty	The requirements writer, and perhaps also the reader, is unfamiliar with this area of requirements. This is either slowing the writer down (unfinished requirements) or causing premature commitment (incorrect requirements). Uncertainty should decrease over time. Detailed feedback on the requirements may be unwelcome early on.
Complexity	The requirements specify something complex, and the difficulty of dealing with the complexity leads to unfinished, ambiguous or conflicting requirements. They may also be copied from previous requirements to reuse a successful structuring mechanism.
Concurrent delay	The requirements writer has yet to perform the work towards the requirement; the details depend on the results of processes that are incomplete. This often happens in large projects with multiple subsystems and complex interfaces. Eventually, the requirement will be properly defined. In this case, feedback is likely to be welcome.
Pressure	The requirements writer is under pressure; the final requirement has not yet been defined. This could arise from areas such as inter-organisational politics, financial arrangements or resourcing. Feedback to suggest clarifications may not be effective.
Language gap	The requirements writer uses language differently to the requirements reader. He may be writing in a non-native language but using native idioms and grammar, or he may apply the rules of the language differently to the expectation of the reader. This subject is studied at length in linguistic approaches to requirements [16].
Context gap	The uncertainty arises from the difference in information available to the reader compared to the writer. The writer may make assumptions that are not made explicit, or the reader may know more about the target platform. It will be useful to document explicit assumptions to help support decision-making.
Intent gap	It may be unclear how the requirements writer intends to constrain the implementation. This can occur with abstractions; distinguishing between deliberate and accidental generality can be difficult. Another possibility is a set of “soft” requirements to trade off, presented as hard constraints. This may happen if the contract does not allow for appropriate negotiation. Issues such as these indicate that extra effort in design trade-offs and flexible architecture may be appropriate.
Medium	The communications medium between writer and reader may constrain the information that may be represented in a cost-effective way. The medium includes the tools and experience available to the writer and reader. For example, the reader may be able to read text and diagrams, but not view links between the two.

With the requirement volatility captured, the requirement is then restated into the same form as the original requirement. This is the shadow requirement, and represents what the engineer will actually work to. The final step is to double-check the result by checking that the original requirement, as stated, is one possible instantiation of the shadow requirement.

Consider the sample requirement 110 again:

110. The system shall provide an indication of the IDG oil temperature status to the aircraft via ARINC.

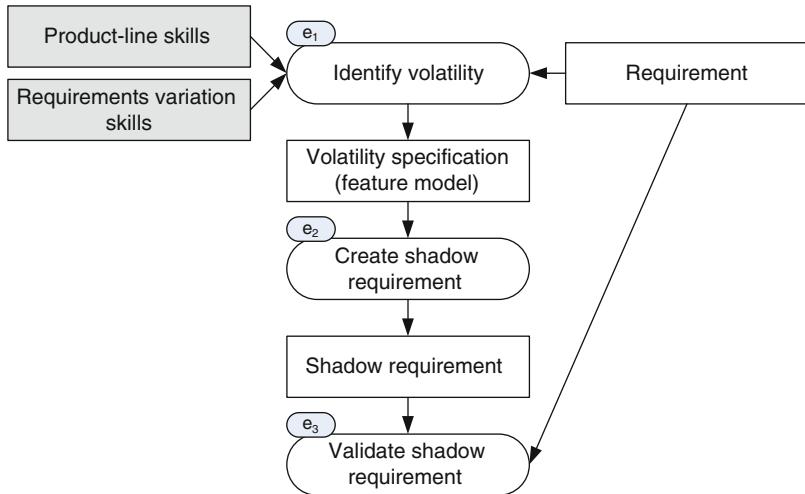


Fig. 8.2 Overview of deriving flexibility requirements. Volatility (expected changes) is specified and then factored into the requirement to create a family of related requirements

The volatility might be specified as follows:

Stable part:

- Sending data over ARINC, part of overall ARINC communications.
- Sending IDG oil temperature or derived values, depending on reading or synthesising value

Volatile part:

- Data to be sent
- Message format/encoding

Likely changes:

- Data to be sent is one or more of:
 - IDG oil temperature reading
 - IDG oil temperature limits, rate of change
 - Details of faults with that reading
 - Presence/absence of faults
 - Details of current sensing method
 - Value encoding will depend on data to be sent

Instantiations:

- 110a. The system shall provide its measurement of IDG oil temperature to the aircraft via ARINC.
- 110b. The system shall provide a count of current IDG oil temperature faults to the aircraft via ARINC.
- 110c. The system shall provide IDG oil temperature, operating limits and rate of change to the aircraft via ARINC.

One possible shadow requirement for the IDG oil temperature requirement would be:

110x. The system shall provide (IDG oil temperature|IDG oil temperature faults|IDG oil temperature presence/absence of faults|IDG oil temperature, operating limits and rate of change) to the aircraft via ARINC [using protocol (P)].

An alternative approach is to derive explicit flexibility requirements to guide the implementation:

F.110a. The impact of changes to the feature of the IDG oil temperature to send on the delivery of IDG information in requirement 110 shall be minimised.

F.110b. The impact of changes to the communication format on the delivery of IDG information to the aircraft shall be minimised.

8.2.5 Suggesting Design Approaches

It is not intended that the commitment uncertainty approach should constrain the type of implementation chosen to accommodate the identified uncertainty. Nevertheless, it is useful to give advice on the type of design approach that is likely to be successful, as a way to further overcome task uncertainty and improve the likelihood of quickly arriving at a suitable design.

The advice is based on a recognition that a design approach will typically respond to a group of requirements. We take as input the scale of the volatility in that group of requirements and produce a suggested list of design approaches to consider, in a particular order. The intent is not to restrict the engineer to these design approaches, nor to constrain the engineer to select the first approach for which a design is possible; the aim is simply to help the engineer to quickly arrive at something that is likely to be useful.

Our approach is therefore much coarser than more considered and involved approaches such as those of Kruchten [11] or Bosch [3]. The mapping is shown in Table 8.3. In this table, the scale of design volatility is broadly categorised as “parameter” when the volatility is in a single parameterisable part of the requirements; “function” when the behaviour changes in the volatile area; and “system” when the volatility is in the existence or structure of a whole system. The engineer is encouraged to choose whichever of these designations best matches the volatility, and then use his existing engineering skills to arrive at designs that are prompted by the entries under that heading: “Parameterisation” to include suitable data types and parameters in the design; “Interfaces and Components” to consider larger-scale

Table 8.3 Mapping from volatility scale to suggested design approaches

Parameter	Function	System
Parameterisation	Interfaces and components	Interfaces and components
Interfaces and components	Parameterisation	Auto-generation
Auto-generation	Auto-generation	

interfacing and decoupling; and “Auto-Generation” to build a domain-specific language or component configuration scripting system to accommodate the volatility.

8.2.6 Ordering Design Decisions

In software, major design decisions are traded off and captured in the software architecture; functionality is then implemented with respect to this architecture. In some complex design domains, however, there are multiple competing design dependencies that can be difficult to resolve. To assist in making progress in this context, we provide a framework that tracks design dependencies and resolves design decisions hierarchically to produce the complete design. The intended effect is that the areas that are most volatile are those that are least fundamental to the structure of the design. This technique makes use of product-line concepts to represent optionality.

In addition to dependencies between design decisions and (parts of) requirements, any part of a design may depend on part of an existing design commitment. This includes both communicating with an existing design element and reusing or extending an existing element. These dependencies are the easiest to accommodate with indirection and well-defined interfaces. Contextual domain information is also important, and most design commitments are strongly related to domain information. The dependency on domain information can be managed through parameterisation or indirection. The process of uncertainty analysis provides additional exposure of contextual issues, helping to reduce the risk associated with missing context.

In our prototype modelling approach, we explicitly represent dependencies between design elements in a graphical notation, shown in Fig. 8.3. This example represents the decision to add the IDG oil temperature parameter to the ARINC table as defined in the interface control document. The decision is prompted by the

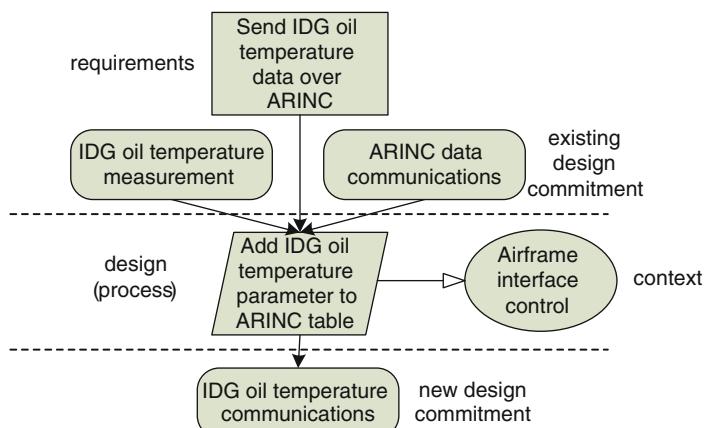


Fig. 8.3 Representing dependencies between design elements

requirement to send the information, the availability of the information, and the availability of a suitable communication mechanism. The result of the decision is a new entry in a data table to allow the communication of the appropriate value. While this example is perhaps trivial, it illustrates the important distinction between decisions (processes that the user may engage in) and designs (the artefacts that result from design activity).

Commitment uncertainty analysis associates volatility with context, requirements and design. This may be annotated alongside the decision tracing diagram. To retain familiarity and compatibility with existing approaches, we base this representation on conventional feature modelling notations, as shown in Fig. 8.4. A feature model view of design decision volatility is a powerful visual tool to help appreciate the impact of volatility on the design approach. It is expected that this

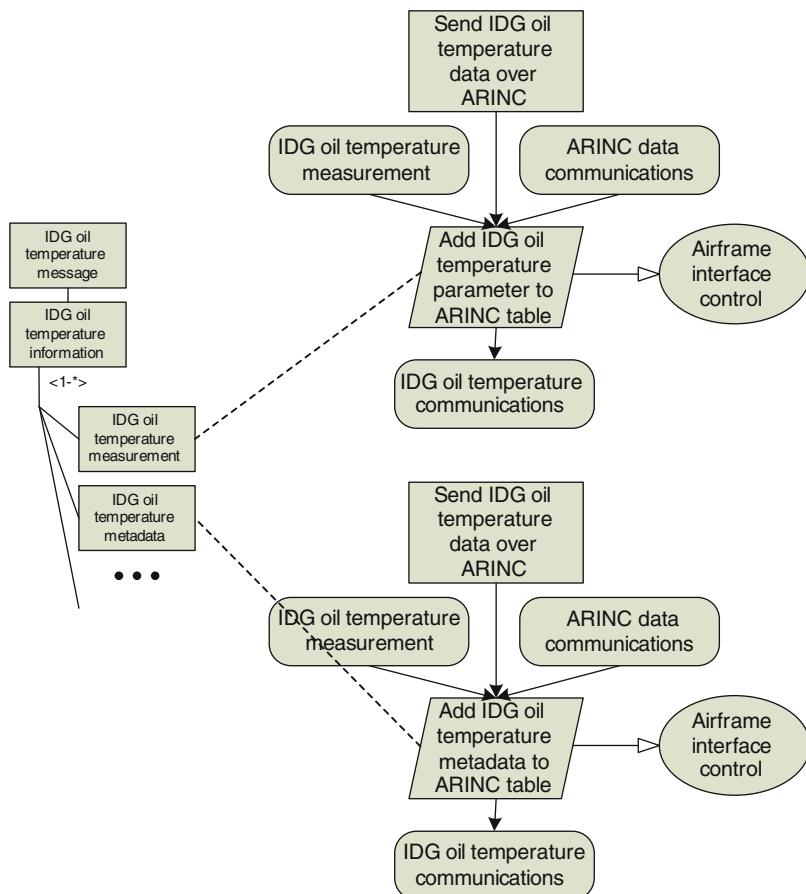


Fig. 8.4 Representing volatility with design decisions. The annotation on the *left-hand side* shows features (*rectangles*) with dependencies (*arcs*) and selection constraints (*arc label <1-*>* in this example)

type of visualisation will be of most benefit when communicating involved technical risk deliberations to interested stakeholders.

Once a body of design decisions has been produced, individual design commitments may be resolved together to create design solutions. The granularity of resolution is not fixed; it will depend on the complexity of the design decisions, their dependencies and their volatility. Similarly, the order of design decisions is not fixed. Any design decision that changes will have an impact on later design decisions, so the intent of design decision resolution is to defer more volatile decisions as late as possible in the decision sequence. Ordinarily, the design decision sequence is chosen by creating a partial order from the design decision dependencies, adding volatility information and then creating a total order from the resulting dependencies. When adding the volatility information, it is important to start from the most volatile decision and then descend in order of volatility, adding volatility relationships where they do not conflict with existing relationships.

In extreme cases, some reengineering will be needed to arrive at an appropriate design. For example, it may be advantageous to break a design dependency in order to accommodate a volatility dependency. This will typically prompt a refactoring of the existing commitments to accommodate the additional variation from the volatile design and allow for the design dependency using dependency injection and inversion of control.

8.3 Empirical Evaluation

8.3.1 Quantitative Analysis of Effectiveness

In this section, we present the design and results of an experiment to test the theoretical effectiveness of the commitment uncertainty approach. For this experiment, we used four instances of the requirements from an engine controller project; a preliminary (draft) document P and issued requirements I_1 , I_2 and I_3 , from which we elicited changes made to the requirements over time. Since the requirements in this domain are generally expressed at a relatively low level – particularly with respect to architectural structure – we consider that the requirements are, for the purposes of this experiment, equivalent to the design.

In the experiment, we created two more flexible versions of P : P_t using conventional architectural trade-off techniques, and P_u using commitment uncertainty techniques. The hypothesis is that, when faced with the changes represented by I_{1-3} , P_u accommodates those changes better than P_t , and both P_u and P_t are better at accommodating changes than P . We consider a change to be accommodated if the architecture of the design provides flexibility that may be used to implement the required change. Table 8.4 shows the data for P_u , and Table 8.5 is the equivalent data for P_t . In each table, the **ID** column gives the associated requirement ID, then

Table 8.4 Uncertainty analysis on randomly-selected requirements

ID	Scope	Derived	I ₁ -P	I ₁ -P _u	I ₂ -P	I ₂ -P _u	I ₃ -P	I ₃ -P _u
6	Y	Y	Y	Y	Y	Y	Y	Y
20	Y	Y	Y	Y	Y	Y	Y	Y
32	Y	Y	Y	Y	Y	Y	Y	Y
99	Y	Y	Y	Y	Y	Y	Y	Y
105	Y	Y	Y	Y	Y	Y	Y	Y
22	Y	Y	Y	Y	Y	Y	Y	Y
127	Y	Y	N	Y	N	Y	N	Y
5	Y	N	Y	Y	Y	Y	Y	Y
39	Y	Y	N	Y	N	Y	N	Y
49	Y	Y	Y	Y	Y	Y	N	Y
26	Y	Y	Y	Y	Y	Y	Y	Y
16	Y	Y	Y	Y	Y	Y	Y	Y
113	Y	Y	Y	Y	Y	Y	Y	Y
118	Y	Y	N	Y	N	Y	N	Y
119	Y	Y	Y	Y	Y	Y	Y	Y
50	Y	N	Y	Y	Y	Y	Y	Y
107	Y	Y	N	Y	N	Y	N	Y
56	N	N	Y	Y	Y	Y	Y	Y
12	Y	N	Y	Y	Y	Y	Y	Y
60	Y	Y	Y	Y	Y	Y	Y	Y
64	Y	N	Y	Y	Y	Y	Y	Y
70	Y	Y	Y	Y	Y	Y	Y	Y
71	Y	Y	Y	Y	Y	Y	Y	Y
110	Y	Y	N	Y	N	Y	N	N
2	N	N	Y	Y	Y	Y	Y	Y
73	N	N	Y	Y	Y	Y	Y	Y
76	Y	Y	Y	Y	Y	Y	Y	Y
123	Y	Y	N	Y	N	Y	N	Y
137	N	N	Y	Y	Y	Y	Y	Y
140	N	N	Y	Y	Y	Y	Y	Y
148	N	N	Y	Y	Y	Y	Y	Y
151	N	N	Y	Y	Y	Y	Y	Y
155	N	N	Y	Y	Y	Y	Y	Y
159	N	N	Y	Y	Y	Y	Y	Y
162	N	N	Y	Y	Y	Y	Y	Y
93	Y	Y	N	Y	N	Y	N	Y
94	Y	Y	N	Y	N	Y	N	Y
120	Y	Y	Y	Y	Y	Y	Y	Y
Y total		28	24	30	38	30	38	37
Filtered		28	24	20	28	20	28	27

the **Scope** column identifies whether the requirement was in scope for commitment uncertainty and the **Derived** column indicates whether a derived requirement was produced from the analysis. The remaining columns evaluate the two sets of requirements – the original set and the set augmented with derived requirements

Table 8.5 Trade-off analysis on randomly-selected requirements

ID	Scope	Derived	I1-P	I1-Pt	I2-P	I2-Pt	I3-P	I3-Pt
19	Y	Y	Y	Y	Y	Y	Y	Y
30	Y	Y	N	N	N	N	N	N
32	Y	N	N	N	N	N	N	N
102	Y	Y	N	N	N	N	N	N
106	Y	N	N	N	N	N	N	N
127	Y	N	N	N	N	N	N	N
10	Y	N	Y	Y	Y	Y	Y	Y
38	Y	N	Y	Y	Y	Y	N	N
48	Y	Y	Y	Y	Y	Y	Y	Y
131	Y	N	Y	Y	Y	Y	Y	Y
16	Y	Y	Y	Y	Y	Y	Y	Y
113	Y	Y	Y	Y	Y	Y	Y	Y
118	Y	Y	N	Y	N	Y	N	Y
42	Y	N	Y	Y	Y	Y	Y	Y
50	Y	N	Y	Y	Y	Y	Y	Y
81	Y	N	Y	Y	Y	Y	Y	Y
27	Y	N	N	N	N	N	N	N
62	Y	Y	Y	Y	Y	Y	Y	Y
67	Y	Y	Y	Y	Y	Y	Y	Y
71	Y	Y	Y	Y	Y	Y	Y	Y
111	Y	Y	N	Y	N	Y	Y	Y
3	Y	N	Y	Y	Y	Y	Y	Y
31	N	N	Y	Y	Y	Y	Y	Y
76	Y	Y	Y	Y	Y	Y	Y	Y
6	Y	N	Y	Y	Y	Y	Y	Y
56	N	N	Y	Y	Y	Y	Y	Y
109	N	N	Y	Y	Y	Y	Y	Y
78	N	N	Y	Y	Y	Y	Y	Y
136	N	N	Y	Y	Y	Y	Y	Y
142	N	N	Y	Y	Y	Y	Y	Y
148	N	N	Y	Y	Y	Y	Y	Y
151	N	N	Y	Y	Y	Y	Y	Y
154	N	N	Y	Y	Y	Y	Y	Y
160	N	N	Y	Y	Y	Y	Y	Y
88	Y	N	Y	Y	Y	Y	Y	Y
92	N	N	Y	Y	Y	Y	Y	Y
96	Y	Y	Y	Y	Y	Y	Y	Y
108	N	N	Y	Y	Y	Y	Y	Y
Y total		26	13	30	32	30	32	31
Filtered		26	13	18	20	18	20	19

from the additional analysis. At the bottom of the table, the totals are presented as both raw totals and a filtered total that excludes the requirements that were outside the scope of architectural flexibility provision.

Table 8.6 Summary of uncertainty analysis against original requirements. Data show significant ($\chi^2, p < 0.05$) improvement over original system

	Y	N		Y	N		Y	N
I ₁ -P	20	8	I ₂ -P	20	8	I ₃ -P	19	9
I ₁ -P _u	28	0	I ₂ -P _u	28	0	I ₃ -P _u	27	1

Table 8.7 Summary of trade-off analysis against original requirements. No significant improvement

	Y	N		Y	N		Y	N
I ₁ -P	18	8	I ₂ -P	18	8	I ₃ -P	18	8
I ₁ -P _t	20	6	I ₂ -P _t	20	6	I ₃ -P _t	19	7

Table 8.8 Summary of uncertainty analysis against trade-off analysis. Data show significant ($\chi^2, p < 0.05$) improvement of uncertainty analysis against trade-off analysis

	Y	N		Y	N		Y	N
I ₁ -P _u	28	0	I ₂ -P _u	28	0	I ₃ -P _u	27	1
I ₁ -P _t	20	6	I ₂ -P _t	20	6	I ₃ -P _t	19	7

To analyse the hypothesis, we compare the ability of one design to accommodate change with the ability of another. This produces contingency tables, displayed in Tables 8.6–8.8.

The results indicate that the commitment uncertainty analysis made a significant improvement in the ability to accommodate change, while the more conventional trade-off analysis was not as capable. Some caveats should, however, be stated here. The significance measure here provides an indication of internal validity. In this particular engineering domain, the use of requirements to stand for designs is appropriate, since the design process is characterized by the iterative decomposition of requirements. We note, however, that our findings may not be applicable in other domains. In terms of the external validity of the study, it is important to note that our experiment differed from real-world practice in that we were external observers of the project, with access to the entire lifecycle history. In practice, it may be more difficult for interested parties to make appropriately flexible commitments early on in a project. It would be interesting to repeat the study in a live project, deriving flexibility requirements to which designers were prepared to commit their choices and then observing the degree to which these requirements proved useful in accommodating later change. It should also be noted that this work concerns an embedded software system, where there are considerable constraints on the design and implementation, and there are objective tests of system functionality and effectiveness. Design drivers in other domains may, of course, differ markedly: for example, a successful design might be one which opens up a new market or incorporates some innovative functionality. In these cases, the nature of the requirements and design processes are likely to differ markedly from the aerospace domain, and the effectiveness of our proposed approach may be less clear.

8.3.2 Qualitative Evaluation of Design Selection

In this study, we investigated the ability of the design prompt sequence approach to correctly identify appropriate design targets for the implementation of uncertainty-handling mechanisms. The study is based on an internal assessment of technical risk across a number of engine projects conducted in 2008. We extracted eight areas that had been identified as technical risks that were in scope for being addressed with architectural mechanisms. For each identified technical risk, we elicited uncertainties and then used the design prompt sequence to generate design options. Finally, we chose a particular design from the list and recorded both the position of the design in the list and the match between the chosen design and the final version of the requirements.

As an example, consider the anonymised risk table entry in Table 8.9. The individual uncertainties for this particular instance are elaborated and documented in a custom tabular format shown in Table 8.10.

The design prompts for “Function, Concurrent” are presented in the order, components/interfaces, parameterisation and then auto-generation.

The options identified are:

1. Use an abstract signal validation component with interfaces that force the designer to consider raw and validated input signals, power interrupts and signals for faults. This design ensures that each component encapsulates any uncertainty regarding its own response to power interrupt.

Table 8.9 Example risk table entry

Risk area	Specific risk	Particular instances
Requirements – flow-down	Have system-level requirements for reaction to power supply interrupts been decomposed into software requirements?	Project Hornclaw refit software.

Table 8.10 Custom tabular format for documentation of uncertainty

Certainties		
After a power interrupt, the system initialises afresh and its RAM and program state no longer represent the state of the environment.		
The system can determine some information about the state of the environment from non-volatile memory.		
The only part of the system that will be out of sync after a power interrupt is input validation.		

Uncertainties		
Type	Definition	Rationale
Function concurrent	Required signal validation after a power interrupt	Derivation from the technical risk concept “requiremens for reaction to power supply interrupts”

2. Use a parameterised signal validation component that selects its response to power interrupt from a list of possible responses. The list should be based on domain expertise and experience in designing robust power interrupt management schemes. This is applicable if the range of possible responses can be captured easily in such a list, and as long as the use of the response selection mechanism is consistent with certification guidelines for configurable components.
3. An auto-generation system may be appropriate for complex parameterisation. The input configuration is derived from the range of possible input validation responses to power interrupt. The input language to the auto-generation system should be easy to use and should be similar to other auto-generation input languages in use. For this option, the language itself captures and manages the uncertainty.

Assessment of these available choices shows that the abstract interface is the easiest to implement; the parameterisation and auto-generation approaches carry more specific details of the available interrupt-management schemes, which is not necessarily a net benefit at this time.

With such a small study, it is difficult to quantify the net benefit of the design prompt sequence; nevertheless, we feel it is useful to present some observations on the outcome of the study. The results are shown in Table 8.11.

Firstly, the study showed that the prompts were able to suggest multiple design options for each technical risk area. Contrary to expectations, the first design alternative was not necessarily the alternative that was eventually chosen; moreover, the last design choice was never selected for use in this study. This suggests that it may not be directly beneficial to create too many different design choices, although there may be an indirect benefit from the comparison of the second design choice to the third choice in establishing its relative merit. It should also be noted that, by forcing designers to consider multiple alternative design solutions (contrary to their usual practice), the technique potentially reduces the danger of the “first plausible design syndrome”, whereby designers commit themselves to the first apparently workable solution, unwilling to move on from it even when problems are identified with the design. Put another way, this could be seen as a way of

Table 8.11 Outcome of qualitative design prompt sequence study

Technical risk area	Uncertainties	Design options	Chosen design	Match to final requirements
New feature	2	3	1	Estimated good
Architecture scope	4	3, 3	2, 2	Perfect
Comprehensive requirements	7	3	None	N/A
Comprehensive requirements	3	3	2	Estimated good
Incremental development	1	0	N/A	N/A
Power interrupt requirements	1	3	1	Estimated good
State transitions	2	4	1	Perfect
New feature	2	3	2	Good

encouraging engineers to delay design commitments [25] which are viewed by some as underpinning lean processes [26].

Secondly, it was useful during the study to note the applicability of the design choices to communicate contextual assumptions from the design phase for validation when changes occur. For example, some design options could result in unused inputs once changes are made, which could impact on testability.

Lastly, we found that in some areas one uncertainty would lead on to further uncertainties. This was particularly the case in novel design areas, where an uncertainty structure arose from consideration of the suggested design alternatives. We expect that this is more closely related to a pattern-based approach to product-line feature modelling than directly to uncertainty analysis, and the phenomenon would merit closer study.

Again, we should express some caveats concerning the wider applicability of these observations. The study reported in this chapter is small, and, because of this, it is difficult to extrapolate its findings to a wider context. However, we do believe the construct validity to be appropriate – that the experiment is able to show, in principle, relationships between the scale of uncertainty and the type of design solution that is most appropriate to address the requirement. It should also be noted that the application domain is a very stable one: it is relatively easy to draw on previous experience to derive alternative design solutions. This may be more difficult in a less stable domain, although it may be that more useful alternative designs can in fact be identified in such environments. There is then a trade-off between the evaluation of alternatives and the technique's capacity, ultimately, to help designers in the derivation of better design solutions.

In terms of the practical application of the ideas presented here, it would be most appropriate to view the approach as a contribution to process architecture, perhaps in the context of product-line development or a framework such as BAPO [27]. Our approach should sit as one of a range of mechanisms that allow the designers to make commitments at appropriate points in the process.

8.4 Summary and Research Outlook

We have presented a synthesis of concepts from uncertainty, risk management and product lines to address the issue of requirements uncertainty that prevents the engineer from making a design commitment. The intended use of the technique is to rapidly suggest possible risk areas and highlight options for a lower-risk implementation that includes flexibility to accommodate particular variations from the requirement as stated. These prompts aim to inspire the engineer into creating a solution that is engineered for specific potential uncertainties, rather than forcing either a brittle implementation that cannot respond to change or an over-engineered solution that is difficult to manage over time.

In our evaluation of the technique, we found that there is significant potential for this type of analysis to suggest design flexibility that is warranted by subsequent

changes between early project requirements and final issued project requirements. Several questions still remain unanswered, however. Most importantly, how much effort is involved in creating the derived requirements and flexible design versus the time taken to refactor the design at a later stage? It is this comparison that is most likely to persuade engineers that the technique has merit. In support of this, we emphasise the positive results that have been obtained so far and focus on the practical aspects of the approach – its lightweight nature and the ability to apply it only where immediate risks present themselves. Similarly, in how many cases is flexibility added to the design but never used later on? The presence of a large amount of unused functionality may be a concern particularly in the aerospace industry and especially if it prevents the engineer from obtaining adequate test coverage.

For future work in this area, we have identified four themes. Firstly, we are interested in integration of the concepts of commitment uncertainty into a suitable metamodeling framework such as decision traces [28] or Archium [29]. Second, it would be interesting to deploy appropriate tool support based on modern metamodeling [30]. Third, there is potential benefit in a richer linguistic framework to support more detailed uncertainty analysis and feedback to requirements stakeholders, and lastly, further experimentation is needed to understand the nature of appropriate design advice, design patterns and commitment-uncertainty metrics.

Acknowledgment The research presented in this chapter was conducted at the Rolls-Royce University Technology Centre in Systems and Software Engineering, based at the University of York Department of Computer Science. We are grateful to Rolls-Royce plc. for supporting this research.

References

1. Stokes DA (1990) Requirements analysis. In: McDermid J (ed) Software engineer's reference book. Butterworth-Heinemann, Oxford
2. Schmid K, Gacek C (2000) Implementation issues in product line scoping. In: Frakes WB (ed) ICSR6: software reuse: advances in software reusability. Proceedings of the 6th international conference on software reusability. (Lecture notes in computer science), vol 1844. Springer, Heidelberg, pp 170–189
3. Bosch J (2000) Design and use of software architectures. Addison-Wesley, Reading
4. Weiss D, Ardisi M (1997) Defining families: the commonality analysis. In: ICSE'97: proceedings of the 19th international conference on software engineering, Boston, May 1997. ACM Press, New York
5. Kang K, Cohen S, Hess J et al (1990) Feature-Oriented Domain Analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie-Mellon University Software Engineering Institute
6. Czarnecki K, Kim CH P, Kalleberg KT (2006) Feature models are views on ontologies. In: SPLC2006: proceedings of the 10th international conference on software product lines. Baltimore, Maryland, USA. IEEE Computer Society Press, Los Alamitos, California, USA
7. Czarnecki K, Helsen S, Eisenecker U (2004) Staged configuration using feature models. In: Nord R L (ed) SPLC 2004: proceedings of the 9th international conference on software product lines, Boston. (Lecture notes in computer science), vol 3154. Springer, Heidelberg

8. Bontemps Y, Heymans P, Schobbens PY et al (2004) Semantics of FODA feature diagrams. In: Proceedings of the international workshop on software variability management for product derivation
9. Weiss DM, Lai CTR (1999) Software product-line engineering: a family-based development process. Addison-Wesley, Reading
10. Czarnecki K, Eisenecker C (2000) Generative programming. Addison-Wesley, Massachusetts
11. Kruchten P (2004) A taxonomy of architectural design decisions in software-intensive systems. In: Proceedings of the second Groningen workshop on software variability management
12. Parnas DL (1979) Designing software for ease of extension and contraction. *IEEE Trans Software Eng* 5(2):128–138
13. Ebert C, de Man J (2005) Requirements uncertainty: influencing factors and concrete improvements. In: ICSE'05: proceedings of the 27th international conference on software engineering, St Louis, Missouri, USA. ACM Press, New York
14. Saarinen T, Vepsäläinen (1993) Managing the risks of information systems implementation. *Eur J Inform Syst* 2(4):283–295
15. Aldaijy A et al (2006) The effects of requirements and task uncertainty on software product quality. In: Proceedings of the 15th International Conference on Software Engineering and Data Engineering, Cary, NC: International Society for Computers and Their Applications, 8ff
16. Berry DM, Kamsties E, Krieger MM (2003) From contract drafting to software specification: linguistic sources of ambiguity. Technical Report, University of Waterloo, Canada
17. Kamsties E, Paech B (2000) Taming ambiguity in natural language requirements. In: Proceedings of the international conference on system and software engineering and their applications
18. Gervasi V, Nuseibeh B (2002) Lightweight validation of natural language requirements: a case study. *Softw Pract Exper* 32(3):113–133
19. Chantree F, Nuseibeh B, de Roeck A et al (2006) Identifying noxious ambiguities in natural language requirements. IEEE Computer Society Press, Los Alamitos
20. Maynard-Zhang P, Kiper JD, Feather MS (2005) Modeling uncertainty in requirements engineering decision support. In: REDECS'05: proceedings of the workshop on requirements engineering decision support of the 13th IEEE international requirements engineering conference
21. Nuseibeh B, Easterbrook S, Russo A (2001) Making inconsistency respectable in software development. *J Syst Softw* 58(20):171–180
22. Moynihan T (2000) Coping with ‘requirements-uncertainty’: the theories-of-action of experienced IS/software project managers. *J Syst Softw* 53(2):99–109
23. Bush D, Finkelstein A (2003) Requirements stability assessment using scenarios. In: RE2003: proceedings of the 11th IEEE international conference on requirements engineering
24. Stephenson Z (2005) Uncertainty analysis guidebook. Technical Report YCS-2005-387, University of York, Department of Computer Science
25. Thimbleby H (1988) Delaying Commitment. *IEEE Softw* 5(3):78–86
26. Poppendieck M, Poppendieck T (2003) Lean software development: an agile toolkit. Addison-Wesley, Reading
27. van der Linden F, Bosch J, Kansties E, Känsälä K, Obbink H (2004) Software product family evaluation. In: Nord R L (ed) SPLC 2004: proceedings of the 9th international conference on software product lines, Boston. (Lecture notes in computer science), vol 3154. Springer, Heidelberg
28. Stephenson Z (2002) Change management in families of safety-critical embedded systems. PhD Thesis YCST-2003-03, Department of Computer Science, University of York
29. van der Ven JS, Jansen AJG, Nijhuis JAG, Bosch J (2006) Design decisions: the bridge between rationale and architecture. In: Dutoit A H, McCall R, Mistrik I, Paech B (eds) Rationale management in software engineering
30. Gonzalez-Perez C, Henderson-Sellers B (2008) Metamodelling for software engineering. Wiley, Hoboken

Chapter 9

Systematic Architectural Design Based on Problem Patterns

Christine Choppy, Denis Hatebur, and Maritta Heisel

Abstract We present a method to derive systematically software architectures from problem descriptions. The problem descriptions are based on the artifacts that are set up when following Jackson's problem frame approach. They include a context diagram describing the overall problem situation and a set of problem diagrams that describe subproblems of the overall software development problem. The different subproblems should be instances of problem frames, which are patterns for simple software development problems.

Starting from these pattern-based problem descriptions, we derive a software architecture in three steps. An initial architecture contains one component for each subproblem. In the second step, we apply different architectural and design patterns and introduce coordinator and facade components. In the final step, the components of the intermediate architecture are re-arranged to form a layered architecture, and interface and driver components are added.

All artefacts are expressed as UML diagrams, using specific UML profiles. The method is tool-supported. Our tool supports developers in setting up the diagrams, and it checks different validation conditions concerning the semantic integrity and the coherence of the different diagrams. We illustrate the method by deriving an architecture for an automated teller machine.

9.1 Introduction

A thorough problem analysis is of invaluable benefit for the systematic development of high-quality software. Not only is there a considerable risk that software development projects fail when the requirements are not properly understood, but also the artefacts set up during requirements analysis can be used as a concrete starting point for the subsequent steps of software development, in particular, the development of software architectures.

In this chapter, we present a systematic method to derive software architectures from problem descriptions. We give detailed guidance by elaborating concrete

steps that are equipped with *validation conditions*. The method works for different types of systems, e.g., for embedded systems, web-applications, and distributed systems as well as standalone ones. The method is based on different kinds of patterns. On the one hand, it makes use of *problem frames* [1], which are patterns to classify simple software development problems. On the other hand, it builds on architectural and design patterns.

The starting point of the method is a set of diagrams that are set up during requirements analysis. In particular, a *context diagram* describes how the software to be developed (called *machine*) is embedded in its environment. Furthermore, the overall software development problem must be decomposed into simple subproblems, which are represented by *problem diagrams*. The different subproblems should be instances of problem frames.

From these pattern-based problem descriptions, we derive a software architecture that is suitable to solve the software development problem described by the problem descriptions. The problem descriptions as well as the software architectures are represented as UML diagrams, extended by stereotypes. The stereotypes are defined in profiles that extend the UML metamodel [2].

The method to derive software architectures from problem descriptions consists of three steps. In the first step, an initial architecture is set up. It contains one component for each subproblem. The overall machine component has the same interface as described in the context diagram. All connections between components are described by stereotypes (e.g., «call_and_return», «shared_memory», «event», «ui»).

In the second step, we apply different architectural and design patterns. We introduce coordinator and facade components and specify them. A facade component is necessary if several internal components are connected to one external interface. A coordinator component must be added if the interactions of the machine with its environment must be performed in a certain order. For different problem frames, specific architectural patterns are applied.

In the final step, the components of the intermediate architecture are re-arranged to form a layered architecture, and interface and driver components are added. This process is driven by the stereotypes introduced in the first step. For example, a connection stereotype «ui» motivates to introduce a user interface component. Of course, a layered architecture is not the only possible way to structure the software, but a very convenient one. We have chosen it because a layered architecture makes it possible to divide platform-dependent from platform-independent parts, because different layered systems can be combined in a systematic way, and because other architectural styles can be incorporated in such an architecture. Furthermore, layered architectures have proven useful in practice.

Our method exploits the subproblem structure and the classification of subproblems by problem frames. Additionally, most interfaces can be derived from the problem descriptions [3]. Stereotypes guide the introduction of new components. They also can be used to generate adapter components automatically. The re-use of components is supported, as well.

The method is tool-supported. We extended an existing UML tool by providing two new profiles for it. The first UML profile allows us to express the different

models occurring in the problem frame approach using UML diagrams. The second one allows us to annotate composite structure diagrams with information on components and connectors. In order to automatically validate the semantic integrity and coherence of the different models, we provide a number of validation conditions. The underlying tool itself, which is called UML4PF, is based on the Eclipse development environment [4], extended by an EMF-based [5] UML tool, in our case, Papyrus UML [6].

In the following, we first discuss the basic concepts of our method, namely problem frames and architectural styles (Sect. 9.2). Subsequently, we describe the problem descriptions that form the input to our method in Sect. 9.3. In Sect. 9.4, we introduce the UML profile for architectural descriptions that we have developed and which provides the notational elements for the architectures we derive. In Sect. 9.5, we describe our method in detail. Not only do we give guidance on how to perform the three steps, but we also give detailed validation conditions that help to detect errors as early as possible. As a running example, we apply our method to derive a software architecture for an automated teller machine. In Sect. 9.6, we describe the tool that supports developers in applying the method. Sect. 9.7 discusses related work, and in Sect. 9.8, we give a summary of our achievements and point out directions for future work.

9.2 Basic Concepts

Our work makes use of problem frames to analyse software development problems and architectural styles to express software architectures. These two concepts are briefly described in the following.

9.2.1 Problem Frames

Problem frames are a means to describe software development problems. They were proposed by Michael Jackson [1], who describes them as follows: *A problem frame is a kind of pattern. It defines an intuitively identifiable problem class in terms of its context and the characteristics of its domains, interfaces and requirement.* Problem frames are described by *frame diagrams*, which basically consist of rectangles, a dashed oval, and different links between them, see Fig. 9.1. The task is to construct a *machine* that establishes the desired behaviour of the environment (in which it is integrated) in accordance with the requirements.

Plain rectangles denote *domains* that already exist in the application environment. Jackson [1, p. 83f] considers three main domain types:

- “A **biddable domain** usually consists of people. The most important characteristic of a biddable domain is that it is physical but lacks positive predictable

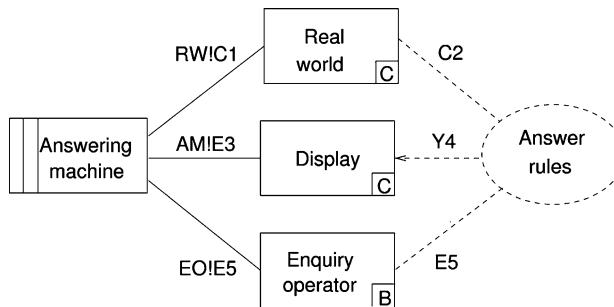


Fig. 9.1 *Commanded Information* problem frame

internal causality. That is, in most situations it is impossible to compel a person to initiate an event: the most that can be done is to issue instructions to be followed.”

Biddable domains are indicated by B (e.g., Enquiry operator in Fig. 9.1).

- “A **causal domain** is one whose properties include predictable causal relationships among its causal phenomena.” Often, causal domains are mechanical or electrical equipment. They are indicated with a C in frame diagrams. (e.g., Display in Fig. 9.1). Their actions and reactions are predictable. Thus, they can be controlled by other domains.
- “A **lexical domain** is a physical representation of data – that is, of symbolic phenomena. It combines causal and symbolic phenomena in a special way. The causal properties allow the data to be written and read.” Lexical domains are indicated by X.

A rectangle with a double vertical stripe denotes the machine to be developed, and *requirements* are denoted with a dashed oval. The connecting lines between domains represent interfaces that consist of *shared phenomena*. Shared phenomena may be events, operation calls, messages, and the like. They are observable by at least two domains, but controlled by only one domain, as indicated by an exclamation mark. For example, in Fig. 9.1 the notation EO! E5 means that the phenomena in the set E5 are *controlled* by the domain Enquiry operator and *observed* by the Answering machine.

To describe the problem context, a connection domain between two other domains may be necessary. Connection domains establish a connection between other domains by means of technical devices. Connection domains are, e.g., video cameras, sensors, or networks.

A dashed line represents a requirement reference, and an arrow indicates that the requirement *constrains* a domain.¹ If a domain is constrained by the requirement,

¹In the following, since we use UML tools to draw problem frame diagrams (see Figure 9.4), all requirement references will be represented by dashed lines with arrows and stereotypes <<refersTo>>, or <<constrains>> when it is constraining reference.

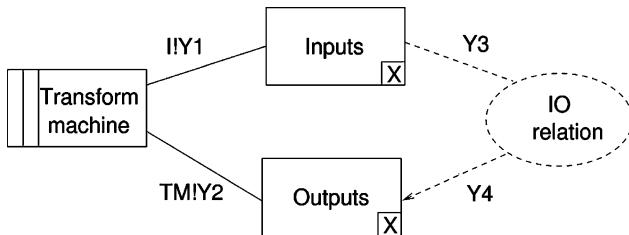


Fig. 9.2 Transformation problem frame

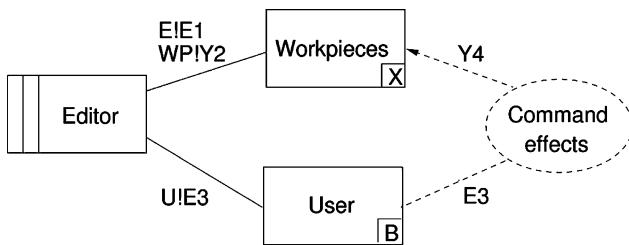


Fig. 9.3 Simple Workpieces problem frame

we must develop a machine, which controls this domain accordingly. In Fig. 9.1, the *Display* domain is constrained, because the *Answering* machine changes it on behalf of *Enquiry* operator commands to satisfy the required *Answer* rules.

The *Commanded Information* frame in Fig. 9.1 is a variant of the *Information Display* frame where there is no operator, and information about the states and behaviour of some parts of the physical world is continuously needed. We present in Fig. 9.4 the *Commanded Behaviour* frame in UML notation. That frame addresses the issue of controlling the behaviour of the controlled domain according to the commands of the operator. The *Required Behaviour* frame is similar but without an operator; the control of the behaviour has to be achieved in accordance with some rules. Other basic problem frames are the *Transformation* frame in Fig. 9.2 that addresses the production of required outputs from some inputs, and the *Simple Workpieces* frame in Fig. 9.3 that corresponds to tools for creating and editing of computer processable text, graphic objects etc.

Software development with problem frames proceeds as follows: first, the environment in which the machine will operate is represented by a *context diagram*. Like a frame diagram, a context diagram consists of domains and interfaces. However, a context diagram contains no requirements. Then, the problem is decomposed into subproblems. Whenever possible, the decomposition is done in such a way that the subproblems fit to given problem frames. To fit a subproblem to a problem frame, one must instantiate its frame diagram, i.e., provide instances for its domains, interfaces, and requirement. The instantiated frame diagram is called a *problem diagram*.

Besides problem frames, there are other elaborate methods to perform requirements engineering, such as i* [7], Tropos [8], and KAOS [9]. These methods are goal-oriented. Each requirement is elaborated by setting up a goal structure. Such a goal structure refines the goal into subgoals and assigns responsibilities to actors for achieving the goal. We have chosen problem frames and not one of the goal-oriented requirements engineering methods to derive architectures, because the elements of problem frames, namely domains, may be mapped to components of an architecture in a fairly straightforward way.

9.2.2 Architectural Styles

According to Bass, Clements, and Kazman [10],

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

Architectural styles are patterns for software architectures. A style is characterised by [10]:

- A set of component types (e.g., data repository, process, procedure) that perform some function at run-time,
- A topological layout of these components indicating their run-time interrelationships
- A set of semantic constraints (for example, a data repository is not allowed to change the values stored in it)
- A set of connectors (e.g., subroutine call, remote procedure call, data streams, sockets) that mediate communication, coordination, or cooperation among components.

When choosing a software architecture, usually several architectural styles are possible, which means that all of them could be used to implement the functional requirements. In the following, we will mostly use the *Layered* architectural style for the top-level architecture. The components in this layered architecture are either *Communicating Processes* (active components) or used with a *Call-and-Return* mechanism (passive components).² We use UML 2 composite structure diagrams to represent architectural patterns as well as concrete architectures.

²The mentioned architectural styles are described in [11].

9.3 Problem Description

To support problem analysis according to Jackson [1] with UML [2], we created a new UML profile. In this profile stereotypes are defined. A stereotype extends a UML meta-class from the UML meta-model, such as `Association` or `Class` [12].

In the following subsections, we describe our extensions to the problem analysis approach of Jackson (Sect. 9.3.1), we explain how the different diagrams can be created with UML and our profile (Sect. 9.3.2), we describe our approach to express connections between domains (Sect. 9.3.3), and we enumerate the documents that form the starting point for our architectural design method in Sect. 9.3.4. We illustrate these concepts on an ATM example in Sect. 9.3.5.

9.3.1 Extensions

In contrast to Jackson, we allow more than one machine domain in a problem diagram so that we can model distributed systems. In addition to Jackson's diagrams, we express technical knowledge (that we know or can acquire before we proceed to the design phases) about the machine to be built and its environment in a *technical context diagram* [3]. In a technical context diagram we introduce connection domains describing the direct technical environment of the machine, e.g., the platform, the operating system or mail server. Additionally, we annotate the technical realisation of all connections as described in Sect. 9.3.3. With UML it is possible to express aggregation and composition relations between classes and to use multiplicities. Thus we can express that one domain is part of another domain, e.g., that a lexical domain is part of the machine. UML distinguishes between active and passive classes. Active classes can initiate an action without being triggered before. Passive classes just react to some external trigger. Since domains are modelled as classes, they now can also be active or passive. Biddable domains are always active, and lexical domains are usually passive.

9.3.2 Diagram Types

The different diagram types make use of the same basic notational elements. As a result, it is necessary to explicitly state the type of diagram by appropriate stereotypes. In our case, the stereotypes are `<<ContextDiagram>>`, `<<ProblemDiagram>>`, `<<ProblemFrame>>`, and `<<TechnicalContextDiagram>>`. These stereotypes extend (some of them indirectly) the meta-class `Package` in the UML meta-model.

According to the UML superstructure specification [2], it is not possible that one UML element is part of several packages. For example a class `Customer` should

be in the context diagram package and also in some problem diagrams packages.³ Nevertheless, several UML tools allow one to put the same UML element into several packages within graphical representations. We want to make use of this information from graphical representations and add it to the model (using stereotypes of the profile). Thus, we have to relate the elements inside a package explicitly to the package. This can be achieved with a dependency stereotype «*isPart*» from the package to all included elements (e.g., classes, interfaces, comments, dependencies, associations).

The ***context diagram*** (see e.g., Fig. 9.8) contains the machine domain(s), the relevant domains in the environment, and the interfaces between them. Domains are represented by classes with the stereotype «*Domain*», and the machine is marked by the stereotype «*Machine*». Instead of «*Domain*», more specific stereotypes such as «*BiddableDomain*», «*LexicalDomain*» or «*CausalDomain*» can be used. Since some of the domain types are not disjoint, more than one stereotype can be applied on one class.

In a ***problem diagram*** (see e.g., Fig. 9.9), the knowledge about a sub-problem described by a set of requirements is represented. A problem diagram consists of sub-machines of the machines given in the context diagram, the relevant domains, the connections between these domains and a requirement (possibly composed of several related requirements), as well as of the relation between the requirement and the involved domains. A requirement refers to some domains and constrains at least one domain. This is expressed using the stereotypes «*refersTo*» and «*constrains*». They extend the UML meta-class *Dependency*. Domain knowledge and requirements are special statements. Furthermore, any domain knowledge is either a fact (e.g., physical law) or an assumption (usually about a user's behaviour).

The ***problem frames*** (patterns for problem diagrams) have the same kind of elements as problem diagrams. To instantiate a problem frame, its domains, requirement and connections have to be replaced by concrete ones. Figure 9.4 shows the commanded behaviour problem frame in UML notation, using our profile.

9.3.3 Associations and Interfaces

For phenomena between domains, we want to keep the notation introduced by Jackson. Our experience is that this notation is easy to read and easy to maintain. In Jackson's diagrams, interfaces between domains (represented as classes) show that there is at least one phenomenon shared by the connected classes. In UML, associations describe that there is some relation between two classes. We decided to use associations to describe the interfaces in Jackson's diagrams. An example for such an interface is depicted in Fig. 9.5. The association $AD! \{ showLog \}$ has the

³Alternatively, we could create several Customer classes, but these would have to have different names.

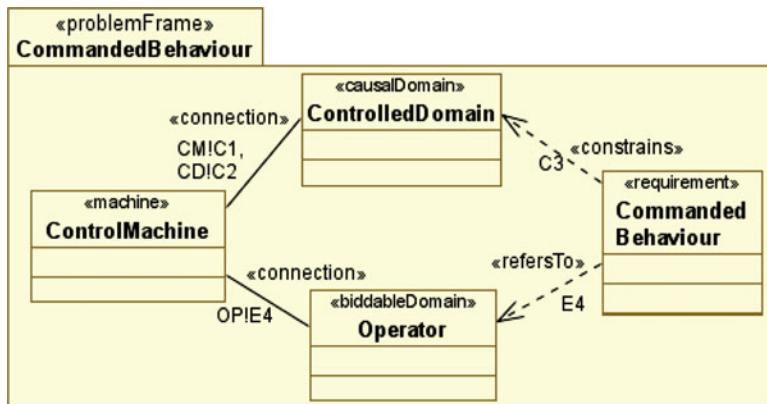


Fig. 9.4 *Commanded Behaviour* problem frame



Fig. 9.5 Interface class generation – drawn

stereotype «connection» to indicate that there are shared phenomena between the associated domains. The AdminDisplay controls the phenomenon showLog. In general, the name of the association contains the phenomena and the controlling domain. We represent different sets of shared phenomena with a different control direction between two domains by a second interface class.

Jackson's phenomena can be represented as operations in UML interface classes. The interface classes support the transition from problem analysis to problem solution. Some of the interface classes in problem diagrams become external interfaces of the architecture. In case of lexical domains, they may also be internal interfaces of the architecture. A «connection» can be transformed into an interface class controlled by a domain and observed by other domains. To this end, the stereotypes «observes» and «controls» are defined to extend the meta-class Dependency in the UML meta-model. The interface should contain all phenomena as operations. We use the name of the association as name for the interface class. Figure 9.6 illustrates how the connection given in Fig. 9.5 can be transformed into such an interface class.

To support a systematic architectural design, more specific connection types can be annotated in problem descriptions. Examples of such stereotypes which can be used instead of «connection» are, e.g., «network_connection» for network connections, «physical» or «electrical» for physical connections, and «ui» for user interfaces (see e.g., Fig. 9.8). Our physical connection can be specialised into hydraulic flow or hot air flow. These flow types are defined in SysML [13].

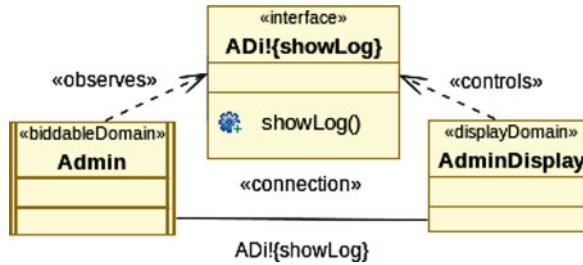


Fig. 9.6 Interface class generation – transformed

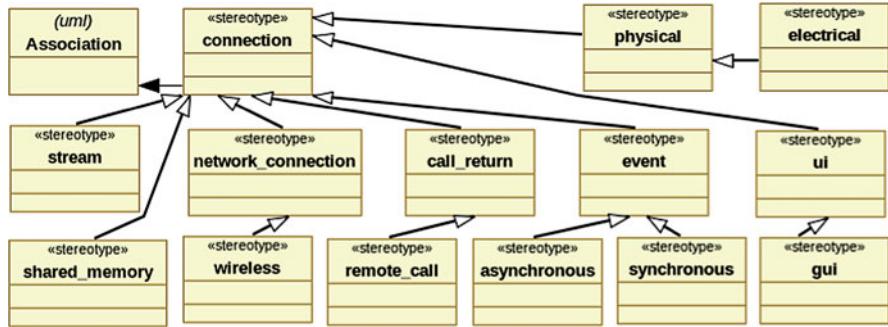


Fig. 9.7 Connection Stereotypes

For the control signal flow type in SysML, depending on the desired realisation, the stereotypes `<<network_connection>>`, `<<event>>`, `<<call_return>>`, or `<<stream>>` can be used. Figure 9.7 shows a hierarchy of stereotypes for connections. This hierarchy can be easily extended by new stereotypes.

For these stereotypes, more specialised stereotypes (not shown in Fig. 9.7) can be defined that consider the technical realisation, e.g. events (indicated with the stereotype `<<event>>`) can be implemented using Windows Message Queues (`<<wmq>>`), Java Events (`<<java_events>>`), or by a number of other techniques. Network connections (`<<network_connection>>`) can be realised, e.g., by HTTP (`<<http>>`) or the low-level networking protocol TCP (`<<tcp>>`).

9.3.4 Inputs and Prerequisites for Architectural Design

As a prerequisite, our approach needs a coherent set of requirements. The architectural design starts after the specification is derived and all *frame concerns* [1] have been addressed. To derive software architectures, we use the following diagrams from requirements analysis:

- Context diagram,
- Problem diagrams, and
- Technical context diagram

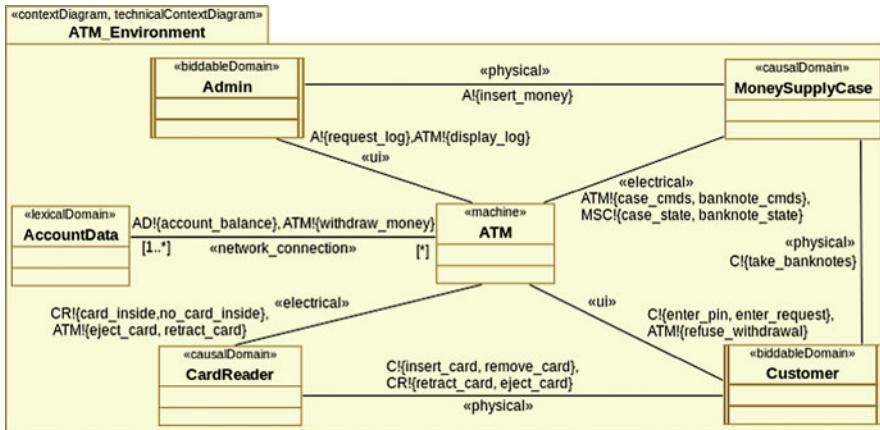


Fig. 9.8 The ATM context diagram/technical context diagram

Moreover, it may be necessary to know any restrictions that apply concerning the interaction of the machines with their environment. For example, in the automatic teller machine, the user must first authenticate before s/he may enter a request to withdraw a certain amount of money. In the following, we refer to this information as *interaction restrictions*.

9.3.5 The Automatic Teller Machine (ATM)

As a running example, we consider an automatic teller machine (ATM). Its context diagram – which is identical to the technical context diagram⁴ – is shown in Fig. 9.8. According to this diagram, Customers can interact with the ATM in the following way:

- Withdraw money by inserting their banking card into the CardReader (`insert_card`),
- Enter their PIN (`enter_PIN`),
- Enter a request to withdraw a certain amount of money (`enter_request`),
- Remove their card from the CardReader, and
- Take money from the MoneySupply_Case.

The ATM context diagram in Fig. 9.8 contains the ATM as the machine to be built. In the ATM environment, we can find the Admin responsible for checking the logs of the ATM with the phenomenon `request_log` and for filling the `MoneySupply_Case` with money (phenomenon `insert_money`).

⁴The technical context diagram is identical to the context diagram, because it is not necessary to describe new connection domains representing the platform or the operating system.

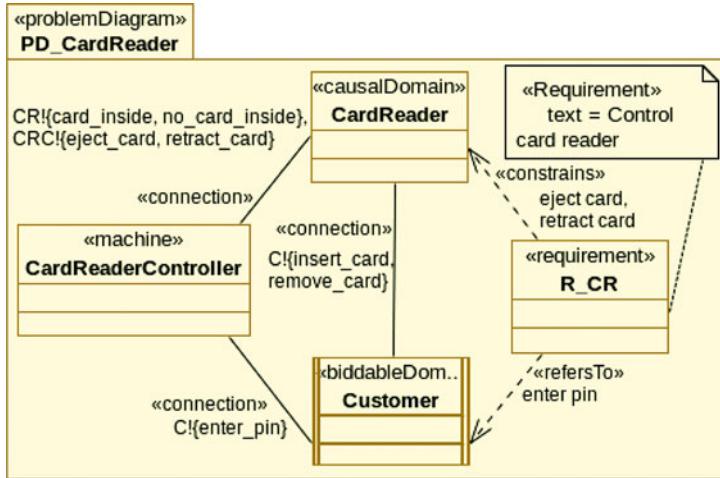


Fig. 9.9 Problem diagram for the card reader controller in UML notation

In some cases, it is possible that the ATM refuses a withdrawal (`refuse_withdrawal`). Each ATM is connected with the `AccountData` of at least one bank. We use multiplicities to express this aspect.

The different domains are annotated with appropriate specialised «domain» stereotype, e.g., the `Customer` is biddable and the `AccountData` is lexical. The connections are marked with specialisations of the stereotype «connection», e.g., a user interface («ui») between `Customer` and ATM, and a physical connection («physical») between `Customer` and `CardReader`.

The card reader controller subproblem in Fig. 9.9 is an instance of a variant of Commanded Behaviour (see Fig. 9.4). In this variant, we introduce a physical connection between the `Customer` and the `CardReader` that models the fact that the customer can physically insert a card into the card reader. Although the phenomena of that interface are used by the `CardReader` to inform the `CardReaderController` whether there is a card inside the card reader, they have no interface with the machine.

A subproblem problem diagram is given in Fig. 9.10. It concerns the `BankInformationMachine` and is an instance of a variant of the commanded information frame. (see Fig. 9.1).

The interfaces in context diagram are refined and split to obtain the interfaces in the problem diagrams. For example, `MSC!{ case_state, banknote_state }` is refined into `MSC!{ case_is_open, case_is_closed, banknotes_removed }`. Connection domains, e.g. a `AdminDisplay` are introduced. Additionally, domains are combined or split. For example, `MoneySupplyCaseCard Reader (MSCCR)` combines `MoneySupplyCase` (MSC) and `CardReader` (CR).

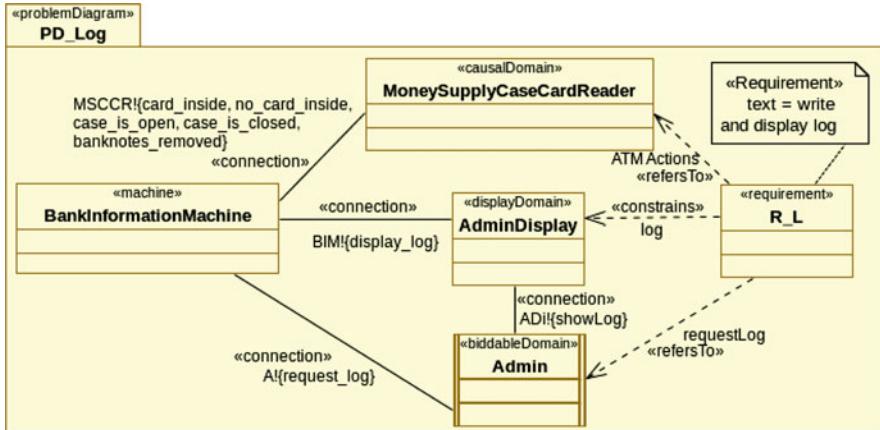


Fig. 9.10 Problem diagram for the administrator log check

9.4 Architectural Description

For each machine in the context diagram, we design an architecture that is described using composite structure diagrams [2]. In such a diagram, the components with their ports and the connectors between the ports are given. The components are another representation of UML classes. The ports are typed by a class that uses and realises interfaces. An example is depicted in Fig. 9.12. The ports (with this class as their type) provide the implemented interfaces (depicted as lollipops) and require the used interfaces (depicted as sockets), see Fig. 9.11.

In our UML profile we introduced stereotypes to indicate which classes are components. The stereotype «Component» extends the UML meta-class Class. For re-used components we use the stereotype «ReusedComponent», which is a specialisation of the stereotype «Component». Reused components may also be used in other projects. This fact must be recorded in case such a component is changed. A machine domain may represent completely different things. It can either be a distributed system (e.g., a network consisting of several computers), a local system (e.g., a single computer), a process running on a certain platform, or just a single task within a process (e.g., a clock as part of a graphical user interface). The kind of the machine can be annotated with the stereotypes «distributed», «local», «process», or «task». They all extend the UML meta-class Class.

For the architectural connectors, we allow the same stereotypes as for associations, e.g. «ui» or «tcp», described in Sect. 9.3.2. However, these stereotypes extend the UML meta-class Connector (instead of the meta-class Association).

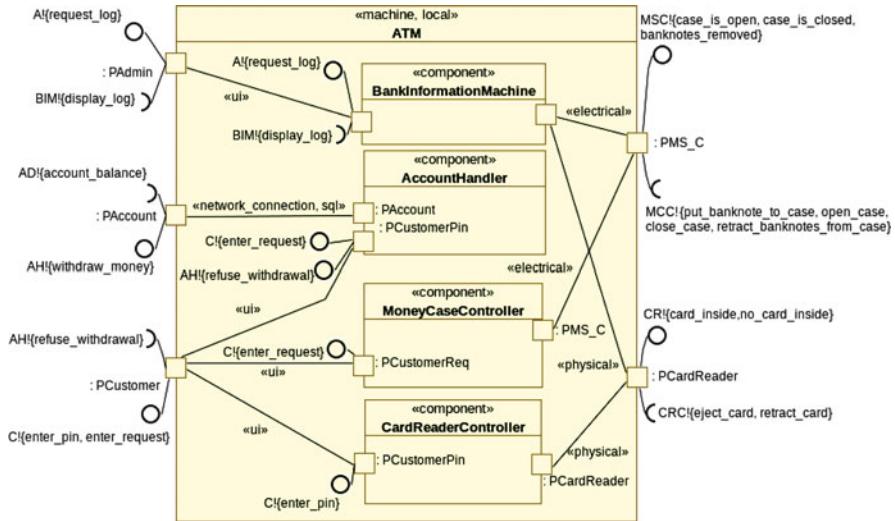


Fig. 9.11 The ATM initial architecture

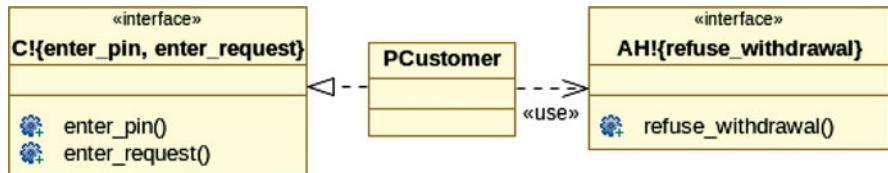


Fig. 9.12 Port Type of PCustomer

9.5 Deriving Architectures from Problem Descriptions

We now present our method to derive software architectures from problem descriptions in detail. For each of its three steps, we specify the input that is needed, the output that is produced, and a procedure that can be followed to produce the output from the input. Of course, these procedures are not automatic, and a number of decisions have to be taken by the developer. Such developer decisions introduce the possibility to make errors. To detect such errors as early as possible, each step of the method is equipped with validation conditions. These validation conditions must be fulfilled if the developed documents are semantically coherent. For example, a passive component cannot contain an active component. The validation conditions cannot be complete in a formal sense. Instead, they constitute necessary but not sufficient conditions for the different documents to be semantically valid. New conditions can be defined and integrated in our tool as appropriate.

Our method leads the way from problem descriptions to software architectures in a systematic way, which is furthermore enhanced with quality assurance measures and tool support (see Sect. 9.6).

9.5.1 Initial Architecture

The purpose of this first step is to collect the necessary information for the architectural design from the requirements analysis phase, to determine which component has to be connected to which external port, to make coordination problems explicit (e.g. several components are connected to the same external domain), and to decide on the machine type and to verify that it is appropriate (considering the connections). At this stage, the submachine components are not yet coordinated.

The *inputs* for this step are the technical context diagram and the problem diagrams. The *output* is an initial architecture, represented by a composite structure diagram. It is set up as follows. There is one *component* for a machine with stereotype «machine», and it is equipped with ports corresponding to the interfaces of the machine in the technical context diagram, see Fig. 9.11.

Inside this component, there is one component for each submachine identified in the problem diagrams, equipped with ports corresponding to the interfaces in the problem diagrams, and typed with a class. This class has required and provided *interfaces*. A controlled interface in a problem diagram becomes a required interface of the corresponding component in the architecture. Usually, an observed interface of the machine in the problem diagram will become a provided interface of the corresponding component in the architecture. However, if the interface connects a lexical domain, it will be a required interface containing operations with return values (see [14, Sect. 9.3.1]). The ports of the components should be connected to the ports of the machine, and stereotypes describing the technical realisation of these connectors are added. A stereotype describing the type of the machine (local, distributed, process, task) is added, as well as stereotypes «ReusedComponent» or «Component» to all components. If appropriate, stereotypes describing the type of the components (local, distributed, process, task) are also added.

The initial architecture of the ATM is given in Fig. 9.11. Starting from the technical context diagram in Fig. 9.8, and the problem diagrams (including the ones given in Figs. 9.9 and 9.10), the initial ATM architecture has one component, ATM, with stereotype «machine, local» and the ports (typed with :PAdmin, :PAccount, :PCustomer, :PMS_C, :PCardReader) that correspond to the interfaces of the machine in the technical context diagram. The components (CardReaderController, BankInformationMachine, MoneyCase Controller, and AccountHandler) correspond to the submachines identified for this case study (e.g., CardReaderController in Fig. 9.9, and BankInformationMachine in Fig. 9.10). Phenomena at the machine interface

in the technical context diagram (e.g. CR! { card_inside} , A! { request_log} , BIM! { display_log}) now occur in external interfaces of the machine. Phenomena controlled by the machine are associated with provided interfaces (e.g. BIM! { display_log}), and phenomena controlled otherwise (e.g. by the user), are associated with required interfaces (e.g., A! { request_log}).

Note that connections in the technical context diagram in Fig. 9.8 not related to the ATM (such as the one between Admin and MoneySupply_Case) are not reflected in this architecture.

The ports have a class as a type. This class uses and realises interfaces. For example, as depicted in Fig. 9.12, the class PCustomer uses the interface AH!{ refuse_withdrawal} and realises the class C!{ enter_pin, enter_request} . The ports with this class as a type provide the interface C! { enter_pin, enter_request} (depicted as a lollipop) and requires the interface HL! { refuse_withdrawal} (depicted as a socket).

We have defined two sets of validation conditions for this first phase of our method. The first set is common to all architectures (and hence should be checked after each step of our method), whereas the second one is specialised for the initial architecture. We give a selection of the validations conditions in the following. The complete sets can be found in [15].

Validation conditions for All architectures:

- VA.1 Each machine in all problem diagrams must be a component or a re-used component in the architectural description.
- VA.2 All components must be contained in a machine or another component.
- VA.3 For each operation in a *required* interface of a port of a component, there exists a connector to a port *providing* an interface with this operation, or it is connected to a re-used component.
- VA.4 The components' interfaces must fit to the connected interfaces of the machine, i.e., each operation in a required or provided interface of a component port must correspond to an operation in a required or provided interface of a connected machine port.
- VA.5 Passive components cannot contain any active components.
- VA.6 A class with the stereotype «Task» cannot contain classes with the stereotype «Process», «Local», or «Distributed».
A class with the stereotype «Process» cannot contain classes with the stereotype «Local» or «Distributed».
A class with the stereotype «Local» cannot contain classes with the stereotype «Distributed».

Validation conditions specific to the Initial architecture:

- VI.1 For each provided or required interface of machine ports in the architecture, there exists a corresponding interface in the technical context diagram.
- VI.2 For each machine in the technical context diagram:
Each stereotype name of all associations to the machine (or a specialization of this stereotype) must be included in the set of stereotype names of the

connectors from the internal components to external interfaces inside the machine.

- VI.3 Each stereotype name of the connectors from components to external interfaces inside an architectural machine component (or their supertypes) must be included in the set of associations to the corresponding machine domain in the technical context diagram.

As already noticed, these validation conditions can be checked automatically, using the tool described in Sect. 9.6.

9.5.2 *Intermediate Architecture*

The purpose of this step is to introduce coordination mechanisms between the different submachine components of the initial architecture and its external interfaces, thus obtaining an implementable intermediate architecture. Moreover, we exploit the fact that the subproblems are instances of problem frames by applying architectural patterns that are particularly suited for some of the problem frames. Finally, we decide whether the components should be implemented as active or passive components.

The *input* to this step are the initial architecture, the problem diagrams as instances of problem frames, and a specification of interaction restrictions⁵ (see Sect. 9.3.4). The *output* is an intermediate architecture that is already implementable. It contains coordinator and facade components as well as architectural patterns corresponding to the used problem frames. The intermediate architecture is annotated with the stereotype «intermediate_architecture» to distinguish it from the final architecture.

The intermediate architecture is set up as follows. When several internal components are connected to one external interface in the initial architecture, a *facade* component⁶ is added. That component has one provided interface containing all operations of some external port and several used interfaces as provided by the submachine components. In our ATM example, several components are connected with external interface :PCustomer in Fig. 9.11; therefore a CustomerFacade component is added in Fig. 9.13.

If interaction restrictions have to be taken into account, we need a component to enforce these restrictions. We call such a component an a *coordinator* component. A coordinator component has one provided interface containing all operations of some external port and one required interface containing all operations of some

⁵Our method does not rely on how these restrictions are represented. Possible representations are sequence diagrams, state machines, or grammars.

⁶See the corresponding design pattern by Gamma et al. [16]: “Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystems easier to use.”

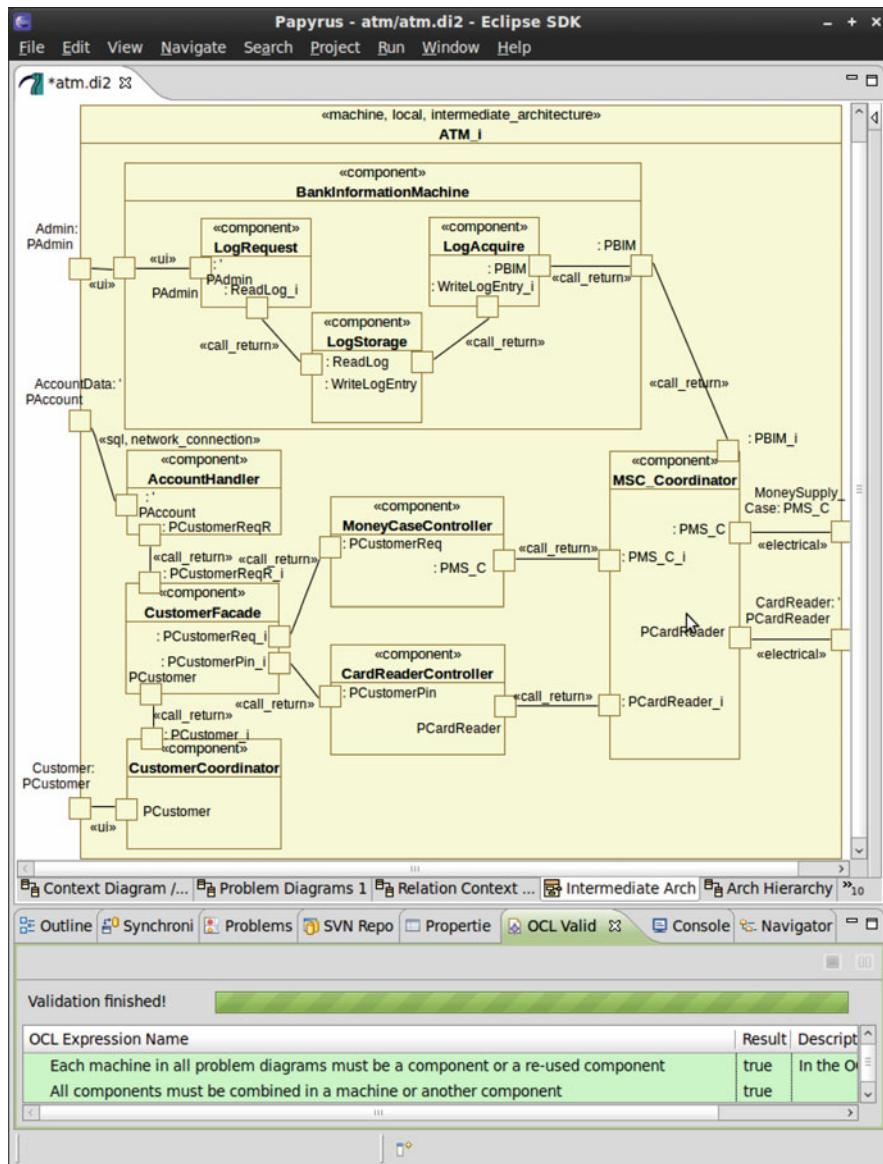


Fig. 9.13 Screenshot of the ATM intermediate architecture

internal port. To ensure the interaction restrictions, a state machine can be used inside the component. Typically, coordinator components are needed for interfaces connected to biddable domains (also via connection domains). This is because often, a user must do things in a certain order. In our example, a user must first authenticate before being allowed to enter a request to withdraw money. Therefore,

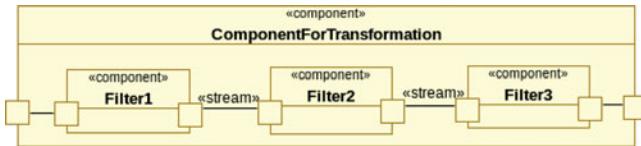


Fig. 9.14 Pattern for component realising transformation

we introduce a `CustomerCoordinator` in Fig. 9.13. Moreover, we need a `MSC_Coordinator` component, because money should only be put into the money supply case after the user has taken his or her card from the card reader.

Figure 9.13 also contains a sub-architecture for the component `BankInformationMachine`. This sub-architecture is an instance of the architectural pattern associated with the *commanded information* problem frame. This pattern contains components that are associated with the acquisition, the storage, and the request for information.

Figure 9.14 shows the architectural pattern for transformation problems. It is a pipe-and-filter architecture. The architectural pattern for the *required behaviour* frame (not shown here) requires the machine to be an active component.

After adding facade and coordinator components and applying architectural patterns related to problem frames, we have to decide for each component if it has to be active or not. In the case of the ATM, all components are reactive (even if the `CardReaderController` and the `MoneyCaseController` use timers for timeouts). For new connectors, their technical realisation should be added as stereotypes. For the ATM example, we use the stereotype `<<call_return>>`. Finally, for all newly introduced components it has to be specified if they are a `<<Component>>` or a `<<ReusedComponent>>`. In Fig. 9.13, we have no re-used components.

To validate the intermediate architecture, we have to check (among others) the following conditions (in addition to the conditions to given in Sect. 9.5.1).

Validation conditions for the interMediate architecture:

- VM.1 All components of the initial architecture must be contained in the intermediate architecture.
- VM.2 The connectors connected to the ports in the intermediate architecture must have the same stereotypes or more specific ones than in the initial architecture.
- VM.3 The stereotypes `<<physical>>` and `<<ui>>`, and their subtypes are not allowed between components.

9.5.3 Layered Architecture

In this step, we finalise the software architecture. We make sure to handle the external connections appropriately. For example, for a connection marked

«gui», we need a component handling the input from the user. For «physical» connections, we introduce appropriate driver components, which are often re-used.

We arrange the components in three layers. The highest layer is the *application layer*. It implements the core functionality of the software, and its interfaces mostly correspond to high-level phenomena, as they are used in the context diagram. The lowest layer establishes the connection of the software to the outside world. It consists of user interface components and *hardware abstraction layer* (HAL) components, i.e., the driver components establishing the connections to hardware components. The low-level interfaces can mostly be obtained from the technical context diagram. The middle layer consists of *adapter* components that translate low-level signals from the hardware drivers to high-level signals of the application components and vice versa. If the machine sends signals to some hardware, then these signals are contained in a required interface of the application component, connected to an adapter component. If the machine receives signals from some hardware, then these signals are contained in a provided interface of the application component, connected to an adapter component.

The *input* to this step are the intermediate architecture, the context diagram, the technical context diagram, and the interaction restrictions. The *output* is a layered architecture. It is annotated with the stereotype «layered_architecture» to distinguish it from the intermediate architecture. Note, however, that a layered architecture can only be defined for a machine or component with the stereotype «local», «process» or «task». For a distributed machine, a layered architecture will be defined for each local component.

To obtain the layered architecture, we assign all components from the intermediate architecture to one of the layers. The submachine components as well as the facade components will belong to the application layer. Coordinator components for biddable domains should be part of the corresponding (usually: user) interface component, whereas coordinator components for physical connections belong to the application layer. As already mentioned, connection stereotypes guide the introduction of new components, namely user interface and driver components. All components interfaces must be defined, where guidance is provided by the context diagram (application layer) and the technical context diagram (external interfaces).

The final software architecture of the ATM is given in Fig. 9.15. Note that we have two independent application components, one for the administrator and the other handling the interaction with the customers. This is possible, because there are no interaction restrictions between the corresponding subproblems. However, both applications need to access the log storage. Therefore, the component LogStorage does not belong to one of the application components. Each of the biddable domains Admin and Customer is equipped with a corresponding user interface. For the physical connections to the card reader and the money supply case, corresponding HAL and adapter components are introduced. Because the connection to the account data was defined to be a «network_connection» already in the initial architecture, the final architecture contains a DB_HAL component.

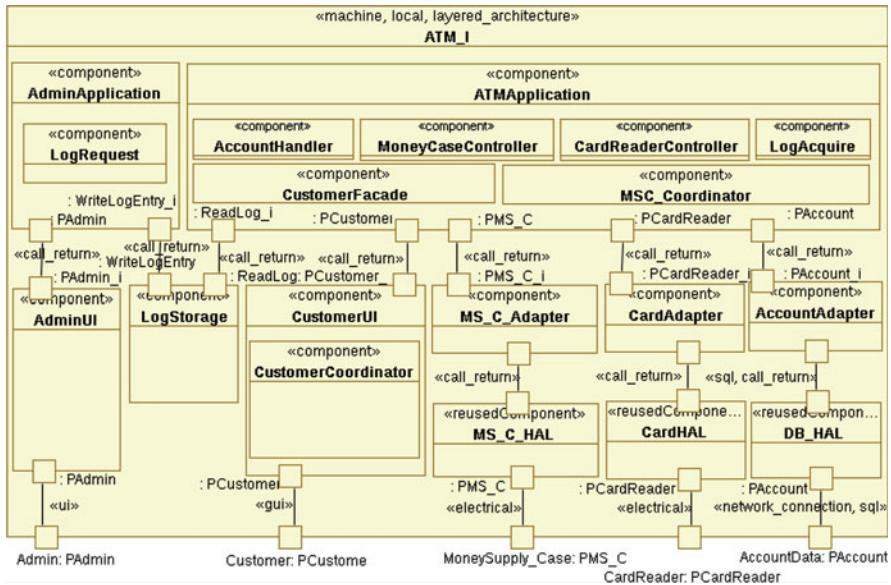


Fig. 9.15 The ATM layered architecture

The validation conditions to be checked for the layered architecture are similar to the validation conditions for the intermediate architectures. Conditions VM.3 must also hold for the layered architecture, and conditions VM.1 and VM.2 become

- VL.1 All components of the intermediate architecture must be contained in the layered architecture.
- VL.2 The connectors connected to the ports in the layered architecture must have the same stereotypes or more specific ones than in the intermediate architecture.

This final step could be carried out in a different way – resulting in a different final architecture – for other types of systems, e.g., when domain-specific languages are used.

9.6 Tool Support

The basis of our tool called *UML4PF* [17] is the Eclipse platform [4] together with its plug-ins EMF [5] and OCL [18]. Our UML-profiles described in Sects. 9.3 and 9.4 are conceived as an eclipse plug-in, extending the EMF meta-model. We store all our OCL constraints (which formalise the validation conditions given in Sect. 9.5) in one file in XML-format. With these constraints, we check the validity and consistency of the different models we set up during the requirements

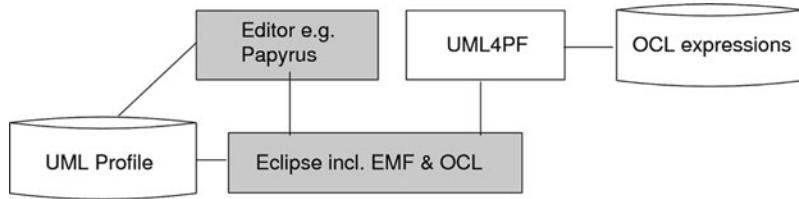


Fig. 9.16 Tool Realisation Overview

analysis and architectural design phases. An overview of the context of our tool is provided in Fig. 9.16. Gray boxes denote re-used components, whereas white boxes describe those components that we created.

The functionality of our tool UML4PF comprises the following:

- It checks if the developed models are valid and consistent by using our OCL constraints.
- It returns the location of invalid parts of the model.
- It automatically generates model elements, e.g., it generates observed and controlled interfaces from association names as well as dependencies with stereotype <<isPart>> for all domains and statements being inside a package in the graphical representation of the model.

The graphical representation of the different diagram types can be manipulated by using any EMF-based editor. We selected Papyrus UML [6], because it is available as an Eclipse plug-in, open-source, and EMF-based. Papyrus stores the model (containing requirements models and architectures) with references to the UML-profiles in one XML-file using EMF. The XML format created by EMF allows developers to exchange models between several UML tools. The graphical representation of the model is stored in separate file. Since UML4PF is based on EMF, it inherits all strengths and limitations of this platform. To use Papyrus with UML4PF to develop an architecture, developers have to draw the context diagram and the problem diagrams (see Sect. 9.3). Then they can proceed with deriving the specification, after UML4PF has generated the necessary model elements. Next, the requirements models are automatically checked with UML4PF.

Re-using model elements from the requirements models, developers create the architectures as described in Sect. 9.5. After each step, the model can be automatically validated. UML4PF indicates which model elements are not used correctly or which parts of the model are not consistent. Figure 9.13 shows a screenshot of UML4PF. As can be seen below the architectural diagram, several kinds of diagrams are available for display. When selecting the OCL validator, the validation conditions are checked, and the results are displayed as shown at the bottom of the figure. Fulfilled validation conditions are displayed in green, violated ones in red.

All in all, we have defined about 80 OCL validation conditions, including 17 conditions concerning architectural descriptions. The time needed for checking only depends on EMF and is about half a second per validation condition.

The influence of the model size on the checking time is less than linear. About 9,800 lines of code have been written to implement UML4PF.

The tool UML4PF is still under development and evaluation. Currently it is used in a software engineering class at the University Duisburg-Essen with about 100 participants. In this class, the problem frame approach and the method for architectural design described in this chapter are taught and applied to the development of a web application. The experience gained from the class will be used to assess (and possibly improve) the user-friendliness of the tool.

Moreover, UML4PF will be integrated into the tool WorkBench of the European network of excellence NESSoS (see <http://www.nessos-project.eu/>). With this integration, we will reach a wider audience in the future. Finally, the tool is available for download at <http://swe.uni-due.de/en/research/tool/index.php>.

9.7 Related Work

Since our approach heavily relies on the use of patterns, our work is related to research on problem frames and architectural styles. However, we are not aware of similar methods that provide such a detailed guidance for developing software architectures, together with the associated validation conditions.

Lencastre et al. [19] define a meta-model for problem frames using UML. Their meta-model considers Jackson's whole software development approach based on context diagrams, problem frames, and problem decomposition. In contrast to our meta-model, it only consists of a UML class model without OCL integrity constraints. Moreover, their approach does not qualify for a meta-model in terms of MDA because, e.g., the class `Domain` has subclasses `Biddable` and `Given`, but an object cannot belong to two classes at the same time (cf. Figs. 5 and 11 in [19]).

Hall et al. [20] provide a formal semantics for the problem frame approach. They introduce a formal specification language to describe problem frames and problem diagrams. However, their approach does not consider integrity conditions.

Seater et al. [21] present a meta-model for problem frame instances. In addition to the diagram elements formalised in our meta-model, they formalise requirements and specifications. Consequently, their integrity conditions (“wellformedness predicate”) focus on correctly deriving specifications from requirements. In contrast, our meta-model concentrates on the structure of problem frames and the different domain and phenomena types.

Colombo et al. [22] model problem frames and problem diagrams with SysML [13]. They state that “*UML is too oriented to software design; it does not support a seamless representation of characteristics of the real world like time, phenomena sharing [...]’*. We do not agree with this statement. So far, we have been able to model all necessary means of the requirements engineering process using UML.

Charfi et al. [23] use a modelling framework, *Gaspard2*, to design high-performance embedded systems-on-chip. They use model transformations to move

from one level of abstraction to the next. To validate that their transformations were performed correctly, they use the OCL language to specify the properties that must be checked in order to be considered as correct with respect to Gaspard2. We have been inspired by this approach. However, we do not focus on high-performance embedded systems-on-chip. Instead, we target general software development challenges.

Choppy and Heisel give heuristics for the transition from problem frames to architectural styles. In [24], they give criteria for choosing between architectural styles that could be associated with a given problem frame. In [25], a proposal for the development of information systems is given using *update* and *query* problem frames. A component-based architecture reflecting the repository architectural style is used for the design and integration of the different system parts. In [26], the authors of this paper propose architectural patterns for each basic problem frame proposed by Jackson [1]. In a follow-up paper [27], the authors show how to merge the different sub-architectures obtained according to the patterns presented in [26], based on the relationship between the subproblems. Hatebur and Heisel [3] show how interface descriptions for layered architectures can be derived from problem descriptions.

Barroca et al. [28] extend the problem frame approach with *coordination* concepts. This leads to a description of *coordination interfaces* in terms of *services* and *events* together with required properties, and the use of *coordination rules* to describe the machine behaviour.

Lavazza and Del Bianco [29] also represent problem diagrams in a UML notation. They use component diagrams (and not stereotyped class diagrams) to represent domains. Jackson's interfaces are directly transformed into used/required classes (and not observe and control stereotypes that are translated in the architectural phase). In a later paper, Del Bianco and Lavazza [30] suggest enhance problem frames with scenarios and timing.

Hall, Rapanotti, and Jackson [31] describe a formal approach for transforming requirements into specifications. This specification is then transformed into the detailed specifications of an architecture. We intentionally left out deriving the specification describing the dynamic behaviour within this chapter and focus on the static aspects of the requirements and architecture.

9.8 Conclusion and Perspectives

We have shown how software architectures can be derived in a systematic way from problem descriptions as they are set up during the requirements analysis phase of software development. In particular, our method builds on information that is elicited when applying (an extension of) the problem frame approach. The method consists of three steps, starting with a simple initial architecture. That architecture is gradually refined, resulting in a final layered architecture. The refinement is guided by patterns and stereotypes. The method is independent of

system characteristics – it works e.g., for embedded systems, for web-applications, and for distributed systems as well as for local ones. Its most important advantages are the following:

- The method provides a systematic approach to derive software architectures from problem descriptions. Detailed guidance is given in three concrete steps.
- Validation conditions for each step help to increase the quality of the results. These conditions can be checked automatically.
- The subproblem structure can be exploited for setting up the architecture.
- Most interfaces can be derived from the problem descriptions.
- Only one model is constructed containing all the different development artifacts. Therefore, traceability between the different models is achieved, and changes propagate to all graphical views of the model.
- Frequently used technologies are taken into account by stereotypes. The stereotype hierarchy can be extended for new developments.
- Stereotypes guide the introduction of new components.
- Adapters can be generated automatically (based on stereotypes).
- The application components use high-level phenomena from the application domain. Thus, the application components are independent of the used technology.
- Re-use of components is supported.

The method presented in this chapter can be extended to take other software development artefacts into account. For example, sequence diagrams describing the externally visible behaviour of machine domains can be used to derive behavioural descriptions of the architectural components. In the future, we will extend our approach to support the development of design alternatives according to quality requirements, such as performance or security, and to support software evolution. On the long run, the method can also be extended to cover further phases of the software development lifecycle.

Acknowledgments We would like to thank our anonymous reviewers for their careful reading and constructive comments.

References

1. Jackson M (2001) Problem Frames. Analyzing and structuring software development problems. Addison-Wesley, Boston
2. UML Revision Task Force (2009) OMG Unified Modeling Language: Superstructure, available at <http://www.omg.org/spec/UML/2.2/Superstructure/PDF/>, last checked: 2011-06-14
3. Hatebur D, Heisel M (2009) Deriving software architectures from problem descriptions. In: Software Engineering 2009 – Workshopband pp 383–302 GI
4. Eclipse – An open development platform (2008) May 2008. <http://www.eclipse.org/>, last checked: 2011-06-14
5. Eclipse Modeling Framework Project (EMF) (2008) May 2008. <http://www.eclipse.org/modeling/emf/>, last checked: 2011-06-14

6. Papyrus UML Modelling Tool (2010) Jan 2010. <http://www.papyrusuml.org>, last checked: 2011-06-14
7. Yu E (1997) Towards modelling and reasoning support for early-phase requirements engineering. In: Proceedings of the 3 rd IEEE Intern. Symposium on RE pp 226–235
8. Bresciani P, Perini A, Giorgini P, Giunchiglia F, Mylopoulos J (2004) Tropos: an agent oriented software development methodology. *Auton Agents Multi-Agent Syst* 8(3):203–236
9. Bertrand P, Darimont R, Delor E, Massonet P, van Lamsweerde A (1998) GRAIL/KAOS: an environment for goal driven requirements engineering. In: ICSE'98 – 20th International Conference on Software Engineering, New York
10. Bass L, Clements P, Kazman R (1998) Software architecture in practice. Addison-Wesley, Massachusetts, first edition
11. Shaw M, Garlan D (1996) Software architecture. Perspectives on an emerging discipline. Prentice-Hall, Upper Saddle River
12. UML Revision Task Force (2009) OMG Unified Modeling Language: Infrastructure. available at <http://www.omg.org/spec/UML/2.0/>, last checked: 2011-06-14
13. SysML Partners (2005) Systems Modeling Language (SysML) Specification. see <http://www.sysml.org>, last checked: 2011-06-14
14. Côté I, Hatebur D, Heisel M, Schmidt H, Wentzlaff I (2008) A systematic account of problem frames. In: Proceedings of the European conference on pattern languages of programs (EuroPLoP) Universitätsverlag Konstanz, pp 749–767
15. Choppy C, Hatebur D, Heisel M (2010) Systematic architectural design based on problem patterns (technical report). Universität Duisburg-Essen
16. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns – elements of reusable object-oriented Software. Addison Wesley, Reading, MA
17. UML4PF (2010) <http://swe.uni-due.de/en/research/tool/index.php>, last checked: 2011-06-14
18. UML Revision Task Force (2006) OMG Object Constraint Language: Reference. <http://www.omg.org/docs/formal/06-05-01.pdf>
19. Lencastre M, Botelho J, Clericuzzi P, Araújo J (2005) A meta-model for the problem frames approach. In: WiSME'05: 4th workshop in software modeling engineering
20. Hall JG, Rapanotti L, Jackson MA (2005) Problem frame semantics for software development. *Softw Syst Model* 4(2):189–198
21. Seater R, Jackson D, Gheyi R (2007) Requirement progression in problem frames: deriving specifications from requirements. *Requirements Eng* 12(2):77–102
22. Colombo P, del Bianco V, Lavazza L (2008) Towards the integration of SysML and problem frames. In: IWAAPF'08: Proceedings of the 3 rd international workshop on applications and advances of problem frames, New York, ACM, pp 1–8
23. Charfi A, Gamatié A, Honoré A, Dekeyser J-L, Abid M (2008) Validation de modèles dans un cadre d'IDM dédié à la conception de systèmes sur puce. In: 4èmes Journées sur l'Ingénierie Dirigée par les Modèles (IDM 08)
24. Choppy C, Heisel M (2003) Use of patterns in formal development: systematic transition from problems to architectural designs. In: recent trends in algebraic development techniques, 16th WADT, Selected Papers, LNCS 2755, Springer Verlag, pp 205–220
25. Choppy C, Heisel M (2004) Une approche à base de “patrons” pour la spécification et le développement de systèmes d’information. In: Proceedings approches formelles dans l’assistance au développement de Logiciels – AFADL’2004, pp 61–76
26. Choppy C, Hatebur D, Heisel M (2005) Architectural patterns for problem frames. *IEE Proc Softw Spec Issue Relating Softw Requirements Architectures* 152(4):198–208
27. Choppy C, Hatebur D, Heisel M (2006) Component composition through architectural patterns for problem frames. In: Proc. XIII Asia Pacific Software Engineering Conference (APSEC), IEEE, pp 27–34
28. Barroca L, Fiadeiro JL, Jackson M, Laney RC, Nuseibeh B (2004) Problem frames: a case for coordination. In: coordination models and languages, Proc. 6th international conference COORDINATION, pp 5–19

29. Lavazza L, Bianco VD (2006) Combining problem frames and UML in the description of software requirements. Fundamental approaches to software engineering. Lecture Notes in Computer Science 3922, Springer, pp 199–21
30. Lavazza L, Bianco VD (2008) Enhancing problem frames with scenarios and histories in UML-based software development. Expert Systems 25:28–53
31. Hall JG, Rapanotti L, Jackson M (2008) Problem oriented software engineering. Solving package router control problem, IEEE Transactions on Software Engineering 34(2):226–241

Chapter 10

Adaptation Goals for Adaptive Service-Oriented Architectures*

Luciano Baresi and Liliana Pasquale

Abstract Service-oriented architecture supports the definition and execution of complex business processes in a flexible way. A service-based application assembles the functionality provided by disparate, remote services in a seamless way. Since the architectural style prescribes that all features be provided remotely, these applications adapt to changes and new business needs by selecting new partner services to interact with. Despite the success of the architectural style, a clear link between the actual applications and the requirements they are supposed to meet is still missing. The embedded dynamism also imposes that requirements properly state how an application can evolve and adapt at runtime. We solve these problems by extending classical goal models to represent both conventional (functional and non-functional) requirements and adaptation policies. The goal model is then used to automatically devise the application's architecture (i.e., the composition) and its adaptation capabilities. It becomes a live, runtime entity whose evolution helps govern the actual adaptation of the application. All key elements are exemplified through a service-based news provider.

10.1 Introduction

In these years, Service-oriented Architecture (SoA) has proven its ability to support modern, dynamic business processes. The architectural paradigm fosters the provision of complex functionality by assembling disparate services, whose ownership – and evolution – is often distributed. The composition, oftentimes rendered in BPEL [18], does not provide a single integrated entity, but it only interacts with services that are deployed on remote servers. This way of working fosters reusability by gluing existing services, but it also allows one to handle new business needs by adding, removing, or substituting the partner services to obtain (completely)

*This research has been funded by the European Commission, Programmes: IDEAS-ERC, Project 227977 SMScom, and FP7/2007–2013, Projects 215483 S-Cube (Network of Excellence).

different solutions. So far the research in this direction has been focused on proposing more and more dynamic service compositions, neglecting the actual motivations behind them. How to implement a service-based application has been much more important than understanding what the solution has to provide and maybe how it is supposed to evolve and adapt. A clear link between the actual architectures – also referred to as service compositions – and the requirements they are supposed to meet is still missing. This lack obfuscates the understanding of the actual technological infrastructure that must be deployed to allow the application to provide its functionality in a robust and reliable way, but it also hampers the maintenance of these applications and the alignment of their functionality with the actual business needs. These considerations motivated the work presented in this chapter. We firmly believe that service-based applications must be conceived from clearly stated requirements, which in turn must be unambiguously linked to the services that implement them. Adaptation must be conceived as a requirement in itself and must be properly supported through the whole lifecycle of the system. It must cope with both the intrinsic unreliability of services and the changes imposed by new business perspectives. To this aim, we extend a classical goal model to provide an innovative means to represent both conventional (functional and non-functional) requirements and adaptation policies. The proposal distinguishes between crisp goals, the satisfiability of which is boolean, and fuzzy goals, which can be satisfied at different degrees; adaptation goals are used to render adaptation policies.

The information provided in the goal model is then used to automatically devise the application's architecture (i.e., the composition) and its adaptation capabilities. We assume the availability of a suitable infrastructure [5] – based on a BPEL engine – to execute service compositions. Goals are translated into a set of abstract processes (a-la BPEL) able to achieve the objectives stated in the goal model; the designer is in charge of selecting the actual composition that best fits stated requirements. Adaptation is supported by performing supervision activities that comprise data collection, to gather execution data, analysis, to assess the application's behavior, and reaction – if needed – to keep the application on track. This strict link between architecture and requirements and the need for continuous adaptation led us to consider the goal model a full-fledged runtime entity. Runtime data trigger the countermeasures embedded in adaptation goals, and thus activate changes in the goal model, and then in the applications themselves.

The chapter is organized as follows. Section 10.2 presents the goal model to express the requirements of the systems. Section 10.3 describes how goals are translated into service-based applications. Section 10.4 illustrates some preliminary evaluation; Sect. 10.5 surveys related works and Sect. 10.6 concludes the chapter.

10.2 Goal Model

This section introduces the goal model to represent requirements. Conventional (functional and non-functional) requirements are expressed by adopting KAOS [14], a well-known goal model, and RELAX [27], a relatively new notation for

expressing the requirements of adaptive systems. The goal model is also augmented with a new kind of goals, adaptation goals, which specify how the model can adapt to changes. The whole proposal is illustrated through the definition of a news provider called Z.com [8]. The provider wants to offer graphical news to its customers with a reasonable response time, but it also wants to keep the cost of the server pool aligned with its operating budget. Furthermore, in case of spikes in requests it cannot serve adequately, the provider commutes to textual content to supply its customers with basic information with acceptable delay.

10.2.1 KAOS

The main features provided by KAOS are goal refinement and formalization. Goal refinement allows one to decompose a goal into several conjoined sub-goals (AND-refinement) or into alternative sub-goals (OR-refinement). The satisfaction of the parent goal depends on the achievement of all (for AND-refinement) or at least one (for OR-refinement) of its underlying sub-goals. The refinement of a goal terminates when it can be “operationalized,” that is, it can be decomposed into a set of operations. Figure 10.1 shows the KAOS goal model of the news provider. The general objective is to provide news to its customers (G1), which is AND-refined into the following sub-goals: Find news (G1.1), Show news to requestors (G1.2), Provide high quality service (G1.3), and Maintain low provisioning costs (G1.4) (in terms of the number of servers used to provide the service). News can be

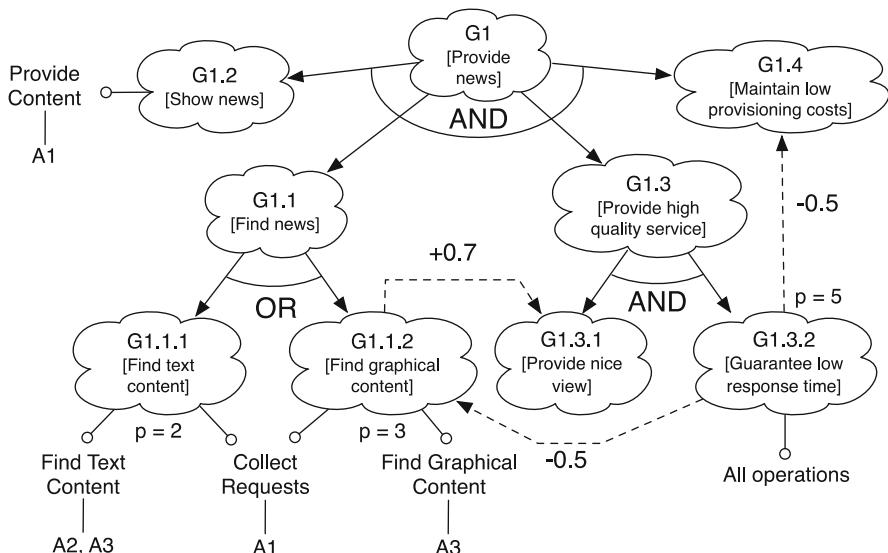


Fig. 10.1 The KAOS goal model of the news provider

Table 10.1 Definition of the example's goals

	$r: NewsReq, nc: NewsCollection,$ $ReceiveRequest(r) \wedge nc.keyword = "" \wedge nc.date = null \Rightarrow$ $\Diamond_{t < x} (\exists n \in nc.news / nc.keyword = r.keyword \vee nc.date = r.date \wedge n.text \neq null)$
G1.1.1	$r: NewsReq, nc: NewsCollection, ReceiveRequest(r) \wedge$ $nc.keyword = "" \wedge nc.date = null \wedge (r.keyword \neq "" \vee r.date \neq null) \Rightarrow$ $\Diamond_{t < x} (\exists n \in nc.news / nc.keyword = r.keyword \vee nc.date = r.date \wedge$ $n.text \neq "" \wedge (\exists i \in n.images))$
G1.1.2	$nc: NewsCollection, (\exists n \in nc.news) \Rightarrow \Diamond_{t < y} ShowNews(nc)$
G1.2	$r: NewsReq, nc: NewsCollection, ReceiveRequest(r) \wedge nc.keyword = "" \wedge$ $nc.date = null \wedge (r.keyword \neq "" \vee r.date \neq null) \Rightarrow \Diamond_{t < RT_MAX} ShowNews(nc)$
G1.3.2	servers: int, servers < N _{MAX}
G1.4.1	

provided both in textual and graphical mode (see OR-refinement of goal G1.1 into G1.1.1 and G1.1.2). Textual mode consumes less bandwidth and performs better in case of many requests. Customer satisfaction is increased by providing news in a nice format and within short response times (see AND-refinement of goal G1.3 into G1.3.1 and G1.3.2). G1.3.1 is a soft goal since there is not a clear-cut criterion to assess it, that is, whether news is provided in a nice way.

Goals are associated with a priority depending on their criticality. For example goal G1.1.1 has lower priority ($p = 2$) than goal G1.1.2 ($p = 3$), since providing news in graphical mode is more important than providing news in text mode. Goals can contribute (either positively or negatively) to the satisfaction of other goals. This is represented in the goal model through contribution links – dashed lines in Fig. 10.1 – and an indication of the contribution ($x \in [-1,1]$). For example, despite the graphical mode is slower, it positively contributes to the customer satisfaction (contribution link between goals G1.1.2 and G1.3.1). Short response times may require the adoption of the text mode to provide the news (see the negative link between goal G1.3.2 and goal G1.1.2) or may increase the provisioning costs since they may require a higher number of servers in the pool (see the link between goal G1.3.2 and G1.4).

Goals are formalized in Linear Temporal Logic¹ (LTL) [21] or First Order Logic (FOL). The definition of the leaf goals of Fig. 10.1 is reported in Table 10.1. For example, goal G1.1.2 states that if the system receives a request for a given keyword and date, it must provide related news within x time units. Provided news must be about supplied keyword and date, and must come with images. Note that we cannot provide a formal definition for goal G1.3.1, since it is soft. Instead, the satisfaction of this goal can be inferred from its incoming contribution links, by performing an arithmetic mean on the satisfaction of each contributing goal weighted by the value given to each contribution link. The formalism beneath the goals of Fig. 10.1 is a preliminary attempt to bridge the gap between the non-formal world of the stakeholders and the formal world of the machine. This

¹For this example, we use operator *sometimes in the future* (\Diamond).

Table 10.2 Definition of the example’s operations

Name:	Collect requests
In/Out:	$r: ReceiveRequest, nc: NewsCollection$
DomPre:	$nc.state = default$
DomPost:	$nc.state = req_initialized$
ReqPre:	$nc.keyword = "" \wedge nc.date = null \wedge (r.keyword \neq "" \vee r.date \neq null)$
TrigPre:	$ReceiveRequest(r)$
ReqPost:	$nc.keyword = r.keyword \wedge nc.date = r.date \wedge Collect(nc.keyword, nc.date)$
Name:	Find graphical content
In/Out:	$nc: NewsCollection$
DomPre:	$nc.state = req_initialized$
DomPost:	$nc.state = news_received$
ReqPre:	$nc.keyword = "" \wedge nc.date = null \wedge (r.keyword \neq "" \vee r.date \neq null)$
TrigPre:	$Collect(nc.keyword, nc.date)$
ReqPost:	$\forall n \in nc.news: n.keyword = nc.keyword \vee nc.date = n.date \wedge n.text \neq null \wedge (\exists i \in n.images / i.content \neq "") \wedge (\exists n \in nc.news)$
Name:	Provide content
Input:	$nc: NewsCollection$
ReqPre:	$(\exists n \in nc.news)$
TrigPre:	$@(nc.state = news_received)$
ReqPost:	$ShowNews(nc)$

formalism relies on background knowledge of the domain and may have several nuances of meaning.

Operationalization [14] is the process that allows one to (semi-automatically) infer the operations that “implement” goals, and thus in our work that partner services must provide. An operation is defined through name, input and output values, and pre- and post-conditions. Required preconditions (*ReqPre*) define when the operation can be executed. Triggering conditions (*TrigPre*) define how the operation is activated. Required post-conditions (*ReqPost*) define additional conditions that must be true after execution. Domain pre- (*DomPre*) and post-conditions (*DomPost*) define the effects of the operation on the domain. Table 10.2² shows the result of the operationalization applied to the case study (except for operation *Find Text Content*). For example, operation *Find Graphical Content* moves the system from a state in which a container for the news that have to be collected is initialized (*DomPre*) to a state in which a set of suitable news is available (*DomPost*). This operation is triggered as soon as the collection of news matching provided keyword and date is started (*TrigPre*). The effect of this operation is to collect a set of news that matches the date and keyword provided by the user (*ReqPost*). The definition of operation *Find Text Content* is similar to operation *Find Graphical Content* except for the required post-condition that is specified as follows:

²Operator @ has the following meaning 0: $@P \equiv \bullet(\neg P) \wedge P$

ReqPost:

$$\forall(n \in nc.news \mid n.keyword = nc.keyword \vee nc.date = n.date \wedge (\exists i \in n.images \mid i.content \neq "")) \wedge \exists n \in nc.news$$

The goal model also specifies a set of agents able to perform one or more operations. According to our point of view, agents represent the providers of the services that will be used in the composition. For example, agent A1 is the user issuing the requests and to whom news must be shown. Agent A3 can find news in both text and graphical mode, while agent A2 can only find news in text mode.

10.2.2 Fuzzy Goals

The definition of goals through LTL formulae allows one to assess whether a goal is satisfied, but there is no way to say if it is only satisfied partially. For example, the definition of goal G1.3.2 only allows one to assess whether the global response time does not exceed the maximum threshold (RT_{MAX}), but it provides no information about the distance between the actual value and RT_{MAX} . Furthermore the definition of goal G1.4 only specifies whether the number of servers is lower than a certain value N_{MAX} , but it says nothing about the actual number of servers used in the pool. These are only a couple of examples that made us introduce fuzzy goals, and express their satisfaction level through real numbers between 0 and 1. Fuzzy goals are rendered through the operators already introduced in RELAX [27] to represent non-critical requirements: AS EARLY/LATE AS POSSIBLE ϕ , for temporal quantities, AS CLOSE AS POSSIBLE TO q ϕ , to assess the proximity of quantities or frequencies (ϕ) to a certain value (q), AS MANY/FEW AS POSSIBLE ϕ , for quantities (ϕ). This way goals G1.3.2 and G1.4 can be redefined in terms of these operators as follows:

G1.3.2: AS EARLY AS POSSIBLE t

G1.4: AS FEW AS POSSIBLE servers

Goal G1.3.2 now says that the response time t must be as short as possible, while goal G1.4 says that the number of servers must be as low as possible. The assessment of goals G1.3.2 and G1.4 is guided by the membership functions shown in Fig. 10.2 that assign a satisfaction value between 0 and 1, depending on

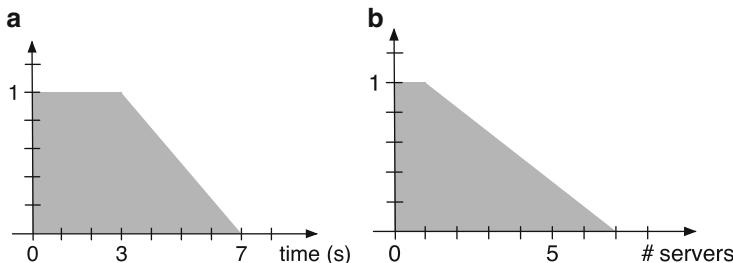


Fig. 10.2 Membership functions for goals G1.3.2 (a) and G1.4 (b)

the actual response time (Fig. 10.2a) and the number of used servers (Fig. 10.2b), respectively. For example, as for goal G1.3.2 if the response time is less than 3 s, the satisfaction is 1, if the response time is between 3 s and 7 s the satisfaction has a value between 0 and 1, and if the response time is greater than 7 s the satisfaction is 0. These functions are limited³ and, in general, have a triangular or trapezoidal shape. The severity of membership functions can be measured in terms of the gradient of the inclined sides. The severity can be tuned according to the priority of a goal (the higher the priority is, the steeper the membership function becomes).

10.2.3 Adaptation Goals

Adaptation goals augment the KAOS model to describe and tune the adaptation capabilities associated with the system-to-be that are necessary to react to changes or to the low satisfaction of conventional goals. An adaptation goal defines a sequence of corrective actions to preserve the overall objective of the system. Each adaptation goal is associated with a *trigger* and a set of *conditions*. The trigger states when the adaptation goal must be activated. Conditions specify further necessary restrictions that must be true to allow the corresponding adaptation actions to be executed. Conditions may refer to properties of the system (e.g., satisfaction levels and priorities of other goals, or adaptation goals already performed) or domain assumptions. Each adaptation goals is operationalized through *actions*.

- **Add, remove, or modify a conventional goal;**
- **Add, remove, or modify an adaptation goal;**
- **Add or remove an operation;**
- **Add or remove an entity;**
- **Perform an operation**, moves the process execution to the activity in which the operation, provided as parameter, starts to be performed (i.e., the first activity in the process flow associated with that operation);
- **Perform a goal**, moves the process execution to the activity in which the goal, provided as parameter, starts to be active (i.e., the first activity in the process flow associated with the first operation of the goal);
- **Substitute agent.**

Adaptation actions can be applied globally, on all (next/running) process instances, or locally (only on the application instance for which the triggers and conditions of that adaptation goal are satisfied). Adaptation goals may also conflict when they are associated with conflicting goals (i.e., a couple of goals linked by a contribution link with a negative weight). In this case, we trigger the adaptation goal associated with the goal with the highest priority.

³Membership functions do not continue to be greater than 0 when the response time is infinite.

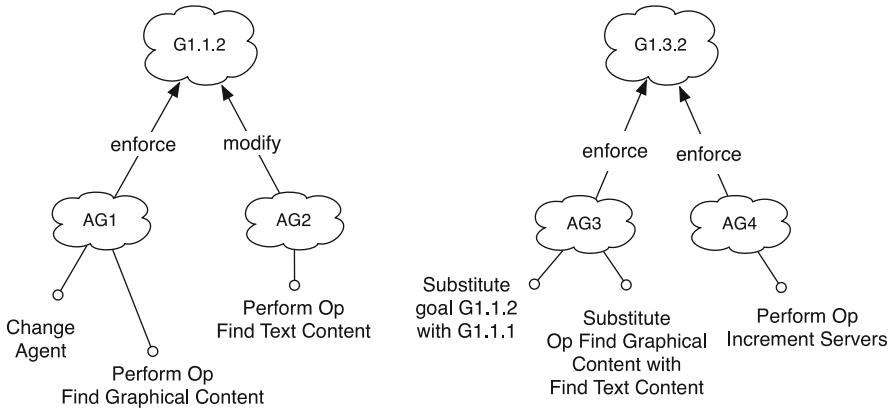


Fig. 10.3 Adaptation goals for the news provider

The adaptation goals envisioned for our example are shown in Fig. 10.3. Adaptation goals AG1 and AG2 are triggered when goal G1.1.2 is violated (i.e., its satisfaction is less than 1). AG1 is performed when the satisfaction of goal G1.1.2 is less than 0.7 and comprises two basic actions: it changes the agent that performs operation *Find Graphical Content* with another one (e.g., A5) and executes the same operation. These actions are applied locally, only for the instance of the goal model (and indeed, the process instance) for which the triggers and conditions hold true. The objective of this countermeasure is to enforce the satisfaction of goal G1.1.2. Adaptation goal AG2 is applied when the satisfaction of goal G1.1.2 is less than 0.7 and AG1 has been already applied. AG2 performs operation *Find Text Content*, and enforces a modified version of goal G1.1.2 (i.e., enforces goal G1.1.1 instead of G1.1.2). AG2 is also applied locally. Adaptation goals AG3 and AG4 are triggered when goal G1.3.2 is violated. In particular they are applied when the average value of the end-to-end response time of the news provider is greater than 3 s (conditions). AG3 enforces the satisfaction of goal G1.3.2 by switching to textual news (i.e., it substitutes goal G1.1.2 with goal G1.1.1 and operation *Find Graphical Content* with *Find Text Content*). AG3 is applied globally on all process instances. If AG3 is not able to reach its objective, AG4 is applied. Instead, it tries to enforce the satisfaction of goal G1.3.2, by incrementing the number of servers in the pool according to the severity of violation (it performs operation *Increment Servers*). Operation *Increment Servers* can only be performed by agent A4 and modifies the number of servers used by the load balancer. AG4 is also applied on all process instances. Adaptation goals AG1 is in conflict with AG3 and AG4 since they try to enforce conflicting goals. According to our policy, AG3 and AG4 are triggered first, since they are associated with goal G1.3.2, which has higher priority ($p = 5$) than G1.1.2 ($p = 3$).

10.3 From Goals to Self-adaptive Compositions

This section illustrates our proposal to transform the goal model into running, self-adaptive service-oriented compositions. The operationalization of conventional goals is used to derive suitable compositions, while adaptation goals help deploy probes needed to collect enough data for the runtime evaluation of goals' satisfaction. They are also in charge of adaptation actions.

10.3.1 Runtime Infrastructure

The runtime infrastructure works at two different levels of abstractions: process and goal level.

- The **process level** provides a BPEL engine to support the execution of the process instances. It also performs data collection and adaptation activities. Data collection activities gather the runtime data needed to update the state of entities, detect events, and evaluate the satisfaction of goals. Data to be collected can be internal (they belong to the process state), or external (they belong to the environment, and are retrieved by invoking external probes). Adaptation activities apply the actions associated with adaptation goals. Different probes and adaptation components can be easily plugged-in to obtain a complete execution platform.
- The **goal level** keeps a live goal model for each process instance, and updates it by means of the data collected at process level. Every time an instance of the goal model is updated, the infrastructure re-computes the satisfaction of conventional goals. Specific analyzers can be plugged-in to when necessary, depending on the kind of constraint (i.e., LTL, FOL, fuzzy) that must be evaluated to assess a goal. The goal level also evaluates the triggers and conditions of the adaptation goals and decides when adaptation must be performed. Adaptation actions can affect both the goal model and the process instances. The interplay between the goal and process levels is supported in the infrastructure by a bidirectional mapping between the elements of the two levels.

Figure 10.4 shows the overall architecture of the runtime infrastructure. The *BPEL Engine* is an instance of ActiveBPEL Community Edition Engine [1] augmented with aspects [13] to collect internal data and start/stop the process' execution when necessary. The *Data Collector* coordinates the different probes; the *Adaptation Farm* oversees the activities of recovery components. The *Supervision Manager*, based on JBoss rule engine [22], receives data from the process level, and triggers the updates of the goal level. Also the *Goal Reasoner* is based on JBoss rule engine: for each running process instance it keeps a goal model in its working memory and updates it. The *Goal Reasoner* asks the *Analysis Farm*, which coordinates analyzers, to (re-)compute the (degree of) satisfaction of the different

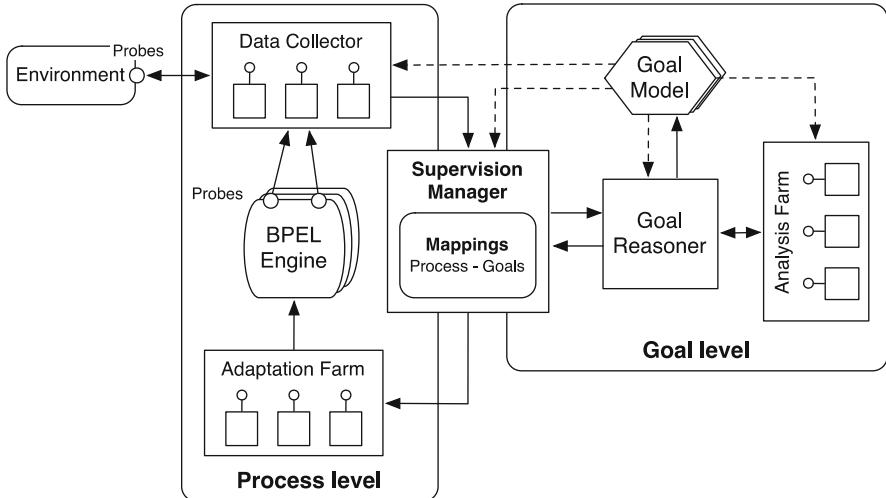


Fig. 10.4 Runtime infrastructure

leaf goals every time new data from the process level feed the goal model. The *Goal Reasoner* evaluates the triggers and conditions associated with adaptation goals and initiate their execution if needed. This means that the *Goal Reasoner* can modify the goal model and propagate the effects of adaptation at the process level. These effects are then applied onto the process instances by using the *Recovery Farm*, through the *Supervision Manager*.

10.3.2 Service Compositions

Service compositions are rendered as BPEL processes. Their activities, events, and partner services have a direct mapping onto the operations, entities, and agents of the goal model. Our assumption is that all operations associated with the same goal define a sequence and are not interleaved with the operations associated with other goals. The definition of a complete process requires the composition of these sequences and the transformation of their operations into the “corresponding” BPEL activities. Each sequence is defined by encoding the operations associated with each goal in Alloy to check whether there exists a possible sequence of operations whose execution guarantees the satisfaction of the corresponding goal. Interested readers can refer to [19] for a complete presentation. In general, a sequence s_1 can unconditionally precede s_2 if the ending operation of s_1 , op_1 , and the starting operation of s_2 , op_2 satisfy (10.1). While a sequence s_1 conditionally precedes s_2 if the ending operation of s_1 and the starting operation of s_2 , op_2 , satisfy (10.2). In this last case an *if* activity is inserted in the BPEL process between s_1 and s_2 , and its condition must correspond to the required precondition of op_2 .

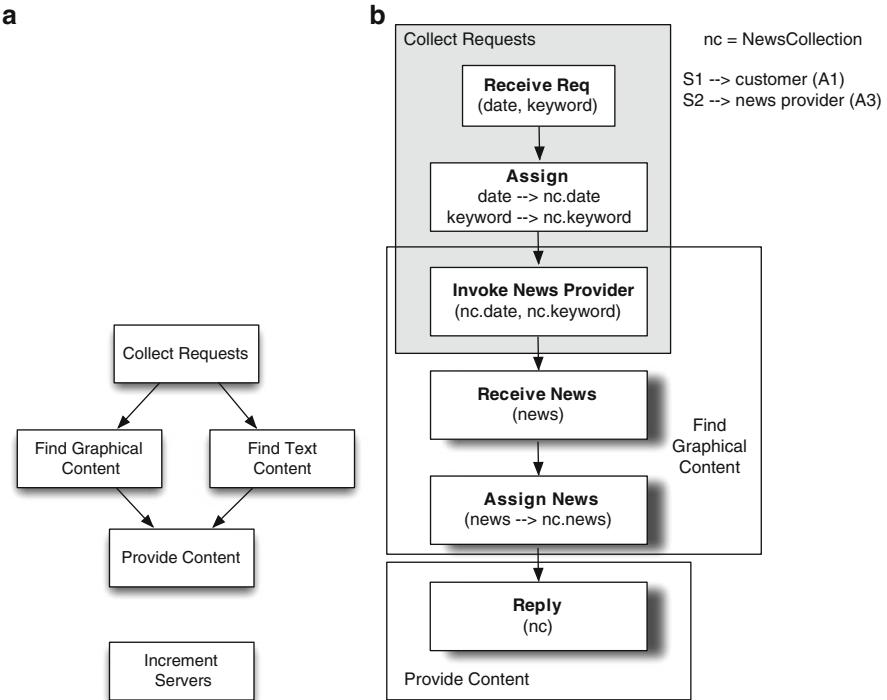


Fig. 10.5 (a) Two possible sequences of operations and (b) an abstract BPEL process

$$(domPost(op1) \rightarrow domPre(op2)) \wedge \\ (reqPost(op1) \rightarrow reqPost(op2) \wedge trigPre(op2)) \quad (10.1)$$

$$(domPost(op1) \rightarrow domPre(op2)) \wedge \\ (trigPre(op1) \rightarrow trigPre(op2)) \wedge (reqPre(op2) \rightarrow reqPost(op1)) \quad (10.2)$$

For example Fig. 10.5a shows two possible sequences of operations. Since operation *Find Text Content* and *Find Graphical Content* are mutually exclusive, we select the first one to satisfy goal G1.1.2 ($p = 3$). This is because it is more critical than goal G1.1.1 ($p = 2$), which is associated with operation *Find Text Content*. The generation of BPEL activities is semi-automatic. When an operation, in the goal domain, is translated into different sequences of BPEL activities, the user must select the most appropriate. Rules for translating operations into BPEL activities are the following:

1. If a required postcondition only contains an event, we generate one of the following activities: *invoke*, *invoke-receive*, or *reply*.
2. If the triggering precondition does not contain any event and the required postcondition changes some entities, we generate an *assign* for each change.

3. If the triggering precondition contains an event, it is translated into a *pick*. If the event refers to a temporal condition, it is translated into a *pick on_alarm*.
4. If rule 1 and 2 are true at the same time, we generate an *invoke-receive* followed by the set of *assigns*.
5. If rule 1 and 3 are verified at the same time, we generate an *invoke*, or a *pick*, or an *invoke-receive*.
6. If rule 2 and 3 are verified at the same time, we generate either a *pick* or a *receive*, and then a set of *assigns*.

The operations devised for the news provider are translated into the sequence of activities shown in Fig. 10.5a. Operations *Collect Requests* and *Find Graphical Content* follow rules 2 and 3. Since event *Collect* appears in the definition of both operations, *invoke News Provider* is generated only once. Operation *Provide Content*, instead, follows rule 1. The same applies also for those operations used in adaptation goals. For example, operation *Increment Servers* follows rule 1 and is simply translated into an *invoke*.

After identifying a proper sequence of BPEL activities, we must link entities and agents to proper process variables and partner links. Each entity used in the operationalization of conventional goals is rendered as an internal variable of the process. For example, our process has an internal variable called *NewsCollection*, which corresponds to entity *NewsCollection*. We also create a partner link for each agent and we assume that the user manually inserts its endpoint reference. In our example, we need partner services S1, S2, S3, S5 that match agents A1, A2, and A3, A5 respectively. Furthermore, we map agent A4 to another service S4 in charge of modifying the number of adopted servers.

10.3.3 Adaptation

The interplay between the process and goal levels is supported by the mapping of Table 10.3. Each conventional goal, which represents a functional requirement (i.e., it is operationalized), is mapped onto the corresponding *sequence* activity in the BPEL process (XPath expression). If the goal represents a non-functional requirement, but its nearest ancestor goal is operationalized, it is associated with the same *sequence* of its parent goal. The XPath expression provides the scope for both

Table 10.3 Mapping goals to runtime data

Conventional goal (leaf)	XPath to the sequence in the BPEL process
Operation	<ul style="list-style-type: none"> • XPath to the first activity associated with the operation • XPath to the last activity associated with the operation
Agent	Partner link
Entity	Internal or external data
Event	XPath to a corresponding process activity
Adaptation goal	Recovery actions at process level

possible adaptation actions and for assessing the satisfaction of the goal (i.e., it defines the activities that must be probed to collect relevant data). Each operation is associated with the first and the last BPEL activities, associated with it through two XPath expressions. Each agent is associated with a partner service; the user manually inserts the actual binding. All events are mapped to an XPath pointing to the corresponding activity in the BPEL process. This activity must represent an interaction of the process with its partner services (e.g., *invoke*, *pick*, *receive*). Each adaptation goal is associated with a set of actions that must be performed at process level.

Data collection specifies the variables that must be collected at runtime to update the live instance of the goal model associated with the process instance. Data are collected by a set of probes that mainly differ on how (push/pull mode), and when (periodically/when certain events take place) data must be collected. If data are collected in push mode, the *Supervision Manager* just receives them from the corresponding probes, while if they are collected in pull mode, the *Supervision Manager* must activate the collection (periodically or at specific execution points) through dedicated rules. To evaluate the degree of satisfaction of each goal, its formal definition must be properly translated to be evaluated by the selected analyzer. The infrastructure provides analyzers for FOL and LTL expressions, for crisp goals, and also provide analyzers to evaluate the actual satisfaction level of fuzzy goals. To this end, we built on our previous work and exploit the monitoring components provided by ALBERT [3], for LTL expressions, and Dynamo [4] for both FOL expressions and fuzzy membership functions.

To enact adaptation goals at runtime, the *Goal Reasoner* evaluates a set of rules on the live instances of the goal model available in its working memory. Each adaptation goal is associated with three kinds of JBoss rules. A **triggering rule**, activates the evaluation of the trigger associated with the goal. A **condition rule** evaluates the conditions linked to the goal. If the two previous rules provide positive feedback, an **activation rule** is in charge of the actual execution of the adaptation actions. They are performed when an adaptation goal can potentially fire (i.e., the corresponding Activation fact is available in the working memory) and is selected by the rule engine to be performed, among the other adaptation goals that can be performed as well. It executes the actions associated with that adaptation goal. For example, the triggering rule associated with AG1 is the following:

when

Goal(id=="G1.1.2", satisfaction < 1, \$pid: pID)

then

wm.insert(new Trigger("TrigAG1", pid));

It is activated when the satisfaction of goal G1.1.2 is less than 1. This rule inserts a new *Trigger* fact in the working memory of the Goal Reasoner, indicating that the trigger associated with adaptation goal AG1 is satisfied for process instance *pid*.

The corresponding condition rule is:

when

```
$t: Trigger(name == "TrigAG1", $pid: pID)
Goal(id=="G1.1.2", satisfaction < 0.7, pID == pid)
$adGoal: AdaptationGoal(name=="AG1",
$maxNumAct: maxAct, numOfActivations < $maxNumNAct)
```

then

```
wm.remove($t); wm.insert(new Activation("AG1", pid));
```

It is activated when the condition associated with AG1 (the satisfaction of goal G1.1.2 is less than 0.7) is satisfied, the trigger of AG1 has taken place, and AG1 has been tried less than a maximum number of times (*maxNumAct*). It inserts a new fact in the working memory (*Activation*), to assert that the adaptation actions associated with goal AG1, for the process instance and the goal model corresponding to *pid* can be performed.⁴ The action rule is:

salience 3

activation-group recovery

when

```
$a: Activation(name == "AG1", $pid: pID)
$ag: AdaptationGoal(name=="AG1", pID == pid)
```

then

```
List<Action> actions = new ArrayList<Action>();
actions.add(new SubstituteAgent("A3","A5"));
actions.add(new Perform("Find Graphical Content"));
ag.numOfActivations++;
Adaptation adapt = new Adaptation("AG1", actions, "instance", pid);
adapt.perform(); wm.remove(a);
```

Action rules have a priority (*salience*) equal to that of the goal they refer to (G1.1.2, in this case) and are always associated with activation-group *recovery*. This means that, once the rule with the highest priority fires, it automatically cancels the execution of the other adaptation goals that could be performed at the same time. Adaptation actions are performed when the triggers and conditions of the adaptation goal are satisfied (e.g., the corresponding activation object (*a*) is asserted in the working memory). The example rule performs the adaptation actions (*adapt.perform()*) on process instance (*pid*). Finally, it removes the object (*a*) that activated this adaptation.

⁴Note that if an adaptation goal is applied globally, there is no need to identify the process instance on which adaptation must be performed.

Adaptation actions associated with AG1 have no consequences on the goal model since they only require that the process re-execute the activities associated with operation *Find Graphical Content* by using another agent. At the process level these actions are applied locally and substitute partner service S1 with another one and restore the execution to operation *Find Graphical Content*. This is achieved through a Dynamo recovery directive that configures AOP probes to intercept the process execution before activity *invoke News Provider* and invoke operation *rebind(S3, S2.wsdl)*, that takes in input the name of the partner service to be substituted and the wsdl exposed by the new partner service to be adopted. After this operation is performed, the execution can proceed. This is only feasible with stateless services: in general, the application of an adaptation action cannot compromise the internal state of the process and that of its partner services.

If we consider adaptation goal AG3, it is applied globally and substitutes goal G1.1.2 and operation *Find Graphical Content* with goal G1.1.1 and operation *Find Text Content*, respectively. To this aim, we deploy a new version of the process, shown in Fig. 10.6, for the next process instances. To apply AG3 on the running process instances we intercept the process execution just before activity *invoke News Provider* is performed. If a process instance has overtaken this execution point, it cannot be migrated. At this point, we substitute the activities associated with operation *Find Graphical Content* with the activities of the alternative

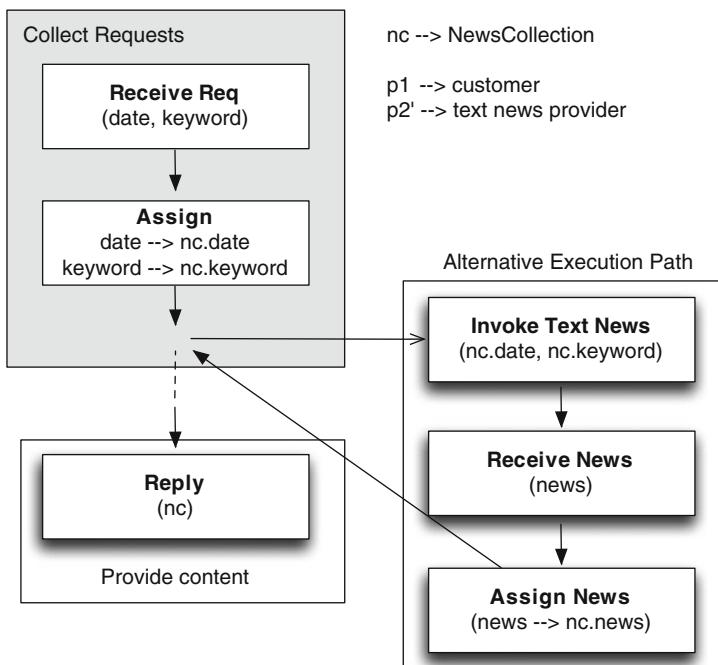


Fig. 10.6 Adapted process for Z.com

execution path, shown in Fig. 10.6. Then, the process execution proceeds, performing the activities of the alternative execution path.

10.4 Preliminary Validation

The validity of the proposed goal model has been evaluated by representing some example applications commonly used by other approaches proposed to model self-adaptive systems: an intelligent laundry [6], a book itinerary management system [23], and a garbage cleaner [16]. These experiments said that our goal model proved to be expressive enough to represent the main functionality of these systems together with their adaptation scenarios.

In the first case study, a laundry system must distribute assignments to the available washing machines and activate their execution. The system must also guarantee a set of fuzzy requirements stating that the energy consumed must not exceed a maximum allowed and the number of clothes that have to be washed must be low. These requirements are fuzzy since their satisfaction depends on the number of clothes to be washed and the amount of energy consumed, respectively. The satisfaction level of the energy consumed allows us to tune the duration of the washing programs accordingly. The adaptation goals devised for this case study also allow us to detect transient failures (e.g. the washing machine turns off suddenly) and activate an action that performs an operation to restart a washing cycle.

The itinerary booking system must help business travellers book their travels and receive updated notifications about the travel status (e.g., delays, cancelled flights). These notifications can be sent via email or SMS depending on the device the customer is actually using (i.e., laptop or mobile phone). Since sending an SMS is the most convenient option, we decided to adopt it in the base goal model of this case study. Suitable adaptation goals allow us to detect, through a trigger (i.e., whether the mobile phone is turned off) and a condition (i.e., whether the email of the customer's secretary is part of the information provided by the customer), when the customer's mobile phone is turned off, and apply an adaptation action that sends an email to him/her.

In the cleaner agent scenario, each agent is equipped with a sensor to detect the presence of dust and its driving direction. In case an agent finds a dirty cell, it must clean it, putting the dust in its embedded dustbin. The adaptation goals envisioned for this example allow the cleaner agent to recharge its battery when the load level is low. Furthermore, they allow us to cover a set of failure prevention scenarios. For example, adaptation goals can detect known symptoms of battery degeneration (e.g., suddenly reduced lifetime or voltage) and perform an operation to alert a technician, or get a new battery. Adaptation goals can also detect the presence of obstacles in the driving direction of an agent and activate two actions: stop the agent and change the driving direction, when possible.

These exercises demonstrated to be very useful to highlight both the advantages and disadvantages of our approach. We can perform accurate and precise

adaptations by assessing the satisfaction degree of soft goals and tuning the adaptation parameters accordingly, as described before. The usage of triggers and conditions makes it possible to react after system failures or context changes, and also model preventive adaptations to avoid a failure when known symptoms take place.

We adopt a priority-based mechanism to solve conflicts among adaptations that can be triggered at the same time. This mechanism is still too simplistic in certain situations. For example a vicious cycle may exist when a countermeasure A has a negative side effect on another goal, and that goal's countermeasure B has a negative side effect on the first goal as well. These cases can be handled by tuning the conditions of the countermeasures involved, which would become pretty complex. For this reason, other decision-making mechanisms should be adopted, like trade-off optimization functions. Finally our goal model does not provide any reasoning mechanism to automatically detect possible adaptations in advance, after changes in the context and in the stakeholders' requirements take place.

10.5 Related Work

Our proposal aims to provide a goal-based methodology to model the requirements of service compositions, that is, the architecture of service-based applications. Cheng et al. [7] proposed a similar approach for self-adaptive systems in general. The authors detect the reasons (threats) that may cause uncertainty in the satisfaction of goals, and propose three strategies for their mitigation: add new functionality, tolerate uncertainty, or switch to a new goal model that is supposed to repair the violation. Instead our strategies do not constraint the ways a goal model can be modified, but they can have different objectives and severity. These features allow us to solve conflicts among strategies and provide ways to apply them at runtime. Also Goldsby et al. [10] use goal models to represent the non-adaptive behavior of the system (business logic), the adaptation strategies (to handle environmental changes) and the mechanisms needed by the underlying infrastructure to perform adaptation. These proposals only handle adaptation by enumerating all alternative paths at design time. In contrast, we support the continuous evolution of the goal model by keeping a live goal model for each process instance and by modifying it at runtime. Different works have tried to link compositions with the business objectives they have to achieve. For example, Kazhamiakin et al. [12] adopt Tropos to specify the objectives of the different actors involved in choreography. Tropos tasks are refined into message exchanges, suitable annotations are added to express conditions on the creation and fulfillment of goals, and assume/guarantee conditions are added to the tasks delegated to partner services. These elements enable the generation of annotated BPEL processes. These processes can only be verified statically through model checking, ours also embed self-adaptation capabilities. Another similar approach is the one proposed by Mahfouz et al. [15], which models the goals of each actor and also the dependences among them. Actor dependencies

take place when a task performed by an actor depends on another task performed by a different actor. Dependencies are then translated into message sequences exchanged between actors, and objectives into sets of local activities performed in each actor's domain. The authors also propose a methodology to modify choreography according to changes in the business needs (dependencies between actors and local objectives). Although this approach traces changes at requirements level, it does not provide explicit policies to apply these changes at runtime.

The idea of monitoring requirements was originally proposed by Fickas et al. [9]. The authors adopt a manual approach to derive monitors able to verify requirements' satisfaction at runtime. Wang et al. [26] use the generation of log data to infer the denial of requirements and detect problematic components. Diagnosis is inferred automatically after stating explicitly what requirements can fail. Robinson [24] distinguishes between the design-time models, where business goals and their possible obstacles are defined, and the runtime model, where logical monitors are automatically derived from the obstacles and are applied onto the running system. This approach requires that diagnostic formulae be generated manually from obstacle analysis. Despite a lot of work focused on monitoring requirements, only few of them provide reconciliation mechanisms when requirements are violated. Wang et al. [26] generate system reconfigurations guided by OR-refinements of goals. They choose the configuration that contributes most positively to the non-functional requirements of the system and also has the lowest impact on the current configuration. To ensure the continuous satisfaction of requirements, one needs to adapt the specification of the system-to-be according to changes in the context. This idea was originally proposed by Salifu et al. [25] and was extensively exploited in different works [2, 20] that handled context variability through the explicit modeling of alternatives. Penserini et al. [20] model the availability of execution plans to achieve a goal (called ability), and the set of pre-conditions and context-conditions that can trigger those plans (called opportunities). Dalpiaz e al. [2] explicitly detect the parameters coming from the external environment (context) that stimulate the need for changing the system's behavior. These changes are represented in terms of alternative execution plans. Moreover the authors also provide precise mechanisms to monitor the context. All these works are interesting since they address adaptation at requirements level, but they mainly target context-aware applications and adaptation. They do not consider adaptations that may be required by the system itself because some goals cannot be satisfied anymore, or new goals are added. We foresee a wider set of adaptation strategies and provide mechanisms to solve conflicts among them.

Despite our solution is more tailored to service-based applications, many works [11, 16] focus on multi agent systems (MAS). Morandini et al. [16], like us, start from a goal model, Tropos4AS [17] which enriches TROPOS with soft goals, environment entities, conditions relating entities and state transitions, and undesired error states. The goal model is adopted to implement the alternative system behaviors that can be selected given some context conditions. Huhns et al. [11] exploit agents to support software redundancy, in terms of different implementations, and provide software adaptation. The advantage here is that agents can be

added/removed dynamically; this way, the software system can be customized at runtime and become more robust. The main advantage of these agent-based systems is their flexibility, since adaptation actions are applied at the level of each single component. On the other hand, MAS provide no guarantees that agents cannot perform conflicting actions or that the main system's objectives are always achieved. Our approach, instead, is centralized and declares adaptation actions at the level of the whole system. Adaptation is simply achieved by adding, removing, and substituting components, since the SOA paradigm does not allow us to change the internal behavior of a component.

10.6 Conclusions

This chapter proposes an innovative approach to specify adaptive service-oriented architectures/applications. The work extends the KAOS goal model and accommodates both conventional (functional and non-functional) requirements and the requirements on how the system is supposed to adapt itself at runtime. Goals can be *crisp*, when their satisfiability is boolean, *fuzzy*, when they can also be partially satisfied, and related to *adaptation*, when they specify adaptation policies. The proposal also explains how to map the “comprehensive” goal model onto the underlying architecture. Conventional goals are used to identify the best service composition that fits stated requirements. Adaptation goals are translated in data collection directives and sequences of concrete adaptation actions. The first assessment provided positive and interesting results. We are already working on extending the tool support and on adopting our proposal to model other self-adaptive service compositions.

Our proposal represents a preliminary step towards linking the non-formal world of the stakeholders with the formal world of the machine. We assume not to have a full codification of the system and, instead, try to find a tradeoff between a completely specified system, where adaptation is fully automated, and the informal requirements of stakeholders. Finally we also assume to rely on experts anadopt customized solutions.

References

1. Active Endpoints (2010) The ActiveBPEL engine. <http://www.activevos.com/community-open-source.php>. Accessed 1 June 2010
2. Ali R, Dalpiaz F, Giorgini P (2009) A goal modeling framework for self-contextualizable software. In: Halpin TA et al (eds) BMMDS/EMMSAD'09. 14th international conference on exploring modeling methods in systems analysis and design, Amsterdam, June 2009. Lecture notes in business information, vol 29. Springer, pp 326–338
3. Baresi L, Bianculli D, Ghezzi C, Guinea S, Spoletini P (2007) Validation of web service compositions. IET Software 1(6):219–232

4. Baresi L, Guinea S (2011) Self-supervising BPEL processes. *IEEE Transactions on Software Engineering* 37(2):247–263
5. Baresi L, Guinea S, Pasquale L (2008) Integrated and composable supervision of BPEL processes. In: Bouguettaya A et al (eds) ICSOC'08. 6th international conference of service oriented computing, Sydney, December 2008. Lecture notes in computer science, vol 5364. Springer, Heidelberg, pp 614–619
6. Baresi L, Pasquale L, Spoletini P (2010) Fuzzy goals for requirements-driven adaptation. In: Proceedings of the 18th international requirements engineering conference, Sydney, Australia
7. Cheng BHC, Sawyer P, Bencomo N, Whittle J (2009) A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty. In: Shurr A et al (eds) MoDELS'09. 12th international conference on model driven engineering languages and systems, Denver, October 2009. Lecture notes in computer science, vol 5795. Springer, Heidelberg, pp 468–483
8. Cheng SW, Garlan D, Schmerl B (2006) Architecture-based self-adaptation in the presence of multiple objectives. In: Proceedings of the 2nd international workshop on self-adaptation and self-managing systems, Shanghai, China
9. Fickas S, Feather MS (1995) Requirements monitoring in dynamic environments. In: Proceedings of the 2nd international symposium on requirements engineering, York, England
10. Goldsby HJ, Sawyer P, Bencomo N, Cheng BHC, Hughes D (2008) Goal-based modeling of dynamically adaptive system requirements. In: Proceedings of the 15th international conference on engineering of computer-based systems, Belfast, Ireland, 31 March–4 April 2008
11. Huynh MN, Holderfield VT, Gutierrez RLZ (2003) Robust software via agent-based redundancy. In: Proceedings of the 2nd international joint conference on autonomous agents & multiagent Systems, Melbourne, Australia
12. Kazhamiakin R, Pistore M, Roveri M (2004) A framework for integrating business processes and business requirements. In: Proceedings of the 8th international conference on enterprise distributed object computing, Monterey, California, USA
13. Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes CV, Loingtier J, Irwin, J (1997) Aspect-oriented programming. In: Aksit M (eds) ECOOP'97. 11th European conference on object-oriented programming, Jyvaskyla, Finland, June 1997. Lecture notes in computer science, vol 1241. Springer, pp 220–242
14. van Lamsweerde A (2009) Requirements engineering: from system goals to UML models to software specifications. Wiley, Chichester
15. Mahfouz A, Barroca L, Laney RC, Nuseibeh B (2009) Requirements-driven collaborative choreography customization. In: Baresi L et al (eds) ICSOC-ServiceWave'09. 7th international joint conference ICSOC-ServiceWave, Stockholm, Sweden, November 2009. Lecture notes in computer science, vol 5900. Springer, Heidelberg, pp 144–158
16. Morandini M, Penserini L, Perini A (2008) Modelling self-adaptivity: a goal-oriented approach. In: Brueckner SA (eds) SASO'08. 2nd international conference on self-adaptive and self-organising systems, Springer, Los Alamitos, pp 469–470
17. Morandini M, Penserini L, Perini A (2008) Towards goal-oriented development of self-adaptive systems. In: Proceedings of the 3rd international workshop on software engineering for adaptive and self-managing systems, Leipzig
18. OASIS (2007) Web services business process execution language version 2.0. <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>. Accessed 1 June 2010
19. Pasquale L (2011) A goal-oriented methodology for self-supervised service compositions. Dissertation, Politecnico di Milano
20. Penserini L, Perini A, Susi A, Mylopoulos J (2007) High variability design for software agents: extending tropos. *Trans Auton Adaptive Syst* 2(4):75–102
21. Pnueli A (1977) The temporal logic of programs. In: Proceedings of the 18th annual symposium on foundations of computer science, Providence, Rhode Island, USA, 31 October–22 November 1977

22. Proctor M et al (2009) Drools. <http://www.jboss.org/drools/>. Accessed 1 June 2010
23. Qureshi NA, Perini A (2009) Engineering adaptive requirements. In: Proceedings of the 4th international workshop on software engineering for adaptive and self-managing systems, Vancouver, BC, Canada
24. Robinson WN (2003) Monitoring web service requirements. In: Proceedings of the 11th international requirements engineering conference, Monterey Bay, CA, USA
25. Salifu M, Yu Y, Nuseibeh B (2007) Specifying monitoring and switching problems in context. In: Proceedings of the 15th international requirements engineering conference, New Delhi, India
26. Wang Y, Mylopoulos J (2009) Self-repair through reconfiguration: a requirements engineering approach. In: Proceedings of the 24th international conference on automated software engineering, Auckland, New Zealand
27. Whittle J, Sawyer P, Bencomo N, Cheng BHC (2009) RELAX: incorporating uncertainty into the specification of self-adaptive systems. In: Proceedings of the 17th international requirements engineering conference, Atlanta, GA, USA, 31 August–4 September 2009

Chapter 11

Business Goals and Architecture

Len Bass and Paul Clements

Abstract Software systems exist to fulfill the business goals of organizations – developing organizations, purchasing organizations, user organizations, and others – and these goals turn out to be a significant source of requirements. These requirements are not currently well understood and are seldom captured explicitly, yet they can have a profound effect on the architecture for a system. Systems that meet the published requirements but do not satisfy the important business goals are considered failures. Our research has produced a standard categorization of business goals that can be used to aid in elicitation and capture. We have also created a form for a business goal scenario, which is an expression to capture a stakeholder's business goal in an unambiguous form, along with its pedigree, by which we mean its source, value, and stability. Finally, we have used these developments to create a lightweight method that architects can use to interview important stakeholders and elicit and record their business goals. The method also includes a step in which the architectural ramifications of the business goal (in the form of quality attribute requirements) are compared against the existing requirements for the system, to help spot missing requirements and requirements that have no basis in business goals. All of this is intended to empower an architect to ask the right questions early in the development process, when the architecture is being crafted.

11.1 Introduction

Computer systems are constructed to satisfy business goals. Yet, even knowing this, determining the business goals for a system is not an easy chore. First, there are multiple stakeholders for a system, all of whom usually have distinct business goals. Secondly, there are business goals that are not explicitly articulated and must be actively elicited. Finally, the initial statement of some business goals is unreasonable in their strictness.

If systems are constructed to satisfy business goals then that makes it imperative that the requirements that drive the design of the system reflect those business goals.

Yet in our experience, requirements documents are generally deficient in providing these requirements, which are often most important from an architect's perspective because they drive quality attribute goals. There are at least two possible reasons for this deficiency.

The authors of the requirements documents do not have access to all of the business goals for a particular system. These goals may not have been captured in any systematic fashion. Or perhaps the development organization is not the same organization as the one that originated the requirements, in which case the business goals of the developers were likely not recognized. Or, as is often the case, an organization's business goals only exist implicitly or do not cover the aspirations and needs of any but the highest-level executives.

Requirements documents tend to dwell on the functional requirements, not the quality attribute requirements (by which we mean requirements for performance, modifiability, security, and the like). Yet the quality attribute requirements are the ones that drive architectural design.

In either case, the result is that the architect lacks critical information on which to base the design.

Even if requirements do reflect business goals, they seldom capture the requirements' pedigree. For example, suppose a competitor's product has a response time for a particular operation of 2 s. We want our product to be superior, so our product's requirement will call for a response time of 1.8 s for a similar operation. Yet if, during development time, the competitor releases a system with a response time of 1.6 s, it's very doubtful that our requirements document will change. If the architect knows that the goal is to beat the competitor's response time with respect to this particular operation, then the architect would know that with the release of the competitor's new version the requirement has changed in the system the architect is currently developing.

A system's architect needs to know those business goals that impinge on the architecture. The architecture must reflect the priorities and schedule needs of its stakeholders. For example, if time to market is a dominant business goal, then one set of architectural choices is appropriate. But if some aspect of quality predominates, then another set. Business goals may be expressed to the architect in terms of quality attribute requirements or in terms of constraints on the architecture.

In this paper, we present a body of knowledge that allows more precise elicitation and specification of business goals. The body of knowledge is based on an analysis of the business literature and its application to specific systems.

We first discuss which businesses have business goals with respect to a particular system. We then present a canonical set of business goals derived from the business literature. We describe a formal syntax that can be used to capture business goals. We discuss how to relate these goals to quality attribute requirements. We present a variety of different engagement models to elicit business goals and discuss the positive and negatives of each engagement model. We close with a summary of an elicitation method we have created and used to elicit business goals and tie them to

an architecture: the Pedigreed Attribute eLicitation Method (PALM). PALM is described in fuller detail elsewhere by the authors [3].

11.2 Whose Business Goal is It?

Multiple organizations are involved in the development of a large computer system. This is the case even if the organizations are different divisions of the same corporation. Each of those organizations has its own set of goals when it comes to the system. There is sometimes an assumption that the business goals for a system are embedded in the requirements for that system. That may not be the case. For example, when one organization contracts with another to produce a system, then even if the requirements embody the business goals of the contracting organization, they will probably not embody the business goals of the development organization. In fact, how well requirements capture the business goals even for the contracting organization is an open question.

Some of the organizations that are involved in the development of a computer system and potential business goals for them are enumerated below. It is possible that all of these organizations are the same organization but for large computer systems, this is an unlikely occurrence.

Acquiring organization. The acquiring organization is the organization that pays for and manages the development. It is responsible for communicating its needs to the development organization. Its primary concerns should be cost, schedule, and suitability for the operating and maintenance organizations.

Developing organization. The developing organization is the organization that has primary responsibility for developing the system. In addition to being responsive to the acquiring organization's goals, it also has concerns about profit, its workforce, and developing and utilizing reusable assets.

Sub-contractor to developing organization. From the perspective of a subcontractor, the developing organization is the acquiring organization and they have the same basic types of goals as the developing organization.

Operating organization. The operating organization has goals that reflect how useful the system being developed will be given their business or mission. They also have concerns about the cost of operation in terms of its impact on their workforce's skill level, numbers, and distribution requirements.

Maintenance organization. The organization responsible for maintaining the system once it has been delivered is concerned about the quality of the documentation and the skill level required to maintain the system.

A computer system should, ideally, satisfy the union of all of these goals and, in theory, if the goals do not conflict, achieving them all is possible. In practice, the goals from multiple organizations will usually conflict and the development team must make trade-offs among the goals. Providing guidance to the architect as to

how to make trade-offs is one important reason the architect needs to be familiar with the business goals for a system no matter which organization originated them.

11.3 A Canonical Set of Business Goals

We surveyed the business literature to understand the types of business goals that organizations have with respect to a particular system. The literature is silent on the business goals for specific projects and so we use the business goals for organizations as a basis for our work. Our literature survey is summarized in [3, 4] where we discuss the process of developing the list. Some of the most important citations are [1, 2, 5–9, 11]. The following business goal categories resulted from applying an affinity grouping to the categories harvested from our survey:

1. **Growth and continuity of the organization.** How does the system being developed (or acquired) contribute to the growth and continuity of the organization? In one experience using these categories, the system being developed was the sole reason for the existence of the organization. If the system was not successful, the organization would cease to exist. Other topics that might come up in this category deal with market share, creation and success of a product line, and international sales.
2. **Meeting financial objectives.** This category includes revenue generated or saved by the system. The system may be for sale, either in stand-alone form or by providing a service, in which case it generates revenue. A customer might be hoping to save money with the system, perhaps because its operators will require less training than before, or the system will make processes more efficient. Also in this category is the cost of development, deployment, and operation of the system. But this category can also include financial objectives of individuals – a manager hoping for a raise, for example, or a shareholder expecting a dividend.
3. **Meeting personal objectives.** Individuals have various goals associated with the construction of a system. Depending on the individual, they may range from “I want my company to lead the industry” to “I want to enhance my reputation by the success of this system” to “I want to learn new technologies” to “I want to gain experience with a different portion of the development process than in the past.” In any case, it is possible that technical decisions are influenced by personal objectives.
4. **Meeting responsibility to employees.** In this category employees are usually employees involved in development or employees involved in operation. Responsibility to employees involved in development might include ensuring that certain types of employees have a role in the development of this system or it might include providing employees the opportunities to learn new skills. Responsibility to employees involved in operating the system might include safety considerations, workload considerations, or skill considerations.

5. **Meeting responsibility to country.** Government systems, almost by definition, are intended to meet responsibility to country. Other topics that might come up in this category deal with export controls, regulatory conformance, or supporting government initiatives.
6. **Meeting responsibility to society.** Some organizations see themselves as in business to serve society. For these organizations, the system under development is helping them meet those responsibilities. But all organizations must discharge a responsibility to society by obeying relevant laws and regulations. Other topics that might come up under this category are resource usage, “green computing,” ethics, safety, open source issues, security, and privacy.
7. **Meeting responsibility to shareholders.** There is overlap between this category and the financial objectives category but additional topics that might come up here are liability protection and certain types of regulatory conformance such as adherence to the Sarbanes-Oxley Act.
8. **Managing market position.** Topics that might come up in this category are the strategy used to increase or hold market share, staying competitive, various types of intellectual property protection, or the time to bring a product to market.
9. **Improving business processes.** Although this category partially overlaps with meeting financial objectives, reasons other than cost reduction exist for improving business processes. It may be that improved business processes enable new markets, new products, or better customer support.
10. **Managing quality and reputation of products.** Topics that might come up in this category include branding, recalls, types of potential users, quality of existing products, and testing support and strategies. In this category, “products” can mean an organization’s for-hire services.

There's obvious overlap among these categories, but we are not proposing a taxonomy but rather a framework to aid in elicitation of a project's business goals. Each category functions as a basis for a semi-structured process to elicit business goals. Each category is a prompt to begin a conversation: “What kind of business goals do you have dealing with responsibility to your employees?” for example. In one exercise we conducted, a manager thought about this particular question for a minute and then told us about wanting the new product to be successful so the branch office wouldn't close, forcing re-location or layoff. That could have come up (but didn't) when we asked about market share goals or financial objective goals.

In addition, how business goals change over time as a reflection of the changing environment must also be considered to determine how much flexibility the architect must design into the system. The elements of the environment that may change are legal, social, financial, competitive, and technological [9]. Developing a technology roadmap is a common activity for a long lived project but other changeable facets are often neglected. For many organizations, social and regulatory environments change at least as fast as technologies, and they should be accounted for. For other organizations, the financial and competitive environments can be

extremely dynamic. These need to be considered to support the architect in the design process.

These business goal and change categories aid in business goal elicitation by prompting a discussion of which categories apply, and then attempting to capture a concrete instance of a goal or goals in each applicable category. Given a business goal, now it becomes possible for the architect to investigate how such a goal might affect quality attribute requirements for the system.

One may view this kind of elicitation as a form of goal-oriented requirements elicitation as described by Anton [1], with the initial goals pre-specified as a canonical list of broad business goals categories.

11.4 Business Goals and Architectural Requirements

The relation between business goals and architectural requirements is threefold.

1. Some business goals have nothing to do with the architecture. For example, a business goal to reduce expenses may be achieved by turning the thermostat down (or up, depending on season).
2. Some business goals directly impact the architecture and are expressed as constraints to the architect. For example, an organization may have a business arrangement with a supplier and the business goal is that “all developments shall use vendor X software” will dictate architectural choices.
3. Some business goals are manifested as quality attribute requirements. For example, “Our product will be faster than our competitor’s product” will be expressed as a performance requirement.

Figure 11.1 shows the relations we have described.

11.5 Relating Business Goals to Quality Attribute Requirements

Since quality attribute requirements play an important role in designing the architecture we not only need to elicit business goals but relate those business goals to quality attribute needs. In essence, we attempt to fill out the following Table 11.1

Fig. 11.1 Business goals can affect architectures directly, indirectly, or not at all

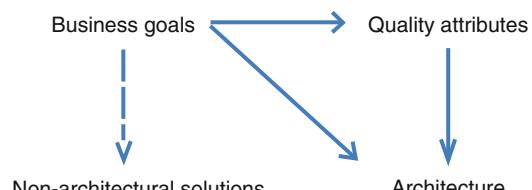


Table 11.1 Business goals and quality attributes

Business goal	Availability	Modifiability	Performance	...
	Relation between business goal 1 and 1 and availability	Relation between business goal 1 and modifiability	Relation between business goal 1 and performance	Relation between business goal 1 and other quality attributes
Business goal 1				
Business goal 2				
...				

Not all cells in the table will be filled out; a particular business goal may not manifest in requirements for every quality attribute of interest. Moreover, cells in the table may be filled in vaguely at first, prompting the architect to have extended discussions about the precise nature of the quality attribute requirements precipitated by a business goal.

But armed with a table like this, the architect can now make informed designed trade-offs, especially when combined with information about the value of achieving each business goal, thus prioritizing the needs.

11.6 A Syntax for Describing Business Goals

Capturing business goals and then expressing them in a standard form will let them be discussed, analyzed, argued over, rejected, improved, reviewed – in short, all of the same activities that result from capturing any kind of requirement. One syntax that could be used to describe business goals is that produced under the guidance of TOGAF [10]. We use a syntax more focused on the relation between business goal and architecture, however. Our business goal scenario template has six parts. They all relate to the system under development, the identity of which is implicit. The parts are:

1. **Goal-subject.** This is the stakeholder who owns the goal, who wishes it to be true. The stakeholder might be an individual, an individual in an identified organization if more than one organization is in play, or (in the case of a goal that has no one owner and has been assimilated into an organization) the organization itself.
2. **Goal-object.** This is the entity to which the goal applies. A goal-object will typically be one of: individual, system, portfolio, organization's employees, organization's shareholders, organization, nation, or society.
3. **Environment.** This is the context for this goal. Osterwalder and Pigneur [9] identify social, legal, competitive, customer, and technological environments. Sometimes the political environment is key; this is a kind of social factor.
4. **Goal.** This is any business goal able articulated by the person being interviewed.

5. **Goal-measure.** This is a measurement to determine how one would know if the goal has been achieved. The goal-measure also usually includes a time component, stating the time by which the goal should be achieved.
6. **Pedigree and value.** The pedigree of the goal tells us the person who stated the goal, the degree of confidence that person has in it, and the goal's volatility and value. The value of a goal can be expressed by how much its owner is willing to spend to achieve it or its relative importance compared to other goals. Relative importance may be given by a ranking from one (most important) to n (least important), or by assigning each goal a value on a fixed scale such as 1–10 or high/medium/low. In this part we can also capture information about the goal such as related goals, or potential obstacles to meeting the goal.

Note the information added through this syntax (in addition to the business goal itself). All of this additional information is intended to establish the pedigree of the goal.

The source of the goal. This enables the architect to have at least one source to gather additional information about the goals.

The object of the goal. This includes the organization that has the goal. This enables the architect to understand the relations among the various organizations.

The justification for the goal. This enables the architect to push back on goals that cause great difficulty in implementation.

The value of the goal. This enables the architect to prioritize the various goals and make trade-offs among them.

11.7 Engagement Models

The body of knowledge we have presented provides the basis for eliciting architecturally relevant business goals from the stakeholders. This elicitation could be executed in a variety of different engagement models. The engagement model that we have direct experience with is having a workshop with a variety of stakeholders; we have reported previously on our results using this engagement model [3, 4]. In this section, we discuss a variety of different potential engagement models and then we discuss some considerations that must be balanced when choosing an engagement model. (All engagement models can include beforehand inspection and analysis of relevant documentation.)

A face to face workshop. A face to face workshop can involve all of the stakeholders or just stakeholders from one of the involved organizations. It has the properties of openness, at least among the attendees, immediacy, and group dynamics. It has the drawbacks of travel costs and scheduling difficulty. Finding a time when all of the stakeholders can come together has proven to be a difficult task, even without consideration of cost.

Phone interviews. Phone interviews can be one-on-one or group oriented. They can be mediated by one of the modern collaboration tools or can just be telephone based. Phone interviews avoid travel cost but, if they are done with a group, have the same scheduling problems. Group teleconferences have the problem of keeping attendees attention. Interruptions are much easier with phone interviews than with face to face interactions.

Questionnaires and forms. Questionnaires and forms to fill out allow the stakeholders to contribute at their convenience. Constructing questionnaires and forms that are unambiguous without hands on guidance by the elicitor is very difficult and time consuming. Questionnaires and forms are also difficult to follow up to gather additional information.

Hybrid. It is possible to have hybrid engagements, as well. For example, one might use a form or questionnaire to gather initial information and follow up with a phone conference at some later date.

Some of the considerations that go into choosing an engagement model for eliciting business goals are

The cost to stakeholders. Each stakeholder from whom goals are to be elicited must spend a non-trivial amount of time providing input. This time can be sequential or broken up. Sequential time is frequently difficult to arrange but reduces the context switching cost for the stakeholder. Another cost to the stakeholders is travel time if the elicitation is performed in a remote location.

The cost to those performing the elicitation. If every relevant stakeholder is interviewed individually, the time for those performing the elicitation is very high. If relevant stakeholders are interviewed face to face, the travel costs for those performing the elicitation is also high.

The immediacy of the elicitation. The elicitation can be done synchronously either face to face or utilizing various communication technologies. It could also be done asynchronously through questionnaires or forms. Synchronous interactions enable the elicitor to direct a conversion and use follow up questions to achieve clarity. Asynchronous interactions allow the stakeholder to provide information at their convenience.

Openness. The elicitation could be done with all of the stakeholders having access to the input provided by any of the stakeholders or with the input of stakeholders being kept confidential. Some stakeholders may not wish to have other stakeholders aware of their business goals. This argues for confidentiality. The architect, on the other hand, must justify decisions in terms of business goals and this argues for openness.

Multiplicity. The elicitation could be done one-on-one or with a group. One-on-one elicitations provide the maximum opportunity for confidentiality and openness to the elicitors. Group dynamics often result in participants adding items they may otherwise not have brought up.

No one engagement method is best for all situations and, in general, the choice of engagement method is usually a matter of determining the least problematic method.

11.8 PALM

We developed the Pedigreed Attribute eLicitation Method (PALM) to put the principles discussed earlier in this chapter into practice, and to provide architects with a simple, repeatable, effective method to help them elicit and capture business goals that apply to a system they are designing. PALM was created and piloted using the “face-to-face workshop” engagement model discussed earlier, but other models should be equally effective.

PALM is a seven-step method. The steps are:

1. **PALM overview presentation:** Overview of PALM, the problem it solves, its steps, its expected outcomes.
2. **Business drivers presentation:** Briefing of business drivers by project management. What are the goals of the customer organization for this system? What are the goals of the development organization? This is a lengthy discussion that gives the opportunity of asking questions about the business goals as presented by project management.
3. **Architecture drivers presentation:** Briefing by the architect on the driving (shaping) business and quality attribute requirements.
4. **Business goals elicitation.** Using the standard business goal categories to guide discussion, we capture the set of important business goals for this system. Business goals are elaborated, and expressed as business goal scenarios. We consolidate almost-alike business goals to eliminate duplication. Participants then prioritize the resulting set to identify the most important ones.
5. **Identifying potential quality attributes from business goals.** For each important business goal scenario, participants describe a quality attribute that (if architected into the system) would help achieve it. If the QA is not already a requirement, this is recorded as a finding.
6. **Assignment of pedigree to existing quality attribute drivers.** For each architectural driver named in Step 3, we identify which business goal(s) it is there to support. If none, that's recorded as a finding. Otherwise, we establish its pedigree by asking for the source of the quantitative part: E.g.: Why is there a 40 ms performance requirement? Why not 60 ms? Or 80 ms?
7. **Exercise conclusion.** Review of results, next steps, and participant feedback.

PALM is primarily a tool to empower architects to ask important questions of salient stakeholders appropriately early in the life cycle. PALM can help architects in the following ways.

First, PALM can be used to sniff out missing requirements early in the life cycle. For example, having stakeholders subscribe to the business goal of improving the

quality and reputation of their products may very well lead to (for example) security, availability, and performance requirements that otherwise might not have been considered.

Second, PALM can be used to inform the architect of business goals that directly affect the architecture without precipitating new requirements. One example is a system requiring a data base in order to utilize the database team. There is no standard name for this property (“full-employment-ability?”) nor would it be expected to show up as a “requirement.” Similarly, if an organization has the ambition to use the product as the first offering in a new product line, this might not affect any of the requirements for that product (and therefore not merit a mention in the project’s requirements specification). But this is a crucial piece of information that the architect needs to know early so it can be accommodated in the design.

Third, PALM can be used to discover and carry along additional information about existing requirements. For example, a business goal might be to produce a product that out-competes a rival’s market entry. This might precipitate a performance requirement for, say, half-second turnaround when the rival features one-second turnaround. But if the competitor releases a new product with half-second turnaround, then what does our requirement become? A conventional requirements document will continue to carry the half-second requirement, but the goal-savvy architect will know that the real requirement is to beat the competitor, which may mean even faster performance is needed.

Fourth, PALM can be used to examine particularly difficult quality attribute requirements to see if they can be relaxed. We know of more than one system where a quality attribute requirement proved quite expensive to provide, and only after great effort, money, and time were expended trying to meet it was it revealed that the requirement had no analytic basis, but was merely someone’s best guess or fond wish at the time.

Fifth, different stakeholders have different business goals for any individual system being constructed. The acquirer may want to use the system to support their mission; the developer may want to use the system to launch a new product line. PALM provides a forum for these competing goals to be aired and resolved.

PALM can be used to developing organizations as well as acquiring organizations. Acquirers can use PALM to sort out their own goals for acquiring a system, which will help them to write a more complete request for proposals (RFP). Developing organizations can use PALM to make sure their goals are aligned with the goals of their customers.

We do not see PALM as anointing architects to be the arbiter of requirements, unilaterally introducing new ones and discarding vexing ones. The purpose of PALM is to empower the architect to gather necessary information in a systematic fashion.

Finally, we would hope and expect that PALM (or something like it) would be adopted by the requirements engineering community, and that within an organization requirements engineers would be the ones to carry it out.

11.9 Experience with PALM

We applied PALM to a system being developed by the Air Traffic Management unit of a major U.S aerospace firm. To preserve confidentiality, we will call this system The System Under Consideration (TSUC) and summarize the exercise in brief.

TSUC will provide certain on-line services to the airline companies to help improve the efficiency of their fleet. Thus, there are two classes of stakeholders for TSUC – the aerospace firm and the airline companies. The stakeholders present when we used PALM were the chief architect and the project manager for TSUC.

Some of the main goals that were uncovered during this use of PALM were:

How the system was designed to effect the user community and the developer community

The fact that TSUC was viewed as the first system in a future product line.

Issues regarding the lifetime of TSUC in light of future directions of regulations affecting air traffic.

The possibility of TSUC being sold to additional markets.

Issues related to the governance strategy for the TSUC product.

The exercise helped the chief architect and the project manager share the same vision for TSUC, such as its place as the first instance in a product line and the architectural and look-and-feel issues that flow from that decision.

The ten canonical business goals ended up bringing about discussions that were wide ranging and, we assert, raised important issues unlikely to have been thought of otherwise. Even though the goal categories are quite abstract and unfocussed, they were successful in triggering discussions that were relevant to TSUC. The result of each of these discussions was the capture of a specific business goal relevant to TSUC.

11.10 Conclusions

Knowing the business goals for a system is important to enable an architect to make choices appropriate to the context for the system. Each organization involved in the construction and operation of the system will have its own set of goals that may conflict.

We presented a body of knowledge suitable for eliciting the business goals from stakeholders. The assumption is that the canonical list of goals will act as the beginning of a conversation that will result in the elicitation of multiple business goals. Capturing these goals in a fixed syntax ensures that all of the information for each goal has been recorded and provides a common format for the reader of the goals.

Based on this body of knowledge there are a variety of different engagement models that will allow the elicitor to gain the business goal from a stakeholder.

These engagement models differ in cost to stakeholders, cost of elicitation, openness, and multiplicity.

Finally, we have put the principles and ideas into practice with a worked-out method called PALM, which we have used to demonstrate the practicality and usefulness of these ideas.

Future research opportunities exist for continuing to clarify the relation between business goals and quality attribute requirements; for applying the method under different engagement models and making the appropriate modifications; and for measuring the effectiveness of the approach by (for example) trying to measure cost savings derived from early discovery of missing requirements.

References

1. Antón A (1997) Goal identification and refinement in the specification of information systems, Ph.D. Thesis. Georgia Institute of Technology
2. Carroll AB (1991) The pyramid of corporate social responsibility: toward the moral management of organizational Stakeholders. *Bus Horiz* 34:39–48
3. Clements P, Bass L (2010) Relating business goals to architecturally significant requirements for software. Technical report CMU/SEI CMU/SEI-2009-TN-026
4. Clements P, Bass L (2010) Using business goals to inform software architecture. In: Proceedings requirements engineering 2010, Sydney, Sept 2010
5. Fulmer RM (1978) American Management Association [Questions on CEO Succession]
6. Hofstede G, van Deusen CA, Mueller CB, Charles TA (2002) What goals do business leaders pursue? a study in fifteen countries. *The business goals network source. J Int Bus Stud* 33 (4):785–803, Palgrave Macmillan Journals, 2002
7. McWilliams A, Siegel D (2001) Corporate social responsibility: a theory of the firm perspective. *Acad Manage Rev* 26(1):117–127
8. Mitchell RK, Agle BR, Wood DJ (1997) Toward a theory of Stakeholder identification and salience: defining the principle of who and what really counts. *Acad Manage Rev* 22 (4):853–886, Academy of Management. <http://www.jstor.org/stable/259247>
9. Osterwalder A, Pigneur Y (2004) An ontology for e-business models. In: Currie W (ed) Value creation from e-business models. Butterworth-Heinemann, Oxford, pp 65–97
10. TOGAF (2009) The open group architecture framework, Version 9, <http://www.opengroup.org/architecture/togaf9-doc/arch>
11. Usunier J-C, Furrer O, Perrinjaquet A (2008) Business goals compatibility: a comparative study. Institut de Recherche en Management (IRM), University of Lausanne, Ecole des HEC, Switzerland. Working paper: <http://www.hec.unil.ch/irm/Research/Working%20Papers>

Part III

Experiences from Industrial Projects

Chapter 12

Experiences from Industrial Projects

P. Avgeriou, J. Grundy, J. Hall, P. Lago, and I. Mistrík

The aim of the book is to develop the bridge between two ‘islands’: Software Architecture and Requirements Engineering. However, in Software engineering, there is another gap that needs to be bridged between two different types of communities: industry and academia. Industrial practitioners work under hard constraints and often do not have the luxury of trying out research results, let alone embedding them in their everyday practice. In contrast, academic researchers face the pressure of ‘publish or perish’ and often struggle with finding the right industrial context in which to validate their work. Nevertheless, when the right synergy is established between the two communities, there can be substantial progress of the state of the art.

In the Software Architecture field, the results of a successful partnership between industry and academia are manyfold. Examples include the IEEE Recommended Practice for Architectural Description of Software-Intensive Systems [1] and its successor, the upcoming ISO-IEC Std. 42010 [2]; they constitute a common conceptual framework respected in both communities. Methods for architecture design [3], as well as evaluation [4] have been derived in both environments and have been successfully applied in practice. The reusability of architecture design has been boosted with the publication of numerous architecture [5] and design patterns [6] that were mined in academic and industrial developmental environments. Finally the most recent advance in the field, the management of Architecture Knowledge, has sprung out from academic research [7] but has quickly had an impact in industrial practice.

Similarly, in the Requirements Engineering field, again standards have arrived, including IEEE 830–1998 [8], which describes both possible and desirable structures, content and qualities of software requirements. Other academic/industry requirements tools have become ubiquitous both in software and wider afield, including use cases [9], requirements elicitation, specification and analysis [10], and scenario analysis [11].

As well as requirements/architectures, then, this third part of the book bridges academia/industry at the same time, relating successful collaborations of industrial

and academic stakeholders with four chapters concerning various aspects of industrial experiences in the field of relating requirements and architecture.

Chapters 13 and 16 relate examples of pragmatic approaches that build on the experience gained by industrial projects: Chapter 13 derives design solutions to recurring problems that are packaged as a reference architecture for design reuse; Chapter 16 documents a number of best practices and rules of thumb in architecting large and complex systems, thus promoting process reuse.

Chapter 14 presents a detailed academic approach of checking architecture compliance that has been validated in an industrial system, providing evidence for the validity of the approach in a real-world case.

Finally Chapter 15 elaborates on a theoretical research result that has come out of an industrial research environment: an approach to the management of artifact traceability at the meta-model level, illustrated with realistic examples.

In more detail:

Chapter 13 by Tim Trew, Goetz Botterweck and Bashar Nuseibeh presents an approach that supports requirements engineers and architects in jointly tackling the hard challenges of a particular demanding domain: consumer electronics. The authors have mined architectures from a number of existing systems in this domain in order to derive a reference architecture. The latter attempts to establish a common conceptual foundation that is generic and reusable across different systems and context. It can be used to derive concrete product architectures by facilitating architects and requirements engineers to successively and concurrently refine the problem and solution elements and make informed design decisions. The process of developing a reference architecture by mining from existing systems, issues encountered and design decisions that resolve them, is simple yet effective for domains where little architecture standardization exists.

Chapter 14 by Huy Tran, Ta'id Holmes, Uwe Zdun, and Schahram Dustdar presents a case study in the challenging field of checking compliance of the architecture (in the SOA domain) to a set of requirements (ICT security issues). The proposed solution has two main characteristics: it defines multiple views to capture the varying stakeholder concerns about business processes, and it is model-driven facilitating the linkage between requirements, architecture, and the actual implementation through traces between the corresponding meta-models. The approach results in semi-formal business processes from the perspective of stakeholders and the checking of compliance of requirements through appropriate links. It is validated in an industrial case study concerning a SOA-based banking system.

Chapter 15 by Jochen M. Küster, Hagen Völzer, and Olaf Zimmermann proposes a holistic approach to relate the different software engineering activities by establishing relations between their artifacts. The approach can be particularly targeted towards relating artifacts from requirements engineering and architecture, as exemplified by their case study. The main idea behind the approach is a matrix that structures artifacts along dimensions that are custom-built for the software engineering process at hand. The authors propose two default dimensions as a minimum: stakeholder viewpoints (in the sense of [1]) and realization levels.

Artifacts are classified into the cells of the matrix according to their nature and subsequently links are established between them that express traceability, consistency or actual model transformation between the artifacts. The approach can be used during method and tool definition across the software engineering lifecycle by utilizing the generic meta-model in order to decide upon the exact relationships to be established between the various artifacts.

Chapter 16 by Michael Stal presents a pragmatic architecting process that has been derived from a number of industrial projects. The process is informally presented as a set of best practices and rules of thumb, as well as a method comprised of a series of steps. The process is based on the principle of eliciting, specifying and prioritizing requirements and subsequently using them as key drivers for architecting. Besides the rooting in the problem space, the approach combines some commonly accepted tenets: piecemeal refinement, risk mitigation, reuse, review, refactoring. The architecture activities are prioritized into four phases (functionality, distribution and concurrency, runtime and design-time qualities) and the system is scoped into three levels (system, subsystem, component).

References

1. IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, IEEE Std 1471–2000 , pp. 1–23, 2000, doi: 10.1109/IEEEESTD.2000.91944
2. ISO-IEC (2010) ISO-IEC Std. 42010: recommended practice for architectural description of software-intensive systems 2010. <http://www.iso-architecture.org/ieee-1471/>, Date of Access: 1 Dec. 2010
3. Hofmeister C, Kruchten P, Nord RL, Obbink H, Ran A, America P (2007) A general model of software architecture design derived from five industrial approaches. *J Syst Softw* 80: 106–126, Elsevier Science Inc
4. Dobrica L, Niemela E (2002) A survey on software architecture analysis methods. *IEEE T Softw Eng* 28:638–653, IEEE Press
5. Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stal M (1996) Pattern-oriented software architecture – a system of patterns. Wiley, West Sussex
6. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison-Wesley, Reading
7. van Vliet H, Avgeriou P, de Boer R, Clerc V, Farenhorst R, Jansen A, Lago P (2009) The GRIFFIN project: lessons learned. In: Babar MA, Dingsøyr T, Lago P, van Vliet H (eds) Software architecture knowledge management: theory and practice. Springer, Heidelberg, pp 137–154
8. IEEE Recommended Practice for Software Requirements Specifications, IEEE Std 830–1998, 1998, doi: 10.1109/IEEEESTD.1998.88286
9. Cockburn A (2001) Writing effective use cases, vol 1. Addison-Wesley, Boston
10. Christel MG, Kang KC (1992) Issues in requirements elicitation. Carnegie-Mellon Univ, Pittsburgh, Software Engineering Inst
11. Hsia P, Samuel J, Gao J, Kung D, Toyoshima Y, Chen C (1994) Formal approach to scenario analysis. *Softw IEEE* 11(2):33–41

Chapter 13

A Reference Architecture for Consumer Electronics Products and its Application in Requirements Engineering

Tim Trew, Goetz Botterweck, and Bashar Nuseibeh

Abstract Consumer electronics (CE) products must be appealing to customers, have features that distinguish them in the market and be priced competitively. However, this must be achieved with limited hardware resources, so requirements engineers and architects must work together to specify an attractive product within these constraints. This requires an architectural description from early in development. The creation of this description is hampered by the lack of consensus on high-level architectural concepts for the CE domain and the rate at which novel features are added to products, so that old descriptions cannot simply be reused. This chapter describes both the development of a reference architecture that addresses these problems and the process by which the requirements and architecture are refined together. The reference architecture is independent of specific functionality and is designed to be readily adopted. The architecture is informed by information mined from previous developments and organised to be reusable in different contexts. The interplay between the roles of requirements engineer and architect, mediated through the reference architecture, is described and illustrated with an example of integrating a new feature into a mobile phone.

13.1 Introduction

Consumer electronics (CE) products, such as TVs, smart phones and in-car entertainment, must be appealing to customers, have features that distinguish them in the market and be priced competitively. Despite falling hardware costs, the resources available for their implementation, such as processing power, memory capacity and speed, and dedicated hardware elements, are still limited. These constraints may restrict the features that can be offered, reduce their capabilities or limit their concurrent availability. Requirements engineers and architects must work together to specify an attractive product within these constraints, which requires an architectural description from the beginning of development.

Given the rate at which novel features are added to product categories such as mobile phones, requirements engineers cannot only rely on past experience. Instead, they have to reason from first principles about how a new feature might be used, how it might interfere with other features, whether implementations developed for other product categories would be acceptable and how requirements may have to be adapted for the feature to be integrated into the overall product at acceptable cost and risk.

This reasoning is hampered by the lack of consensus on high-level architectural concepts for the CE domain. In contrast, the information processing domain has widely-recognized industry standards and concepts, such as transactions and the transparencies supported by distributed processing middleware [1], which are rarely relevant for embedded software. As an example of the differences, a transaction, a core concept in information processing, is rarely used in the lower levels of embedded software. This is because any software action that changes the state of the hardware may be immediately observable by end-users and cannot be rolled-back unnoticed.

In this chapter, we describe a reference architecture for CE products, which facilitates the creation of concrete architectures for specific products, and show how this reference architecture can be used by requirements engineers to ensure that products are both attractive for consumers and commercially viable. The reference architecture was developed within Philips Electronics and NXP Semiconductors, Philips' former semiconductor division. Philips/NXP developed both software-intensive CE products and the semiconductor devices that are critical elements of their hardware. These semiconductor devices are associated with considerable amounts of software (over one million lines of code for an advanced TV), and are sold on the open market.

The reference architecture addresses the limited consensus on concepts in this domain, and avoids the need for architects to become familiar with many abstract concepts before it can be used. This is by proposing recommended design solutions for each element in the architectural structure, each followed by a process for reviewing the design decisions in the light of the specific product requirements. The content of the reference architecture is based on the experience of many product developments, and the granularity of its structure is determined by the architectural choices that must be made. Since architects must map new features onto the reference structure, they are confronted with architectural choices and their consequences for the requirements from the outset. We present a process for the concurrent refinement of requirements and architecture.

In Sect. 13.2, we describe the architectural concerns of requirements engineers, with examples of requirements and architectural choices that should be refined together and the types of architectural information required in each case. This includes the requirements for both complete products and individual COTS components, which may have to be integrated into a variety of architectures.

Section 13.3 discusses how reference architectures and other forms of architectural knowledge have been used in software development and considers how they can address a broad application domain, independent of specific functionality.

Section 13.4 describes the scope of the domain of CE products and identifies some of the characteristics and requirements of the domain that distinguish it from others. Section 13.5 describes how, given the limited consensus on high-level concepts relevant to the architecture of embedded software, appropriate information was mined from earlier developments. It then describes how the information was organised into our reference architecture. Finally, Sect. 13.6 describes a process in which the requirements engineer and architect use the reference architecture to refine the requirements and gives an example of its use in the integration of a novel feature into a mobile phone.

13.2 Architectural Concerns of Requirements Engineers

Nuseibeh’s “Twin Peaks” model describes both a concurrent process for requirements engineering and architecting and the relationship between requirements, architecture and design artefacts [2]. While this provides an overall framework, on its own it is not concrete enough to provide guidance for the development of CE products. Our reference architecture aims to pre-integrate elements of design and architecture, so that, when developing the architecture for a new product, it will both provide guidance on the decisions that must be made and give insight into the refinement of the requirements.

As a first step in the development of a reference architecture, we consider the types of architectural information that are relevant when establishing the requirements for a CE product. Specifying these products requires a careful balance between functionality and product cost, while meeting the constraints of performance, quality and power consumption. The company that is first to introduce a new feature, at the price-point acceptable for the mainstream range of a product category, can achieve substantial sales.

13.2.1 Aligning Requirements and Resources

Balance between functionality and price – Achieving the balance between functionality and selling price requires early insight into alternatives for how a feature might be implemented, and the hardware consequences for each of the options. Many CE products have real-time requirements, both firm performance requirements for signal processing, (e.g., audio/video processing or software-defined radio), and soft ones to ensure that the product appears responsive to the user. If it is only found late in development that these requirements cannot be met, then features may have to be dropped or downgraded, undermining the value proposition for the product. As reported by Ran, by the mid-1980s embedded products had already reached a level of complexity where it was no longer possible to reason about their performance without architectural models that characterise

their behaviour [3]. Therefore, there should be an architectural model from the very outset as the basis for refining requirements and architecture together, to specify a product that makes the best use of its resources.

Visibility and resolution of resource conflicts – The requirements engineer must be able to ensure that the resource management policies that resolve conflicts between features result in a consistent style of user interaction. CE products often have several features active concurrently, which not only impacts performance, but can also result in contention for non-sharable resources and thereby feature interaction [4]. Although resource management policies might be considered to be a purely architectural issue, they can affect the behaviour at the user interface. Therefore, the requirements engineer must be able to understand both the nature of the resource conflicts, to be able to anticipate that feature interaction could occur, and the options for their resolution.

An example of this is the muting of TV audio, which is used by several features, such as the user mute, automatic muting while changing channels or installing TV channels, and the child lock, in which the screen is blanked and the sound muted when a programme's age rating is too high [5]. Since these features can be active concurrently, feature interaction will result, and it must be possible to articulate policies across the features. These policies should be directly traceable to the architecture to ensure that they are correctly implemented.

Consequently, it must be possible to map features in the requirements specification onto elements of the software architecture to ascertain which features can co-exist and for the software architecture to be able to represent the different resource management policies for resolving resource conflicts.

13.2.2 Architectural Compatibility of Software from External Suppliers

Major CE companies used to develop all their software in-house in order to have complete control over its requirements and to be able to fully optimize the implementation for their hardware architectures. However, this became uneconomic as the number of product features increased and they now purchase software for non-differentiating features. While features available on the open market are usually governed by standards, these standards largely focus on the interface between the product and its environment and rarely address the APIs between the implementation of this feature and the remainder of the product.¹ For instance, the standards for

¹There have been many industry standardization initiatives for particular product categories for interfaces below the application layer, such as LiMo for mobile phones [6], the MPEG Multimedia Middleware (M3W) for audio/video platforms [7] and OpenMAX for media processing libraries [8]. However, to date, none has been widely-adopted in the market. Contributors to this lack of adoption are the high degree of technical and market innovation in the CE domain and the unstable structure of the industry, which is in a transition away from vertically-integrated CE companies [9].

Conditional Access systems for controlling viewing in pay-TV systems specify how a TV signal is encrypted and the interface to the user's smart card, but not the particular functions that should be called to activate decryption in the TV.

Consequently, each supplier develops their own API, with the expectation that it can be integrated into the architecture of any of their customers' products, but with little knowledge of how the architectures may differ between customers. Although integration problems can arise from many sources, a significant class of problems result from the dynamic behaviour of the software, particularly since multiple threads often have to be used to satisfy performance requirements. Different companies may adopt different policies for aspects such as scheduling, synchronization, communication, error handling and resource management. Failure of the supplier and integrator to achieve a mutual understanding of their policies can lead to complex integration problems. A second source of problems is when the functionality of components from different suppliers overlaps, so that the components do not readily co-exist. The requirements engineer of the component supplier should be aware of these potential problems, which will be described in more detail shortly. Conversely, the requirements engineer of the product integrator should be aware that such mismatches can occur and that these might be too risky to resolve if there is a tight deadline for delivery.

The reference architecture should allow the compatibility between a component and the remainder of the product to be assessed at an early stage. Ruling out a COTS component on these grounds, despite having an attractive feature list, allows the requirements engineer to focus on less risky alternatives. This may require abandoning low-priority requirements that are only supported by that component. To be able to detect incompatibilities, the options for the behaviour of the software must be explicit in the reference architecture, while being described independently of the structure of the software. This independence is required because the lack of API standardisation results in suppliers using different structural decompositions for the same functionality. We therefore capture alternative behavioural policies as *architectural texture*, which Ran describes as the “recurring microstructure” of the architecture [3] and which van der Linden characterizes as “the collection of common development rules for realising the system” [10]. Kruchten’s *architectural mechanisms* for persistency and communication [11] are concrete implementations of behavioural policies. The identification of the alternative policies to include in our reference architecture and the structuring of its *architectural texture* are described in Sect. 13.5.

These issues are described in greater detail in the following paragraphs:

Policies for error handling and resource management – From a requirements perspective, policies for error handling and resource management can affect the behaviour observed by the end-user and, hence, must be assessed for their acceptability. For example, the vendor of a TV electronic programme guide may have a policy of displaying their logo when the guide is activated, but the overall product may support the restart of a specific feature if it fails. This restart would aim to restore the feature to as close to its previous state as possible, and with minimal

disturbance to the user. However, this recovery would be compromised if the guide also displays the logo during this restart.

Degree of control by supplied components – Another source of incompatibility with supplied components is the scope of their functionality and the degree of control that they expect to have over the platform. Multi-function CE devices may integrate best-of-breed functionality from several suppliers. Problems can occur if the *required* interfaces of a component are too low, so that the component encapsulates the control of the hardware resources it requires, or if the *provided* interfaces are too high level.

The first case can cause two types of problems: either it is not possible for this feature to execute concurrently with another that should share the same resource, or it is not possible to achieve a smooth transition between features that require exclusive access to the resource. The *required* interfaces of these components should always be high enough that it is possible to insert a resource management mechanism below them. However, new features are often originally conceived for products in which they would always have exclusive access. Then provisioning for an additional layer might have appeared to be an unnecessary overhead and an additional source of complexity. It may only be years later, when the product is extended with functionality from another category, that the problem emerges.

As an example of restrictions on concurrent execution, consider the potential conflicts between interactive services and video recording in a TV. Both features must be able to both monitor broadcast data continuously and to select new stations. Both features must be active continuously and must be able to share the resources. However, they may not have been designed with that in mind. All terrestrial digital TVs in the UK have supported interactive services from the outset. However, it was only a decade later that digital video recording was integrated into TVs. In planning the extension of the TV to include the recording functionality, it may have been thought that it is only necessary to add the new feature, whereas it may also have been necessary to acquire a new interactive TV engine from a different source and to develop a resource manager. If the TV is scheduled for launch within a tight window, the additional risk associated in this change may result in the introduction of the recording feature being deferred.

Even if features are not to be active concurrently, excessively low-level *required* interfaces can impair the end-user experience. For instance, as Wi-Fi home networks became common, stand-alone adapters were developed to allow consumers to browse for content on their PCs or the Internet and then to decode the video streams, which could then be fed to a conventional TV. When this functionality was later integrated within the TV, it was desirable to reuse the same software components. However, previously these had exclusive control of the video decoders, but in the new context this control has to be handed over to the conventional broadcast TV receiver. If the Wi-Fi browser does not have provision for this, it may be necessary to reinitialize the component whenever the feature is selected, causing it to lose internal state information and taking an excessive time. The requirements engineer must be aware of such consequences of reusing a proven

component in this new context to be able to decide whether the resulting behaviour will be acceptable for the end-user.

Having *provided* interfaces at too high a level can compromise the consistency of the user interface. The supplier of a component of a resource-intensive feature has to ensure that it can operate reliably with the available hardware resources, e.g. memory capacity, processing power or interconnect bandwidth, and possibly with minimal power dissipation. This is most easily achieved with resource managers that are not only aware of the current global state of the component, but also of the desired future state of the component so that state transitions can be planned in ways that avoid transient exhaustion of resources. For instance, in a product with multi-stream audio/video processing, the semiconductor supplier may wish to have complete control of the processing of these streams and of the transitions between different stream configurations. This can be achieved by raising the level of the *provided* interface, so that the client only makes a single declarative request for a new configuration, much in the style of SOA. This enables the supplier to both provide a component that can be fully-validated, independent of the behaviour of the customer's software, and allows the supplier to innovate by evolving their hardware/software tradeoffs without affecting their customers' code. These properties of dependability and evolvability are important non-functional attributes for component supplier, but they can lead to two problems for the product integrator. Firstly, in this example, the integrator may be reluctant to reveal the stream configurations that it plans to use and, secondly, the supplier's state transition strategy may differ from that used in other features, resulting in inconsistent overall product behaviour.

An architectural model is required that allows this tension to be discussed without either party exposing critical intellectual property (IP), possibly providing the motivation for the parties to enter a closer commercial partnership where requirements can be discussed more freely. Therefore, the architecture should represent the responsibilities of the components, while being independent of their specific functionality.

This section has identified some situations in which requirements and architectural choices should be refined together, both (1) to achieve a satisfactory balance between functionality and product cost and (2) to ensure that resource management policies result in a consistent user interface. It has also addressed the selection of COTS components, both by identifying policies that have some influence on the behaviour observed by the end-user, and by rapidly screening components for architectural compatibility, so that unsuitable components can be disregarded at an early stage. In each case, the types of architectural information required has been highlighted, including identifying the resources used by any feature and the options for managing resource conflict, and representing the scope of different COTS components, in terms of the levels of their *provided* and *required* interfaces. This must be done with a reference architecture which is abstracted from the concrete product line architecture, both to allow these decisions to be made at an early stage in development, before a refined architecture is available, and to protect the IP of the parties.

Having described the support that a reference architecture should provide the requirements engineer, the remainder of the chapter describes how such an architecture was developed for the CE domain and illustrates how it can be used in practice. As a first step in this, the next section reviews how industry develops and uses reference architectures in general.

13.3 Reference Architectures in Software Development

Before describing how our reference architecture was developed and how it can be applied, we will introduce the form and use of reference architectures in some more mature application domains and what lessons can be learnt for the development of our architecture.

The role of reference architectures in software development is well-established; the Rational Unified Process uses them to capture elements of existing architectures, which have been proven in particular contexts, for reuse in subsequent developments [11, 12]. Reference architectures can exist at many levels of abstraction and can take many forms, depending on the context in which they are to be applied. The Open Group Application Framework (TOGAF) introduces the *architecture continuum* to describe the degree of abstraction a reference architecture has from an organisation-specific architecture [13]. TOGAF describes the characteristics of potential architectures in this continuum, ranging from *Foundation Architectures* to *Organization-Specific Architectures* and provides a *Technical Reference Model* (TRM) as an example Foundation Architecture. The TRM is a taxonomy of applications, services and service qualities. The service taxonomy is specific to information processing applications. For our purposes, we require a model that is less specific to an application domain, since the definition of services changes rapidly as the functionality supported by a product category evolves. We also require a model that provides more technical guidance, while being independent of the functionality being supported.

The OASIS reference architecture for service-oriented architecture (SOA) [14] is an example of such an architecture, since it captures the information that is important for a successful SOA, independent of its functionality. In this case, the overall structure of the SOA relates to the structure of the business, so that there is a straightforward mapping from the requirements to the structure of the software architecture. This mapping is more complex in an embedded system, with aspects of a particular feature being implemented at different layers in the system, e.g. based on the need to support variability of requirements and hardware and to separate operations with different temporal granularity [15]. Therefore, in contrast to the SOA reference architecture, our reference architecture for CE products should provide guidance on structuring the software.

Eeles and Cripps' classification of architectural assets [16] uses axes of *granularity* and *level of articulation (or implementation)*. At a fine grain, they identify *Architectural Styles*, *Architectural Patterns*, *Design Patterns* and *Idioms*, which are

at a suitable level of articulation for our purposes. However, at a coarser grain, their *Reference Model* is more domain-specific, being comparable to the TOGAF TRM. We seek a reference architecture that can aggregate the fine grain architectural information and provide guidance on its use, while still being independent of the application domain.

POSA4 [17] presents an extensive pattern language that addresses these aims. This provided inspiration for some aspects of our reference architecture but it does not provide sufficient guidance for determining the overall structure of an embedded system for two reasons. Firstly, rather than giving specific guidance, it raises a set of general questions about the behaviour of an application, the variability that it must support and its life expectancy and then describes the characteristics of the architectural styles and patterns, relying on the insights of the architects, who must be familiar with a wide range of concepts before the language can be applied. Secondly, developing the structure of embedded software is particularly challenging because it is usually a hybrid of architectural styles. For example, in the structure in Fig. 13.5, the software is largely structured as *layers*, but the operating system may be orthogonal to these, being accessible by all layers. Moreover, different architectural styles may be used in different layers, e.g. the *media/signal processing* may employ *pipes and filters* and the user applications may use *model-view-controller*.

POSA4 addresses the problem of how to interpret the general questions in the context of a specific application by preceding its pattern language with an extensive example of the development of a warehouse management system. This approach of using a running example is also used by Moore et al. in their B2B e-commerce reference architecture [18]. Here, the reader can draw parallels between these examples and their own applications by using the widely-accepted concepts of the information processing domain. This approach is less effective for embedded software because of the limited consensus on higher-level concepts.

Considering how better support might be given to architects, Kruchten states that “architecture encompasses significant decisions” about the software [11], therefore we might expect that the reference architecture will have made some decisions, which are applicable throughout its scope, and identify decision topics that have to be addressed for the current system. In their model of architectural knowledge de Boer et al. state, “decision making is viewed as proposing and ranking Alternatives, and selecting the alternative that has the highest rank . . . based on multiple criteria (i.e. Concerns)” [19]. The reference architecture should provide guidance for making such decisions.

Reed provides an example of a reference architecture for information processing, using an N-tier architecture and identifying the decision topics for each tier [12]. Here the decision criteria can be described concisely and unambiguously since they are based on widely-understood concepts and established technology standards. While this example is a valuable illustration of the role of reference architectures in supporting the creation of a wide variety of applications, many more decisions are required to cover the whole information processing domain.

A more extensive example is Zimmermann et al.'s reusable architectural decision model for developing a SOA, containing 300 decisions [20]. To guide the architect through the decisions, the decision model is structured by an extension of IBM's 7-layer SOMA model for SOA development [21]. However, for this guidance to be effective, and for the consequences of the decisions to be fully appreciated, the architects should already be familiar with the concepts in SOMA, otherwise the initial effort required to adopt it will inhibit the reference architecture's deployment. This consensus is lacking in the CE domain, as highlighted by the problems of enforcing several hundred architectural rules for a single concrete CE architecture, developed across multiple sites, reported by Clerc et al. [22]. The adoption of a reference architecture in Zimmermann's form would be even more challenging, given the broader scope of the domain and the lack of an initial structure in which to position the decisions. Therefore, while architects claim that they do not want to be unduly constrained, and following early trials with a structure comparable to Zimmermann's, we concluded that our reference architecture had to be more prescriptive. Therefore, rather than beginning with a sequence of decision topics from which the architect would develop their architecture, it begins by proposing a design, followed by the decision topics, with alternatives and design rationale, that should be considered where the architects believe the recommended design to be inappropriate.

Many of the decisions relate to the satisfaction of non-functional requirements (NFRs), comparable to TOGAF's *service qualities*. We have extended Gross and Yu's approach to guiding the selection of design patterns, given the product's NFRs [23], which is itself based on Chung et al.'s NFR Framework [24].

Muller and Hole report on the views of architects, developing embedded software in several industries, on the role of reference architectures and how they can be developed [25]. They show how reference architectures are informed by feedback from the field, in terms of both new stakeholder requirements that would be satisfied if the architecture could be changed, and the constraints that should be imposed on new architectures to avoid problems that have occurred in practice. They note that one of the main objectives for developing a reference architecture might be to ensure that integration effort is low and, indeed, this was the starting point for developing our architecture. However, before describing how the architecture was developed, we will first scope the domain of CE products and identify the viewpoints that the reference architecture should contain.

13.4 Required Scope and Viewpoints of the Reference Architecture

When developing the reference architecture, we have to address of the scope of the CE domain to be covered by the architecture and the identification of appropriate viewpoints. These can be considered in relation to the business aims that motivated

the development of the architecture, which go beyond the needs of the requirements engineer. Indeed, the search for the form and content of the reference architecture was driven by the desire to avoid integration problems. The overall set of business aims were as follows:

- To enable requirements engineers to ensure that the product makes the best use of its resources and to ensure that resource management policies result in a consistent user interface, as introduced in Sect. 13.2.1. This is particularly important the first time that a feature is incorporated into a product category.
- To support requirements engineers in the selection of software components from external suppliers, as introduced in Sect. 13.2.2.
- To support software component suppliers in establishing the requirements for components to be supplied to CE manufacturers, as introduced in Sect. 13.1.
- To enable architects to exchange best practices across different product categories, having different concrete architectures. This is particularly to avoid problems during component integration and to improve maintainability.
- To support internal standardization to facilitate reuse as the requirements of different product categories converge.

Note that these aims do not include aspects, such as hardware-software co-design, where specialised analytical models, such as Synchronous Data Flow [26], specific to the nature of the processing, are used to optimise the system architecture. While such optimisation is critical to the success of the product, it normally addresses only a small proportion of the code. The overall software architecture must ensure that the remainder of the software does not compromise the performance of these critical elements.

In selecting the application domain to be addressed by the reference architecture, we have taken the broad domain of CE products, rather than developing separate reference architectures for each product category, such as TVs and mobile phones. This is for several reasons. During the requirements phase, we need to be able to handle the expansion in the scope of functionality supported by a product category, whether with an entirely novel feature or a feature that was originally developed for another category. By abstracting from specific functionality we are able to provide support for feature combinations that had not been anticipated. A broad scope is also required to exchange best practices and to promote reuse between product categories. Without a reference architecture, the differences in requirements and business models can obscure their commonalities. A broader reference architecture will be exposed to a wider range of design choices, which makes it more effective when incorporating COTS components, and will be more satisfactory for architects to use since it cannot be overly prescriptive. Finally, the effort of developing a broadly scoped architecture can be recouped over more development projects.

The scope of the CE application domain is characterised in two ways:

1. An abstract context diagram for a generic CE product, shown in Fig. 13.1. This informal diagram is annotated with examples of actors and protocols.
2. A list of the general capabilities of these products, namely:

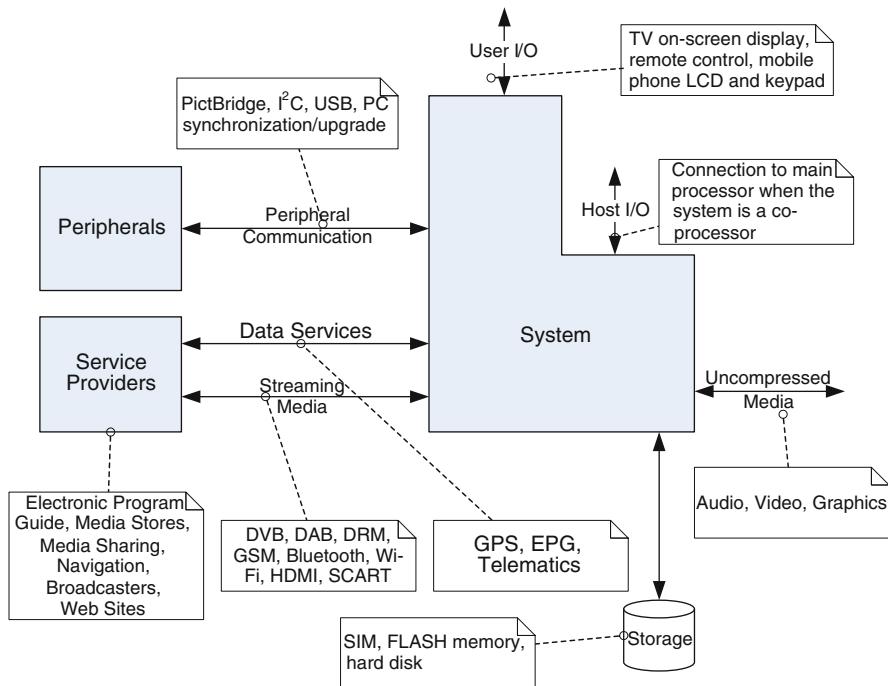


Fig. 13.1 Context diagram for the domain of CE products

- The reception, generation, storage, processing and rendering of audio, video and graphics.
- Interaction with data services.
- Communication through wired and wireless connections.
- Interaction with peripheral devices.
- Interaction with a user, either supporting the physical user interface or, if only a co-processor is being developed, its control interface to a host processor.

The following are the general non-functional requirements and constraints of the products in this domain:

Requirements: The products must meet firm and soft real-time performance constraints. Their user interfaces must be responsive, even when actions are inherently time-consuming. Most actions should be interruptible, with the system transitioning smoothly to respond to the new command. Actions can be triggered by both user commands and spontaneous changes in a product's environment. Several features may be active concurrently. Products are usually members of a product line, whose members may address different geographic regions or ranges of features.

Constraints: The products have limited resources, such as application-specific hardware and processing power and memory. They have limited user interfaces,

in which feature interaction must be carefully managed, rather than providing virtualized interfaces using a windowing system. Because any change to the state of the hardware may be directly observable by end-users, product-level requirements may impose constraints on the exact sequence in which the sub-steps of an action are made. For instance, when changing the channel of a TV, some manufacturers favour only displaying the picture when all its parameters are known, whereas others display it at the earliest opportunity and then adapt its presentation as its characteristics, such as its aspect ratio, are detected. While the post-conditions are the same in both cases, the user experience is quite different.

Since our reference architecture is used to create the software architecture description for a specific product line, its viewpoints are aligned with those of the existing architecture descriptions. Philips and NXP Semiconductors used Obbink et al.'s Component-Oriented Platform Architecting (COPA) method [27], which addresses the development of a product family from the perspectives of *business*, *architecture*, *process* and *organisation* (BAPO), as elaborated by van der Linden et al. [28]. COPA's *architecture* perspective has five viewpoints: the purely commercial viewpoints of *customer* and *application*, the purely technical viewpoints of *conceptual* and *realization*, and a shared *functional* viewpoint that represents the traditional product line requirements specification [29].

To support requirements engineering it might appear to be best to focus on the commercial viewpoints, which characterize business value of the software and the context in which it will be used. For instance, the COPA *application* viewpoint is comparable to the TOGAF *Industry Architecture* [13]. However, we see that apparently similar products, such as TVs sold in the retail market and set-top boxes supplied to cable TV operators, have quite different business models and value propositions. Similarly, the business model of a supplier of components into these markets will be very different from that of a product integrator, which will usually translate into differences in the technical viewpoints to support a greater degree of variability. Consequently, the commercial viewpoints are usually specific to a business unit, which can develop views that are more specific than is appropriate for a reference architecture covering a broad domain.

In NXP Semiconductors, the technical viewpoints are documented according to a "Software Architecture Design" (SAD) template, described and evaluated by van Dinther et al. [30]. This template is used in several companies and is an extension of Kruchten's "4 + 1" model [31]. The template uses three viewpoints, *conceptual*, *logical* and *physical*, which approximate to COPA's *functional*, *conceptual* and *realization* viewpoints. For clarity, we will use COPA's terminology in the remainder of this chapter. The SAD template informally describes the information that should be included in each viewpoint, each of which is divided into *static* and *dynamic* parts.

The *functional* view in an instantiated SAD is application-specific, containing the requirements and variability model for the concrete product line. In contrast, the corresponding view in our reference architecture is primarily intended to orientate new users. It is limited to the illustration of its scope, shown in Fig 13.1, together

with an elaboration of the capabilities, requirements and constraints listed at the beginning of this section. The generic elements of the *realization* viewpoint relate to the rules for the directory structure and permitted dependencies, together with coding standards, which are addressed in the company’s reuse standards and will not be elaborated further.

The primary focus of the reference architecture is COPA’s *conceptual* viewpoint. Here, we require a *structural* model that is abstracted from concrete architectures and from particular applications, while still being sufficiently specific to address issues of resource requirements and component scoping, introduced in Sect. 13.2. The *texture* of the reference architecture must identify the decision topics that must be addressed with regard to behaviour. In practice, the granularity of the *structural* model also had to be fine enough to express the alternative policies in the *architectural texture*, e.g. to be able to express that alternative policies allocate responsibilities to different components.

As noted by Muller and Hole, the reference architecture can be informed by proven concepts and known problems in existing architectures [25]. Given that it was not known at the outset what information the reference architecture should contain, nor how it should be structured, this approach was taken to develop insight incrementally. This was first by mining reusable architectural information from previous developments and then by structuring this information into a reference architecture that can be used from the beginning of a new development, as will be described in the following section.

13.5 Developing a Reference Architecture for the CE Domain

Given that few concepts from information processing can be applied to embedded software, generally-recognised concepts for embedded software are only found at the level of state machines and the services offered by real-time operating systems. These concepts are too low-level for the early stages of architectural development. Furthermore, the software architectures of concrete embedded products are highly influenced by the associated hardware architecture and the diversity that this introduces obscures the commonalities across product categories that could form a reference architecture to support early decision-making.

The main challenges in the development of our reference architecture were ascertaining what information should be documented and how it should be structured. For instance, while the inclusion of a *structural* model in a reference architecture is uncontentious, what should it contain? The definition of “architecture” in IEEE 1471 includes “the fundamental organization of a system embodied in its *components*, their relationships to each other . . .” [32] but what should be the semantics of a *component* in a model abstracted from any specific product?

It was even less certain *a priori* what decision topics and other information should be included in the architectural *texture*. However, Kruchten states that one of the purposes of the architectural description is to “be able to understand how the

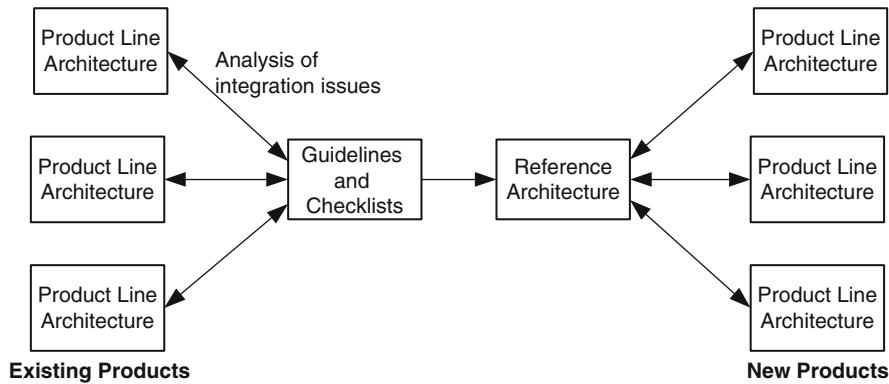


Fig. 13.2 Information flow in the development of the reference architecture

system works” and to “be able to work on one piece of the system” [11]. Consequently, one way of identifying the necessary information is through the study of the root causes of failures that occurred during integration and to record and abstract those that resulted from insufficient architectural information. Furthermore, architectures should support evolution and a similar approach can be taken with components that have poor maintainability.

Our reference architecture was therefore developed in two phases. Firstly the problems encountered during the integration and maintenance of existing products were studied to obtain guidelines and checklists that could be used to review new concrete architectures. Secondly, the understanding gained here was used in the construction of a reference architecture that would support the creation of concrete architectures, providing support from the earliest phase of the development. The information flow is illustrated in Fig. 13.2.

13.5.1 Mining Information from Earlier Developments

As noted by Jackson, “the lessons that can be learned from engineering failures are most effective when they are highly specific” [33]. Therefore, when setting expectations for what will be achieved from the study of previous projects, we expect much more than merely reiterating that decisions should be recorded for each of the aspects identified in COPA, e.g. initialization, termination, fault handling [27]. However, it is challenging to gain insights on developments incorporating COTS components, given the limited information that is usually provided and the reluctance of suppliers to discuss problems in detail. Therefore, in searching for decision topics to include in the architectural *texture*, we first exploited the experience of multi-site development within a single company. Here, while architectural decisions must be explicit because of the limited communication between the sites, we were not hampered by IP issues. It was only after

developing an understanding of the architectural issues that affect integration that the study was broadened to include developments incorporating third-party software and issues encountered during subsequent maintenance.

We began with a study of 900 failures that occurred during the initial integration of the sub-systems for a product-line of TVs, developed across multiple sites [34]. These sub-systems contained no legacy code, were developed according to a new architecture, and had not yet accrued changes as the result of evolution. Furthermore, all versions of code and documents, together with comments in the problem tracking system, were available. As such, these failures were an ideal candidate for identifying policies that should have been defined at an architectural level. Many of these related to component communication and synchronization.

This study did not merely result in a catalogue of the problems encountered and the design patterns that would have avoided them. Such a catalogue would have been difficult to reuse by different architects in the context of another development. Instead, a framework was created that not only identified decision topics for the observed problems, but also identified new problems and decision topics that could occur in other contexts. This framework is illustrated in Fig. 13.3.

The key to achieving this generalisation was the identification, for each architecture-related problem, the property (*specific intent*) that had been violated and the general nature of the interaction between the components (*interaction context*). Examples of interaction contexts are *synchronous function calls* and *asynchronously-communicating state machines*, which are straightforward for architects to recognise within any architecture. The *generic intents* are an abstraction of the *specific intents*, formulated in a way that they can be reinterpreted in different interaction contexts, thereby anticipating new decision topics. Table 13.1 shows examples of intents and their specializations.

For each *intent*, several alternative *policies* might be identified that can guarantee its satisfaction. The implementation of each policy is documented in terms of a *design pattern*. In some cases, the choice of policy is arbitrary but it must be consistent throughout the architecture. Inconsistencies may arise when incorporating third-party components.

As an example, the problem of notification handlers reading uninitialized variables, listed in Table 13.1, can arise when a server completes an action on one thread before the function call that requested that action, made on another thread, returns to its client. The reference architecture identifies three different policies that can satisfy this intent. However, these are mutually incompatible, so a global choice must be made.

More often, the choice of policy will be guided by the NFRs for the product. Here, Gross and Yu's approach to selecting design patterns is used to illustrate the relative merits of the alternatives in relation to the NFRs [23]. We extend their notation by adding the intents to their *softgoal interdependency graphs*, together with the argumentation of how these are satisfied by each of the design patterns.

It may be that different choices will be made at different layers in the architecture. For instance, in addressing the problem of ensuring the correct ordering of notifications, listed in Table 13.1, there is a trade-off between performance and

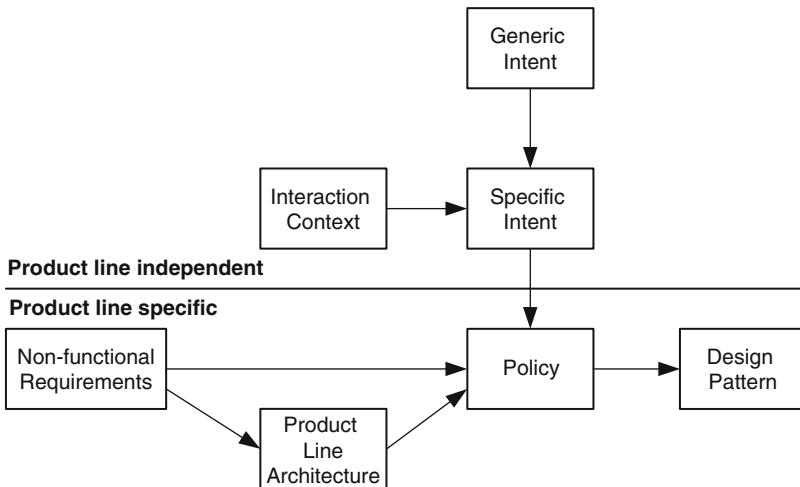


Fig. 13.3 Framework for reasoning about policies for communication and synchronisation

Table 13.1 Example *intents* and their specializations for specific *interaction contexts*

Interaction contexts	Intents	
Generic	Variables must be initialized before they are used	Designs should be insensitive to the order of completion of unconstrained activities
Notification handling	Variables that will be read by a notification handler must be set before the handler executes.	The notifications of a specific event should be generated and delivered in the same order.
Power-up	Avoid cyclic dependencies between sub-systems during initialization.	

configurability. Rather than making a single decision for the entire product, a policy that favours performance may be selected for the lower levels of the software and a policy giving greater configurability may be used at higher levels. The reference architecture is a vehicle for the requirements engineer to understand how the efficiency of different parts of the software contributes towards the overall product performance.

Figure 13.3 shows how, for a concrete product line, the choice of policy would be recorded in the *architecture*. In contrast, the reference architecture would contain a decision topic with the design alternatives.

Having developed this understanding of integration issues and established a framework for structuring decision topics, several analyses were undertaken of the integration of software from external suppliers [35]. Since this software had been developed with no knowledge of the architecture of the products in which it was to be integrated, these studies revealed a much larger range of policy

mismatches, e.g. in relation to resource management and error handling. These mismatches were included as new alternatives in the reference architecture.

Another important class of mismatches related to the scoping of the functionality supported by components, which caused some of the problems introduced in Sect. 13.2.2, for which further examples will be given in Sect. 13.6. These mismatches gave insight into the granularity required of the reference architecture's structural model to be able to compare the scopes of components.

Finally, as the product line architectures were subjected to adaptive maintenance, further studies were undertaken of components having low maintainability. Although the principles of object-oriented design, as articulated by Martin [36], would have addressed these problems, these are less easy to apply in the C programming language, which dominates embedded software development. Furthermore, the flexibility that these principles support is often at the expense of performance, so guidance is required on their application. Here, it is crucial for architects and requirements engineers to have a shared roadmap to ensure that components are structured to support anticipated changes in requirements.

13.5.2 *The Organisation of the Reference Architecture*

As introduced in Sect. 13.4, the reference architecture is primarily intended to support the creation of COPA's *conceptual* viewpoint for the concrete product line [27]. Its purpose is to facilitate communication between its stakeholders: the requirements engineer, the software architect, the project manager and the test architect. It must support the mapping of requirements onto a structural model so that the resource usage of each requirement can be ascertained. It must define concepts, such as different styles of resource management, which facilitate reasoning about the product.

The principal organisation of the *conceptual* viewpoint is provided by the architectural *structure* and *texture*, as introduced in Sect. 13.4 and shown in Fig. 13.4. The architecture is documented as dynamic web pages, which link the elements in the structure to pages that guide the user through the decision topics relevant for that element and its interaction with its immediate neighbours. These decision topics are underpinned by the orthogonal texture axis, which addresses the topics in greater depth but in a more abstract context, enabling consistent choices to be made throughout the product.

Additionally, given the variation in the *conceptual* views of existing architectures, a new user requires some orientation. This is provided through the *scope* of the reference architecture, which is an abstracted form of the COPA *functional* viewpoint [27]. It contains the context diagram, already shown in Fig. 13.1, which includes the mapping of examples of concrete actors and protocols, used in existing products, to the abstract representations used in the remainder of the architecture. It also lists the general capabilities of CE products, already described in Sect. 13.4, together with their typical dynamic behaviour at a product

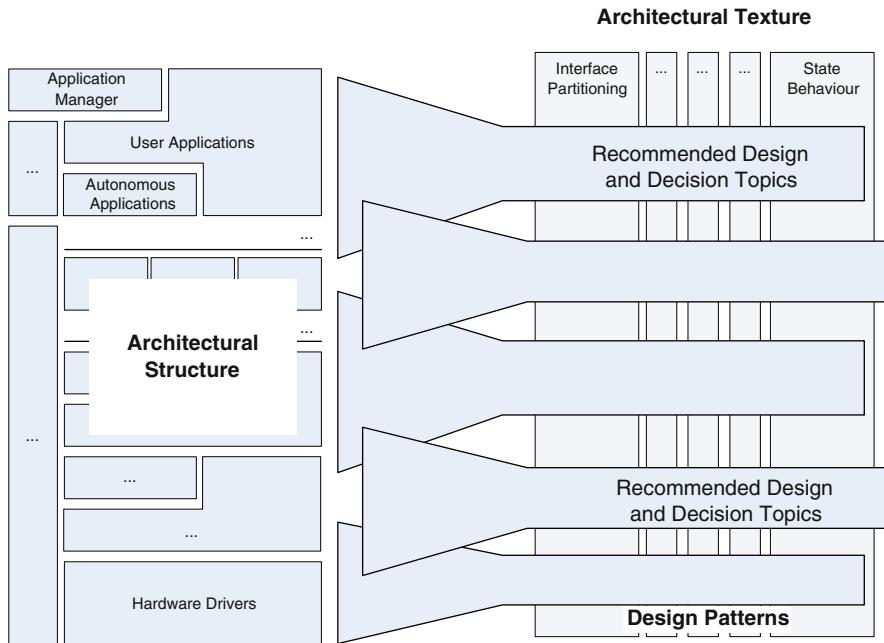


Fig. 13.4 Primary organisation of the reference architecture

level and typical non-functional requirements. This assists the architect in interpreting the concepts of the framework in the context of their specific product.

The top levels of the architectural structure and texture will now be described in more detail. This will be followed by a description of how they are linked through the lower levels of the reference architecture, such as through the recommended designs and decision topics.

Structure: Normally a structural model shows a partition of the software that can be traced to concrete system elements [3]. However, as discussed in Sect. 13.3, our reference architecture is more generic. As shown in Fig. 13.4, the top level of the structure is the primary entry point to the architectural guidance, so its abstraction level must be high enough to be broadly applicable, yet concrete enough to be usable in practice. To achieve this, we have adopted Wirsfs-Brock and McKean's responsibility-driven design (RDD) [37]. In RDD the *role* of a class is defined as "a set of related responsibilities," which may be abstracted from their specific functionality. The structural model, illustrated in Fig. 13.5, identifies the roles that are normally present in a CE product. Following the RDD approach, the roles are annotated with their purpose and responsibilities. In use, RDD provides the method for mapping the requirements for a specific product onto the responsibilities that define these roles. This mapping is assisted by annotating many of the roles with examples from concrete architectures that would be familiar to all architects in the company.

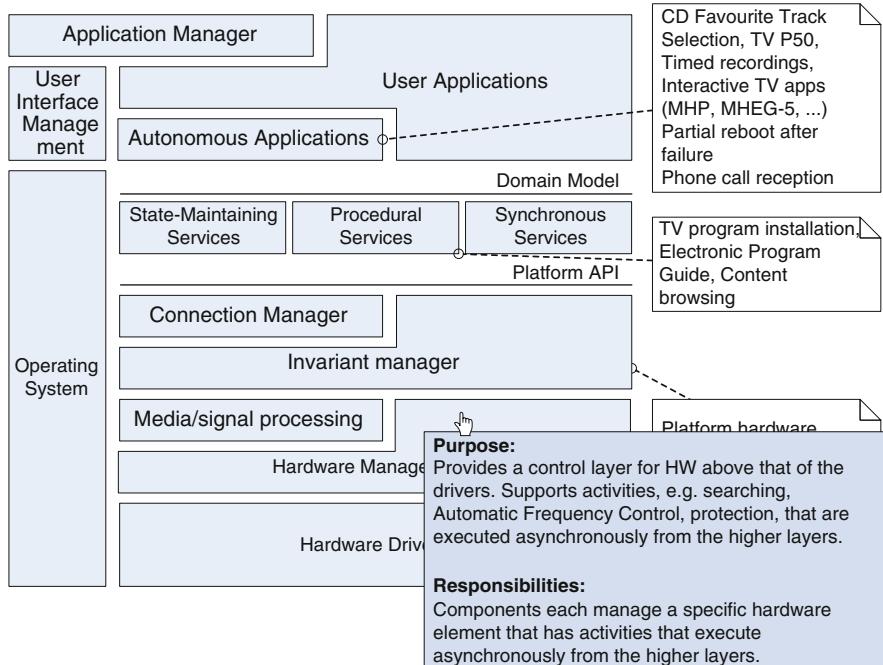


Fig. 13.5 Structural model of the reference architecture with an example role description

The structural model, shown in Fig. 13.5, is sufficiently fine-grained to be able to compare the scopes of different components and to distinguish between different sets of decision topics identified in the studies in Sect. 13.5.1. For example, although the purposes of the three *services* roles shown in Fig. 13.5 are comparable, they have different behavioural characteristics and are therefore distinct in the model.

Considering the problems of scoping, introduced in Sect. 13.2, each role relating to a particular feature should be allocated to a single concrete component if integration problems are to be avoided. This allocation is clarified in more detail in the *recommended design* linked to the role, an example of which is in Fig. 13.6, where the *collaborations* of the roles, the other element required by the RDD method, are made explicit. These relationships are not present in the top-level structural model since they can vary, depending on the decisions made. For instance, the *invariant manager* in Fig. 13.5 might be implemented as a single state machine or, as described by van Ommering, by protocol handlers integrated into each component in the next layer down [38].

Texture: Our model largely follows the classification used in POSA4 [17], but with the contents adapted from the concerns of distributed systems to those of embedded software. It contains guidelines on both the structure of the software, e.g. for interface and component partitioning to support variability and evolution, and on

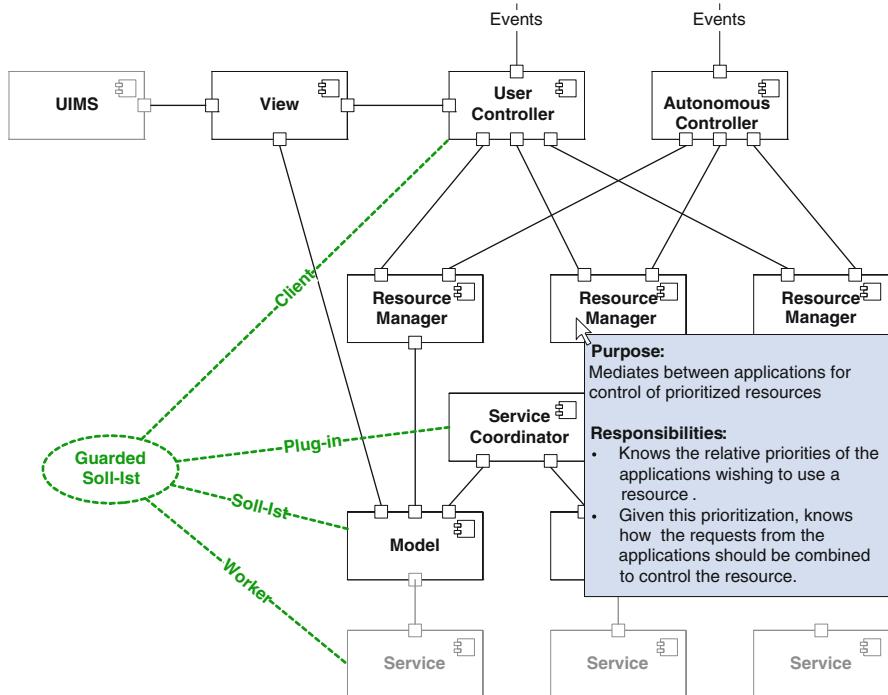


Fig. 13.6 Recommended design for the applications roles and their interfaces to the services, highlighting one of the three domain-specific patterns used in this design

the rules or decision topics for behaviour. Some examples of the categories of behavioural guidelines are:

Synchronization: This category includes the rules or decision topics identified in the studies of integration failures described in Sect. 13.5.1. Here the rules or decision topics are classified according to their interaction contexts and are therefore reusable throughout the architecture.

State behaviour: This extends the taxonomy of modal behaviour in *POSA4* [17] to cover the much larger set of state-related patterns referenced in the architecture, providing both consistency in their description and a broader perspective on the options available to architects.

Resource management: The different classes of resources are an important concept to help the requirements engineer to understand how architectural choices affect feature interaction. The category identifies three different classes of resource management policies, namely *synchronized*, *prioritized* and *virtual*, and the issues that must be considered with each class. These definitions and arguments are widely-referenced throughout the architecture.

Having described the top level of the architecture, we will describe how the structural model is linked to more detailed recommendations and guidance on

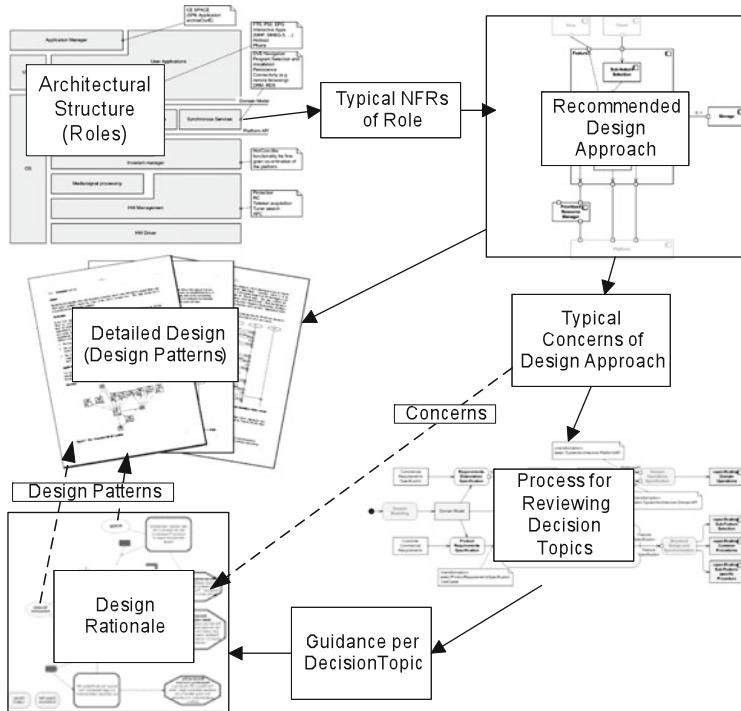


Fig. 13.7 Example of steps in the guidance through architectural decisions

architectural decisions. Each of the roles is hyperlinked to a textual description of the most relevant NFRs, a recommended design approach and guidance on selecting design alternatives, as illustrated in Fig. 13.7.

The recommended design approach is expressed in terms of a UML component model, in which the roles and responsibilities in the top-level structural model are expressed at a finer grain. Unlike the top-level structural model, component connectors now indicate the collaborations between components. Figure 13.6 shows an example component model, including a tooltip describing the purpose and responsibilities of one of the components. More detailed information is provided through design patterns, documented in a conventional form as part of the orthogonal architectural texture. The model allows the user to display the different patterns in which the components collaborate, one of which is shown in Fig 13.6.

While the individual pattern descriptions are independent of where in the structure of the architecture they might be applied, Fig. 13.7 shows how the recommended design is also linked to a description of the concerns addressed by that design in its particular context. These concerns will include a re-interpretation of the *intents*, identified in Sect. 13.5.1, for the current role in the architecture.

This is followed by a diagram illustrating the process for reviewing the decision topics in the light of the particular product requirements or the characteristics of

pre-existing components. A UML activity diagram is used, showing the tasks and resulting work products. Each task in this model is hyperlinked to decision topics, such as the alternative policies for handling notifications, introduced in Sect. 13.5.1. Each topic has a detailed discussion of the forces involved and examples of decisions taken in earlier product developments, obtained from the studies described in Sect. 13.5.1. Throughout the guidance, hyperlinks are made to definitions and discussions in the architectural texture, where the issues are described in a more general context. This both allows consistent decisions to be made throughout the product and reduces the amount of material that must be presented in the context of each of the individual roles in the architecture.

Finally, the design rationale for each decision topic is presented using our extension of Gross and Yu's approach to selecting between design alternatives, based on their support for different NFRs [23]. As described in Sect. 13.5.1, we add the *intent* as a goal that must be satisfied by all design alternatives.

Early trials of the use of the architecture confirmed that the approach of beginning with a recommended design had a shallower learning curve compared with that of a pure pattern language, such as that in *POSA4* [17], in which there are no default decisions. Such pattern languages require that the architect has a good initial grasp of many abstract concepts.

A general principle behind the use of web pages to document the reference architecture is that a user should be provided with the essence of recommendations in the first instance, but that it is easy to drill down and get more details when required. For example, Fig. 13.6 shows both tooltips and dynamic content, used for the overlay of different design patterns. The latter provides navigation to other pages through hyperlinks. Indeed, the ability to provide details on demand is the key to presenting a full design rationale in a compact and comprehensible form.

Having described the development and organisation of our architecture, the following section describes how it can be used during requirements engineering.

13.6 Usage of the Reference Architecture by Requirements Engineers

Our reference architecture both allows requirements and architectural decisions to be assessed together and prompts the requirements engineer to elicit how particular issues, e.g. resource management, should be handled. In this regard, the architecture also implicitly includes some of the *concerns* of Jackson's Problem Frames [39], another element of the "Twin Peaks" approach. Figure 13.8 is an informal activity diagram that illustrates the role of the reference architecture in requirements-related activities.

The diagram includes three feedback loops from the architecture to the functional requirements specification:

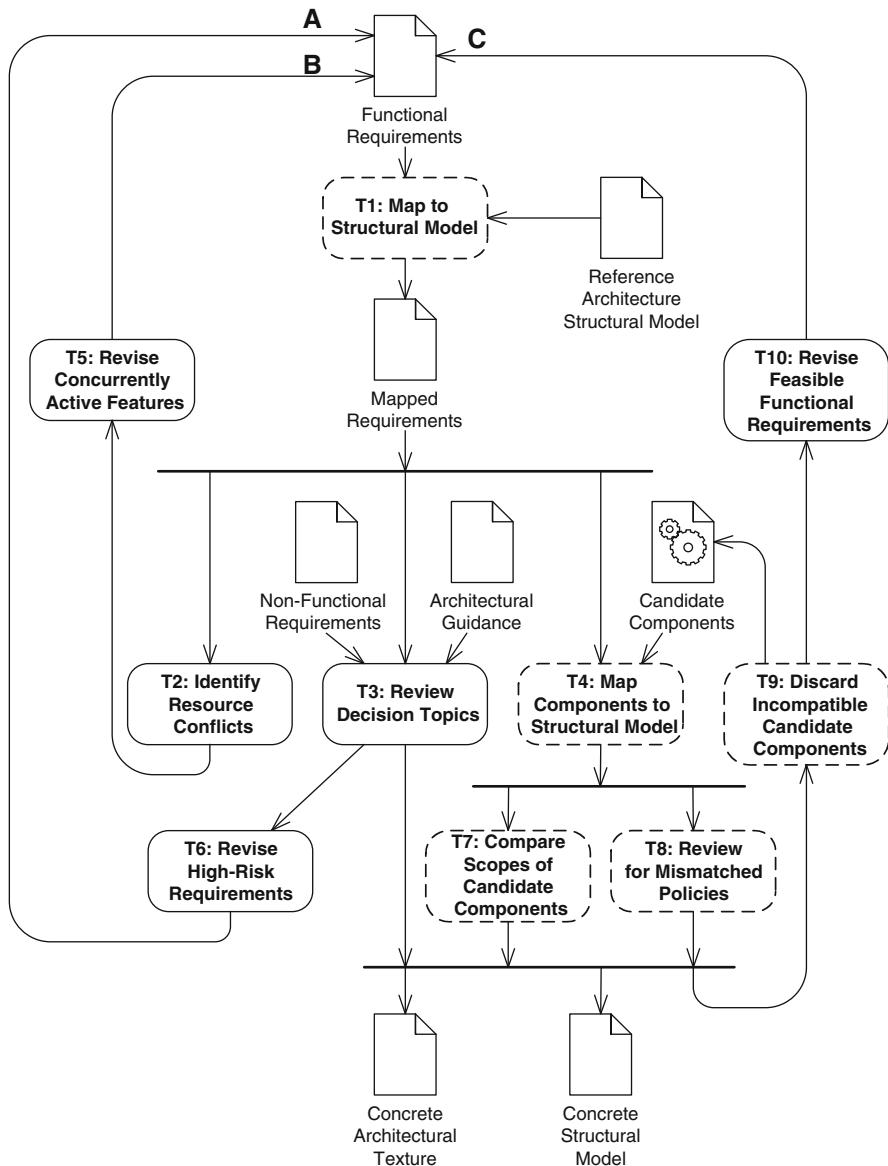


Fig. 13.8 Role of the reference architecture in requirements-related tasks. Tasks involving requirements engineers have a solid outline

A. Revise the requirements, having reviewed the architectural decisions that would be required to satisfy the related NFRs. This might discard requirements with a high technical risk.

- B. Identifies cases where contention for resources restricts the concurrent availability of features. Where features cannot be active concurrently, new requirements may be added relating to the transition between those features.
- C. Where third-party components are to be used, low-priority requirements are removed if they are only supported by components that are architecturally-incompatible with the remainder of the product.

The use of the reference architecture will be illustrated by an example of the integration of the PictBridge protocol into a mobile phone. This will show how feature interaction can be detected and resource management policies assessed.

PictBridge [40] is a standard that allows a user to select, crop and print photographs, using a camera connected directly to a printer, without requiring a PC. The standard only addresses the protocol between the camera and printer, and not the camera's user interface or how the feature should be implemented in the camera.

Consider establishing the requirements the first time that this feature was integrated into a mobile phone, where it is to be implemented by a COTS component that has previously only been integrated in a conventional camera. The requirements engineer must:

- Determine a complete set of end-user requirements for the PictBridge feature.
- Identify potential feature interaction with the remainder of the phone's features and identify how they can be resolved satisfactorily.

These aims are addressed, with reference to Fig. 13.8, with the following sequence of activities:

- *T1: Map the functional requirements of the PictBridge feature onto the roles in the structural model.* The component implementing the protocol is an example of a *procedural service* (see Fig. 13.5), which is one that executes a series of actions, normally running to completion. The PictBridge component will need access to the hardware drivers for the USB interface and the memory in which the photographs are stored. In addition, the feature will require a user interface.
- *T4: Map the scope of candidate PictBridge COTS components onto the structural model.*
 - Survey the COTS components that implement the PictBridge feature.
 - The scope of each promising candidate is identified from studying the features it supports and its interface specification. For instance, are its *provided* interfaces restricted to the PictBridge protocol, or does the component also provide some user interface functionality?
- *T7: Compare the scope of the candidate PictBridge COTS components with those of other features.* Will it be possible to maintain a consistent user interface across all features? If not then *T9: discard incompatible candidate components*. If these components also support some unique functionality that cannot otherwise be implemented, *T10: revise the feasible functional requirements*.

- *T2: Identify resource conflicts.* Identify the features that could be active concurrently and detect feature interaction. Since the user interface of most phones only permits one application to be selected at a time, we are primarily concerned with interference from features of the phone that are *autonomous applications* (see Fig. 13.5), i.e., features that make calls to the services without having been explicitly selected through the user interface. For a phone, these are incoming telephone calls and text messages. How should the product react when a call or message is received when the PictBridge feature is active?
 - What are the consequences for the user interface? PictBridge implementations on cameras normally retain control of the user interface while the photos are being printed. Would it be possible to continue printing in the background on a phone, so that it could continue to be used for other purposes? The architectural guidance for the user applications and their interface to the services includes a recommended design for managing the transfer of resources between applications, shown earlier in Fig. 13.6. Do the available components have the necessary synchronization functions to implement such design patterns?
 - Considering the lower levels of the structural model, can both features be active concurrently? Does the file system support concurrent access from multiple applications and are there sufficient memory and processor resources to support both features?
 - Based on this analysis, *T5: revise the requirements for concurrently active features.*
- *T8: Review the COTS components for mismatched policies.* The policies for how the feature should be activated and terminated should be consistent with those of other features. Many cameras activate the PictBridge feature only when the camera is switched on while connected to a printer, whereas a phone user would not expect to have to switch the phone off and on in the same way. Mismatches often occur in state behaviour when components developed for one category of product are integrated into a product of another category. Mismatches may also occur in the selection of design alternatives, such as those for handling notifications, introduced in Sect. 13.5.1. Such mismatches can be detected by architects during the later steps in Fig. 13.7 and may require unacceptably complex glue code to integrate the component into the remainder of the system. Again, following this analysis, *T9: discard incompatible candidate components* and, if necessary, *T10: revise the feasible functional requirements* to remove those only supported by the discarded components.

A benefit of using the reference architecture, even when a concrete architecture already exists for the mobile phone, is that it supports the comparison of the scopes of COTS components implementing different features. This makes it easier to detect feature interaction and identify requirements for resource management.

13.7 Conclusions

Establishing the requirements for a CE product has many challenges, such as identifying user needs or desires for a diffuse market, identifying features that will differentiate a product from the competition, finding the right price/performance points and developing a simple and intuitive interaction style.

Much of the requirements specification for a CE product addresses the interaction between features, either because they are active concurrently or because they can be activated spontaneously by events in the environment. Even if all the software were to be bespoke, support is required to identify the sources of feature interaction, which arise both from resources that cannot be shared and from performance constraints. These problems are compounded when features are implemented by COTS components, which may initially have been developed for different product categories, having different overall requirements.

We have developed a reference architecture that covers a broad range of CE products. The breadth is required so that it can support the addition of novel features that were not anticipated at the time of the architecture's creation, to enable the exchange of best practice between development groups and to promote reuse across product categories. The abstraction level is set high enough to cover this broad scope, while still being concrete enough to have clear relevance for product development. The architecture addresses the lack of consensus on architectural concepts in the CE product domain by proposing a structure of roles with recommended designs, while providing guidance on alternative design choices. This is based on architectural information mined from earlier multi-site and COTS-based developments.

Architectural texture provides consistency, with design guidelines that can be used throughout the architecture. This information would also facilitate the creation of variants of the architecture for other industries with similar technical characteristics, e.g. automotive engine management or medical image acquisition.

Because it provides an initial structure, our reference architecture is of benefit from the beginning of the requirements phase when identifying resource constraints or conflicts. For COTS-based developments, the architecture provides a framework for comparing the scope of functionality of COTS components, both to identify which features can be active concurrently and to ensure a consistent interaction style.

Our architecture was developed in the context of a CE manufacturer with a broad product portfolio. A company with a narrower range of products might be tempted to move along TOGAF's architecture continuum, representing more application-specific information. However, this would give little support for the integration of novel functionality.

The broad scope of our architecture is also valuable for COTS component suppliers, for whom it can be difficult to anticipate all the architectures used by potential customers. The design alternatives used in our reference architecture give an insight into what might be encountered. The architecture also provides a vehicle

for detailed discussions with customers without either party exposing their IP, which will be of increasing value as the CE industry transitions away from vertically-integrated companies towards supply chains or ecosystems.

References

1. International Organization for Standardization (1998) ISO/IEC 10746–1:1998, information technology – open distributed processing – reference model: overview. International Organization for Standardization, Geneva
2. Nuseibeh B (2001) Weaving together requirements and architectures. *IEEE Computer* 34(3):115–117
3. Ran A (2000) ARES conceptual framework for software architecture. In: Jazayeri M, Ran A, van der Linden F (eds) *Software architecture for product families*. Addison-Wesley, Redwood City
4. Calder M, Kolberg M, Magill EH, Reiff-Marganiec S (2003) Feature interaction: a critical review and considered forecast. *Computer Networks* 41(1):115–141
5. Tun TT, Jackson M, Laney R, Nuseibeh B, Trew T (2009) Specifying features of an evolving software system. *Softw Pract Exper* 39(11):973–1002
6. LiMo Foundation (2010) Welcome to LiMo. <http://www.limofoundation.org>. Accessed 6 June 2010
7. International Organization for Standardization (2007) ISO/IEC 23004–2:2007, information technology – multimedia middleware – part 2: Multimedia application programming interface (API). International Organization for Standardization, Geneva
8. Khronos Group (2010) OpenMAX-the standard for media library portability. <http://www.khronos.org/openmax/>. Accessed 6 June 2010
9. Suoranta R (2006) New directions in mobile device architectures. Paper presented at the 9th EUROMICRO conference on digital system design: architectures, methods and tools. Dubrovnik, Croatia, 28 August–1 September 2006
10. van der Linden F (2005) Documenting variability in design artefacts. In: Pohl K, Böckle G, van der Linden F (eds) *Software product line engineering*. Springer, Berlin, pp 116–134
11. Kruchten P (2000) The rational unified process: an introduction. Addison-Wesley, Redwood City
12. Reed P (2002) Reference architecture: the best of best practices. <http://www.ibm.com/developerworks/rational/library/2774.html>. Accessed 6 June 2010
13. The Open Group (2009) The open group application framework (TOGAF). The Open Group, San Francisco
14. Organization for the Advancement of Structured Information Standards (2008) Reference architecture for service oriented architecture. Organization for the Advancement of Structured Information Standards, Burlington
15. Burns A, Baxter G (2006) Time bands in system structure. In: Besnard D, Gacek C, Jones CB (eds) *Structure for dependability: computer-based systems from an interdisciplinary perspective*. Springer, London, pp 74–88
16. Eeles P, Cripps P (2010) The process of software architecting. Addison-Wesley, Boston
17. Buschmann F, Henney K, Schmidt DC (2007) *Pattern-oriented software architecture: a pattern language for distributed computing*. Wiley, Chichester
18. Moore B, McDougall A, Rolshausen E, Smith S, Stehle B (2003) B2B e-commerce with webSphere commerce business Edition v5.4. Report SG24-6194-00. IBM, Raleigh
19. de Boer RC, Farenhorst R, Lago P, van Vliet H, Clerc V, Jansen A (2007) Architectural knowledge: getting to the core. Paper presented at the third international conference on the quality of software-architectures, Medford, 12–13 July 2007

20. Zimmermann O, Zdun U, Gschwind T, Leymann F (2008) Combining pattern languages and reusable architectural decision models into a comprehensive and comprehensible design method. Paper presented at the seventh working IEEE/IFIP conference on software architecture, Vancouver, 18–22 Feb 2008
21. Arsanjani A (2004) Service-oriented modeling and architecture: how to identify, specify, and realize services for your SOA. <http://www.ibm.com/developerworks/library/ws-soa-design1/>. Accessed 7 June 2010
22. Clerc V, Lago P, van Vliet H (2007) Assessing a multi-site development organization for architectural compliance. Paper presented at the sixth working IEEE/IFIP conference on software architecture, Mumbai, 6–9 Jan 2007
23. Gross D, Yu E (2001) From non-functional requirements to design through patterns. Requirements Engineering 6(1):18–32
24. Chung L, Nixon BA, Yu E, Mylopoulos J (2000) Non-functional requirements in software engineering. Kluwer, Boston
25. Muller G, Hole E (2007) Reference architectures; why, what and how. System architecture forum. http://www.architectingforum.org/whitepapers/SAF_WhitePaper_2007_4.pdf
26. Lee EA, Messerschmitt DG (1987) Static scheduling of synchronous data flow programs for digital signal processing. IEEE Transactions on Computers 36(1):23–35
27. Obbink H, Müller J, America P, van Ommering R (2000) COPA: a component-oriented platform architecting method for families of software-intensive electronic products. Paper presented at the software product line conference, Denver, 28–31 Aug 2000
28. van der Linden F, Schmid K, Rommes E (2007) Software product lines in action: the best industrial practice in product line engineering. Springer, Berlin
29. Pohl K, Weyer T (2005) Documenting variability in requirements artefacts. In: Pohl K, Böckle G, van der Linden F (eds) Software product line engineering. Springer, Berlin, pp 89–113
30. van Dinther Y, Schijfs W, van den Berk F, Rijnierse K (2001) Architectural modeling, introducing the architecture metamodel. Paper presented at the landelijk architectuur congress, Utrecht
31. Kruchten P (1995) Architectural blueprints – the “4+1” view model of software architecture. IEEE Software 12(6):42–50
32. The Institute of Electrical and Electronics Engineers, Inc. (2000) IEEE Std 1471–2000, IEEE recommended practice for architectural description of software-intensive systems. The Institute of Electrical and Electronics Engineers, Inc, New York
33. Jackson M (2010) Engineering and software. In: Nuseibeh B, Zave P (eds) Software requirements and design: the work of Michael Jackson. Good Friends Publishing, Chatham
34. Trew T (2005) Enabling the smooth integration of core assets: defining and packaging architectural rules for a family of embedded products. Paper presented at the software product line conference, Rennes, 26–29 Sept 2005
35. Trew T, Soopenberg G (2006) Identifying technical risks in third-party software for embedded products. Paper presented at the fifth international conference on COTS-based software systems, Orlando, 13–16 Feb 2006
36. Martin RC (2003) Agile software development: principles, patterns and practices. Prentice Hall, Upper Saddle River
37. Wirfs-Brock R, McKean A (2003) Object design: roles, responsibilities, and collaborations. Addison-Wesley, Boston
38. van Ommering R (2003) Horizontal communication: a style to compose control software. Software Practice and Experience 33(12):1117–1150
39. Jackson M (2001) Problem frames: analyzing and structuring software development problems. Addison-Wesley, Harlow
40. Camera and Imaging Products Association (2007) White paper on CIPA DC-001-2003 Rev 2.0: Digital photo solutions for imaging devices. Camera and Imaging Products Association. Tokyo, Japan

Chapter 14

Using Model-Driven Views and Trace Links to Relate Requirements and Architecture: A Case Study

Huy Tran, Ta'id Holmes, Uwe Zdun, and Schahram Dustdar

Abstract Compliance in service-oriented architectures (SOA) means in general complying with laws and regulations applying to a distributed software system. Unfortunately, many laws and regulations are hard to formulate. As a result, several compliance concerns are realized on a per-case basis, leading to ad hoc, hand-crafted solutions for each specific law, regulation, and standard that a system must comply with. This, in turn, leads in the long run to problems regarding complexity, understandability, and maintainability of compliance concerns in a SOA. In this book chapter, we present a case study in the field of compliance to regulatory provisions, in which we applied our view-based, model-driven approach for ensuring the compliance with ICT security issues in business processes of a large European company. The research question of this chapter is to investigate whether our model-driven, view-based approach is appropriate in the context of the case. This question is generally relevant, as the case is applicable to many other problem of requirements that are hard to specify formally (like the compliance requirements) in other business cases. To this end, we will present lessons learned as well as metrics for measuring the achieved degree of separation of concerns and reduced complexity.

14.1 Introduction

As the number of elements involved in an architecture grows, the complexity of design, development, and maintenance activities also extremely increases along with the number of the elements' relationships, interactions, and data exchanges – and becomes hardly manageable. We have studied this problem in the context of process-driven, service-oriented architectures (but observed similar problems in other kinds of architectures as well) [23]. Two important issues are (among other issues) reasons for this problem: First, the process descriptions comprise various tangled concerns, such as the control flow, data dependencies, service invocations, security, compliance, etc. This entanglement seriously reduces many aspects of

software quality such as the *understandability*, *adaptability*, and *Maintainability*. Second, the differences of language syntaxes and semantics, the difference of granularity at different abstraction levels, and the lack of explicit links between process design and implementation languages hinder the *reusability*, *understandability*, and *traceability* of software components or systems being built upon or relying on such languages.

In our previous work we introduced a novel approach for addressing the aforementioned challenges. Our approach exploits a combination of the concept of architectural views [11] – a realization of the *separation of concerns* principle [6] – and the model-driven development paradigm (MDD) [22] – a realization of the *separation of abstraction levels*. This approach has been implemented in the View-based Modeling Framework – an extensible development framework for process-driven, service-oriented architectures (SOAs) [23]. In this chapter, we present a case study in the field of compliance to regulatory provisions in which we applied our approach for complying to ICT security issues in a business process of a large European banking company. In particular, the case study illustrates how our approach helps achieving the following major contributions: *first*, it captures different perspectives of a business process model in separated (semi-)formalized view models in order to adapt to various stakeholders' expertise; *second*, it links to the requirements of the system via a special requirements meta-data view formally modeling the parts of the requirements information needed in the model-driven architecture; *third*, it reduces the complexity of dependency management and enhances traceability in process development via explicit trace links between code, design, and requirements artifacts in the model-driven architecture. We also present lessons learned and preliminary quantitative evaluations on the case study to support the assessment of our approach regarding some aspects of software quality such as the understandability, adaptability, and maintainability.

The rest of the chapter is organized as follows. In Sect. 14.2 we introduce a working application scenario extracted from the business processes of an European banking company. Next, an overview of compliance in service-oriented architectures is provided in Sect. 14.3. Section 14.4 presents a qualitative analysis of our approach applied in the application scenario that illustrates how the aforementioned contributions can be achieved. The lessons learned and quantitative evaluations are provided in Sect. 14.5. We discuss the related work in Sect. 14.6 and conclude.

14.2 Case Study: A Loan Approval Process

Throughout this study, we use a loan approval process of a large European banking company to illustrate the application of our approach in the domain of process-driven SOAs. The banking domain must enforce security and must be in conformity with the regulations in effect. Particular measures like separation of duties, secure logging of events, non-repudiable action, digital signature, etc., need to be

considered and applied to fulfil the mandatory security requirements in order to comply with norms and standards of the banking domain as well as European laws and regulations. In particular, the company emphasizes the necessity of preventing the frauds, preserving the integrity of data, insuring a secure communication between the customers and the process, and protecting customer privacy. Figure 14.1 depicts the core functionality of the loan approval process by using BPMN¹ – a notational language widely used in industry for designing business processes.

At the beginning of the process, a credit broker is assigned to handle a new customer's loan request. He then performs preliminary inspections to ensure that the customer has provided valid credit information (e.g., saving or debit account). Due to the segregation of duties policy of the bank, the inspection carried out by the credit broker is not enough to provide the level of assurance required by the bank. If the loan enquired by the customer is less than one million euros, a post-processing clerk will take over the case. Otherwise, the case is escalated to a supervisor. In this stage, the customer's credit worthiness is estimated through a larger set of data including sums of liabilities, sums of assets, third-party loans, etc. Finally, if no negative reports have been filed, the loan request is handed over to a manager who judges the loan risk and officially signs the loan contract. The customer shall receive either a loan declined notification or a successful loan approval.

14.3 Compliance in Process-Driven SOAs

Services are autonomous, platform-independent entities that can be described, published, discovered, and loosely coupled by using standard protocols [19]. Service-oriented architecture (SOA) is the main architectural style for service-oriented computing. In the scope of this chapter, we exemplify our approach for process-driven SOAs – a particular kind of SOAs utilizing processes to orchestrate services [10] – because enterprises increasingly use process-centric information systems to automate their business processes and services.

Generally speaking, IT compliance means conforming to laws and regulations applying to an IT system such as the Basel II Accord², the Financial Security Law of France³, the Markets in Financial Instruments Directive (MiFID)⁴, and the Sarbanes-Oxley Act (SOX)⁵. These laws and regulations are designed to cover issues such as auditor independence, corporate governance, and enhanced financial

¹<http://www.omg.org/spec/BPMN/1.1>

²<http://www.bis.org/publ/bcbs107.htm>

³<http://www.senat.fr/leg/pjl02-166.html>

⁴<http://www.fsa.gov.uk/pages/About/What/International/mifid>

⁵http://frwebgate.access.gpo.gov/cgi-bin/getdoc.cgi?dbname = 107_cong_bills&docid = fh3763enr.tst.pdf

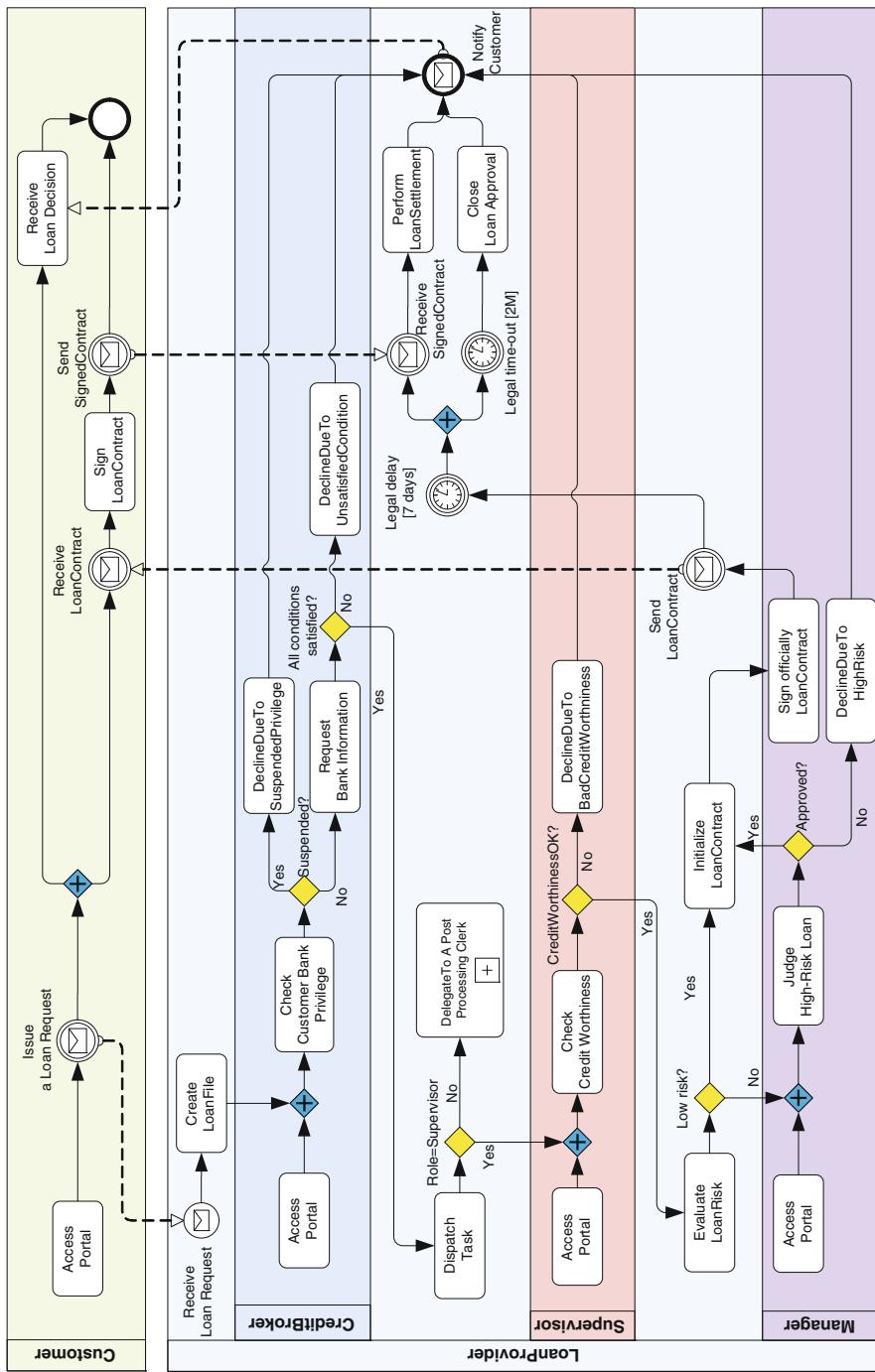


Fig. 14.1 Overview of the loan approval process

disclosure. Nevertheless, laws and regulations are just one example of compliance concerns that might occur in process-driven SOAs. There are many other rules, policies, and constraints in a SOA that have similar characteristics. Some examples are service composition and deployment policies, service execution order constraints, information exchange policies, security policies, quality of service (QoS) constraints, and so on.

Compliance concerns stemming from regulations or other compliance sources can be realized using various *controls*. A control is any measure designed to assure a compliance requirement is met. For instance, an intrusion detection system or a business process implementing separation of duty requirements are all controls for ensuring systems security. As regulations are not very concrete on how to realize the controls, the regulations are usually mapped to established *norms* and *standards* describing more concretely how to realize the controls for a regulation. Controls can be realized in a number of different ways, including manual controls, reports, or automated controls (see Fig. 14.2). Table 14.1 depicts some relevant

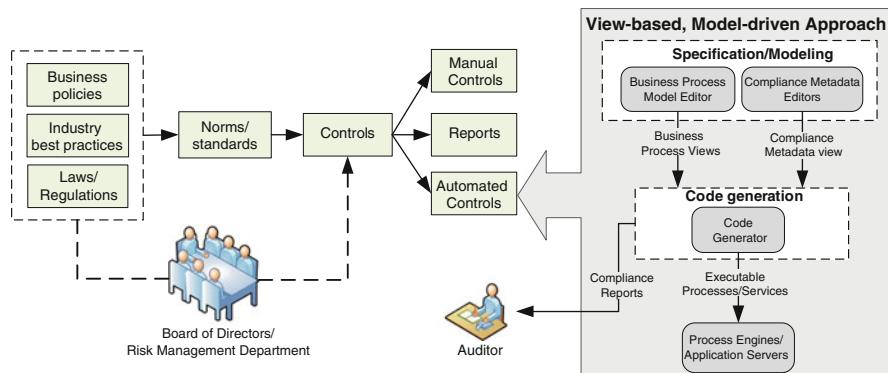


Fig. 14.2 Overview of the view-based, model-driven approach for supporting compliance in SOAs

Table 14.1 Compliance requirements for the loan approval process

Compliance	Risks	Control
Order Approval	R1: Sales to fictitious customers are not prevented and detected	C2: Customer's identifications are verified with respect to identification types and information, customer's shipping and billing addresses are checked against some pre-defined constraints (countries, post code, phone number, etc.).
Segregation of Duties (SoD)	R2: Duties are not properly segregated (SOX 404)	C3: The status of the account verification must be checked by a Financial Department staff. The customer's invoice must be checked and signed by a Sales Department staff.

compliance requirements that the company must implement in the loan approval process in order to comply with the applicable laws and regulations.

14.4 View-Based, Model-Driven Approach: Overview

14.4.1 View-Based Modeling Framework

Our view-based, model-driven approach has been proposed for addressing the complexity and fostering the flexibility, extensibility, adaptability in process-driven SOA modeling development [23]. A typical business process in a SOA comprises various tangled concerns such as the control flow, data processing, service invocations, event handling, human interactions, transactions, to name but a few. The entanglement of those concerns increases the complexity of process-driven SOA development and maintenance as the number of involved services and processes grow. Our approach has exploited the notion of architectural views to describe the various SOA concerns. Each view model is a (semi)-formalized representation of a particular SOA or compliance concern. In other words, the view model specifies entities and their relationships that can appear in the corresponding view.

Figure 14.3 depicts some process-driven SOA concerns formulated using VbMF view models. All VbMF view models are built upon the fundamental concepts of the Core model shown in Fig. 14.4. Using the view extension mechanisms described in [23], the developers can add a new concern by using a *New-Concern-View* model that extends the basic concepts of the Core model (see Fig. 14.4) and defines additional concepts of that concern. The new requirements meta-data view, which is presented in Sect. 14.4.2, is derived using VbMF extension mechanisms for

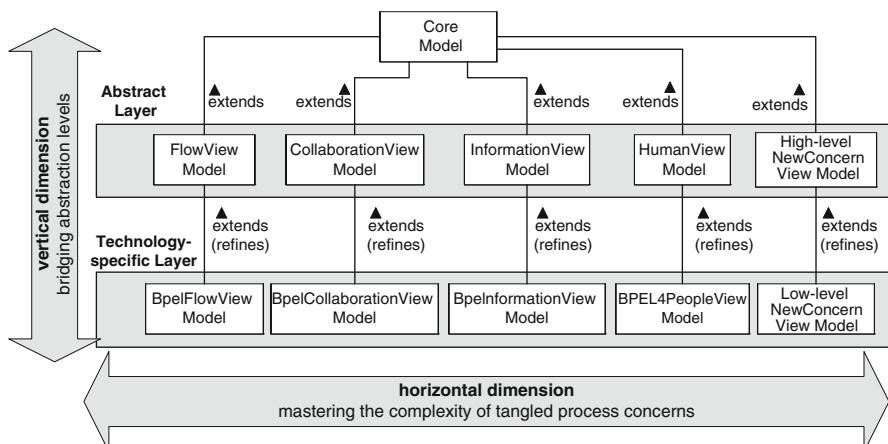


Fig. 14.3 Overview of the view-based modeling framework

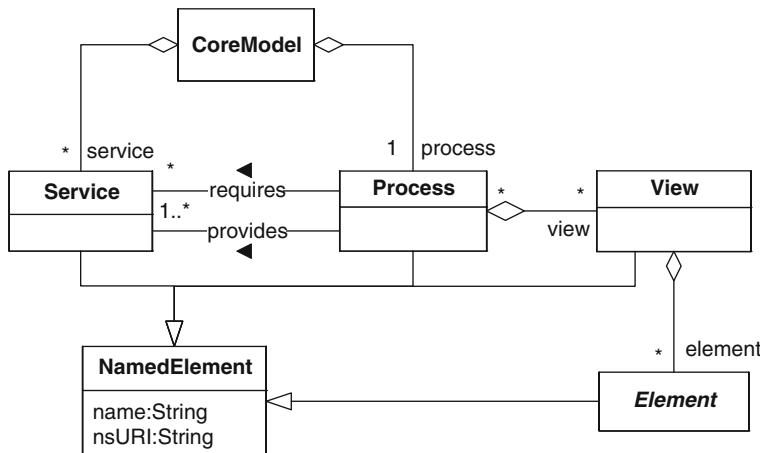


Fig. 14.4 Core model – the foundation for VbMF extension and integration

representing parts of the requirements information needed in process-driven SOAs and links them to the designs described using view models. As a result, the Core model plays an important role in our approach because it provides the basis for extending and integrating view models, and establishing and maintaining the dependencies between view models [23, 27].

There are various stakeholders involved in process development at different levels of abstraction. For instance, business experts require high-level abstractions that offer domain or business concepts concerning their distinct knowledge and expertise while IT experts merely work with low-level, technology-specific descriptions. The MDD paradigm provides a potential solution to this problem by separating the platform-independent and platform-specific models [22].

Leveraging this advantage of the MDD paradigm, VbMF has introduced a model-driven stack that has two basic layers: abstract and technology-specific. The abstract layer includes the views without the technical details such that the business experts can understand and manipulate them. Then, the IT experts can refine or map these abstract concepts into platform-and technology-specific views. For specific technologies, such as BPEL and WSDL, VbMF provides extension view models that enrich the abstract counterparts with the specifics of these technologies [23]. These extension views belong to the technology-specific layer shown in Fig. 14.3.

Some activities during the course of process development may require information of multiple concerns, for instance, communications and collaborations between different experts, generation the whole process implementation, and so on. VbMF offered view integration mechanisms for combining separate views to provide a richer or a more thorough view of a certain process [23]. Finally, VbMF code generation can be used to produce executable process implementation and deployment configurations out of these views.

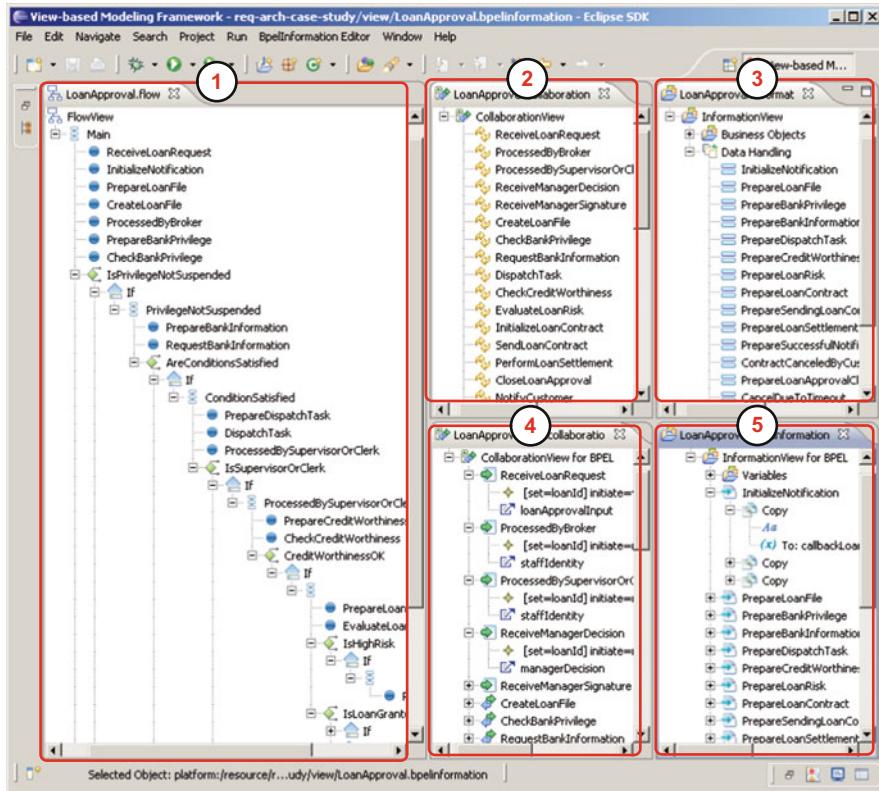


Fig. 14.5 The loan approval process development in VbMF: (1) The FlowView, (2–3) The high-level collaborationView and informationView, and (4–5) The low-level, technology-specific BpelCollaborationView and BpelInformationView

Figure 14.5 shows the loan approval process implemented using VbMF. These views are inter-related implicitly via the integration points from the Core model [23]. The detail of these views as well as their aforementioned relationships shall be clarified in Sect. 14.4.3 on the trace dependencies between VbMF views.

14.4.2 Linking to the Requirements: A Compliance Meta-data View

In this section, we present a Compliance Meta-data view for linking parts of the requirements and the design views of a SOA system. On the one hand, this view enables stakeholders such as business and compliance experts to represent compliance requirements originating from some compliance sources. On the other hand, it allows to annotate process-driven SOA elements described using VbMF (e.g., the

ones shown in Fig. 14.5) with the elicited compliance requirements. That is, we want to implement a compliance *control* for, e.g., a compliance regulation, standard, or norm, using a process or service.

The Compliance Meta-data view provides domain-specific architectural knowledge (AK) for the domain of a process-driven SOA for compliance: It describes which parts of the SOA, i.e., which services and processes, have which roles in the compliance architecture (i.e., *are they compliance controls?*) and to which compliance requirements they are linked. This knowledge describes important architectural decisions, e.g., why certain services and processes are assembled in a certain architectural configuration. In addition, the Compliance Meta-data view offers other useful aspects to the case study project: From it, we can automatically generate compliance documentation for off-line use (i.e., PDF documents) and for online use. Online compliance documentation is, for instance, used in monitoring applications that can explain the architectural configuration and rationale behind it, when a compliance violation occurs, making it easier for the operator to inspect and understand the violation.

A compliance requirement may directly relate to a process, a service, a business concern, or a business entity. Nonetheless compliance requirements not only introduce new but also depict orthogonal concerns to these: although usually related to process-driven SOA elements, they are often pervasive throughout the SOA and express independent concerns. In particular, compliance requirements can be formulated independently until applied to a SOA. As a consequence, compliance requirements can be *reused*, e.g., for different processes or process elements.

Figure 14.6 shows our proposed Compliance Meta-data view model. Annotation of specific SOA elements with compliance meta-data is done using compliance *Controls* that relate to concrete implementations such as a process or service (these are defined in other VbMF views). A *Control* often realizes a number of *ComplianceRequirements* that relate to *ComplianceDocuments* such as a *Regulation*, *Legislation*, or *InternalPolicy*. Such *RegulatoryDocuments* can be mapped to *Standards* that represent another types of *ComplianceDocument*. When a compliance requirement exists, it usually comes with *Risks* that arise from a violation of it. For documentation purposes, i.e., off-line uses, and for the implementation of compliance controls the *ControlStandardAttributes* help to specify general meta-data for compliance controls, e.g., if the control is automated or manual (*isAutomatedManual*). Besides these standard attributes, individual *ControlAttributes* can be defined for a compliance control within a certain *ControlAttributeGroup*.

To provide for extensibility, we have realized a generic modeling solution: a *NamedElement* from the Core model can implement a *Control*. This way not only *Services* and *Processes* can realize a compliance control but as the View-based Modeling Framework is extended also other *NamedElements* can be specified to implement a *Control*. In order to restrict the arbitrary use, an OCL constraint is attached to the *Control* that can be adapted if necessary (i.e., the set of the *getTargetClasses* operation is extended with a new concept that can implement a *Control*).

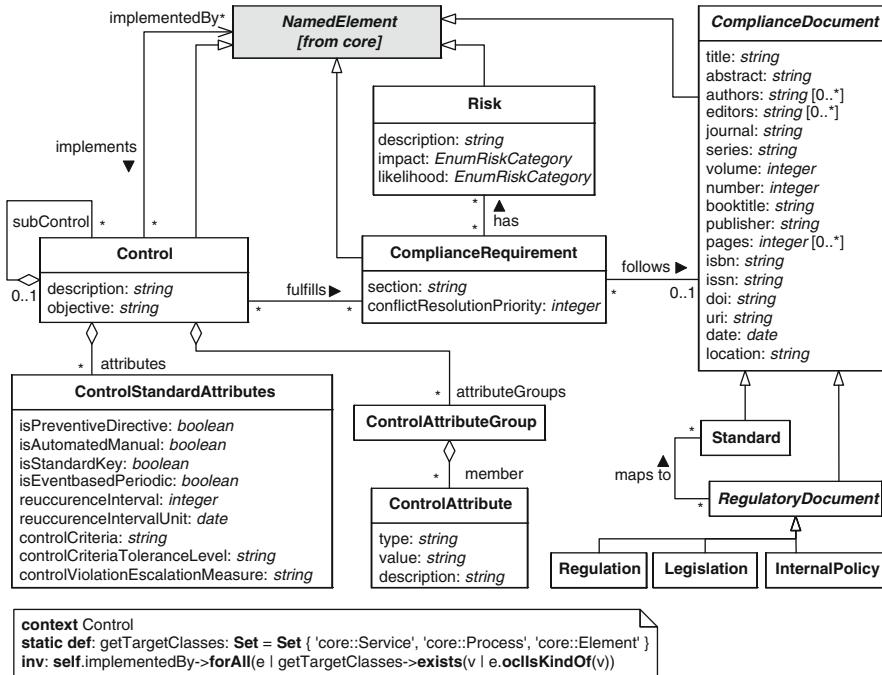


Fig. 14.6 The Compliance meta-data view model

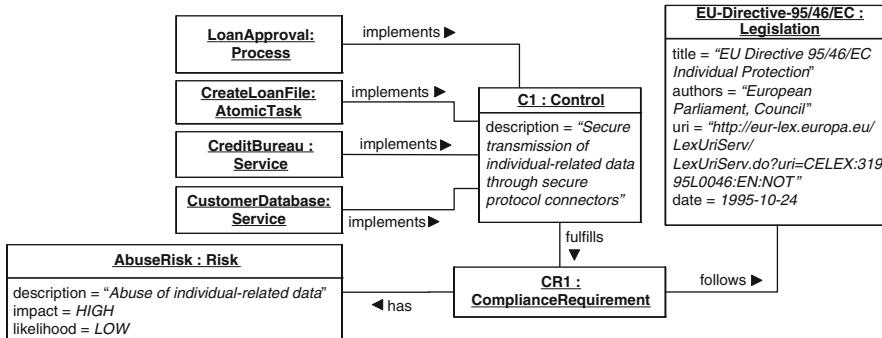


Fig. 14.7 Excerpt of the compliance meta-data view of the loan approval process

Figure 14.7 shows an excerpt of the Compliance Meta-data view of the loan approval process that illustrates a directive from the European Union on the protection of individuals with regard to the processing of personal data. The compliance control *C1*, which fulfills the requirements *CR1*, is implemented by the elements of the loan approval process such as the process named *LoanApproval*, the task named *CreateLoanFile*, and the services named *CreditBureau* and *CustomerDatabase*. Those elements are modeled in VbMF as presented in

Figures 1.5. The compliance requirement *CRI* follows the legislative document and is associated with an *AbuseRisk*.

In this way, the views in Figures 1.5 provide the architectural configuration of the processes and services whilst Fig. 14.7 provides the compliance-related rationale for the design of this configuration. Using the Compliance Meta-data view, it is possible to specify compliance statements such as *CRI is a compliance requirement that follows the EU Directive 95/46/EC on Individual Protection⁶ and is implemented by the loan approval process* within VbMF.

The aforementioned information is useful for the project in terms of compliance documentation, and hence likely to be maintained and kept up-to-date by the developers and users of the system, because it can be used for generating the compliance documentation that is required for auditing purposes. If this documentation is the authoritative source for compliance stakeholders then it is also likely that they have an interest in keeping this information up to date. In doing so they may be further supported with, e.g., imports from other data sources. But in this model also important AK is maintained: In particular the requirements for the process and the services that implement the control are recorded. That is, this information can be used to explain the architectural configuration of the process and the services connected via a secure protocols connector. Hence, in this particular case this documented AK is likely to be kept consistent with implemented system and, at the same time, the rationale of the architectural decision to use secure protocol connectors does not get lost.

14.4.3 *Model-Driven Traceability: Linking Architecture, Code, and Requirements*

In the previous section we introduce the View-based Modeling Framework for modeling and developing processes using various perspectives that can be tailored for particular interests and expertise of the stakeholders at different levels of abstraction. We present in this section our view-based, model-driven traceability approach (VbTrace) realized as an additional dimension to the model-driven stack of VbMF [27]. VbTrace aims at supporting stakeholders in (semi-)automatically establishing and maintaining trace dependencies between the requirements, architecture, and implementations (i.e., process code artifacts) in VbMF [27].

As we mentioned in Sect. 14.4.2, the relationships between the requirements and elements of a process represented in terms of VbMF views have been gradually established during the course of process development and stored in a Compliance Meta-data view. Now we elaborate how our traceability approach helps linking the various process views and code artifacts. The trace links between low-level,

⁶<http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:31995L0046:EN:HTML>

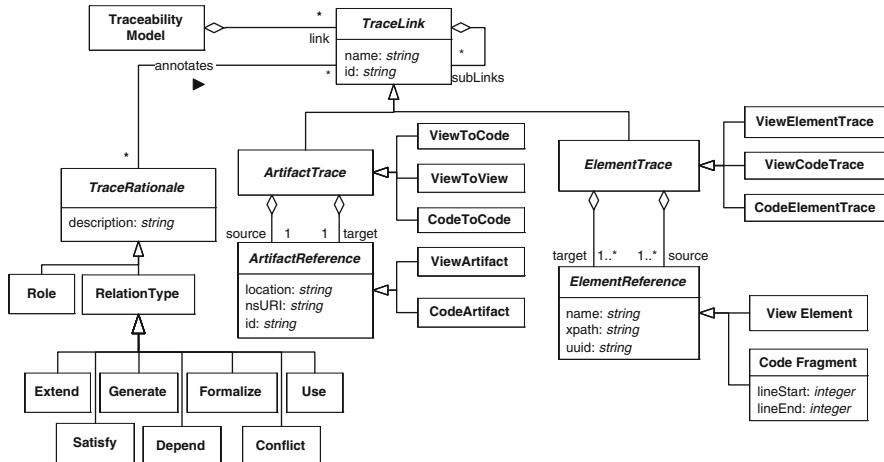


Fig. 14.8 The traceability view model

technology-specific views and code artifacts can be (semi-)automatically derived during the VbMF forward engineering process by using extended code generators or during the VbMF reverse engineering process by using extended view-based interpreters [27]. The relationships between a view and its elements are intrinsic while the relationships between different views are established and maintained according to the name-based matching mechanism for integrating and correlating views (cf. [23] for more details).

Figure 14.8 presents the traceability view model – a (semi-)formalization of trace dependencies between development artifacts. The traceability view model is designed to be rich enough for representing trace relations from process design to implementation and be extensible for further customizations and specializations. There are two kinds of *TraceLinks* representing the dependencies at different levels of granularity: *ArtifactTraces* describing the relationships between artifacts such as view models, BPEL and WSDL files, and so on; *ElementTraces*, describing the relationships between elements of the same or different artifacts such as view elements, BPEL elements, WSDL messages, XML Schema elements, and so forth. The source and target of an *ArtifactTrace* are *ArtifactReferences* that refers to the corresponding artifacts. *ElementTraces*, which are often sub-links of an *ArtifactTrace*, comprises several source and target *ElementReferences* pointing to the actual elements inside those artifacts. Each *TraceLink* might adhere to some *TraceRationales* that comprehend the existence, semantics, causal relations, or additional functionality of the link. The *TraceRationale* is open for extension and must be specialized later depending on specific usage purposes [27].

In order to represent trace dependencies of the various view models at different levels of granularity, VbTrace has introduced three concrete types of *TraceLinks*: *ViewToViews* describe internal relationships of VbMF, i.e., relationships between

view models and view elements, *ViewToCodes* elicit the traceability from VbMF to process implementations, and finally, *CodeToCodes* describe the relationships between the generated schematic code and the associated individual code. Along with these refined trace links between process development artifacts, we also extend the *ElementTrace* concept by fine-grained trace link types between elements such as *ViewElementTrace*, *ViewCodeTrace*, and *CodeElementTrace*. Last but not least, formal constraints in OCL have been defined in order to ensure the integrity and support the verification of the views instantiated from the traceability view model [27]. In the subsequent sections, we present a number of working scenarios to demonstrate how VbTrace can help establishing and maintaining trace dependencies.

14.4.4 Traceability Between VbMF Views

As we mentioned in Sect. 14.4, the stakeholders might either formulate an individual view or communicate and collaborate with each other via combined views that provide richer or more thorough perspectives of processes [23]. For instance, a discussion between a business expert and an IT specialist might require the orchestration of the loan approval process activities along with the interactions between the process and other processes or services. The combination of the FlowView and either the CollaborationView or the BpelCollaborationView based on the name-based matching approach described in [23, 24] can offer such a perspective. Figure 14.9 illustrates the trace relationships of such combinations. The main purpose of view integration is to enhance the flexibility of VbMF in

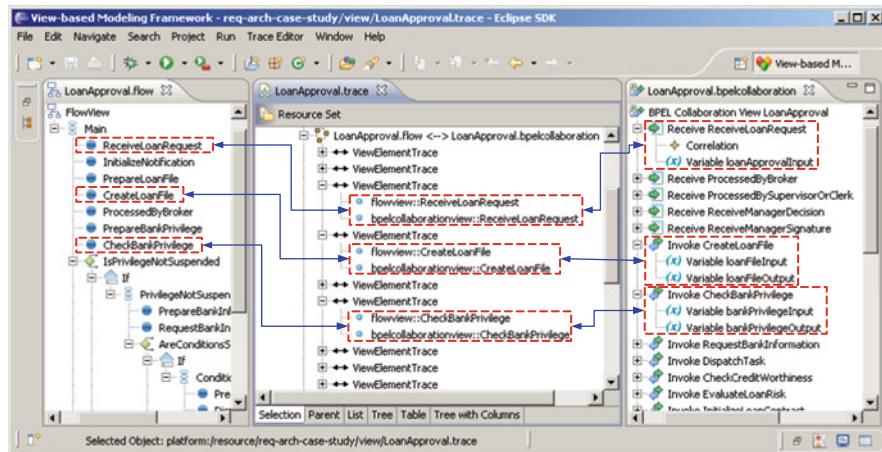


Fig. 14.9 Illustration of trace links between the flowView (left) and BpelCollaborationView (right) of the loan approval process

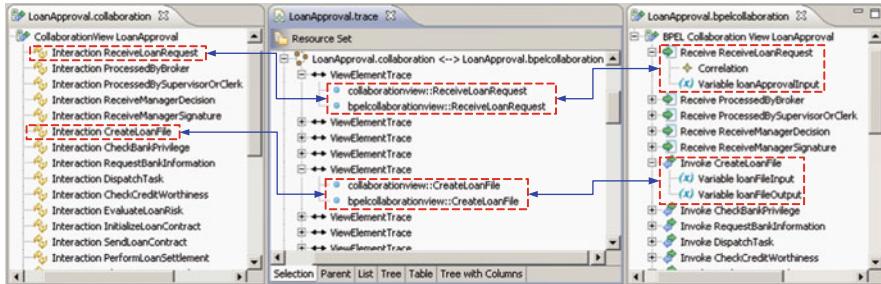


Fig. 14.10 Illustration of trace links between the high- (left) and low-level CollaborationView (right) of the loan approval process

providing various tailored perspectives of the process representation. Because those perspectives might be used by the stakeholders for analyzing and manipulating the process model, we record the relationships raised from the above-mentioned combinations in the traceability according to specific stakeholders' actions and augment them with the *Dependency* type. For the sake of readability, we only present a number of selected trace dependencies and use the arrows to highlight the trace links stored in the traceability view.

The refinements of high-level, abstract views to low-level, technology-specific ones are also recorded by using trace links of the type *ViewToView* to support the traceability between two view models as well as a number of *ViewElementTraces* each of which holds references to the corresponding view elements. Figure 14.10 shows an excerpt of the traceability view that consists of a number of trace links between the CollaborationView and BpelCollaborationView of the loan approval process.

14.4.5 Traceability Between VbMF Views and Process Implementations

The relationships between views and process implementation can be achieved in two different ways. On the one hand, process implementation are generated from the technology-specific views such as the BpelCollaborationView, BpelInformationView, etc., [23]. On the other hand, the view-based reverse engineering approach can also automatically extract process views from existing (legacy) implementations [26]. We recorded the trace links in both circumstances to maintain appropriate relationships between view models and process implementations to fully accomplish the traceability path from process designs to the implementation counterparts (see Fig. 14.11).

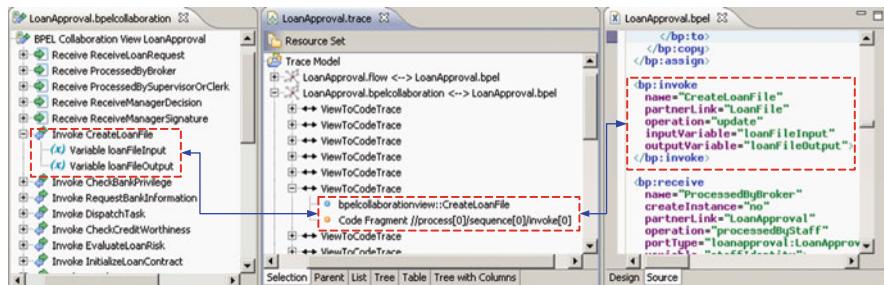


Fig. 14.11 Illustration of trace links between the views (left) and generated BPEL code (right) of the loan approval process

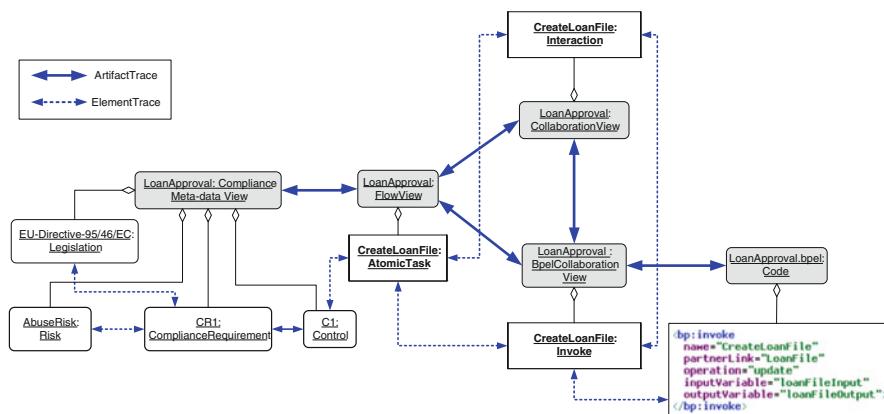


Fig. 14.12 Illustration of a traceability path from requirements through architectural views to code

14.4.6 Example Linking Architectural Views, Code, and Requirements

We present a sample traceability path based on the traceability view established in the previous sections to illustrate how our traceability approach can support linking the requirements, architecture, and code (see Fig. 14.12). The traceability path implies the trace links between the requirements and process elements – derived from the Compliance Meta-data view – followed by the relationships among VbMF views. The process implementation is explored at the end of the traceability path by using the trace dependencies between VbMF technology-specific views and process code.

Let us assume that there is a compliance requirement changed according to new regulations. Without our traceability approach, the stakeholders, such as business, domain, and IT experts, have to dig into the BPEL and WSDL code, identify the

elements to be changed and manipulate them. This is time consuming and error-prone because there is no explicit links between the requirements to and process implementations. Moreover, the stakeholders have to go across numerous dependencies between various tangled concerns, some of which might be not relevant to the stakeholders expertise. Using our approach, the business and domain experts can better analyze and manipulate business processes by using the VbMF abstract views, such as the FlowView, CollaborationView, InformationView, Compliance Meta-data View, etc. The IT experts, who mostly work on either technology-specific views or process code, can better analyze and assess coarse-grained or fine-grained effects of these changes based on the traceability path.

14.5 Evaluation and Lessons Learned

So far we have presented a case study based on the development life cycle of an industrial business process that qualitatively illustrates the major contributions achieved by using our approach. To summarize, these are in particular: First, a business process model is (semi-)formally described from different perspectives that can be tailored and adapted to particular expertise and interests of the involving stakeholders. Second, parts of the requirements are explicitly linked to the system architecture and code by a special (semi-)formalized meta-data view. Third, our view-based traceability approach can help reducing the complexity of dependency management and improving traceability in process development. In addition, we also conducted a quantitative evaluation to support the assessment of our approach. The degree of separation of concerns and the complexity of business process models are measured because they are considered as the predictors of many important software quality attributes such as the understandability, adaptability, maintainability, and reusability [5]. This evaluation focuses on the view-based approach as the foundation of our approach and provides evidence supporting our claims regarding the above-mentioned software quality attributes.

14.5.1 Evaluation

14.5.1.1 Complexity

In practice, there are several efforts aiming at quantifying the complexity of software such as Line of Code [5], McCabe complexity metrics [18], Chidamber-Kemerer metrics [3], etc. However, [16] suggested that these metrics are not suitable for measuring the complexity of MDD artifacts. Lange [16] proposed an approach for measuring model size based on the four dimensions of [5]. Lange's metric is of cognitive complexity that rather reflects the perception and understanding of a certain model from a modeler's point of view [5]. That is, the higher the size

complexity, the harder it is to analyze and understand the system [5]. The complexity used in our study is a variant of Lange's model size metrics [16], which is extended to support specific concepts of process-driven SOAs and the MDD paradigm. It measures the complexity based on *the number of the model's elements and the relationships between them*.

In addition to the loan approval process (LAP) presented in Sect. 14.2, we perform the evaluation of complexity on four other use cases extracted from industrial process including a travel agency process (TAP) from the domain of tourism, an order handling process (OHP) from the domain of online retailing, a billing renewal process (BRP) and a CRM fulfillment process (CFP) from the domain of Internet service provisioning. We apply the above-mentioned model-based size metric for each main VbMF view such as the FlowView (FV), high-level and low-level CollaborationViews (CV/BCV), and high-level and low-level InformationViews (IV/BIV). Even though the correlation of views are implicit performed via the name-based matching mechanism [23], the name-based integration points between high-level views (IP_{high}) and low-level views (IP_{low}) are calculated because these indicates the cost of separation of concerns principle realized in VbMF. Table 14.2 shows the comparison of these metrics of VbMF views to those of process implementation in BPEL technology, which is widely used in practice for describing business processes. Note that the concerns of process implementation are not naturally separated but rather intrinsically scattered and tangled. We apply the same method to calculate the size metric of the process implementation based on its elements and relationships with respect to the corresponding concepts of VbMF views.

The results show that the complexity of each of VbMF views is lower than that of the process implementation. Those results prove that our approach has reduced the complexity of business process model by the notion of (semi-)formalized views. We also measure a high-level representation of process by using an integration of VbMF abstract views and a low-level representation of process by using an integration of VbMF technology-specific views. The numbers say that the complexity of the high-level (low-level) representation is much less than (comparable to) that of the process implementation. The overhead of integration points occurs in both aforementioned integrated representations.

Table 14.2 The complexity of process descriptions and VbMF views

Process	VbMF(Hi)			VbMF(Lo)		IntegrationPoint		Process impl. BPEL/WSDL
	FV	CV	IV	BCV	BIV	IP_{high}	IP_{low}	
Travel agency (TAP)	33	33	43	56	261	17	40	355
Order handling (OHP)	29	36	44	65	285	17	46	383
Billing renewal (BRP)	81	63	85	132	492	48	177	700
CRM fulfilment (CFP)	49	74	78	131	535	31	88	730
Loan approval (LAP)	68	44	48	104	651	34	85	871

Table 14.3 Measures of process-driven concern diffusion

Process (%)	Flow			Collaboration			Information flow		
	Without VbMF	With VbMF	Reduced (%)	Without VbMF	With VbMF	Reduced (%)	Without VbMF	With VbMF	Reduced (%)
TAP	175	17	90.3	186	40	78.5	85	23	72.9
OHP	212	17	92.0	221	46	79.2	93	29	68.8
BRP	411	48	88.3	409	117	71.4	195	69	64.6
CFP	398	31	92.2	407	88	78.4	176	57	67.6
LAP	425	34	92.0	431	68	84.2	188	69	63.3

14.5.1.2 Separation of Concerns

To assess the separation of concerns, we use the Process-driven Concern Diffusion metric (PCD), which is derived from the metrics for assessing the separation of concerns in aspect-oriented software development proposed in [21]. The PCD of a process concern is a metric that counts the number of elements of other concerns which are either tangled in that concern or directly referenced by elements of that concern. The higher the PCD metric of a concern, the more difficult it is for the stakeholders to understand and manipulate the concern. The measurement of PCD metric in all processes mentioned in Sect. 14.5.1.1 are presented in Table 14.3.

A process description specified using BPEL technology often embodies several tangled process concerns. VbMF, by contrast, enables the stakeholders to formulate the process through separate view models. For instance, a process control-flow is described by a BPEL description that often includes many other concerns such as service interactions, data processing, transactions, and so on. As a result, the diffusion of the control-flow concern of the process description is higher than that of the VbMF FlowView. The results show that the separation of concerns principle exploited in our view-based, model-driven approach has significantly reduced the scatter and tanglement of process concerns. We have achieved a significant decrement of the diffusion of the control-flow approximately of 90%, which denotes a better understandability and maintainability of the core functionality of processes. For other concerns, our approach is also shown to notably reduce concern diffusions by roughly 80% for the collaboration concern and about 60% for the information concern, and therefore, improve the understandability, reusability, and maintainability of business process models.

14.5.2 Lessons Learned

The quickly increasing of the complexity of design, development, and maintenance activities and maintenance due to the thriving of the number of elements involved in an architecture is very challenging in the context of process-driven, service-oriented architectures. We also observed similar problems in other kinds of architectures that expose the following common issues. First, a system description,

e.g., an architectural specification, a model, etc., embodies various tangled concerns. As a consequence, the entanglement seriously reduces many aspects of software quality such as the *understandability*, *adaptability*, and *maintainability*. Second, the differences of language syntaxes and semantics, the difference of granularity at different abstraction levels, and the lack of explicit links between these languages hinder the *understandability* and *traceability* of software components or systems being built upon or relying on such languages. Last but not least, parts of the system's requirements are hard to specify formally, for instance, the compliance requirements. These intrinsic issues (among other issues) are ones of reasons which impede the correlating of requirements and the underlying architectures.

Our study showed that it is feasible to facilitate a view-based, model-driven approach to overcome the aforementioned challenges. Our approach enables flexible, extensible (semi-)formalized methods to represent the software system using separate architectural views. The flexibility and extensibility of our approach have been confirmed including the devising and using an additional model-driven requirement view for adding AK meta-data with reasonable effort and a traceability view for supporting establishing and maintaining dependency relationships between the architecture and the corresponding implementations. In particular, this study also provided evidences to confirm that it is possible in the context of a project to record specific AK that is domain-specifically relevant for a project using such a view.

Moreover, the model-driven approach complemented by the traceability view model can help to keep the data in the AK view up-to-date and *consistent* with the project. As a result, the integrity and consistency of the links from requirements to architecture and code can be maintained. To this end, it is reasonable to connect the data recorded in the AK view with other meta-data that needs to be recorded in the project anyway. This would be an additional incentive for developers to document the AK. In our study, compliance in service-oriented systems is illustrated as an area where this is feasible because a lacking or missing compliance documentation can lead to severe legal consequences. Nonetheless, our general approach can also be applied for custom AK without such additional incentives.

There is a limitation in our approach that only specific AK – linked to a domain specific area like compliance – is recorded and other AK might get lost. It is the responsibility of a project to make sure that all relevant AK for understanding an architecture gets recorded. In addition, our view-based, model-driven exploits the notion of architectural views – a realization of the separation of concern principle – and the MDD paradigm – a realization of the separation of levels of abstraction. In particular, the system description is managed and formulated through separate view models that are integrated and correlated via the name-based matching mechanism [23].

On the one hand, this supposes additional investments and training are required at the beginning for instantiating view models, traceability models, and model transformation rules. In case of legacy process-driven SOAs, the view-based reverse engineering might partially help stakeholders quickly coming up with

view models extracted from the existing business process descriptions. However, manual interventions of stakeholders are still needed to analyze and improved the extracted views, and sometimes, the corresponding the traceability models. On the other hand, this also implies that, comparing to a non-view-based or non-model-driven approach, additional efforts and tool supports are necessitated for managing the consistency of views and traceability models as those can be manipulated by different stakeholders as well as enabling change propagation among them. Nonetheless, the maintenance of trace dependencies between views can be enhanced by using hierarchical or ontology-based matching and advanced trace link recovery techniques [1].

However, the project can benefit in the long term regarding the reducing maintenance cost due to the enhancement of understandability, adaptability, and traceability as well as the preserved consistent AK. Nonetheless, it is possible to introduce our approach into a non-model-driven project (e.g., as a first step into model-driven development). For doing this, at least a way to identify the existing architectural elements, such as components and connectors, must be found. But this would be considerably more work than adding the view to an existing model-driven project.

14.6 Related Work

In our study, we applied our prior works that is the view-based, model-driven approach for process-driven SOAs [23–27] in the field of compliance to regulatory provisions. Therefore, more in-depth comparisons and discussions on the related work of the view-based, model-driven approach can be found in [23–26] those of the name-based view integration mechanism can be found in [28], and those of the view-based traceability approach can be found in [27]. To this end, we merely discuss the major related works in the area of bridging requirements, architecture, and code.

A number of efforts provide modeling-level viewpoint models for software architecture [20], 4+1 view model by [14] and the IEEE 1471 standard [11] concentrating on various kinds of viewpoints. While some viewpoints in these works and VbMF overlap, a general difference is, that VbMF operates at a more detailed abstraction level – from which source code can be generated via MDD. Previous works on better support for codifying the AK have been done in the area of architectural decision modeling. Jansen et al. [12] see software architecture as being composed of a set of design decisions. They introduce a generic meta-model to capture decisions, including elements such as problems, solutions, and attributes of the AK. Another generic meta-model that is more detailed has been proposed by [31]. Tyree and Ackerman [29] proposed a highly detailed, generic template for architectural decision capturing.

De Boer et al. [2] propose a core model for software architectural knowledge (AK). This core model is a high-level model of the elements and actions of AK and

their relationships. In contrast to that, our models operate at a lower-level – from which code can be generated (the core model in VbMF is mainly defining integration points for MDD). Of course, the core model by de Boer et al.\ and VbMF could be integrated by providing the model by de Boer et al.\ as a special AK view in VbMF that is linked to the lower-level VbMF models via the matching mechanisms and trace links discussed in this paper.

Question, Options, and Criteria diagrams raise a design question, which points to the available solution options, and decision criteria are associated with the options [17]. This way decisions can be modeled as such. Kruchten et al. [13] extend this research by defining an ontology that describes the information needed for a decision, the types of decisions to be made, how decisions are being made, and their dependencies. Falessi et al. [4] present the Decision, Goal, and Alternatives framework to capture design decisions. Recently, Kruchten et al. [15] extended these ideas with the notion of an explicit decision view – akin to the basic view-based concepts in our approach.

Wile [30] introduced a runtime approach that focuses on monitoring running systems and validating their compliance with the requirements. Grünbacher et al. [7] proposed an approach that facilitates a set of architectural concepts to reconcile the mismatches between the concepts of requirements and those of the corresponding architectures. Hall et al. [8] proposed an extension to the problem-frames approach to support the iteration between problem and solution structures in which architectural concepts can be considered as parts of the problem domain rather than the solution domain. Heckel and Engels [9] proposed an approach to relate functional requirements and software architecture in order to arrive at a consistent overall model in which a meta model is facilitated to provide separate packages for the functional requirements and the architectural view and a third package representing the relation between these views.

In contrast to our work, most of the related work on architectural decision modeling focus on *generic* knowledge capturing. Our approach proposes to capture AK in a *domain-specific* fashion as needed by a project. Hence in our work some AK is not as explicit as in the other approaches. For example, the collaborations of components are shown in the CollaborationView whereas the other approaches rather use a typical component and connector view. The decision drivers and consequences of the decisions are reported in the compliance sources and as risks. That means, our domain-specific AK view adopts the terminology from the compliance field, and it must be mapped to the AK terminology in order to understand the overlaps. None of the related works provide detailed guidelines how to support the AK models or views through MDD. On the contrary, this is a focus of our work. Additionally, using the model-driven development paradigm in our approach gains a twofold advantage. On the one hand, stakeholders working at different abstraction levels are offered tailored perspectives according to their expertise and interests. On the other hand, data in the AK view are preserved and keeping up-to-date and *consistent* with other parts of the project.

14.7 Conclusion

In this book chapter we presented an approach for relating requirements and architecture using model-driven views and automatically generated trace links. We demonstrated the applicability of this approach in the context of a case study in the field of ICT security compliance. The results suggest that using our approach it is possible to describe a business process in (semi-)formal way from different perspectives that can be tailored and adapted to particular expertise and interests of the involved stakeholders. Our quantitative evaluation gives evidence that this approach also has benefits in terms of reduced complexity and concern diffusion. Using a special (semi-)formalized meta-data view, we were able to link parts of the requirements to the system architecture described by these views and the code generated from them. In this context, our view-based traceability approach supports the automated dependency management and hence improves the traceability in process development. Our ongoing work is to complement this framework with an integrated development environment that facilitates collaborative model-driven design with different stakeholders as well as a runtime governance infrastructure that enacts the detection of compliance violations and compliance enforcement according to the monitoring directives generated from compliance DSLs and the Compliance Meta-data view model.

Acknowledgments The authors would like to thank the anonymous reviewers for providing constructive, insightful comments that greatly help to improve this chapter. This work was supported by the European Union FP7 project COMPAS, grant no. 215175 and the European Union FP7 project INDENICA grant no. 257483.

References

1. Antoniol G, Canfora G, Casazza G, De Lucia A, Merlo E (2002) Recovering traceability links between code and documentation. *IEEE Trans Softw Eng* 28(10):970–983
2. de Boer RC, Farenhorst R, Lago P, van Vliet H, Clerc V, Jansen, A (2007) Architectural knowledge: getting to the core. In: *Quality of software architectures (QoSA)*, Boston, pp 197–214
3. Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. *IEEE Trans Softw Eng* 20(6):476–493
4. Falessi D, Becker M, Cantone G (2006) Design decision rationale: experiences and steps towards a more systematic approach. *SIG-SOFT software engineering notes* 31 – workshop on sharing and reusing architectural knowledge 31(5)
5. Fenton N, Pfleeger SL (1997) Software metrics, 2nd edn, a rigorous and practical approach. PWS Publishing Co, Boston
6. Ghezzi C, Jazayeri M, Mandrioli D (2002) Fundamentals of software engineering, 2nd edn. Prentice Hall, Upper Saddle River
7. Grünbacher P, Egyed A, Medvidovic N (2003) Reconciling software requirements and architectures with intermediate models. *J Softw Syst Model* 3(3):235–253
8. Hall JG, Jackson M, Laney RC, Nuseibeh B, Rapanotti L (2002) Relating software requirements and architectures using problem frames. In: *IEEE international conference requirements engineering*. IEEE Computer Society, Essen, pp 137–144

9. Heckel R, Engels G (2002) Relating functional requirements and soft-ware architecture: separation and consistency of concerns. *J Softw Maint Evol Res Pract* 14(5):371–388
10. Henrich C, Zdun U (2006) Patterns for process-oriented integration in service-oriented architectures. In: Proceedings of 11th European conference pattern languages of programs (EuroPLoP 2006), Irsee, pp 1–45
11. IEEE (2000) Recommended practice for architectural description of software intensive systems. Technical report IEEE-Std-1471-2000
12. Jansen AGJ, van der Ven J, Avgeriou P, Hammer DK (2007) Tool support for architectural decisions. In: Sixth IEEE/IFIP working conference software architecture (WICSA), Mumbai
13. Kruchten P, Lago P, van Vliet H (2006) Building up and reasoning about architectural knowledge. In: QoSAs 2006. LNCS, Vol 4214, Springer, Heidelberg, pp 43–58
14. Kruchten P (1995) The 4 + 1 view model of architecture. *IEEE Softw* 12(6):42–50
15. Kruchten P, Capilla R, Duenas JC (2009) The decision view's role in software architecture practice. *IEEE Softw* 26:36–42
16. Lange CFJ (2006) Model size matters. In: Models in software engineering, workshops and symposia at MoDELS 2006. LNCS. Springer, Berlin, pp 211–216
17. MacLean A, Young RM, Bellotti V, Moran T (1991) Questions, options, and criteria: elements of design space analysis. *HumanComput Interact* 6(3–4):201–250
18. McCabe TJ (1976) A complexity measure. *IEEE Trans Softw Eng* 2(4):308–320
19. Papazoglou MP, Traverso P, Dustdar S, Leymann F (2008) Service-oriented computing: a research roadmap. *Int J Cooperative Inf Syst* 17(2):223–255
20. Rozanski N, Woods E (2005) Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley, Boston
21. Sant'Anna C, Garcia A, Chavez C, Lucena C, and v. von Staa A (2003) On the reuse and maintenance of aspect-oriented software: an assessment framework. In XVII Brazilian symposium on software Engineering, Manaus
22. Stahl T, Völter M (2006) Model-driven software development. John Wiley & Sons, Chichester
23. Tran H, Holmes T, Zdun U, Dustdar S (2009) Modeling process-driven SOAs – a view-based approach, IGI global. In: Handbook of research on business process modeling (Chap 2)
24. Tran H, Zdun U, Dustdar S (2007) View-based and model-driven approach for reducing the development complexity in process-driven SOA. In: International conference business process and services computing (BPSC), GI, LNI, vol 116, pp 105–124
25. Tran H, Zdun U, Dustdar S (2008) View-based integra-tion of process-driven SOA models at various abstraction levels. In: First international workshop on model-based software and data integration MBSDI 2008. Springer, Heidelberg, pp 55–66
26. Tran H, Zdun U, Dustdar S (2008b) View-based reverse engineering approach for enhancing model interoperability and reusability in process-driven SOAs. In: Tenth international conference software reuse (ICSR), Springer, Beijing, pp 233–244
27. Tran H, Zdun U, Dustdar S (2009) VbTrace: using view-based and model-driven development to support traceability in process-driven SOAs. *J Softw Syst Model*. doi:10.1007/s10270-009-0137-0
28. Tran H, Zdun U, Dustdar S (2010) Name-based view integration for enhancing the reusability in process-driven SOAs. In: First international workshop on reuse in business process management (rBPM) at BPM 2010. Springer, Heidelberg, pp 1–12
29. Tyree J, Ackerman A (2005) Architecture decisions: demystifying ar-chitecture. *IEEE Softw* 22:19–27
30. Wile DS (2001) Residual requirements and architectural residues. In: Fifth IEEE International symposium on requirements engineering IEEE Computer Society, Toronto, pp 194–201
31. Zimmermann O, Gschwind T, Kuester J, Leymann F, Schuster N (2007) Reusable architectural decision models for enterprise application development. In: Quality of software architecture (QoSAs) 2007. Lecture notes in computer science, Boston

Chapter 15

Managing Artifacts with a Viewpoint-Realization Level Matrix

Jochen M. Küster, Hagen Völzer, and Olaf Zimmermann

Abstract We propose an approach to artifact management in software engineering that uses an *artifact matrix* to structure the artifact space of a project along stakeholder viewpoints and realization levels. This matrix structure provides a basis on top of which relationships between artifacts can be defined, such as consistency constraints, traceability links and model transformations. The management of all project artifacts and their relationships supports collaboration across different roles in the development process as well as change management and agile practices. Our approach is highly configurable to facilitate adaptation to different development methods and processes. It provides a basis to develop and/or to integrate generic tools that can flexibly support such different methods. In particular, it can be leveraged to improve the transition from requirements analysis to architecture design.

15.1 Introduction

The state of the art in requirements engineering and software architecture has advanced significantly in recent years. Mature requirements and software engineering methods such as the Unified Process (UP) [18] specify processes to be followed and artifacts to be created in application development and integration projects. In requirements engineering, informal and formal notations as well as modeling techniques are available to the domain analyst, for example vision statements, strategy maps, business process models, user stories, and use cases [30]. In software architecture, both lightweight and full-fledged architecture design processes exist. Architectural tactics, patterns, and styles help architects to create designs from previously gained experience that is captured in reusable assets [2, 6]. Techniques and tools for architectural decision capturing and sharing have become available [19, 32].

When applying these or other state-of-the-art methods, techniques and tools on large projects, requirements engineers and architects often produce a large amount of rather diverse artifacts. If this is the case, it is challenging to maintain the consistency of these artifacts and to promote their reuse. Likewise, the large body

of existing work makes it difficult for creators of requirements engineering and architecture design methods to assemble domain-specific methods that are both comprehensive and consumable; Service-Oriented Architecture (SOA) design exemplifies this problem [32]. The same issue makes it hard to build tools that flexibly support such methods for different domains.

Relationships between artifacts do not only have to be understood both *within* requirements analysis and *within* architecture design; as motivated in previous chapters of this book, a seamless transition *between* these analysis and design activities is particularly important. For instance, the dependency between architecturally significant requirements and the architectural decisions that are required to satisfy these requirements often remains undocumented. This is unfortunate because the requirements engineer, who is familiar with the application domain, can advise the architect on domain-specific design issues that arise from common requirements in the domain. Examples of such requirements are industry-specific process models and regulatory compliance rules. The architect, on the other hand, has tacit knowledge that makes him/her pre-select certain patterns, technology standards and implementation assets. The traceability links to the requirements that are satisfied by these assets are often not made explicit; therefore it is difficult to evaluate whether a given design is an adequate solution for a particular business domain and problem. According to our industry project experience, this gap between requirements engineering and architecture design delays projects or leads to project failure; application maintenance may also suffer.

Similar problems can be observed between other roles and viewpoints. For instance, test cases should be derived from functional and non-functional requirements; they should also examine design hot spots such as single points of failure (if high availability is a desired quality attribute) and scalability bottlenecks (if many concurrent users are likely to perform complex operations concurrently). Hence, the artifacts created by testers should be aligned with those used by requirements engineers, architects, and developers. At a minimum, terminologies should be aligned, traceability links be defined, and continuous refinement and change of artifacts be supported across these four roles.

The outlined problems in managing requirements engineering, architecture design, and other software engineering artifacts can be abstracted and generalized:

1. *Method definition:* When a software engineering method is defined, either by creating a new method from scratch for a new domain or by instantiating, tailoring, and combining existing methods, one has to define which artifacts of which type to include and which of their dependencies to trace. Specifically for requirements engineering and architecture design, one has to determine what the deliverables of specific analysis or design activities are and how they relate to each other (e.g., use cases and class diagrams defined in the Unified Modeling Language (UML) [28]).
2. *Tool design:* It should be defined which tools, if any, are used to create and manage the artifacts defined in a method. These tools should support the flexible application of a method beyond the provision of simple editors for artifact notations such as UML. Specifically for requirements engineering and architecture design, it should be defined how the requirements engineering and the architecture

design tools are organized and how they interface with each other. E.g., are the same tools used by both roles? If so, are different UML profiles used? How can the required architectural decisions be identified in requirements artifacts?

To address these two general problems, we propose an integrated, model-driven artifact management approach. This general approach can be leveraged to specifically improve the transition from requirements engineering to architecture design. In the center of our approach is the *Artifact and Model Transformation (AMT) Matrix*, which provides a structure for organizing and maintaining artifacts and their dependencies. The default incarnation of our AMT Matrix is based on stakeholder *viewpoints* and analysis/design refinement levels, which we call *realization levels*, two common denominators of contemporary analysis and design methods. Relative to the two general artifact management problems, our AMT approach contributes the following:

1. *Method definition*: With the AMT matrix, our approach provides a generic structure to support the definition of methods. It contributes a metamodel that formalizes the AMT Matrix and its relationship to software engineering artifacts.
2. *Tool design*: With the AMT matrix, our approach provides a foundational structure for designing and integrating tools that provide transformations between different artifacts, create traceability links automatically, validate consistency, and support cross-role and cross-viewpoint collaboration.

Via instantiation and specialization, these two general contributions can be leveraged to solve our original concrete problem of better aligning and linking requirements engineering and architecture design artifacts (e.g., user stories, use cases, logical components, and architectural decisions).

The remainder of the chapter is structured in the following way. In the next section, we clarify fundamental literature terms such as viewpoint and realization level and introduce our general concepts on an informal level. After that, we formalize these concepts in a metamodel. The concepts and their formalization allow us to define cross-viewpoint transformations and traceability links between requirements engineering and architecture design artifacts. These solution building blocks form the core contribution of this chapter. In the remaining sections of the chapter, we provide an example how our concepts can be applied in practice, outline the implementation of a tool prototype and discuss related work. We conclude with summary and a discussion of open issues.

15.2 The Artifact and Model Transformation (AMT) Matrix

Both in requirements engineering and in software architecture design, model-driven software development is applied in various forms. Maturity levels and practitioner adoption vary by domain. For instance, software engineering processes such as Object-Oriented Analysis and Design (OOAD) [4] and modeling languages such as the Unified Modeling Language (UML) [28] are successfully adopted in embedded systems engineering and enterprise application development today. Well-crafted

models facilitate communication between stakeholders; formal models can be processed by tools in support of automation. A key concept of model-driven software development is to construct models that describe the system under construction from different viewpoints and at different levels of abstraction; these models then become the artifacts to be created when following a particular method. The information found in already existing models serves as input to create model artifacts. For example, in OOAD and Component-Based Development (CBD), requirements analysis artifacts such as UML use cases may serve as input to the construction of architecture design artifacts such as functional component models expressed as UML class diagrams [8].

To overcome the artifact management problems identified in the previous section, we structure the model space of a project as an Artifact and Model Transformation (AMT) matrix. The goal of the AMT matrix is to organize the model space according to the concepts in the chosen software engineering method. The default incarnation of our AMT matrix has two dimensions: The horizontal dimension represents disjoint/discrete stakeholder *viewpoints* as defined in the IEEE 42010 specification for architecture descriptions [24]; the vertical dimension of the matrix represents *realization levels* as defined in methods promoting an incremental and iterative refinement of artifacts.

For instance, the architecting process defined by Eeles and Cripps [8] distinguishes stakeholder-specific, role-based viewpoints such as ‘requirements’, ‘functional’, and ‘deployment’; their two realization levels are the platform-independent ‘logical level’ and the platform-specific ‘physical level’.¹ The AMT matrix entry for functional design on the logical refinement level may then list, for example, UML class diagrams and sequence diagrams as the artifact types that populate this matrix entry.

Both AMT matrix dimensions can be configured for a particular software engineering method via the viewpoints and realization levels defined in the method. An AMT matrix for UP, for instance, differs from an AMT matrix for an agile process in terms of number, names, and semantics of viewpoints and realization levels. UP leverages Kruchten’s original 4 + 1 viewpoint model; these viewpoints become the columns of the matrix. Elaborating a design via multiple iterations requires touching already existing artifacts multiple times; by defining one realization level row for each iteration, these different stages of the artifact evolution can be distinguished from each other.

AMT matrix entries can be connected by *traceability links* and *transformations* between artifacts and individual artifact elements (e.g., between steps in a use case model and operations/methods in a UML class diagram). To enforce its design practices, the software engineering method determines which links and transformations are valid. For example, it might not permit to bypass a realization level or to increase the realization level and switch the viewpoint in a single atomic transformation.

¹Note that Eeles and Cripps [8] use the terms ‘logical’ and ‘physical’ for realization levels whereas the 4 + 1 viewpoint model in UP uses them for particular viewpoints.

In the following, we introduce our AMT matrix management approach in three steps:

1. Specify AMT matrix dimensions (i.e., viewpoints and realization levels by default).
2. Position method-specific artifact types in AMT matrix entries (according to their purpose).
3. Populate a project-specific AMT matrix instance with artifacts (according to their type).

Step 1: Specify AMT matrix dimensions. Our first step is preparatory. As outlined above, we propose to organize all types of models and other artifacts defined in a method (and, later on, models and other artifacts created on projects) in a multi-dimensional matrix structure. In this preparatory step, we define how many dimensions are used, what the semantics of these dimensions are, and how these dimensions are structured and sourced.

The default incarnation of an AMT matrix has two dimensions. We decided to combine two problem solving strategies that are promoted by many contemporary software engineering methods and commonly used in many other engineering disciplines:

- Partitioning by stakeholder-specific *viewpoints* [18, 24].
- Incremental refinement via *realization levels* [8, 15].

The two-dimensional default AMT matrix structure resulting from these considerations is shown in Fig. 15.1. In the remainder of this chapter, we work with this default structure; adding dimensions is subject to future research.

Viewpoint Realization Level	Viewpoint A	Viewpoint B	Viewpoint C								
0	<table border="1"> <tr> <td colspan="2">AMT Matrix Entry</td> </tr> <tr> <td colspan="2">Informal Specification</td> </tr> <tr> <td>Static Models</td><td>Behavioral Models</td> </tr> <tr> <td>Static Model Elements</td><td>Behavioral Model Elements</td> </tr> </table>	AMT Matrix Entry		Informal Specification		Static Models	Behavioral Models	Static Model Elements	Behavioral Model Elements	(same structure as A0)	(same structure)
AMT Matrix Entry											
Informal Specification											
Static Models	Behavioral Models										
Static Model Elements	Behavioral Model Elements										
1	(same structure as A0)	(same)	(same)								
2	(same structure)	(same)	(same)								

Fig. 15.1 Artifact and Model Transformation (AMT) matrix structure with entry content

Each entry in the matrix serves one particular, well-defined analysis or design purpose. In the horizontal dimension, the stakeholder viewpoints differ in terms of analysis/design concerns addressed as well as notations used and education/experience required to create artifacts. In the vertical dimension, each level has a certain depth associated to it such as platform-independent conceptual modeling, technology platform-specific but not yet vendor-specific modeling, and executable/installable platform-specific modeling and code. Both informal and formal specifications may be present in each matrix entry; both static structure and dynamic behavior of the system under construction are covered.

Our rationale for making discrete viewpoints a default matrix dimension is the following: Each such viewpoint takes the perspective of a single stakeholder with a particular concern. This makes the artifacts of a viewpoint, i.e., diagrams and models, consumable as it hides unnecessary details without sacrificing end-to-end consistency.²

We decided for realization levels as our second dimension because they allow elaborating analysis results and design artifacts in an iterative and incremental fashion without losing the information from early iterations when refining the artifacts. This is different from versioning a single artifact to keep track of editorial changes, i.e., its evolution in time (in the absence of dedicated artifact management concepts such as those presented in this chapter, the current state of the art is to define naming conventions and use document/file/model versioning to manage artifacts and organize the model space in a project). The same notation can be used when switching from one realization level to another, but more details be added. For instance, a UML class diagram on a logical refinement level might model conceptual patterns and therefore not specify as many UML classes and associations as a Java class diagram residing on the physical realization level. Furthermore, different sets of stereotypes might be used on the two respective levels although both take a functional design viewpoint.

Realization levels support an iterative and incremental work organization which helps to manage risk by avoiding common pitfalls such as big design upfront (a.k.a. analysis paralysis or waterfall) but also the other extreme, ad hoc modeling and developer anarchy.³ Instances of this concept can be found in many places. For instance, database design evolves from the conceptual to the logical to the physical level. Moreover, the Catalysis approach and Fowler in UML Distilled [14] promote similar approaches for UML modeling (from analysis to specification to implementation models). To give a third example, an IBM course on architectural thinking recommends the same three-step refinement for the IT infrastructure (deployment)

²Cross-cutting viewpoints such as security and performance have different characteristics; as they typically work with multiple artifacts, they are less suited to serve as matrix dimensions. However, such viewpoints can be represented a slices (projections) through an AMT matrix, e.g., with the help of keyword tags that are attached to the matrix entries.

³This extreme sometimes can be observed if teams claim to be agile without having digested intent and nature of agile practices.

viewpoint dealing with data center locations, hardware nodes, software images, and network equipment. Finally, the distinction between platform-independent and platform-specific models in Model-Driven Architecture can be seen as an instance of the general approach of refinement levels as well. Additional rationale for selecting viewpoints and realization levels as primary structuring means can be found in the literature [8, 32].

Step 2: Position method-specific artifact types in AMT matrix entries. Our second step is performed by method creators and tool engineers. Each method and each analysis or design tool supporting such method is envisioned to populate the AMT matrix structure from step 1 with artifact types for combinations of viewpoints and realization levels. It is not required to fully populate the matrix in this step; it rather serves as a structuring means. However, gaps should not be introduced accidentally; they should rather result from conscious engineering decisions made by the method creator or tool engineer.

To give an example, we now combine agile practices, OOAD, and CBD; our concepts are designed to work equally well for other methods and notations. Figure 15.2 shows an exemplary AMT matrix; its viewpoints stem from three references [8, 18, 31] and the realization levels are taken from two references [8, 15]. The three AMT matrix entries in the figure belong to the requirements viewpoint (Req-0, Req-1) and the functional design viewpoint (Fun-1). User stories, use cases, and component models (specified as a combination of UML class and sequence diagrams) are the selected artifact types in this example.

Viewpoint \ Realization Level	Requirements Viewpoint (Req)	Functional Design Viewpoint (Fun)										
0 (Informal/Inception)	<p>Requirements Viewpoint @ Realization Level 0 (Req-0)</p> <table border="1"> <tr><td>Vision Document</td></tr> <tr><td>Business Component Heat Map</td><td>Epics and User Stories</td></tr> <tr><td colspan="2">Business Activities</td></tr> </table> <p>refine ↑ trace</p>	Vision Document	Business Component Heat Map	Epics and User Stories	Business Activities		<p>Functional Solution Design Viewpoint @ Realization Level 1 (Fun-1)</p> <table border="1"> <tr><td>Architecture Overview Diagram</td></tr> <tr><td>Component Relationship Diagram (UML)</td><td>Component Interaction Diagram (UML)</td></tr> <tr><td>Components, Connectors</td><td>Component Interactions</td></tr> </table> <p>ensure consistency</p> <p>manage versions of AMT matrix entries</p>	Architecture Overview Diagram	Component Relationship Diagram (UML)	Component Interaction Diagram (UML)	Components, Connectors	Component Interactions
Vision Document												
Business Component Heat Map	Epics and User Stories											
Business Activities												
Architecture Overview Diagram												
Component Relationship Diagram (UML)	Component Interaction Diagram (UML)											
Components, Connectors	Component Interactions											
1 (Logical/Elaboration 1)	<p>Requirements Viewpoint @ Realization Level 1 (Req-1)</p> <table border="1"> <tr><td>Business Case, Development Contract</td></tr> <tr><td>Use Case Diagram (UML)</td><td>Actor-System Interaction Flow (in Use Case)</td></tr> <tr><td>Use Case</td><td>Use Case Step</td></tr> </table> <p>transform ← → trace</p>	Business Case, Development Contract	Use Case Diagram (UML)	Actor-System Interaction Flow (in Use Case)	Use Case	Use Case Step						
Business Case, Development Contract												
Use Case Diagram (UML)	Actor-System Interaction Flow (in Use Case)											
Use Case	Use Case Step											

Fig. 15.2 AMT matrix entry population examples

The AMT matrix for a method produced in steps 1 and 2 answers the following questions:

1. Which viewpoints to use and how many realization levels to foresee (step 1)?
2. Which artifact type(s) to use in each matrix entry, and for what purpose (step 2)?
3. Which notation to select for each artifact and how to customize the selected ones for a particular matrix entry, e.g., syntax profiling (step 2)?
4. Which role to assign ownership of AMT matrix entries to and when in the process to create, read, and update the matrix entries (step 2)?
5. Which techniques and best practices heuristics from the literature to recommend for manual artifact creation and model transformation development (step 2)?
6. Which commercial, open source, in house, or homegrown tools to use for artifact creation, e.g., editors and transformation frameworks (step 2)?

As these questions can be answered differently depending on modeling preferences and method tailoring, the AMT matrix content for a method differentiates practitioner communities (e.g., a practice in a professional services firm or a special interest group forming an open source project).

Having completed steps 1 and 2, the AMT matrix is ready for project use (step 3).

Step 3: Populate a project-specific AMT matrix instance with artifacts. In this step, the AMT matrix structures the project document repository (model space) and is populated with actual artifacts (e.g., models and code) throughout the project.

We continue the presentation of step 3 in the second next section of this chapter. Before that, we formalize the concepts presented so far in a metamodel for artifact management.

15.3 A Metamodel for the AMT Matrix

In this section, we present a metamodel for the AMT matrix. This metamodel may serve as a reference for tool development.

15.3.1 Overview

As our approach targets different audiences (i.e., method creators and tool builders, but also project practitioners), we distinguish between an *AMT metamodel*, a *type-level AMT model* and an *instantiated AMT model*. The involved models and diagrams can be summarized as:

1. AMT metamodel, shown in Fig. 15.3.
2. Type-level AMT model, created by a method creator by instantiating the AMT metamodel. Such a type-level AMT model is created in step 2 of our approach.

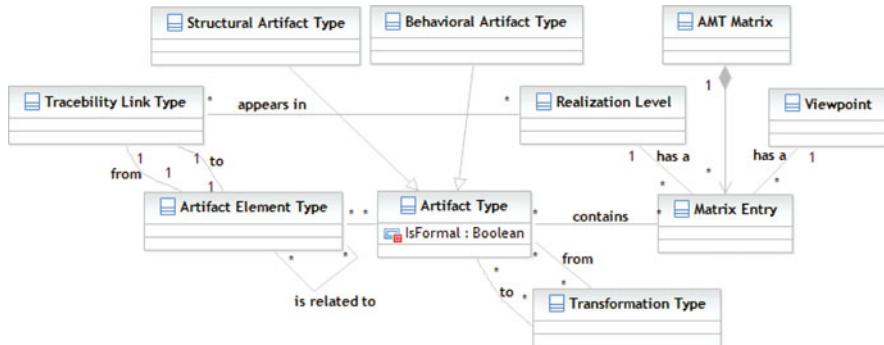


Fig. 15.3 The metamodel for the AMT matrix (represented as a UML class diagram)

3. Instantiated AMT model, which is created by instantiating and populating a type-level AMT model when creating project artifacts on a project and categorizing them according to the AMT matrix approach. Such an instantiated AMT model is created in step 3 of our approach.

Overall, our organization follows well-established principles that are also used in the Meta Object Facility (MOF) [27]. We now describe the AMT metamodel which is used to formalize an AMT Matrix. We outlined in the previous section that an AMT Matrix consists of several Matrix Entries (expressed by the AMT Matrix class and the Matrix Entry class in the metamodel). Each Matrix Entry represents one combination of a viewpoint and a realization level (e.g., Req-0 or Fun-1) and contains a number of Artifact Types. An Artifact Type can either be a Structural or a Behavioral Artifact Type, e.g., a UML class diagram vs. a sequence diagram. Artifact Types consist of Artifact Element Types such as use cases (or use case steps) and UML classes. Artifact Types together with Artifact Element Types can be considered as the metamodel defining a language (i.e., requirements engineering or architecture design notation), i.e., the UML metamodel for UML class diagrams. UML merely serves as an example here; any other formally defined notation can be represented this way. As a metamodel for a language typically consists of a multitude of related model elements, Artifact Element Types are related to each other by an ‘is related to’ association.

Each Matrix Entry is given a Viewpoint which categorizes it (e.g., Requirements and Functional Design viewpoints, see Fig. 15.2 in the previous section). Furthermore, each Matrix Entry is associated to a Realization Level which categorizes the artifacts in the Matrix Entry into realization levels such as the two exemplary ones presented in the previous section (i.e., informal and logical); as motivated in the description of step 1 in Sect. 15.2, other amounts of realization levels and different names can be defined as well (see Sect. 15.5 for examples). An Artifact Type may appear in multiple matrix Entries (e.g., if the same notation is used to create models that serve different purposes). Traceability between artifacts and their elements is supported by the Traceability Link Type which connects Artifact Element Types.

Transformations can be defined as Transformation Types that transform Artifact Types. Examples are use case to component model transformations and links in OOAD/CBD.

The metamodel classes and their associations support the configuration of an AMT Matrix by instantiation, i.e., the creation of a type-level AMT Model. We will now present several examples of creating AMT model instances.

15.3.2 AMT Matrix Modeling Examples (Applying Step 1 and Step 2)

As a first straightforward example, we configure a matrix that supports the example of the previous section. This matrix consists of two viewpoints, a requirements and a functional design viewpoint. There are two realization levels in each viewpoint. On realization level 0 of the requirements viewpoint, there are four artifact types, Vision Document, Business Component Heat Map, Epics and User Stories and Business Activities. Figure 15.4 shows the instantiation of the metamodel to express such an AMT matrix configuration. In Fig. 15.4, ‘Req’ is an instance of the class ‘Requirements Viewpoint’, which is a subclass of ‘Viewpoint’ (the subclass is not shown in Fig. 15.3).

A business component heat map indicates areas of a business that require attention and investment because of market dynamics and the current positioning of an enterprise in the market (relative to competition). A component in such a heat map might be home loan processing (in a banking scenario); a related epic and user story

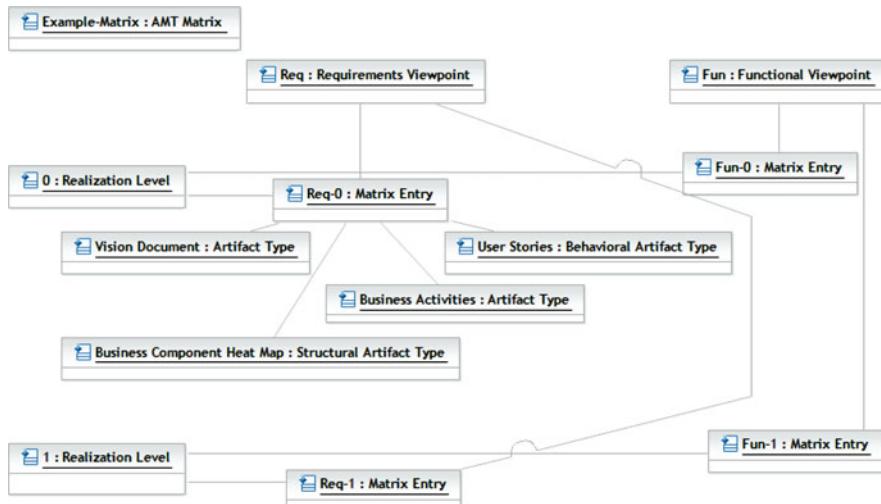


Fig. 15.4 Example of a type-level AMT model (UML object instance diagram)

set might then be ‘modernize and accelerate home loan processing’ and ‘as a retail bank client, I want to be able to apply for a loan online and be able to receive a quote immediately so that I can save time and am able to compare competing quotes’.

These artifacts can be related by transformations that generate parts of artifacts (e.g., an initial set of epics and user stories may be obtained from the heat map). Transformations may also generate traceability links between artifact elements (e.g., to record that a user story has been derived from a heat map).

In addition to the AMT metamodel itself, constraints can be formulated on different levels. Such constraints can pose requirements on a type-level AMT model and also on instantiated AMT models, such as that each configured viewpoint must be represented. Constraints can also be formulated to require a certain relationship in a populated AMT Matrix instance, e.g., that certain traceability links between model elements have to exist. It is also possible to formulate constraints without tying them to a specific AMT Matrix. Such a constraint can state a requirement for an AMT matrix and could require a certain number of artifact types or transformation types to exist.

As a second example, we continue the OOAD/CBD example from step 2 in Sect. 15.2 and discuss an AMT matrix configuration where software architects are advised to analyze the use cases to create an initial component model for the solution under construction. We now perform this design activity in an example and derive an AMT Matrix along the way. We decided to apply the component modeling heuristics for this design step from the literature, specifically ‘UML Components’ by Cheesman/Daniels [7]. This book defines a mature, state-of-the-art OOAD method that is widely adopted in practice by a large population of requirements engineers and architects. Alternatively, other techniques or heuristics could be applied in this activity as well.⁴ With the help of our metamodel, these heuristics can be expressed as model transformations.

In their method, Cheesman/Daniels defined a ‘specification workflow’ which begins with a step called ‘component identification’. Business concepts and use case model are among the input to this step (besides existing interfaces and existing assets). Among other artifacts, it yields ‘component specs & architecture’, captured in a component model artifact [8]. In our exemplary AMT matrix (Fig. 15.2), use case models reside in entry Req-1 and component models in entry Fun-1. Under ‘Identifying Interfaces’, Cheesman/Daniels advise to call out user interface, user dialog, system services, and business services components and trace them back to requirements artifacts. These components offer one operation per use case step.

As the next step, they recommend adding a UML interface to the design for each use case step; for each use case, a dialog type component should be added to the architecture. This scheme can be followed by an architect/UML modeler, or partially automated in a tool targeting the analysis-design interface (or, more specifically, the Req-1 to Fun-1 matrix entry boundary in an AMT matrix).

⁴For instance, domain- and style-specific literature, e.g., on service modeling and SOA design can further assist with this work (see Schloss Dagstuhl Seminar on Software Service Engineering (January 2009) and [29] for examples).

Table 15.1 Artifact and Model Transformation (AMT) matrix for OOAD/CBD (developed in step 2 of our approach)

Matrix entry Metamodel concept \	Requirements engineering, level 1 (Req-1) Role: domain analyst	Functional solution design, level 1 (Fun-1) Role: software architect
Artifact type (structural)	Use case model	Component model (UML class diagram)
Artifact type (behavioral)	Use case scenarios	Component interaction diagrams (UML sequence diagrams) for sunny day and error scenarios
Artifact element type	Use case Use case step Precondition Postcondition	Components as defined in reference architecture Component responsibilities expressed as initial operations in initial system interface Assertion in component specification Out value of operation plus optional assertion
Traceability link type		From component back to use case From operation back to use case step
Transformation type		From use cases to functional components (realization level 1): user interface, user dialog, system services, business services

Table 15.1 maps the AMT metamodel classes from Fig. 15.3 to the artifact types in the OOAD/CBD example from Fig. 15.2 (Sect. 15.2). It also lists an exemplary model transformation implementing the Cheesman/Daniels heuristics for the transition from use cases to components. A full version of this table would provide the complete output of step 2 of our approach for OOAD/CBD.

15.3.3 Additional Metamodel Instantiation Examples and Considerations

An example of a constraint crossing viewpoints is that no logical functional design component in Fun-1 should exist that does not have any traceability link (existence justification) in a requirement, e.g., a use case (on a lower level of refinement, there might be merely technical utility components that only have indirect links to the business requirements). A second example of a constraint is the rule of thumb that no Req-0 user story (see Sect. 15.2) and no Req-1 use case should reference any business entities that have not been defined in a Req-0 glossary or Req-1 OOAD domain analysis model.

Business interfaces (i.e., business type model and business rules) often serve as input to component interface specification work (i.e., providing method signatures). These artifact types can be positioned in our AMT matrix in a similar way as use cases and components; additional traceability link types and transformation types can be defined.

15.4 Example: Instantiation of Artifact Management for an OOAD Project

Once a type-level AMT model has been created (e.g., the OOAD/CBD one from Sects. 15.2 and 15.3), one instance of the type-level AMT model is created per project that employs the method described by the AMT matrix (here: OOAD/CBD). This instantiated AMT model is populated by the project team.

AMT matrix instance for Travel Booking System (applying step 3). In the Travel Booking System example in [7], a ‘Make a Reservation’ use case (for a hotel room as part of a travel itinerary) appears. The use case has the steps ‘Identify room requirements’, ‘System provides price’, and ‘Request a reservation’ (among others). The component identification advice given in the book is to create one dialog type component called ‘Make Reservation’ (derived from use case) and one system interface component providing initial operations such as ‘getHotelDetails()’, ‘getRoomInfo()’, and ‘makeReservation()’ (derived from the use case steps). Use case model and component model for the Travel Booking System are examples of artifacts; the four use cases, the use case steps, and the two components are examples of artifact elements that are linked with traceability links. The transition from the use case to the initial component model can be implemented as a model transformation. Figure 15.5 summarizes these examples of model artifacts, artifact elements, traceability links, and transformations.

All component design activities discussed so far take place on a conceptual, platform-independent realization level; in the Cheesman/Daniels method, component realization (covered in a separate Chapter called ‘Provisioning and Assembly’)

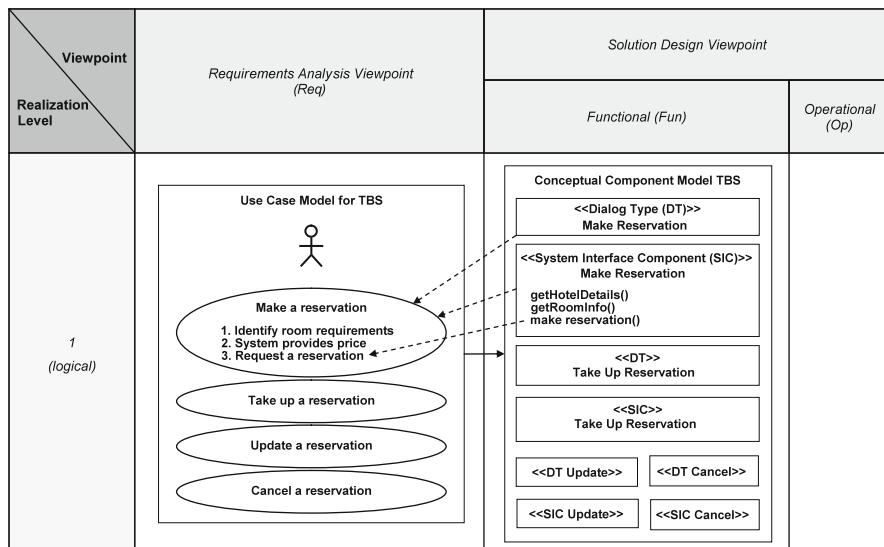


Fig. 15.5 A part of an exemplary AMT matrix instance population

represents the transition from realization level 1 to realization level 2 (or from logical to physical design). All design advice in the book pertains to the functional viewpoint; the operational/deployment viewpoint is not covered. Advice how to place deployment units that implement the specified and realized functional components, e.g., in a Java Enterprise Edition (JEE) technology infrastructure from an application server middleware vendor, is available elsewhere in the literature.

Observations in exemplary application of AMT concepts (discussion). The examples in this section demonstrate that our artifact management ideas are in line with those in established methods and recognized text books; our concepts add value to these assets as they organize the artifacts produced and consumed in a structured, interlinked fashion. They also provide a formal underpinning for such assets that tool builders can use to create method-aware tools of higher value than current Create, Read, Update, Delete (CRUD) editors on metamodel instances.

As a side effect, our examples also indicate that it often makes sense (or even is required) to combine methods, techniques on application development and integration projects. An integrated approach to artifact management facilitates such best-of-breed approach to method engineering.

15.5 AMT Matrix Prototype and Usage Considerations

Prototypical implementation. We have realized AMT matrix support in the *Zurich Artifact Compiler (ZAC)*. ZAC is implemented in Java and Eclipse; its first version has the objective to demonstrate the value and the technical feasibility of our concepts. The current ZAC demonstrator provides Business Process Modeling Notation (BPMN) and UML frontends as well as UML, architectural decisions, and project management tool backends. It is configured with the realization levels and viewpoints from [8] that already served as examples in the previous sections.

The demonstrator supports the following transformations:

- User story to UML use case (Req-0 to Req-1).
- Activity in a process specified in BPMN to UML use case (Req-0 to Req-1, or Req-1 to Req-1⁵).
- UML use case to UML component (Req-1 to Fun-1).
- UML component to architectural decision issues and outcomes (Req-1 to Rat-1).
- User stories to work items in high-level project plan (Req-0 to Mgmt-0), where ‘Mgmt’ stands for ‘Project Management Viewpoint’.

The frontends are implemented as plain Java objects; they process XML files that contain the input models. Backends use Eclipse JET as a template-based transformation framework. The AMT matrix comprises a set of interrelated plain Java objects; the ASTs are realized as graphs, the symbol tables as hash tables.

⁵Depending on the positioning of BPMN in the method used to create and configure the type-level AMT model

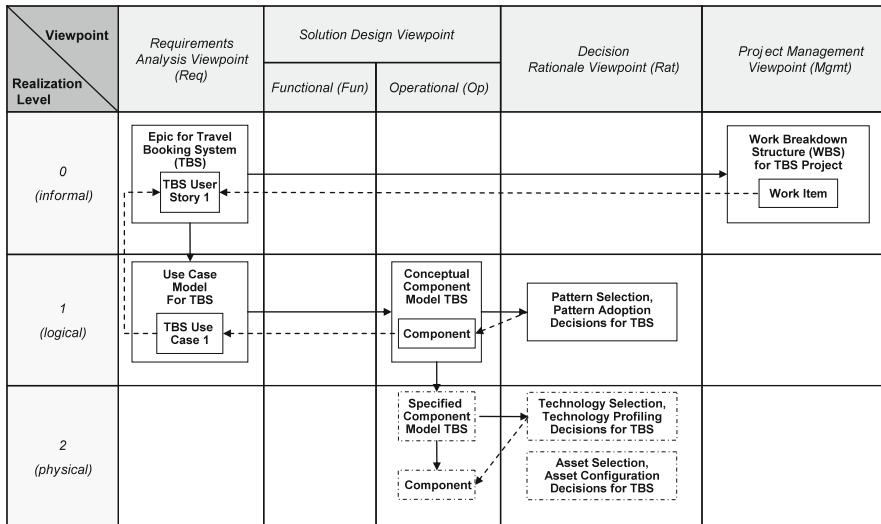


Fig. 15.6 Exemplary AMT matrix configuration (subset of entries)

Figure 15.6 shows an instance of an AMT matrix for OOAD/CBD (i.e. an instantiated AMT model) as it would be developed on a project. The case study from the previous section serves as example here. This instance was developed with the ZAC prototype; it extends the examples given in previous sections of the chapter. The five viewpoints, three realization levels, and various model artifact types (e.g., component relationship diagram) originate from the *IBM Unified Method Framework (UMF)*. UMF is the standard method used by all IBM architects on professional services engagements with clients; under predecessor names, it has been applied on numerous commercial enterprise application development and integration projects since 1998. UMF leverages the UP metamodel, but adds a rich set of method content.

In the prototype, an agile user story elicitation effort (Req-0) yields candidate use cases (Req-1), and candidate components (Fun-1); examples of such artifacts appeared in the previous section of this chapter. A work breakdown structure can be derived from these inputs (Mgmt-0); its work items concern the design and development of the candidate components. The transitions between entries (i.e., realization levels or viewpoints) are accompanied by related architectural decisions that preserve the justifications of the chosen designs, e.g., pattern selection and adoption decisions regarding the conceptual component model (Rat-1). The solid arrows in the figure either represent human design activities or model transformations.⁶ The dotted arrows represent traceability links. For instance, we have suggested

⁶Such transformations are algorithms/functions that accept one or more models as input and return the same or another set of models as output

a pattern-centric decision identification technique connecting Fun-1 and Rat-1 in our previous work [32].

Usage scenarios and roles. Software engineering methods can be characterized by their artifacts, their process (i.e., roles, phases/tasks/activities), the techniques for artefact creation they provide, and their reusable content. Hence, we expect method creators to configure AMT matrices according to particular methods such as UP, domain analysts (requirements engineers) to populate the AMT entries for the requirements viewpoint, software architects to own the entries in the functional and operational design viewpoints (i.e., Fun-1, Fun-2, Op-1, Op-2), and platform specialists to populate the design AMT matrix entries on the physical realization level (e.g., developers for Fun-3). These roles may perform the following activities:

1. Populate existing AMT matrices supporting an out-of-the-box configuration (this is a mere application scenario, corresponding to our step 3 from Sect. 15.2).
2. Develop custom transformations, e.g., in support of software provisioning; in such transformations, models in the operational viewpoint can be leveraged, e.g., when transitioning from a deployment pattern to a concrete software-as-a-service offering (step 2).
3. Configure AMT matrices with other viewpoint models and/or additional realization levels (step 1).
4. Develop additional applications and utilities leveraging the data in an AMT matrix, e.g., effort estimation applications and artifact search utilities (step 2 and step 3).
5. Instantiate the AMT metamodel to support new methods, artifact management tools, or model transformation frameworks (step 1 and step 2).

Another usage scenario for the matrix is to compare methods and position reusable assets; an agile project uses many realization levels, attempts to produce very few artifacts for each entry, and traverses the matrix several times per day (a formalization of continuous integration); a waterfall project has only one realization level (matrix row) and traverses this row once per project.

Discussion. In the current state of the practice, we find tools that produce a number of artifacts without providing an integrated view of all knowledge about the system under construction. This knowledge is contained in the various created artifacts. Moreover, some tools overlap in their functionality and expose their users to semantic dissonances and tedious, error-prone import-export or copy-paste activities as it is not clear to which viewpoint and realization level the tool output belongs. Naming conventions and package hierarchies are used to organize artifacts.

These problems can be overcome if configurable AMT matrix support is introduced to existing and emerging requirements engineering and architecture design (modeling) tools to integrate them into an *Integrated Modeling Environment (IME)* that ties in with state-of-the-art development environments such as Eclipse-based Integrated Development Environment (IDEs). Details of such integration effort are out of scope of this chapter (and this book).

Applying our presented approach requires some effort from method engineers and project teams. For instance, matrix dimensions, viewpoints, and realization levels have to be defined and artifact types and artifacts have to be assigned to matrix entries. However, much of this effort is spent anyway, even without our approach being available (if this is not the case, project teams run the risk of creating unnecessary artifacts). Furthermore, well-established engineering practices such as separating platform-independent from platform-specific concerns and distinguishing stakeholder-specific concerns are easily violated if established concepts such as viewpoints and realization levels are not applied. Our AMT matrix approach supports these well-established concepts natively and combines them to their mutual benefit. What is minimally required from the method engineer and the project team is to reflect and clearly articulate the purpose of each artifact type/artifact in terms of its viewpoint and realization levels. We believe that the necessary effort of following our approach is justified by its various benefits: (1) it enables checking the completeness of artifact types/artifacts across all required stakeholder concerns across the software engineering lifecycle, (2) it enables checking the absence of redundancy across artifact types/artifacts and (3) it allows method engineers and project teams to specify and reason about traceability and consistency of artifacts/artifact types in a more conscious and disciplined way.

Furthermore, we believe that the required effort can be minimized through good tool design. For instance, a tool that supports role-specific configuration of its user interface and is aware of method definitions, e.g., of requirements analysis and architecture design methods, can automatically tag artifacts according to their location within the matrix. This assumes that the tool has been configured with an AMT matrix.

15.6 Related Work

In the software engineering community, viewpoints have been introduced to capture different perspectives of stakeholders involved in the development. The ViewPoints framework [13] introduces a viewpoint to consist of a representation style, a domain, a specification, a work plan and a work record. A viewpoint is used to express the concerns of a particular stakeholder involved in the development. The representation style is used for describing in which language a viewpoint is expressed. The domain is a name given to the part of the world that is seen by the viewpoint. The specification is used to capture a partial system description, using the representation style. The work plan captures the development process knowledge in this viewpoint and the work record the development history.

The ViewPoints framework has been used in [26] to explore the relationship to a software development process. For that purpose, a viewpoint template is introduced where only the representation style and the work plan slot is fixed. A software engineering method is then a configuration of viewpoint templates and their relationships.

If a software system is described from different viewpoints, this immediately raises the issue of consistency and inconsistency. In the ViewPoints framework, this is addressed by defining relationships between viewpoints, called inter-viewpoint rules. These rules can be checked at certain points in the development process.

In the context of the ViewPoints framework, many case studies have been performed. The ViewPoints framework has triggered substantial research in the area of consistency management (see, e.g., [3, 10–12, 16]). The work presented in this paper builds on the original work by Finkelstein et al. and extends their concepts in a method engineering context, combining them with realization levels and ideas from model-driven development.

Recent work by Anwar et al. [1] introduce the VUML profile to support view-based modeling from analysis to coding. Each actor of a system is associated with a viewpoint. In each viewpoint, use cases and scenarios are described and class diagrams are derived. A VUML model is then composed of these partial models. This composition is automated using model transformations. In contrast to our work, Anwar et al. focus on model composition for a fixed set of viewpoint models. They do not address the problem of configuring and defining the relationship between different artifacts.

In the software architecture community [24], an architectural viewpoint consists of a ‘viewpoint name, the stakeholders addressed by the viewpoint, the architectural concerns framed by the viewpoint and the viewpoint language.’ A viewpoint can also include consistency and completeness checks and heuristics, patterns, guidelines. It is explicitly mentioned that a viewpoint is a template for a view. A view itself is an aggregation of models that represents the software system from a specific angle, focusing on some concerns. According to Rozanski and Woods [31], a viewpoint is defined as ‘a collection of patterns, templates, and conventions for constructing one type of view. It defines the stakeholders whose concerns are reflected in the viewpoint and the guidelines, principles, and template models for constructing its views’.

When comparing the different notions of viewpoints one can conclude that viewpoints in the different approaches are very similar. In all cases, a viewpoint consists of stakeholders whose concerns are addressed, a notation (representation style or modelling languages), some method or process knowledge, as well as viewpoint relationships. Our work is in line with all presented definitions and combines disjoint/discrete viewpoints with realization levels in a novel way to structure the artifact landscape (i.e., the model space) in a project to the benefit of the various stakeholders. Cross-cutting viewpoints are not yet covered by our AMT formalization.

Another area of related work is tool and model integration. Milanovic et al. [25] describe how artifacts created in a software development process are stored and managed in the BIZYCLE repository. This repository includes repository management operations such as consistency and metadata management and provides central point of integration.

Using a configuration manager, artifacts and their relationships as well as consistency rules can be defined which are then used by the automatic artifact

management during a project. The ideas presented by Milanovic et al. are similar, however, their focus is on the common repository. It is not described how such a common repository can be technically realized. Our AMT Matrix concept and the metamodel can be used to structure the common repository when realizing an integrated tool infrastructure. Recent work by Broy et al. [5] criticizes the current state-of-the art of tool integration and proposes different ways to improve the situation. We believe that our AMT matrix can be used as one means and basis for seamless model-based development.

Earlier work in the area of consistency management has already recognized that consistency constraints are dependent on the development process and the application domain. Küster [20] describes a method how consistency management can be defined dependent on the process and application domain. Further, the large body of work on consistency checking and resolution (see, e.g., [9, 17, 23]) must be integrated and adapted by tools working with the AMT Matrix in order to achieve full benefit of it. This also applies to the work on traceability which establishes traceability links based on, e.g., transformations and the work on defining and validating model transformations (see, e.g., [21, 22]).

15.7 Conclusion

In this chapter, we first observed several problems encountered by method creators and tool builders as well as requirements engineers and architects on projects. We then generalized these problems into conceptual artifact management issues and argued that these issues are currently not sufficiently addressed by techniques and tools from the software engineering and modeling communities. Based on these observations, we introduced the Artifact and Model Transformation (AMT) matrix which provides a categorization and structuring means for artifacts and their relationships in a project. We developed and presented our solution in three steps:

1. Specify AMT matrix dimensions (i.e., viewpoints and realization levels by default).
2. Position method-specific artifact types in AMT matrix entries (according to their purpose).
3. Populate project-specific AMT matrix instance with artifacts (according to their type).

In its default incarnation, our approach combines two commonly applied complexity management concepts, viewpoints and realization levels, to their mutual benefit. Other and/or additional structuring dimensions can be defined (future work). An AMT matrix can be populated with artifact types to support the needs of a specific method; project teams then create and manage artifact instances of these types by their matrix position. Such a configuration requires the definition of relationships and transformations between artifacts as well as consistency and traceability management. We provided a metamodel for the

AMT Matrix which can be used as a reference model for integrating AMT matrix concepts into tools. We also presented an exemplary OOAD/CBD instantiation of the AMT metamodel and applied it to a sample project. Finally, we discussed the usage scenarios, benefits, and the implications of our approach.

In future work, we plan to evaluate further our approach, e.g., via personal involvement in industry projects (action research) and via joint work with developers of commercial requirements engineering and architecture design tools. Further research includes the elaboration of how consistency and traceability management can be defined on the basis of the AMT matrix as well as adapting existing modeling tools to support the AMT matrix when defining modeling artifacts. Cross-cutting viewpoints also require further study; tagging the matrix entries that address cross-cutting design concerns such as performance and security to slice AMT instances seem to be a particularly promising direction in this regard.

References

1. Anwar A, Ebersold S, Coulette B, Nassar M, Kriouile A (2010) A rule-driven approach for composing viewpoint-oriented models. *J Object Technol* 9(2):89–114
2. Bass L, Clements P, Kazman R (2003) Software architecture in practice. Addison Wesley, Reading
3. Boiten E, Bowman H, Derrick J, Steen M (1997) Viewpoint consistency in Z and LOTOS: a case study. In: Fitzgerald J, Jones CB, Lucas P (eds) FME'97: industrial applications and strengthened foundations of formal methods (Proceedings 4th International symposium of formal methods Europe, Graz, Austria, September 1997). (Lecture notes in computer science), vol. 1313. Springer-Verlag, Heidelberg, pp 644–664
4. Booch G (1991) Object-oriented design. Benjamin-Cummings, Redwood City
5. Broy M, Feilkas M, Herrmannsdorfer M, Merenda S, Ratiu D (2010) Seamless model-based development: from isolated tools to integrated model engineering environments. *Proc IEEE* 98(4):526–545
6. Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stal M (1996) Pattern-oriented software architecture. Wiley, Chichester
7. Cheesman J, Daniels J (2001) UML components. Addison-Wesley, Reading
8. Eeles P, Cripps P (2009) The process of software architecting. Addison-Wesley, Upper Saddle River
9. Egyed A, Letier E, Finkelstein A (2008) Generating and evaluating choices for fixing inconsistencies in UML design models. In: ASE, IEEE 99–108
10. Engels G, Küster JM, Groenewegen L, Heckel RA (2001) Methodology for specifying and analyzing consistency of object-oriented behavioral models. In: Gruhn V (ed) Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, ACM, New York, NY, USA, pp 186–195
11. Finkelstein A, Gabbay D, Hunter A, Kramer J, Nuseibeh B (1993) Inconsistency handling in multi-perspective specifications. In: Sommerville I, Paul M (eds) Proceedings of the fourth European software engineering conference, Springer-Verlag, 84–99
12. Finkelstein A, Gabbay D, Hunter A, Kramer J, Nuseibeh B (1994) Inconsistency handling in multi-perspective specifications. *IEEE Trans Software Eng* 20(8):569–578
13. Finkelstein A, Kramer J, Nuseibeh B, Finkelstein L, Goedicke M (1992) Viewpoints: a framework for integrating multiple perspectives in system development. *Int J Software Engineer Knowledge Engineer* 2(1):31–57

14. Fowler M (2000) UML distilled. Addison-Wesley, Reading
15. Frankel D (2003) Model-driven architecture: applying MDA to enterprise computing. Wiley, Indianapolis
16. Ghezzi C, Nuseibeh B (1998) Special issue on managing inconsistency in software development. IEEE Trans Software Eng vol Vol. 24, No. 11, November 1998, IEEE CS Press, pp. 902–906
17. Groher I, Reder A, Egyed A (2010) Incremental consistency checking of dynamic constraints. In: FASE, (Lecture notes in computer science), vol 6013. Springer, pp 203–217
18. Kruchten P (1999) The rational unified process: an introduction. Addison-Wesley, Boston
19. Kruchten P, Lago P, van Vliet H (2006) Building up and reasoning about architectural knowledge. In: QoSA (Lecture notes in computer science), vol 4214. Springer, pp 43–58
20. Küster JM (2004) consistency management of object-oriented behavioral models. PhD thesis, University of Paderborn
21. Küster JM (2006) Definition and validation of model transformations. Softw Syst Model 5 (3):233–259
22. Küster JM, Abd-El-Razik M (2007) Validation of model transformations – first experiences using a white box approach. In models in software engineering, workshops and symposia at models 2006, Genoa, Italy, 1–6 October 2006 Reports and Revised Selected Papers (Lecture notes in computer science), vol 4364. Springer, pp 193–204
23. Küster JM, Ryndina K (2007) Improving inconsistency resolution with side-effect evaluation and costs. In MoDELS (Lecture notes in computer science), vol 4735. Springer, pp 136–150
24. Maier M, Emery D, Hilliard R (2001) Introducing IEEE 1471. IEEE Computer
25. Milanovic N, Kutsche R-D, Baum T, Cartsburg M, Elmasgunes H, Pohl M, Widiker J (2008) Model&metamodel, metadata and document repository for software and data integration. In: MoDELS (Lecture notes in Computer Science), vol 5301. Springer, pp 416–430
26. Nuseibeh B, Kramer J, Finkelstein A (1994) A framework for expressing the relationships between multiple views in requirements specification. IEEE Trans Software Eng 20 (10):760–773
27. Object Management Group (OMG) (2006) *Meta Object Facility (MOF) core specification, OMG available specification version 2.0*, OMG Document Number formal/06-01-01
28. Object Management Group (OMG) (2009) *OMG unified modeling language, superstructure. version 2.2.*, OMG Document Number formal/2009-02-02
29. Papazoglou M (2007) Web services: principles and technologies. Prentice-Hall, Boston
30. Pohl K (2010) Requirements engineering – fundamentals, principles, and techniques. Springer-Verlag, Berlin
31. Rozanski N, Woods E (2005) Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley, Boston
32. Zimmermann O (2009) An architectural decision modeling framework for service-oriented architecture design. PhD thesis, University of Stuttgart

Chapter 16

Onions, Pyramids & Loops – From Requirements to Software Architecture

Michael Stal

Abstract Although Software Architecture appears to be a widely discovered field, in fact it represents a rather young and still maturing discipline. One of its essential topics which still need special consideration is systematic software architecture design. This chapter illustrates a set of proven practices as well as a conceptual method that help software engineers classify and prioritize requirements which then serve as drivers for architecture design. All design activities follow the approach of piecemeal growth.

16.1 Introduction

Although Software Architecture appears to be a widely discovered field, in fact it represents a rather young and still maturing discipline. One of its essential topics which still need special consideration is systematic software architecture design. Currently, there are many open issues regarding the design process. In particular, does an appropriate practice exist that allows engineers to systematically turn architecturally relevant requirements into high quality design? After all, the quality of a software system depends on its ability to achieve goals of the involved stakeholders. This chapter illustrates a set of proven practices as well as a conceptual method introduced within SIEMENS that helps software engineers classify and prioritize requirements which then serve as drivers for architecture design. All design activities follow the approach of piecemeal growth. For this purpose, two models, the Onion and the Pyramid model, help structure the design process in a requirements – and risk-driven way. Engineers start emphasizing the most important forces before moving to less important ones. Such an approach ensures that engineers implement the most critical requirements, while they may optionally skip less important goals in case of budget or time problems. The method supports and favors but is not constrained to an iterative-incremental process. It introduces change as an additional force in the design process, because change in projects is the rule not the exception. Thus, although engineers will typically not be able to

provide the optimal architecture solution due to such changes, they will at least be able to create solutions with sufficient quality and traceability.

It is important to note, that the introduced architecture design process does not claim to be the only possible approach for systematic architecture design. Nor does the sequence of steps and activities define the only possible order. Rather the process introduces best practices from real and successful system development projects at SIEMENS.

16.2 Systematic Design – The Principle of Piecemeal Growth

16.2.1 Loops

An important topic for designing a software system is the kind of development process the designers should facilitate. In practice, a Big Design Upfront approach cannot work, because its success would have a lot of prerequisites such as the early availability of all requirements in a consistent and complete specification, the stability and suitability of the project-specific technology roadmap, and the early detection and removal of any design errors. In practice, many things keep changing in a software development project – “panta rhei” as the Greek philosopher Heraclitus once said. Thus, professional software architects should prefer piecemeal growth which implies that the software architecture design process reveals [1] the following properties (see Fig. 16.1):

- An *iterative* approach which does not only focus on adding new features in iterations but also emphasizes quality evaluation and refactoring. Before and after extending a software system, necessary steps include evaluating the architecture and refactoring it when needed. Only with regular architecture assessment and refactoring in place we can keep the architecture under control by preventing architecture erosion due to accidental complexity.
- An *incremental* approach where teams or persons develop different parts of system architecture at different times or at different pace and integrate the completed artifacts into the overall software design. The result of such integration is always

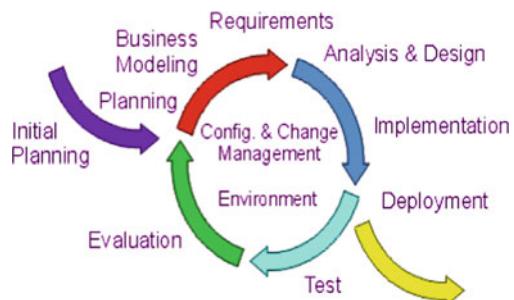


Fig. 16.1 An iterative-incremental design model is appropriate for software design

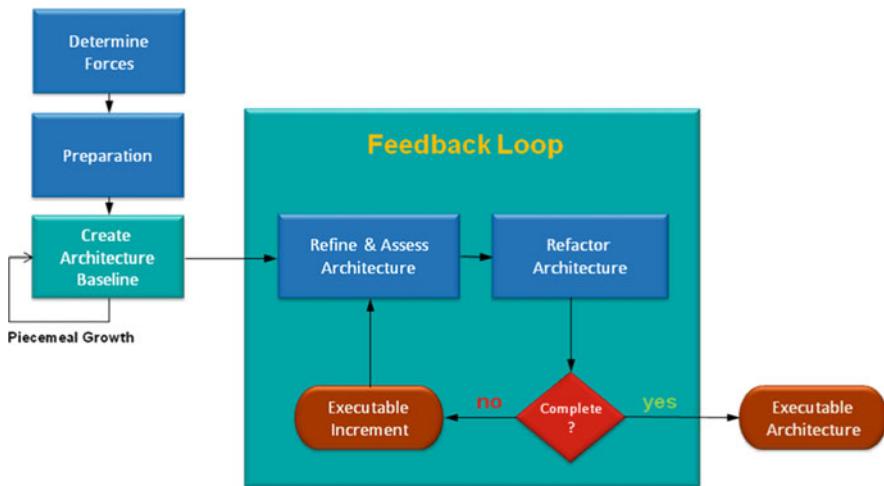


Fig. 16.2 Iterations are introduced in the architecture creation process to help master complexity and to avoid accidental complexity. Piecemeal growth is ensured by iterative loops

a working product. This kind of continuous integration is the approach preferable to any big bang integration.

An important extension to these properties of iterative-incremental design is to embrace change instead of relying on a frozen set of requirements. It should however be noted that for piecemeal growth it is not required to commit to an agile approach with its time-boxed iterations, agile principles, values and development methods.

The principle of piecemeal growth imposes some consequences on software architecture and design (see Fig. 16.2). Software architects should perform their design activities in the following order:

- Identify and prioritize all factors that have an influence on their work such as requirements, risks, business goals, or technology constraints. Prioritization of forces may happen in different ways using existing methods and tools such as leveraging Quality Trees for operational and developmental requirements [2];
- Prepare for the actual design, for instance create a model of the problem domain using Domain Driven Design or other approaches;
- Create a coarse-grained architecture baseline in an iterative loop handling strategic requirements and important tactical requirements;
- Iteratively for each requirement (or a semantically related group of requirements):
 - Evaluate the current state of the software architecture under construction, perform a SWOT analysis (Strengths, Weaknesses, Opportunities and Threats), refactor the software architecture to eliminate the problems found and prepare it for further refinement;

- Refine the architecture baseline with the selected requirement or requirement group until eventually the working software system is ready for rollout and meets its expectations.

16.2.2 Drivers of Systematic Design

To achieve internal architectural qualities such as simplicity, traceability or expressiveness, architects should follow some fundamental guidelines:

- *Requirements should drive all architectural decisions.* There must not be any architecture entity introduced without an associated architecturally relevant requirement. With other words, for introducing an entity into software architecture, software engineers need a concrete design rationale which explains the exact need for this entity. Otherwise architects and developers may come up with design pearls that don't satisfy specific needs but increase the likelihood of accidental complexity. Requirements-driven design also helps with traceability, because each design decision is based on a concrete requirement. In addition, it supports simplicity and expressiveness.
- *Risk-driven development.* Designers, i.e., architects and developers, need to address factors that impose high risks before dealing with low risk issues. Risk mitigation is an important topic for architecture design. Architects should introduce appropriate mitigation strategies for all identified risks. For example, strategic technology decisions such as the use of a particular distribution middleware might stay in conflict with operational qualities such as performance expectations. Thus, a risk mitigation strategy for this concrete example could comprise building a prototype for investigating the technical feasibility of the performance aspect using the prescribed middleware technology.
- *Test-driven Development and Architecture Evolution.* The main property of piecemeal growth is the continuous evolution of the software architecture. Architects add new constituents, and change or even remove architectural parts. Hence, they must ensure that each step in architecture evolution does not invalidate previous steps by affecting whether the software architecture meets its requirements. For this purpose, it is necessary to introduce regular testing activities as well as architecture, design and code introspections. To improve the architectural structures with respect to internal architecture qualities and eliminate unwanted side-effects architects should plan regular architecture refactoring activities.

16.2.3 Architecture From 10,000 Feet – Onion Model

As previously explained, it is the goal of software development projects to ensure that the resulting system meets all architecturally-relevant requirements. The Onion

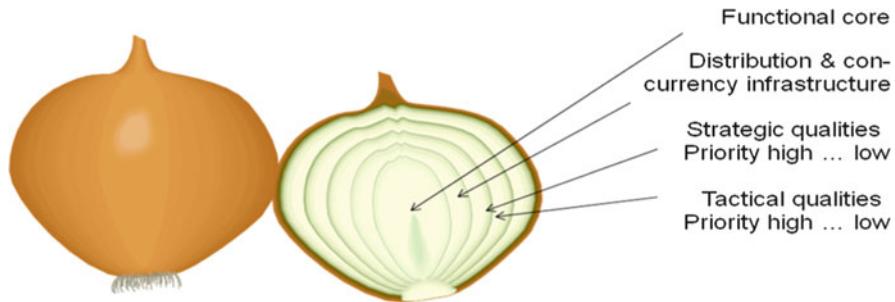


Fig. 16.3 The Onion model structures the architecture and design process into four different phases

model (see Fig. 16.3) is supposed to introduce a systematic model for all necessary architectural design activities.

Its core introduces all functional aspects of a software system, thus representing the problem domain functionality software engineers need to implement. All other requirements, no matter whether of operational, developmental or infrastructural type can only be addressed with these functional responsibilities in mind. For instance, it is not useful to build a flexible or high performance system without any knowledge about the actual parts that need to be flexible or well performing.

Of course, we must have previously assigned unique priorities to all functional requirements – and likewise to all other requirements. Consequently, designers can move step by step from the functional requirement with the highest priority down to the one with the lowest priority. This does not imply that all functional or other requirements must be known upfront nor does it mean we need to cover them in full detail. It only suggests we should at least know a sufficient subset of high priority requirements in advance, where “sufficient” heavily depends on influential factors such as forces from the problem and solution space which is the reason why it proves to be (almost) impossible to specify a rule of thumb. When applying use cases as a means for functional design, we could also adhere to the concept of piecemeal growth by first considering the main scenarios (also known as “Happy Day” scenarios), followed by alternative or exceptional flows (“Rainy Day” scenarios).

After designing the functional core of the software system with a priority – and requirements – driven design process, the two subsequent phases consist of first, addressing infrastructural and then, operational requirements. Note, that we could also view infrastructural requirements as a subset of operational qualities. However, infrastructure prerequisites such as the necessity of distributing logic or executing it concurrently typically reveal an indispensable precondition which operational (and developmental) qualities must take into account. Therefore, it often turns out useful to separate infrastructural qualities with respect to distribution and concurrency from operational qualities and to assign them the highest priorities.

Thus, the first sub step comprises determining an appropriate distribution and concurrency infrastructure. Architecture patterns such as Broker or Half-Sync/Half-Async help find the main constituents of this infrastructure and can be further

refined using design patterns such as Proxy or Monitor Object. In this process, architects should integrate the software architecture they have already designed as functional core with the distribution and concurrency infrastructure. This way, they are beginning to mix in, step by step, elements from the solution space into architectural concepts from the problem space.

In the third phase software architects integrate the remaining operational qualities such as security, performance or reliability ordered by descending priorities. This activity resembles the previous one for introducing the distribution and concurrency infrastructure. Designers take care of the operational quality, conceive an architectural infrastructure to handle the quality, and then integrate the current architecture, i.e. functional core plus infrastructure, with the architecture entities imposed by the current quality. If architecture and design patterns exist for the quality under consideration, designers should leverage existing best practices instead of reinventing the wheel. In the Onion model each operational quality represents an additional onion layer.

As all architecture refinements and extensions are priority-driven, more important requirements will have more impact on the software architecture than less important requirements. For instance, if security is considered more important than performance, we'll end up with a different software system, than if performance were considered the predominant quality. Thus, it is an essential prerequisite to assign unique priorities to each requirement. Otherwise, architects would face ambiguous precedence rules when facing design decisions.

The fourth phase in the Onion model addresses developmental qualities such as maintainability, manageability or modifiability. Again, designers handle these qualities in descending priority order. This activity happens after designing the operational qualities, because developmental properties need to operate on functional and operational entities in the software architecture. With other words, we require those architecture parts to be already available that we expect to modify. For example, modifiability could imply the exchange of a specific functionality such as the communication protocols used. It could also mean to configure algorithms or the sizes of thread pools for performance improvement.

16.2.4 Dealing with Bottom-Up Requirements in the Onion Model

As the Onion model illustrates, the obvious way to create software architecture is via a top-down, breadth-first approach. However, only a few software development projects start as green field projects. In most project contexts various constraints limit the freedom of designers. For example, in the following cases:

- Usage of specific operating systems, legacy components, middleware, standard information systems, or tools is required.
- In Embedded and especially in Real-Time systems, resources such as CPU time or memory are only available in limited quantity.

- For the integration into an existing environment, the software system under development needs to provide interfaces to external systems or use external interfaces.

How can designers cope with such Bottom-up requirements? An appropriate solution requires stakeholders to assign priorities to these like to all other requirements and to use them as input for the kind of top-down design promoted by the Onion model. With this strategy, we can ensure that bottom-up constraints have the necessary impact on the software architecture and its implementation. Of course, stringent limitations such as restricted main memory must get the highest possible priorities as they directly affect the feasibility of all other architecture and technology decisions.

16.2.5 How Deep Should We Go? – The Pyramid Model

A main challenge of software architecture creation is to define a clear boundary between strategic and tactical design. When is strategic design supposed to end and tactical design supposed to start?

We expect software architects to define an architecture baseline as well as guiding principles the developers should adhere to during fine design. If, on one hand, the architecture contains too many levels of abstraction, for instance from the system level down to the method level, first of all, the whole architecture vision might get lost, and secondly complexity becomes unmanageable due to the large number of architectural entities and their relations. On the other hand, an insufficient number of abstraction levels lead to architecture baselines that only scratch the surface of the problem and are open to interpretation by developers.

As a rule of thumb, software architects should follow the Pyramid model (see Fig. 16.4) that proposes three levels of abstraction. This model has been derived from experiences in many projects at SIEMENS. For a large software system, the baseline architecture suggests the following abstractions:

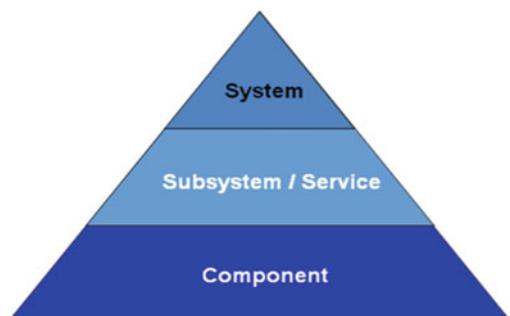


Fig. 16.4 Architects should constrain their work to three levels of abstractions as illustrated by the pyramid model

- *System as a whole*: In this abstraction layer the system is considered a black box that is in charge of a set of responsibilities and that lives in an environment with which it interoperates.
- *Subsystems*: If we leave the black-box view and enter a grey-box view, the system consists of major interrelated building blocks, typically called subsystems, which are responsible to implement the system's core responsibilities.
- *Components*: Subsystems are usually too coarse-grained in order to understand how requirements such as use cases are mapped to the software architecture. Thus, architects introduce interrelated components as constituents of subsystems in order to reveal a more detailed design. In contrast to subsystems that represent a kind of artificial abstraction level, programming platforms often support the notion of components.

Note that terms such as subsystem or component in this context denote architectural entities with different abstraction levels. While subsystem introduce a coarse-grained partitioning of system responsibilities, in the ideal case each subsystem representing one responsibility, components define the fundamental finer-grained building blocks to implement the subsystems.

This model of abstraction is not prescriptive. In a pure Service-Oriented Architecture we would rather introduce system, services and components as abstraction layers, while for small systems abstraction layers such as system, component, and class might be more appropriate. Architects choose whatever model is most suitable with respect to their concrete problem context.

16.2.6 Detailed Process Steps for Architecture Creation

The Onion and the Pyramid model add important means to the tool & value set of software architects as these models provide guidance how to map requirements to architecture decisions and where to set the boundary between software architecture and design. Both models focus on a coarse-grained level of establishing architecture decisions systematically from requirements. However, they do not define an adequate architecture design process with more detailed steps. To establish the coarse-grained feedback model for software architecture design that got introduced in Fig. 16.2, different alternative solutions are possible. Nonetheless, we can identify some common practices software engineers should employ to create the design efficiently and effectively, no matter what concrete problem context they are actually facing. To illustrate the conceptual idea, let us introduce a kind of process pattern that is based upon these practices which it instantiates and concretizes in a concrete process model (see Fig. 16.5).

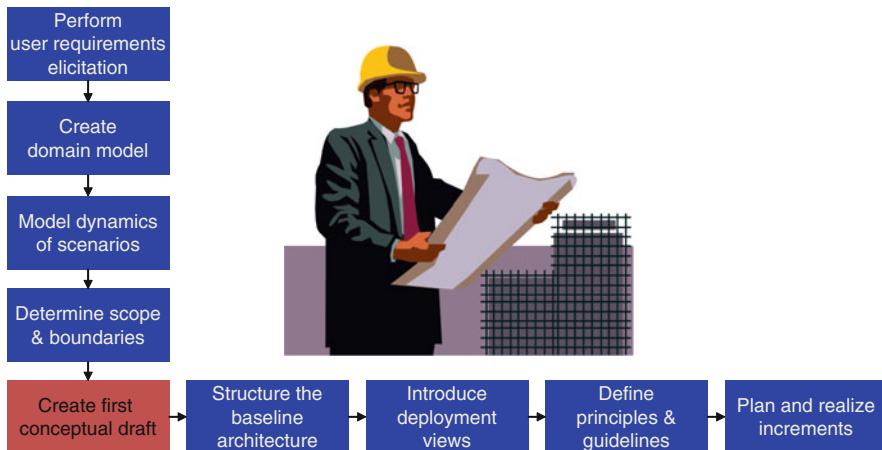


Fig. 16.5 Architects and developers need to execute several steps to create the implementation

16.2.6.1 Preconditions and Assumptions

Our assumptions and preconditions for the applicability of the process pattern are as follows:

- In the context of the system development project, a development process model has been already defined or prescribed by the organization. In the ideal case, the development process uses an iterative-incremental or agile process model, but the suggested process pattern may also be applied in an iterative waterfall model.
- The software architects and developers know the business goals of their organization. Business aspects are important for dealing with architecture decisions and risks.
- We also assume familiarity of the development team with the requirements and other forces and constraints that drive system development, but we don't expect a frozen and complete set of requirements nor their availability in all details. Even if the set of requirements isn't complete, there needs to be a sufficient number of core requirements to start architecture creation, for example high priority use cases and infrastructural/operational requirements. There is no metrics for quantifying "sufficient," as this heavily depends on various factors such as maturity of the problem domain or available expertise and knowledge. The available requirements should have been prioritized by all stakeholders to allow architects and developers decide which direction to use in all decisions. In addition, the requirements and other forces should have been specified with the necessary quality. i.e., they need to be cohesive, complete, correct, consistent, current, unambiguous, verifiable, feasible, mandatory, and externally observable.
- Software architects have already taken the system requirements and figured out the architecturally relevant requirements.

The following sections explain the details of the activities that are introduced in Fig. 16.5.

16.2.6.2 Show Case

The description of the process steps is accompanied by concrete examples from SIEMENS projects with the foremost example being a concrete Warehouse Management system developed for a large organization. Such a system is used to load and retrieve goods into respectively from various kinds of storages using specific storage mechanisms. The transportation of goods is also considered as a special kind of storage such as a belt or a cart. All operations are triggered by orders. Obviously the main use cases comprise manage storages, query storages, load and retrieve goods. In addition, the system is supposed to integrate with existing ERP systems.

Two of the key operational and developmental requirements have been:

- High Performance when loading or unloading
- High Scalability (and Concurrency) in terms of orders processing

16.2.6.3 User Requirements Elicitation

In the first step architects consider only functional aspects. They create a set of use cases with high priorities together with those stakeholders who are in charge of requirements engineering. High priority use cases are covered first because they need to have the most significant impact on the functional core (as described in the Onion model). Use cases in this set should be available as textual use case descriptions (see Fig. 16.6), not as Use Case Diagrams, because Use Case diagrams provide an overview of use cases without offering any details or explanations.

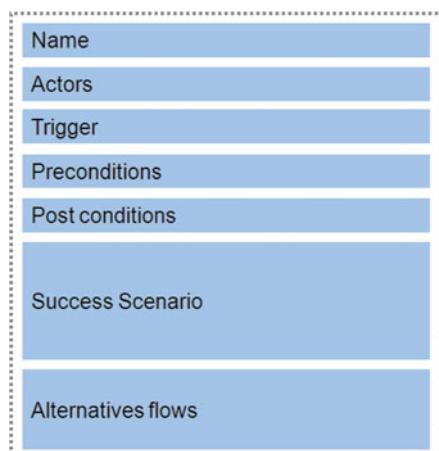


Fig. 16.6 Architects need use cases in a textual description as explained by Alistair Cockburn

If the number of high priority use cases is very large, architects should partition each use cases in a success scenario and alternative flows as appropriate strategy for mastering complexity. They could consider the success scenarios first before continuing with the alternative flows.

16.2.6.4 Create Domain Model

Stakeholders in a project should create a model of the problem domain or use an existing one for efficient and effective communication. Software engineers require a formal model to understand user requirements and their contexts. A formal model could also end up in a DSL (Domain-Specific Language) and serve as a base for automation. While use cases provide a black box view of the software system, especially of its functional aspects, the problem domain model can help understand how to actually implement the use cases, thus moving from a black box view to a white box view. One possible approach for this kind of domain engineering was introduced by Eric Evans in his book [3]. It is essential that architects, requirement engineers and domain experts create the model in cooperation to avoid ambiguities, misunderstandings, inconsistencies and incompleteness. Typically, at least an implicit model of the problem domain is already available that needs to be turned to an explicit and formal model. This model reveals commonalities but also variability which is particularly important for product line engineering. It is essential to mention that creating the domain model and the subsequent step (Model Dynamics of Scenarios) are typically not strongly separated but go partially hand in hand.

In the Show Case a Warehouse Management system had been developed by SIEMENS Industry Logistics. Figure 16.7 reveals a subset of the problem domain model that the software architects introduced.

16.2.6.5 Model Dynamics of Scenarios

With a (subset of the) problem domain model and prioritized and partitioned use cases in place, software architects can map the black-box use cases to white-box scenarios relying solely on problem domain concepts. This helps further refine the



Fig. 16.7 Example – a problem domain model for warehouse management

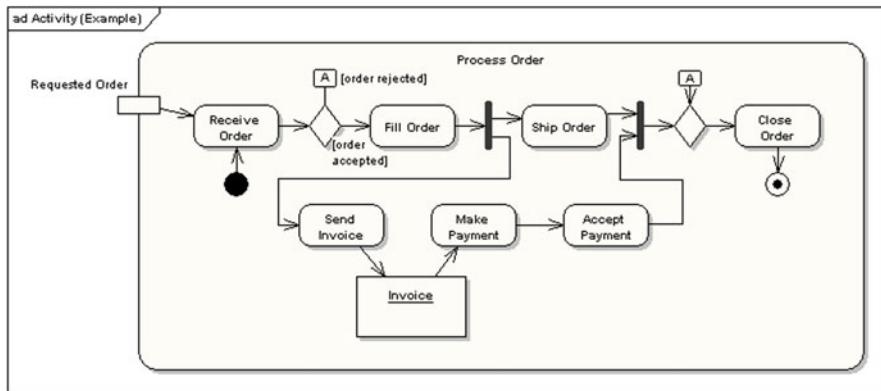


Fig. 16.8 Scenarios are white-box instantiations of black-box use cases

problem domain model by introducing relations and interactions between the various domain entities. Engineers should avoid introducing concepts of the solution domain too early, because otherwise both domains get intertwined without necessity. Which concrete diagram types designers use for this modeling step, is not important. For instance, they may choose UML activity diagrams or UML sequence diagrams depending on their concrete needs and experience.

Figure 16.8 represents an activity diagram which is used for modeling the internal interactions necessary to implement the use case “Process Order” in the Warehouse Management system.

16.2.6.6 Determine Scope & Boundaries

A fundamental issue in every software architecture design is embedding the system under construction into its environment. What are its primary and secondary actors and which interfaces does it provide or require? This information is relevant for system design, as it determines what is or isn’t the responsibility of the software engineers. For example, a Web Shop may implement its own order processing system or use an existing Enterprise Information System such as SAP R/3 instead. To define the system boundaries, two models are of particular value. The UML Use Case diagram, such as in Fig. 16.9, helps identify external actors interacting with the system, while a Context diagram might even define provided and required interfaces. In a product line engineering approach this step is even more demanding as it needs to determine all internal part of the product line platform (commonalities) as well as the type and parameters of product differences (variability).

16.2.6.7 Create Conceptual Draft

After the previous steps, software architects are able to define the architecture baseline. For this purpose, they follow the Onion model and the Pyramid model.

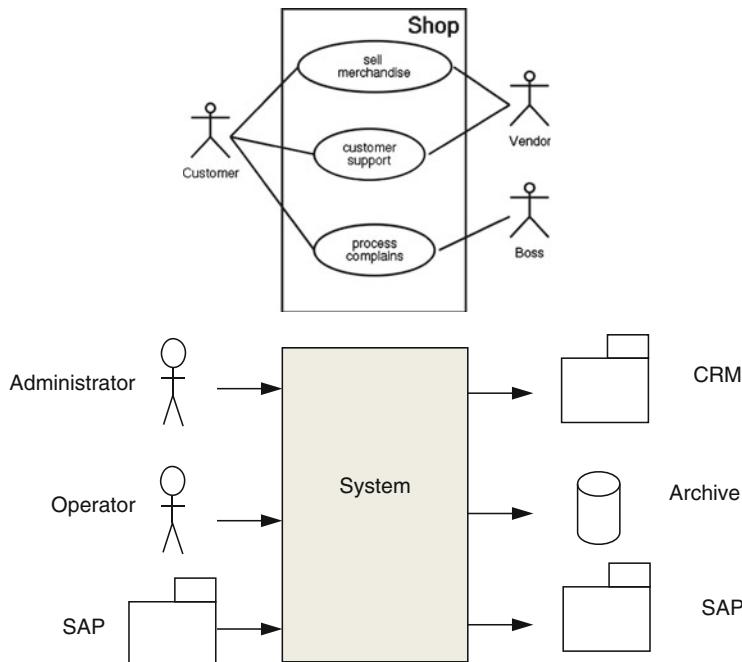


Fig. 16.9 Use case diagrams and context diagrams help identify the system's boundaries

The functional core is extended step by step with different layers representing infrastructural, operational or developmental requirements. Thus, concepts of the solution domain are introduced to the functional entities. At the end of this step, a conceptual high-level software architecture design is available.

It is up to the software architects whether and to what extent they take developmental qualities into account. One alternative strategy might be to cover mandatory and important developmental qualities in this step while postponing all other developmental qualities to the later step refining the architecture baseline through end-to-end slices. Another alternative could consist of addressing all developmental qualities in the later refinement phase. Which approach they choose, depends on the criticality, complexity and relevance of developmental qualities. For example, in a product line engineering organization variability management denotes an important issue for domain engineering and should already be addressed in this architecture creation step.

In Fig. 16.10 an original sketch of the conceptual architectural baseline of the Warehouse Management system was derived from the problem domain object model by applying the Onion model approach. The Pyramid model helped to constrain the architecture to three levels of architecture abstractions (system, subsystem, component). Using architecture and design patterns helped with providing the infrastructural, operational and developmental layers.

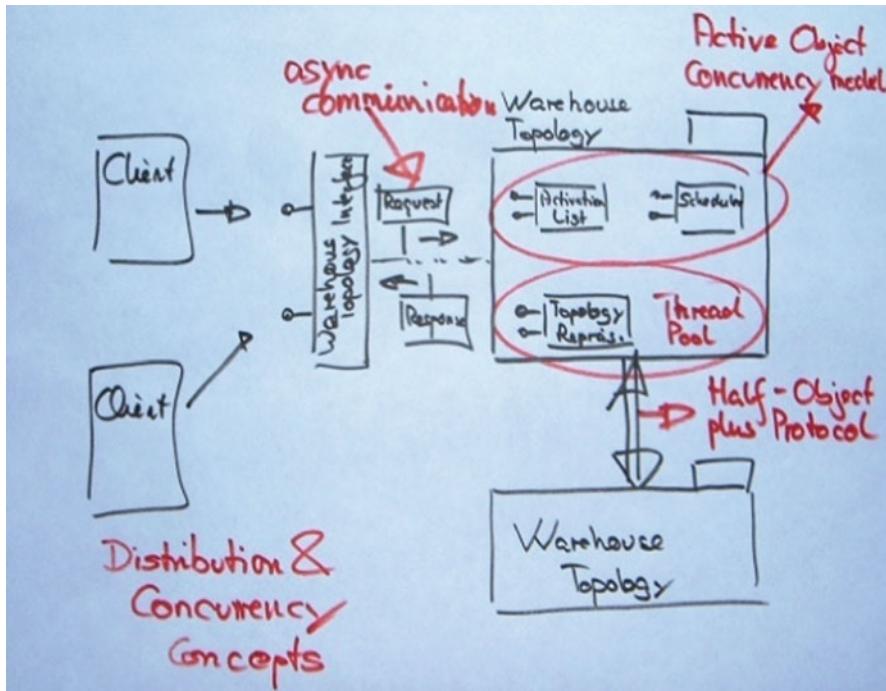


Fig. 16.10 With the onion model and the pyramid model, software architects can derive an initial architecture baseline

16.2.6.8 Structure the Baseline Architecture

If architecture design created an architecture baseline without considering internal quality aspects such as separation of concerns, the activity would result in a rather unstructured architecture baseline. Architecture patterns [4, 5] such as Pipes & Filters or Layers help structure the software architecture to better adapt to future maintenance and evolution. This activity can often be tightly intertwined with the previous activity (Create first Conceptual Draft).

Two SIEMENS examples: Fig. 16.11 demonstrates the three Tier architecture of the Warehouse Management system as enabled by the Layers architecture pattern.

In Fig. 16.12 the high level software architecture of an IP-based telecommunications system built by SIEMENS is visualized. The architects leveraged the Layers patterns to partition functionality into different layers of abstraction, granularity and change.

16.2.6.9 Introduce Deployment Views

As a further step to actual implementation, we need to address deployment aspects, for example by using UML deployment diagrams. Deployment comprises the

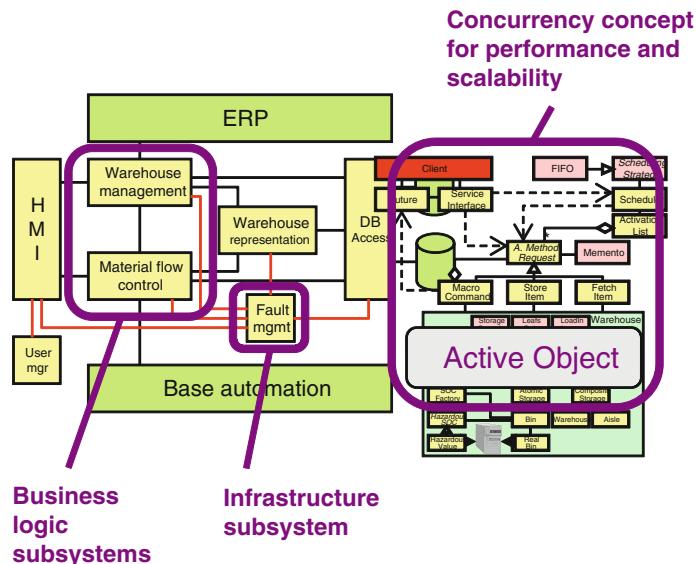


Fig. 16.11 In the warehouse management system the Layers pattern helps creating a three tier-architecture

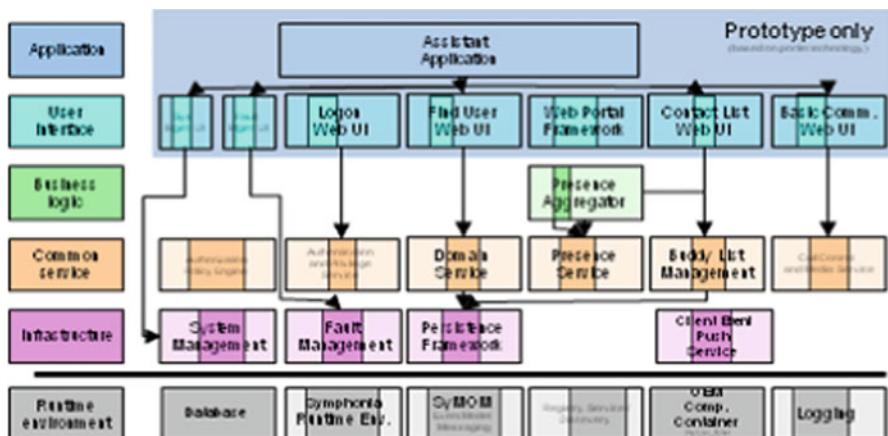


Fig. 16.12 Architecture patterns such as Layers help improve internal software quality and support future evolution and maintenance

mapping of the software architecture to the physical system architecture, mostly driven by operational and developmental aspects. The necessary responsibilities include assigning concrete artifacts such as executable files to computational nodes as well as describing the communication between physical elements.

A possible physical deployment for a Web-based Warehouse Management system is modeled using a deployment diagram (Fig. 16.13):

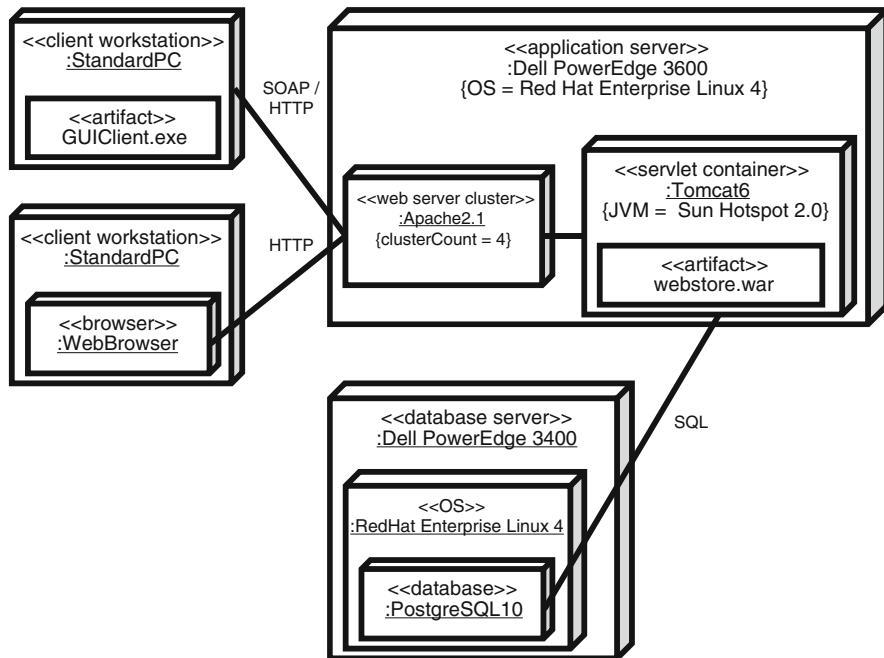


Fig. 16.13 Deployment denotes the mapping of software architecture artifacts to physical entities

16.2.6.10 Define Principles and Guidelines

After availability of the initial architecture baseline, software architects must take additional preparations for the later implementation steps. In order to employ some architecture governance, they are supposed to introduce principles and guidelines developers must adhere to. Otherwise, the refinement of the architecture baseline can lead to lack of internal architecture qualities, for instance to insufficient symmetry, but also to a failure meeting operational or developmental qualities. Typical examples are crosscutting concerns such as coding styles, exception handling or logging. When every developer introduces her own coding style, expressiveness and simplicity will inevitably suffer. Such principles and guidelines comprise (but are not limited) to the following areas:

- Coding Guidelines
- Application of specific software patterns
- Handling of cross-cutting concerns such as error handling, logging, auditing, session management, internationalization
- Design tactics at least for the most critical operational and developmental qualities
- Idioms for applying specific technologies and tools

Fig. 16.14 Software architects introduce guidelines and principles that govern the refinement process



For the most critical topics, software development projects might assign responsibilities to subsystem architects or whole teams who are in charge of providing and enforcing guidelines how developers are expected to handle particular concerns. Such guidelines do not necessarily resemble formal specifications; they might also include tutorials (Fig. 16.14).

16.2.6.11 Plan and Realize Increments

The last but most extensive phase consists of the iterative-incremental refinement of the architectural baseline. Again, this step is driven by the Onion model. In contrast to strategic design it is not limited by the Pyramid model but introduces whole end-to-end slices of the system, eventually resulting in implementation prototypes. The testing strategy gains significant importance for quality assurance as does the application of architecture analysis and the use of appropriate tools.

In Fig. 16.15 the iterative and incremental fine design of the SIEMENS Warehouse Management system is illustrated. The diagram shows an intermediate state of the system after a couple of iterations. For sake of brevity, only a part of the system is presented.

16.2.7 Good Practices

16.2.7.1 Why We Need Good Practices

To survive as software architects, it is important to carry around some kind of treasure chest filled with best practices no matter where we go. Obviously, there are lot of experiences and competences that are tightly connected to a specific problem

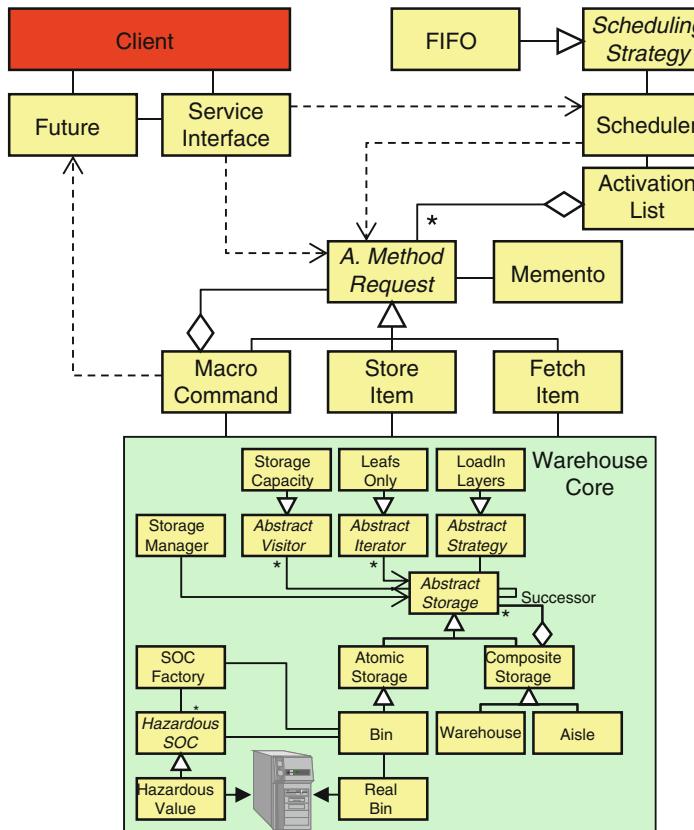


Fig. 16.15 During the last phase, software engineers create the implementation step by step

or solution domain and cannot be easily generalized. But there are also some rules of thumbs, patterns and practices we can leverage independently of our domains. This section provides some common hints and tips without striving for completeness. These guidelines are all taken from my experiences in various projects.

16.2.7.2 Re-use Perspective

The introduced architecture design process introduced works well for software development projects, no matter whether they start from scratch or can be based on existing artifacts. Sometimes, the organization has already gained some familiarity with the problem and solution domain. In this case, a reference architecture might be available that reflects knowledge and experience how to build a software system in the given domain(s). If software architects can leverage a reference architecture, their job is constrained to refining and instantiating the reference architecture with the project-specific forces, i.e., requirements, constraints, and business goals. In the ideal

case, the organization has already matured into a product line organization [6, 7] and can instantiate the whole software system by configuration and adaption of core assets. Even if such systematic re-use isn't established, a common practice is to re-use design or implementation artifacts to reduce costs and time. Software patterns such as design patterns and architecture patterns help introduce proven solutions for problems in specific contexts instead of reinventing the wheel.

16.2.7.3 Software Architecture Review and Refactoring

As already explained, creating a software system should not consist only of adding new features. In all iterations architects need to assess content and quality of the software architecture and get rid of all architectural deficiencies early. Otherwise, top-down design will continuously extend the architecture, making it increasingly difficult or even impossible and too costly to revise earlier decisions. This is where software architectures begin suffering from design erosion until they eventually have to be completely reengineered or rewritten. To identify architecture smells such as dependency cycles, unnecessary abstractions, insufficient coupling or cohesion, or inadequate design decisions architects should apply a method for architecture evaluation. Several methods for software architecture review have been proposed [8–10]. For regular evaluation an experience-based flash review or an ADR (Architecture Design Review) will suffice. Tools for CQM (Code Quality Management) or software architecture assessment show many benefits in this context, keeping designers up-to-date about the state of the software architecture. After software architects have identified deficiencies they should apply software architecture and code refactoring. However, this activity needs to obey the same principles as systematic software architecture design. For instance, all issues with respect to strategic requirements have to be resolved before the tactical issues are being tackled, and inadequate architecture decisions regarding high priority requirements should be addressed before those concerning requirements with low priority.

16.3 Outroduction

On one hand, creating software architecture is easy because every implementation has its implicit software architecture, even if software engineers did not explicitly define one. On the other hand, high quality software architecture is hard to achieve, because it requires software engineers to systematically create a software system that meets all requirements and business goals. Likewise, it is critical to gain high quality in software architecture, because all failures will lead to more costs and time needed for development and maintenance.

Software architecture is both a design artifact and a systematic process which serves as a fundamental vehicle to meet the requirements of customers and development organizations. As change is the rule and not the exception in software

engineering, software systems require modifications, for example to eliminate deficiencies or to adapt to new requirements or technologies. Evolution of software architecture needs to happen systematically, because otherwise design erosion will start to creep into the software system, first polluting some local areas, but finally leading to software architectures that must be completely reengineered or rewritten. Another major reason for design erosion is accidental complexity which occurs if software engineers make wrong decisions, do not understand the requirements, introduce design pearls not rooted in requirements, or implement code that doesn't abide to the software architecture or leading guidelines. A good countermeasure to prevent accidental complexity is based upon strictly following a requirements-driven, risk-driven and test-driven architecture development process applied jointly with architecture evaluation and refactoring activities. This process should support an iterative-incremental approach with piecemeal growth instead of emphasizing on a Big Bang design that simply can't work for fast moving targets like software development projects. With other words, if we can't avoid change we should embrace it. In contrast to common believe, software architecture design is a crosscutting activity of the overall system development. We can't just define a fixed boundary for design activities. Software architecture starts when the business goals are defined and ends when the software system is phased out. Requirements are the major drivers when creating a sustainable software system. To this end, requirements are – or at least should be – a software architect's best friends.

Each problem that I solved became a rule which served afterwards to solve other problems – René Descartes, 1596–1650, in “Discours de la Methode”

References

1. Cockburn A (2000) Writing effective use cases. Addison-Wesley, New York
2. Bass L, Clements P, Kazman R (2003) Software architecture in practice, 2nd edn. Addison-Wesley, Boston
3. Evans E (2003) Domain-driven design: tackling complexity in the heart of software. Addison-Wesley, New York
4. Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stal M (1996) Pattern-oriented software architecture, volume 1 – a system of patterns. Wiley, Chichester
5. Schmidt DC, Stal M, Rohnert H, Buschmann F (2000) Pattern-oriented software architecture, volume 2 – patterns for concurrent and networked objects. Wiley, Chichester
6. Weiss DM, Lai CTR (1999) Software product-line engineering: a family-based software development process. Addison-Wesley, New York
7. Petroski H (1992) To engineer is human – the role of failure in successful design, Reprint edn. Vintage, Chicago
8. Clemens P, Kazman R, Klein M (2002) Evaluating software architectures: methods and case studies. Addison Wesley, New York
9. Maranzano JF, Rozsypal SA, Zimmermann GH, Warnken GW, Wirth PA, Weiss DM (2005) Architecture reviews: practice and experience. IEEE Softw 22(2):34–43
10. Bosch J (2000) Design and use of software architectures – adapting and evolving a product-line approach. Addison-Wesley, New York

11. Clements P, Northrop L (2001) Software product lines: practices and patterns. Addison-Wesley, New York
12. Constantine L, Lockwood L (1999) Software for use: a practical guide to the models and methods of usage-centred design. Addison-Wesley, New York
13. Hohmann L (2003) Beyond software architecture – creating and sustaining winning solutions. Addison-Wesley, New York
14. Maier MW, Rechtin E (2000) The art of systems architecting. CRC Press, Boca Raton
15. Fowler M, Beck K, Brant J, Opdyke W, Roberts D (1999) Refactoring: improving the design of existing code. Addison-Wesley, New York
16. Beck K (1999) Extreme programming explained: embrace the change. Addison-Wesley, New York
17. Kruchten P (2000) The rational unified process, an introduction. Addison-Wesley, New York
18. Alexander C (1979) The timeless way of building. Oxford University Press, Oxford, UK
19. Gabriel RP (1996) Patterns of software. Oxford University Press, Oxford, UK
20. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns – elements of reusable object-oriented software. Addison-Wesley, New York
21. Pattern languages of program design (1995, 1996, 1997, 1999), vols 1–4. Addison-Wesley, New York
22. Rozanski N, Woods E (2008) Software systems architecture. Addison-Wesley, New York

Part IV

**Emerging Issues in Relating Software
Requirements and Architecture**

Chapter 17

Emerging Issues in Relating Software Requirements and Architecture

J. Grundy, P. Avgeriou, J. Hall, P. Lago, and I. Mistrik

Many emerging issues challenge us when attempting to relate software requirements and architectures. These include but are certainly not limited to:

- Do requirements inform or constrain architecture or vice-versa?
- Do new development processes and methodologies e.g. model-driven engineering, agile process, open sourcing, outsourcing impact on the relationship between requirements and architecture, and if so, how?
- Do new technologies and applications e.g. cloud computing, ubiquitous computing, social network applications and end-user computing impact on the relationship between requirements and architecture, and if so how?
- How can developers best identify, document, manage and utilize the myriad of relationships between requirements and architecture? What process and tool support is required to do this?

Many of the previous chapters have touched on, some in detail, the issue of whether we should develop a set of software requirements first and then an architecture for a system based on these requirements, or whether available architectural and technological platforms must be considered when actually eliciting, documenting and using requirements. Traditional wisdom has been that requirements for a system are captured and then a suitable architecture developed to fulfill these [2, 6]. As we have seen, many application domains, development approaches and technology solutions challenge this approach. For example, an understanding of architectural characteristics and technologies available for a problem domain are useful when discussing possible requirements with stakeholders e.g. what is actually feasible and what is not [3]. Similarly, many approaches have adopted round-trip engineering between requirements and architecting where enhancing or changing one necessarily impacts greatly the other. Many of the previous chapters indicate that the field is moving towards a view that requirements and architecture for a software system are necessarily closely related and therefore often need to be engineered, if not in parallel, then closely informing one another in an iterative manner. Some domains the notion of what the requirements are or what the available technology solutions to architect are ill-defined. Open source projects are an example where requirements are

often organic and emerge from user communities over time. Rapid technology change fields like consumer appliance engineering and cloud computing similarly have architectures and technologies rapidly changing during development. Some applications have requirements that are difficult to anticipate during development e.g. social networking applications and mash-ups where an application may be composed of parts and the “requirements” both dynamic and run-time emergent from stakeholders.

A number of new development processes and methods have emerged in recent years which make the relationship between requirements and architecture a challenge to elicit or maintain. Model-driven engineering and Genetic Algorithms (GA) have been applied to taking high-level models and architectural information and synthesizing code and configurations of systems [1, 10]. Could they be used to either generate architectures from suitable requirements models. In a related way, could requirements models be usefully extracted from existing architectural models? Agile processes have received much attention and they bring a very different focus to both capturing and using requirements and working with architectures [7]. In many agile methods architecture is emergent and requirements prioritization and alignment with architecture is highly volatile. Open sourcing, as noted above, is often characterized by highly distributed, heterogeneous teams and both requirements and architecture are often emergent. No one group of stakeholders owns either the requirements or the technology solutions used to realize them. Outsourcing typically requires well-defined architectures to work within along with well-defined requirements. How can more agile processes or domains with emergent requirements and/or architectures use outsourcing approaches? End user computing allows non-technical experts to modify their systems to configure them to their needs. By definition, requirements are highly volatile and emergent and even architectures can be vastly different depending on mash-up integrations.

A variety of new technologies and applications are impacting both requirements and architecture and their relationship. Cloud computing and self-architecting platforms offer a new way to architect systems but also impact on overall system and application requirements, particularly quality of service attributes non-functional requirements [8]. Similarly, ubiquitous computing systems are made up of a wide variety of components and dynamically evolving architectures. They have a range of users and are often context-dependent. This dramatically impacts on requirements and architecture volatility. In a similar way, social networking applications with mash-ups and user-selected applications, and mobile smart phones and iPods with dynamic application content similarly result in highly evolving technology platforms and application sets. These are, in contrast to traditional software systems, heavily user-driven in terms of evolving requirements and necessary architectural solutions, often very heterogeneous, embodied in user-selected and configured diverse applications.

Development process and tool support for requirements engineering and software architecting has become very sophisticated. A number of previous chapters in this book have highlighted this. However, support is still rather limited for managing the diversity and complexity and relationships between software requirements and

architectures [9]. For example, issues of non-functional constraints and their realization by appropriate architecture middleware choices [4] and the economics of evolving requirements on architectural decisions [5]. The above emerging issues will further compound these challenges. This final section of our book contains three very interesting chapters addressing different areas of these emerging issues of relating software requirements and architectures.

Chapter 18 by Outi Räihä, Hadaytullah, Kai Koskimies and Erkki Mäkinen looks at a new approach to generating candidate software architectures for a system based on a set of requirements models. One can think of this as a form of “model driven engineering” where the model is a set of requirements for a software system and the output a candidate architecture for it. A genetic algorithm is used to produce the architecture working from an initial simplistic model and refining the model using fitness functions. These evaluate candidates produced by the genetic algorithm including such measures as simplicity, efficiency and modifiability. Their approach then produces a proposal for a new software architecture for the specified target system. The candidate architecture makes use of various architectural styles and patterns to constrain the result. They study the quality of the candidate architectures by comparing the generated solutions to ones produced by undergraduate students. The results of these comparisons are very interesting!

Chapter 19 by Eoin Woods and Nick Rozanski describes how software architecture can be used to frame, constrain and indeed to inspire the requirements of a system. They observe that historically a system’s requirements and its architecture have been viewed as having a simple relationship. Typically this was assumed to be the requirements driving the architecture and the architecture was designed in order to meet the requirements. They report that in their experience a much more dynamic relationship exists and indeed must be achieved between these activities. They present a model that allows the architecture of a system to be used to constrain the requirements to an achievable set of possibilities, rather than be a “pipe dream” set of possibly unattainable feature. The architectural characteristics can be used to frame these requirements making their implications clearer. New requirements may even be inspired by consideration of a system’s architecture.

Chapter 20 by Rami Bahsoon and Wolfgang Emmerich addresses a very different aspect of requirements and architecture relationships, namely economics in the presence of complex middleware capabilities. They observe that most systems now adopt some form of middleware technology platform in order to more easily and reliably achieve many non-functional requirements for a software system, such as scalability, openness, heterogeneity, availability, reliability and fault-tolerance. An issue is how to evolve non-functional requirements while being able to analyse the impact of these changes on software architectures induced by middleware. Specifically they look at economics and stability implications and the role of middleware in architecture in achieving non-functional requirements in a software architecture. They propose a technique to elicit not only current non-functional requirements for a system but likely evolution of these over the lifetime of the system and economic impact of these. This then allows choice of appropriate middleware solutions that will both achieve non-functional requirements when used to realise an

architecture but balance evolutionary costs. An economics-driven approach using real options theory is used to inform the selection on middleware to induce software architectures in relation to the evolving non-functional requirements.

These three chapters include case studies of the techniques and empirical evaluations of the approaches. As you will see, results are both impressive in terms of the nature of the emerging issues addressed by these three efforts.

References

1. Amoui M, Mirarab S, Ansari S, Lucas C (2006) A GA approach to design evolution using design pattern transformation. *Int J Inf Technol Intell Comput* 1:235–245
2. Bass L, Clements P, Kazman R (2003) Software architecture in practice, 2nd edn. Prentice Hall, Reading
3. Cheng B, de Lemos R, Giese H, Inverardi P, Magee J (2009) Software engineering for self-adaptive systems: a research roadmap. In: Cheng BHC, Lemos R de, Giese H, Inverardi P, Magee J (eds) Software engineering for self-adaptive systems. Lecture notes in computer science, vol 5525. Springer, Amsterdam
4. Emmerich W (2000) Software engineering and middleware: a road map. In: Finkelstein A (ed) Future of software engineering. ACM Press, New York
5. Erdoganmus H, Boehm B, Harriossn W, Reifer DJ, Sullivan KJ (2002) Software engineering economics: background, current practices, and future directions. In: Proceeding of 24th international conference on software engineering. Orlando
6. Hull E, Jackson K, Dick J (2010) Requirements engineering. Springer Verlag, Berlin
7. Larman C (2004) Agile and iterative development: a manager's guide. Addison-Wesley, Reading
8. Menascé DA, Sousa JP, Malek S, Gomaa H (2010) QoS architectural patterns for self-architecting software systems. In: Proceedings of the 7th international conference on autonomic computing and communications. ACM Press, Washington DC, USA, p 195
9. Nuseibeh B(2001) Weaving the software development process between requirements and architectures. In: Proceedings of STRAW 01 the first international workshop from software, Toronto
10. Selonen P, Koskimies K, Systä T (2001) Generating structured implementation schemes from UML sequence diagrams. In: QiaYun L, Riehle R, Pour G, Meyer B (eds) Proceedings of TOOLS 2001. IEEE CS Press, California, USA, p 317

Chapter 18

Synthesizing Architecture from Requirements: A Genetic Approach

Outi Räihä, Hadaytullah Kundi, Kai Koskimies, and Erkki Mäkinen

Abstract The generation of software architecture using genetic algorithms is studied with architectural styles and patterns as mutations. The main input for the genetic algorithm is a rudimentary architecture representing the functional decomposition of the system, obtained as a refinement of use cases. Using a fitness function tuned for desired weights of simplicity, efficiency and modifiability, the technique produces a proposal for the software architecture of the target system, with applications of architectural styles and patterns. The quality of the produced architectures is studied empirically by comparing these architectures with the ones produced by undergraduate students.

18.1 Introduction

A fundamental question of computing is: “What can be automated?” [1]. Is software architecture design inherently a human activity, sensitive to all human weaknesses, or could it be automated to a certain degree? Given functional and quality requirements for a particular system, could it be possible to generate a reasonable software architecture design for the system automatically, thus avoiding human pitfalls (like the Golden Hammer syndrome [2])? Besides being interesting from the viewpoint of understanding the limits of computing and the character of software architecture, we see answers to these questions relevant from a pragmatic viewpoint as well. In particular, if it turns out that systems can successfully design systems, various kinds of software generators can optimize the architecture according to the application requirements, self-sustaining systems [3] can dynamically improve their own architecture in changing environments, and architects can be supported by automated design tools.

A possible approach to automate software architecture design is to mechanize the human process of architecture design into a tool that selects or proposes architectural solutions using similar rules as a human would. A good example of this approach is ArchE, a semi-automated assistant for architecture design [4]: the design knowledge

is codified as a reasoning framework that is applied to direct the design process. In this approach, the usefulness of the resulting architecture largely depends on the intelligence and codified knowledge of the tool. This is a potential weakness as far as automation is concerned: it is hard to capture sufficient design knowledge in a reasoning framework, which decreases the automation level.

An alternative approach is not to mechanize the process and rules of architecture design, but simply give certain criteria for the “goodness” of an architecture, and let the tool try to come up with an architecture proposal that is as good as possible, using whatever technique. This approach requires no understanding of the design process, but on the other hand it requires a characterization of a “good” architecture. If suitable metrics can be developed for different quality attributes of software architectures, this approach is more light-weight than the former. In particular, this approach is more amenable for an automated design process, as it can be presented essentially as a search problem. We will briefly review some existing work related to this approach in Sect. 18.3.

In this chapter, we will follow the latter approach. More precisely, we will make the following assumptions to simplify the research setup. First, we assume that the architecture synthesizer can rely on a “null architecture” that gives the basic decomposition of the functionalities into components, but pays no attention to the quality requirements. We will later show how the null architecture is derived from use cases. Second, we assume that the architecture is obtained by augmenting the null architecture with applications of general architectural solutions. Such solutions are typically architectural styles and design patterns [5]. Third, we assume that the goodness of an architecture can be inferred by evaluating a representation of the architecture mechanically against the quality requirements of the system. Each application of a general solution enhances certain quality attributes of the system, at the expense of others.

With these assumptions, software architecture design becomes essentially a search problem: find a combination of applications of the general solutions that satisfies the quality requirements in an optimal way. However, given multiple quality attributes and a large number of general solutions, the search space becomes huge for a system with realistic size. This leads us to the more refined research problem discussed in this chapter: to what extent could we use meta-heuristic search methods, like genetic algorithms (GA) [6, 7], to produce a reasonable software architecture automatically for certain functional and quality requirements?

The third assumption above is perhaps the most controversial. Since there is no exact definition of a good software architecture, and different persons would probably in many cases disagree on what is a good architecture, this assumption means that we can only approximate the evaluation of the goodness. Obviously, the success of a search method depends on how well we can capture the intuitive architecture quality in a formula that can be mechanically evaluated.

In this chapter, we consider three quality attributes, modifiability, efficiency, and simplicity; these correspond roughly to the ISO9126 quality factors changeability, time behavior and understandability [8], respectively. We base our evaluation of all these factors on existing software metrics [9], but extend them for modifiability and

efficiency by exploiting knowledge about the effect of the solutions on these two quality factors. Optional information given by the designer about certain functionalities is also taken into account. In addition, the designer can give more precise modifiability requirements as change scenarios [10], taken into account in the evaluation of modifiability as well.

Although a number of heuristic search methods could be used here [11], we are particularly interested in GA for two main reasons. First, the structural solutions visible in the living species in nature provide an indisputable evidence of the power of evolution in finding competitive system architectures. Second, crossover can be naturally interpreted for software architecture, as long as certain consistency rules are followed. Crossover can be viewed as a situation where two architects provide alternate designs for a system, and decide to merge their solutions, (hopefully) taking the best parts of both designs.

This chapter proceeds as follows. Background information on genetic algorithms and the proposed evolutionary software architecture generation process are discussed in Sect. 18.2. Existing work related to search-based approaches to software architecture design is briefly reviewed in Sect. 18.3. The GA realization in our approach is discussed in Sect. 18.4, concretized with an example system. An application of the technique for the example system is presented in Sect. 18.5, and an empirical experiment evaluating the quality of the genetically produced software architecture is discussed in Sect. 18.6. Finally, we conclude with some remarks about the implications of the results and future directions of our work.

18.2 Background

18.2.1 *Genetic Algorithms*

Meta-heuristics [12] are commonly used for combinatorial optimization, where the search space can become especially large. Many practically important problems are NP-hard, making exact algorithms not feasible. Heuristic search algorithms handle an optimization problem as a task of finding a “good enough” solution among all possible solutions to a given problem, while meta-heuristic algorithms are able to solve even the general class of problems behind the certain problem. A search will optimally end in a global optimum in a search space, but at the very least it will give some local optimum, i.e., a solution that is “better” than alternative solutions nearby. A solution given by a heuristic search algorithm can be taken as a starting point for further searches or be taken as the final solution, if its quality is considered high enough.

We have used genetic algorithms, which were invented by John Holland in the 1960s. Holland’s original goal was not to design application specific algorithms, but rather to formally study the ways of evolution and adaptation in nature and develop ways to import them into computer science. Holland [6] presents the genetic

algorithm as an abstraction of biological evolution and gives the theoretical framework for adaptation under the genetic algorithm.

In order to explain genetic algorithms, some biological terminology needs to be clarified. All living organisms consist of *cells*, and every cell contains a set of *chromosomes*, which are strings of DNA and give the basic information of the particular organism. A chromosome can be further divided into *genes*, which in turn are functional blocks of DNA, each gene representing some particular property of the organism. Each gene is located at a particular *locus* of the chromosome. When reproducing, *crossover* occurs and genes are exchanged between the pair of parent chromosomes. The offspring is subject to *mutation* where single bits of DNA are changed. The *fitness* of an organism implies the probability that the organism will live to reproduce and carry on to the next generation [7]. The set of chromosomes at hand at a given time is called a *population*.

During the evolution, the population needs to change to fit better the requirements of the environment. The changing is enabled by mutations and crossover between different chromosomes (i.e., individuals), and, due to natural selection, the fittest survive and are able to participate in creating the next generation.

Genetic algorithms are a way of using the ideas of evolution in computer science to find a good solution from a very large search space, the goal obviously being that the found solution is as good as possible. To operate with a genetic algorithm, one needs an encoding of the solution, i.e., a representation of the solution in a form that can be interpreted as a chromosome, an initial population, mutation and crossover operators, a fitness function (to determine the “goodness” of a solution) and a selection operator for choosing the survivors for the next generation.

In addition, there are also many parameters regarding the GA that need to be defined and greatly affect the outcome. These parameters are the population size, number of generations (often used as the terminating condition) and the mutation and crossover probabilities. Having a large enough population ensures variability within a generation, and enables a wide selection of different solutions at every stage of evolution. However, a larger population always means more fitness evaluations and thus requires more computation time. Similarly, the more generations the algorithm is allowed to run, the higher the chances are that it will be able to reach the global optimum. However, again, letting an algorithm run for, say, 10,000, generations will most probably not be beneficial: if the operations and parameters have been chosen correctly, a reasonably good solution should have been found much earlier.

Mutation and crossover probabilities both affect the speed of evolution. If the probabilities are too high, there is the risk that the application of genetic operations becomes random instead of guided. Vice versa, if the probabilities are too low there is the risk that the population will evolve too slowly, and no real diversity will exist. An assumption to be noted with genetic operators is the building block hypothesis, which states that a genetic algorithm combines a set of sub-solutions, or building blocks, to obtain the final solution. The sub-solutions that are kept over the generations usually have an above-average fitness [13]. The crossover operator is

especially sensitive to this hypothesis, as an optimal crossover would thus combine two rather large building blocks in order to produce an offspring.

18.2.2 Overview of Evolutionary Software Architecture Generation

Software architecture can be understood in different ways. The definitions of software architecture usually cover the high-level structure of the system, but in addition to that, often also more process-related aspects like design principles and rationale of design decisions are included [14, 15]. To facilitate our research, we adopt a narrow view of software architecture, considering only the static structural aspect, expressible as a UML (stereotyped) class diagram. In terms of the 4 + 1 views of software systems [16], this corresponds to a (partial) logical view. While a similar approach could be applied to generate other views of software architectures as well, there are some fundamental limitations in using heuristic methods. For example, it is very difficult to produce the rationale for the design decisions proposed by a heuristic method.

A central issue in our approach is the representation of the functional and quality requirements of the system, to be given as input for the genetic synthesis of the architecture. For expressing functional requirements we need to identify and express the primary use cases of the system, and refine them into sequence diagrams depicting the interaction between major components required to accomplish the use cases. This is a manual task, as the major components have to be decided, typically based on domain analysis.

In our approach, a so-called null architecture represents a basic functional decomposition of the system, given as a UML class diagram. No quality requirements are yet taken into account in the null architecture, although it does fulfill the functional requirements. The null architecture can be systematically derived from the use case sequence diagrams: the (classes of the) participants in the sequence diagram become the classes, the operations of a class are the incoming call messages of the participants of that class, and the dependency relationships between the classes are inferred from the call relationships of the participants. This kind of generation of a class diagram can be automated [17], but in the experiments discussed here we have done this manually.

Depending on the quality attributes considered, various kinds of information may need to be associated with the operations of the null architecture. In our study we consider three quality attributes: simplicity, modifiability, and efficiency. Simplicity is an operation-neutral property in the sense that the characteristics of the operations have no effect on the evaluation of simplicity. In contrast, modifiability and efficiency are partially operation-sensitive. For evaluating the modifiability of a system, it is useful to know which operations are more likely to be affected by changes than others. Similarly, for evaluating efficiency it is often useful to know

something about the frequency and resource consumption of the operations. For example, if an operation that is frequently needed is activated via a message dispatcher, there is a performance cost because of the increased message traffic. To allow the evaluation of modifiability and efficiency, the operations can be annotated with this kind of optional information. If this information is insufficient, the method may produce less satisfactory results than with the additional information. However, no actual “hints” on how the GA should proceed in the design process are given. The null architecture gives a skeleton for the system and does not give any finer details regarding the architectures. The information regarding the operations merely helps in evaluating the solutions but influences in no direct way the choices of the GA.

The specific quality requirements of a system are represented in two ways. First, the fitness function used in the GA is basically a weighted sum of the values of individual quality attributes. By changing the weights the user can emphasize or downplay some quality attributes, or remove completely certain quality attributes as requirements. Second, the user can optionally provide more specific quality requirements using so-called scenarios. The scenario concept is inspired by the ATAM architecture evaluation method [10], where scenarios are imaginary situations or sequences of events serving as test cases for the fulfilling of a certain quality requirement. In principle, scenarios could be used for any quality attribute, but their formalization is a major research issue outside the scope of this work. Here we have used only modifiability scenarios, which are fairly easy to formalize. For example, in our case a scenario could be: “With 50% probability operation T needs to be realized in different versions that can be changed dynamically.” This is expressed for the GA tool using a simple formal convention covering most usual types of change scenario contents.

Figure 18.1 depicts the overall synthesis process. The functional requirements are expressed as use cases, which are refined into sequence diagrams. This is done manually by exploiting knowledge of the major logical domain entities having functional responsibilities. The null architecture, a class diagram, is derived mechanically from the sequence diagrams. The quality requirements are encoded for the GA as a fitness function, which is used to evaluate the produced architectures. Weights can be given as parameters to emphasize certain quality attributes, and scenarios can be used for more specific quality (modifiability) requirements. When the evolution begins, the null architecture is used by the GA to first create an initial population of architectures and then, after generations of evolution, the final architecture proposal is presented as the best individual of the last generation. New generations are produced by applying a fixed library of standard architectural solutions (styles, patterns, etc.) as mutations, and crossover operations to combine architectures. The probabilities of mutations and crossover can be given as parameters as well. The GA part is discussed in more detail in Sect. 18.4.

The influence of human input is present in defining the use cases which lead to the null architecture and in giving the parameters for the GA. The use cases must be defined manually, as they depict the functional requirements of a system: automatically deciding what a system is needed for is not sensible. Giving the parameters for

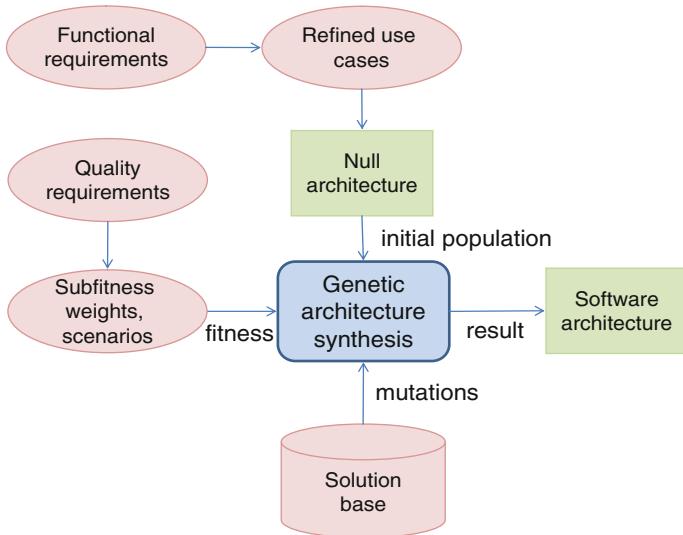


Fig. 18.1 Evolutionary architecture generation

the GA, in turn, is necessary for the algorithm to operate. It is possible to leave everything for the algorithm, and give each mutation the same probability and each part of the fitness function the same weight. In this case, the GA will not favor any design choice or quality aspect over another. If, however, the human architect has a vision that certain design solutions would be more beneficial for a certain type of system or feels that one quality aspect is more important than some other, it is possible to take these into account when defining the parameters.

Thus, the human restricts the GA in terms of defining the system functionality and guides the GA in terms of defining parameters. Additionally, the GA is restricted by the solution base. The human can influence the solution base by “removing solutions,” that is, by giving them probability 0, and thus making it impossible for the GA to use them. But in any case the GA cannot move beyond the solution base: if a pattern is not defined in the solution base, it cannot be used, and thus the design choices are limited to those that can be achieved as a combination of the specified solutions. Currently the patterns must be added to the solution base by manual coding.

18.3 Related Work

Search-based software engineering applies meta-heuristic search techniques to software engineering issues that can be modeled as optimization problems. A comprehensive survey of applications in search-based software engineering has been made by Harman et al. [18]. Recently, there has been increasing interest in

software design in the field of search-based software engineering. A survey on this subfield has been conducted by Räihä [19]. In the following, we briefly discuss the most prominent studies in the field of search-based software design.

Bowman et al. [20] study the use of a multi-objective genetic algorithm (MOGA) in solving the class responsibility assignment problem. The objective is to optimize the class structure of a system through the placement of methods and attributes within given constraints. So far they do not demonstrate assigning methods and attributes “from scratch” (based on, e.g., use cases), but try to find out whether the presented MOGA can fix the structure if it has been modified.

Simons and Parmee [21, 22] take use cases as the starting point for system specification. Data is assigned to attributes and actions to methods, and a set of uses is defined between the two sets. The notion of class is used to group methods and attributes. This approach starts with pure requirements and leaves all designing to the genetic algorithm. The genetic algorithm works by changing the allocation of attributes and methods.

Our work differs from those of Bowman et al. [20] and Simons and Parmee [21, 22] by operating on a higher level. The aforementioned studies concentrate only on class-level structure, and single methods and attributes. Bowman et al. [20] also do not present a method for straightforward design, but are only at the level where the algorithm can correct a set of errors introduced for testing purposes. Simons and Parmee [21, 22] do start from roughly the same level as we do (requirements derived from use cases), but they consider only the assignment of methods and attributes to classes.

Amoui et al. [23] use the GA approach to improve the reusability of software by applying architecture design patterns to a UML model. The authors’ goal is to find the best sequence of transformations, i.e., pattern implementations. Used patterns come from the collection presented by Gamma et al. [24]. From the software design perspective, the transformed design of the best chromosomes are evolved so that abstract packages become more abstract and concrete packages in turn become more concrete. When compared to our work, this approach only uses one quality factor (reusability) instead of several contradicting quality attributes. Further, the starting point in this approach is an existing architecture that is more elaborated than our null architecture.

Seng et al. [25] describe a methodology that computes a subsystem decomposition that can be used as a basis for maintenance tasks by optimizing metrics and heuristics of good subsystem design. GA is used for automatic decomposition. If a desired architecture is given, and there are several violations, this approach attempts to determine another decomposition that complies with the given architecture by moving classes around. Seng et al. [26] have continued their work by searching for a list of refactorings, which deal with the placement of methods and attributes and inheritance hierarchy.

O’Keeffe and Ó Cinnéide [27] have developed a tool for improving a design with respect to a conflicting set of goals. The tool restructures a class hierarchy and moves methods within it in order to minimize method rejection, eliminate code duplication and ensure superclasses are abstract when appropriate. Contrary to most

other approaches, this tool uses simulated annealing. O’Keeffe and Ó Cinnéide [28, 29] have continued their research by constructing a tool for refactoring object-oriented programs to conform more closely to a given design quality model. This tool can be configured to operate using various subsets of its available automated refactorings, various search techniques, and various evaluation functions based on combinations of established metrics.

Seng et al. [25, 26] and O’Keeffe and Ó Cinnéide [27–29] make more substantial design modifications than, e.g., Simons and Parmee [21, 22], and are thus closer to our level of abstraction, but they work clearly from the re-engineering point of view, as a well designed architecture is needed as a starting point. Also, modifications to class hierarchies and structures are still at a lower abstraction level than the design patterns and styles we use, as we need to consider larger parts of the system (or even the whole system). The metrics used by Seng et al. [25, 26] and O’Keeffe and Ó Cinnéide are also simpler, as they directly calculate, e.g., the number of methods per class or the levels of abstraction.

Mancoridis et al. [30] have created the Bunch tool for automatic modularization. Bunch uses hill climbing and GA to aid its clustering algorithms. A hierarchical view of the system organization is created based on the components and relationships that exist in the source code. The system modules and the module-level relationships are represented as a module dependency graph (MDG). The goal of the software modularization process is to automatically partition the components of a system into clusters (subsystems) so that the resultant organization concurrently minimizes inter-connectivity while maximizing intra-connectivity.

Di Penta et al. [31] build on these results and present a software renovation framework (SRF) which covers several aspects of software renovation, such as removing unused objects and code clones, and refactoring existing libraries into smaller ones. Refactoring has been implemented in the SRF using a hybrid approach based on hierarchical clustering, GAs and hill climbing, and it also takes into account the developer’s feedback. Most of the SRF activities deal with analyzing dependencies among software artifacts, which can be represented with a dependency graph.

The studies by Mancoridis et al. [30] and Di Penta et al. [31] again differ from ours on the direction of design, as they concentrate on re-engineering, and do not aim to produce an architecture from requirements. Also they operate on different design levels: clustering in the case of Mancoridis et al. [30] is on a higher abstraction level, while, e.g., removing code clones in the case of Di Penta et al.’s [31] study is on a much more detailed level than our work.

In the self-adaptation approach presented by Menascé et al. [32], an existing SOA based system is adapted to a changing environment by inserting fault-tolerance and load balancing patterns into the architecture at run time. The new adapted architecture is found by a hill climbing algorithm. This work is close to ours in the use of architecture-level patterns and heuristic search, but this approach – as other self-adaptation approaches – use specific run-time information as the basis of architectural transformations, whereas we aim at synthesizing the architecture based on requirements.

To summarize, most of the approaches discussed above are different from ours in terms of the level of detail and overall aim: we are especially interested to shape the overall architecture genetically, while the works discussed above consider the problem of improving an existing architecture in terms of fairly fine-grained mechanisms.

18.4 Realizing Genetic Algorithms for Software Architecture Generation

18.4.1 Representing Architecture

The genetic algorithm makes use of two kinds of information regarding each operation appearing in the null architecture. First, the basic input contains the call relationships of the operations taken from the sequence diagrams, as well as other attributes like estimated parameter size, frequency and variability sensitiveness, and the null architecture class it is initially placed in. Second, the information gives the position of the operation with respect to other structures: the interface it implements and the design patterns [24] and styles [33] it is a part of. The latter data is produced by the genetic algorithm.

We will discuss the patterns used in this work in Sect. 18.4.2. The message dispatcher architecture style is encoded by recording the message dispatcher the operation uses and the responsibilities it communicates with through the dispatcher. Other patterns are encoded as instances that contain all relevant information regarding the pattern: operations involved, classes and interfaces involved, and whether additional classes are needed for the pattern (as in the case of Façade, Mediator and Adapter). All this data regarding an operation is encoded as a supergene. An example of a supergene representing one operation is given in Fig. 18.2.

The chromosome handled by the genetic algorithm is gained by collecting the supergenes, i.e., all data regarding all operations, thus representing a whole view of the architecture. The null architecture is automatically encoded into the chromosome format on the basis of the sequence diagrams. An example of a chromosome is presented in Fig. 18.3. A more detailed specification of the architecture representation is given by Räihä et al. [34, 35].

calls	name	type	frequency	parameter size	variation	class	interface	dispatcher	dispatcher communications	component class	pattern
-------	------	------	-----------	----------------	-----------	-------	-----------	------------	---------------------------	-----------------	---------

Fig. 18.2 A supergene for operation

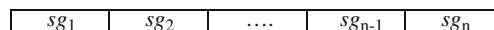


Fig. 18.3 Chromosome for a system with n operations (and n supergenes)

The initial population is generated by first encoding the null architecture into the chromosome form and creating the desired number of individuals. A random pattern is then inserted into each individual (in a randomly selected place). In addition, a special individual is left in the population where no pattern is initially inserted; this ensures versatility in the population.

18.4.2 Mutations and Crossover

As discussed above, the actual design is made by adding patterns to the architecture. The patterns have been chosen so that there are very high-level architectural styles (message dispatcher and client-server), medium-level design patterns (Façade and Mediator), and low-level design patterns (Strategy, Adapter and Template Method). The particular patterns were chosen also because they mostly deal with structure and need very little or no information of the semantics of the operations involved. The mutations are implemented in pairs of introducing a specific pattern or removing it. The dispatcher architecture style makes a small exception to this rule: the actual dispatcher must first be introduced to the system, after which the components can communicate through it.

Preconditions are used to check that a pattern is applicable. If, for example, the “add Strategy”-mutation is chosen for operation o_i , it is checked that o_i is called by some other operation in the same class c and that it is not a part of another pattern already (pattern field is empty). Then, a Strategy pattern instance sp_i is created. It contains information of the new class(es) sc_i where the different version(s) of the operation are placed, and the common interface si_i they implement. It also contains information of all the classes and operations that are dependent on o_i , and thus use the Strategy interface. Then, the value in the class field in the supergene sg_i (representing o_i) would be changed from c to sc_i , the interface field would be given value si_i and the pattern field the value sp_i . Adding other patterns is done similarly. Removing a pattern is done in reverse: the operation placed in a “pattern class” would be returned to its original null architecture class, and the pattern found in the supergene’s pattern field would be deleted, as well as any classes and interfaces related to it.

The crossover is implemented as a traditional one-point crossover. That is, given chromosomes ch_1 and ch_2 that are selected for breeding, a crossover point p is first chosen at random, so that $0 < p < n$, if the system has n operations. The supergenes $sg_1 \dots sg_p$ from chromosome ch_1 and supergenes $sg_{p+1} \dots sg_n$ from ch_2 will form one child, and supergenes $sg_1 \dots sg_p$ from chromosome ch_2 and supergenes $sg_{p+1} \dots sg_n$ from ch_1 another child.

A corrective function is added to ensure that the architectures stay coherent, as patterns may be broken by overlapping mutations. In addition to ensuring that the patterns present in the system stay coherent and “legal,” the corrective function also checks that no anomalies are brought to the design, such as interfaces without any users.

Also the mutation points are selected randomly. However, we have taken advantage of the variability property of operations with the Strategy, Adapter and dispatcher communication mutations. The chances of a gene being subjected to these mutations increase with respect to the variability value of the corresponding operation. This should favor highly variable operations.

The actual mutation probabilities are given as input. Selecting the mutation is made with a “roulette wheel” selection [36], where the size of each slice of the wheel is in proportion to the given probability of the respective mutation. Null mutation and crossover are also included in the wheel. The crossover probability increases linearly in relation to the fitness rank of an individual, which causes the probabilities of mutations to decrease in order to fit the larger crossover slice to the wheel. Also, after crossover, the parents are kept in the population for selection. These actions favor strong individuals to be kept intact through generations. Each individual has a chance of reproducing in each generation: if the first roulette selection lands on a mutation, another selection is performed after the mutation has been administered. If the second selection lands on the crossover slice, the individual may produce offspring. In any other case, the second selection is not taken into account, i.e., the individual is not mutated twice.

18.4.3 Fitness Function

The fitness function needs to produce a numerical value, and is thus composed of software metrics [37, 38]. The metrics introduced by Chidamber and Kemerer [9] have especially been used as a starting point for the fitness function, and have been further developed and grouped to achieve clear “sub-functions” for modifiability and efficiency, both of which are measured with a set of positive and negative metrics. The most significant modifications to the basic metrics include taking into account the positive effect of interfaces and the dispatcher and client-server architecture styles in terms of modifiability, as well as the negative effect of the dispatcher and server in terms of efficiency. A simplicity metric is added to penalize having many classes and interfaces.

Dividing the fitness function into sub-functions gives the possibility to emphasize certain quality attributes and downplay others by assigning different weights for different sub-functions. These weights are set by the human user in order to guide the GA in case one quality aspect is considered more favorable than some other. Denoting the weight for the respective sub-function sf_i with w_i , the core fitness function $f_c(x)$ for architecture x can be expressed as

$$f_c(x) = w_1 * sf_1 - w_2 * sf_2 + w_3 * sf_3 - w_4 * sf_4 - w_5 * sf_5.$$

Here, sf_1 measures positive modifiability, sf_2 negative modifiability, sf_3 positive efficiency, sf_4 negative efficiency and sf_5 complexity. The sub-fitness functions are defined as follows ($|X|$ denotes the cardinality of X):

$$\begin{aligned}
 sf_1 &= |\text{interface implementers}| + |\text{calls to interfaces}| + |\text{calls to server}| + |\text{calls through dispatcher}| * \prod (\text{variabilities of operations called through dispatcher}) \\
 &\quad - |\text{unused operations in interfaces}| * \alpha, \\
 sf_2 &= |\text{indirect calls between operations in different classes}|, \\
 sf_3 &= \sum (\text{operations dependent of each other within same class}) * \text{parameterSize} \\
 &\quad + \sum (\text{usedOperations in same class}) * \text{parameterSize} + |\text{dependingOperations in same class}| * \text{parameterSize}, \\
 sf_4 &= \sum \text{ClassInstabilities} [23] + (|\text{dispatcherCalls}| + |\text{serverCalls}|) * \sum \text{frequencies}, \\
 sf_5 &= |\text{classes}| + |\text{interfaces}|.
 \end{aligned}$$

The multiplier α in sf_1 emphasizes that having unused responsibilities in an interface should be more heavily penalized. In sf_3 , “usedOperations in same class” means the set of operations $o_i \dots o_l$ in class C , which are all used by the same operation o_m from class D . Similarly, “dependingOperations in same class” means the set of operations $o_b \dots o_h$ in class K , which all use the same operation o_a in class L .

It should be emphasized that all these sub-functions calculate a numerical fitness value for the entire system, and do not reward or penalize any specific patterns (apart from dispatcher connections). This fitness value is the basis of the evaluation, and weights are simply used to guide the algorithm, if needed. Each weight can be set to 1, in which case all sub-fitnesses are considered equally important, and the fitness value is the raw numerical value produced by the fitness calculations. All sub-fitnesses are normalized so that their values are in the same range.

Additionally, scenarios can be used for more detailed fitness calculations. Basically, a scenario describes an interaction between a stakeholder and the system [39]. In our approach we have concentrated only on change scenarios. We have categorized each scenario in three ways: is the system changed or is something added; if changed, does the change concern semantics or implementation of the operation, and whether the modification should be done dynamically or statically. This categorization is the basis for encoding the scenarios. In addition, each encoding of a scenario contains information of the operation it affects, and the probability of the scenario occurrence. Räihä et al. [40] explain the scenario encoding in more detail.

Each scenario type is given a list of preferences according to the general guidelines of what is a preferable way to deal with that particular type of modification. These preferences are general, and do not in any way consider the specific needs or properties of the given system.

When scenarios are encoded, the algorithm processes the list of given scenarios, and compares the solution for each scenario to the list of preferences. Each solution is then awarded points according to how well it supports the scenarios, i.e., how high the partial solutions regarding individual operations are on the preference list.

Formally, the scenario sub-fitness function sf_s can be expressed as

$$sf_s = \sum \text{scenarioProbability} * 100 / \text{scenarioPreference}.$$

Adding the scenario sub-fitness function to the core fitness function results in the overall fitness, $f(x) = f_c(x) + w_s * sf_s$.

18.5 Application

18.5.1 Creating Input

As an example system, we will use the control system for a computerized home, called ehome. Use cases for this system are assumed to consist of logging in, changing the room temperature, changing the unit of temperature, making coffee, moving drapes, and playing music. In Fig. 18.4, the coffee making use case has been refined into a sequence diagram.

Since we are here focusing on the architecture of the actual control system, we ignore user interface issues and follow a simple convention that the user interface is represented by a single (subsystem) participant that can receive use case requests. Accordingly, in the null architecture the user interface is in this example represented by a single component that has the use cases as operations.

To refine this use case, we observe that we need further components. The main unit for controlling the coffee machine is introduced as CoffeeManager; additionally, there is a separate component for managing water, WaterManager. If a component has a significant state or it manages a significant data entity (like, say, a data base), this is added to the participant box. In this case, CoffeeManager and WaterManager are assumed to have significant state information.

The null architecture in Fig. 18.5 (made by hand in this study) for the ehome system can be mechanically derived from the use case sequence diagrams. The null architecture only contains use relationships, as no more detail is given for the algorithm at this point. The null architecture represents the basic functional decomposition of the system.

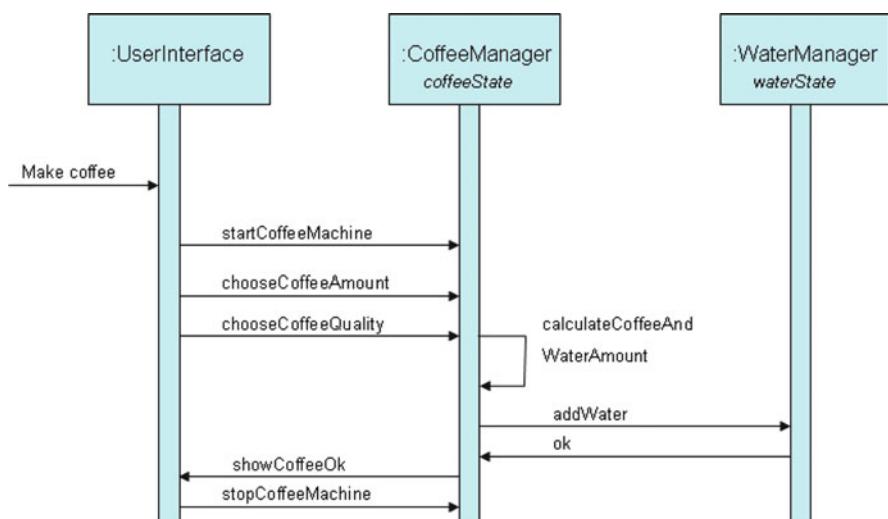


Fig. 18.4 Make coffee use case refined

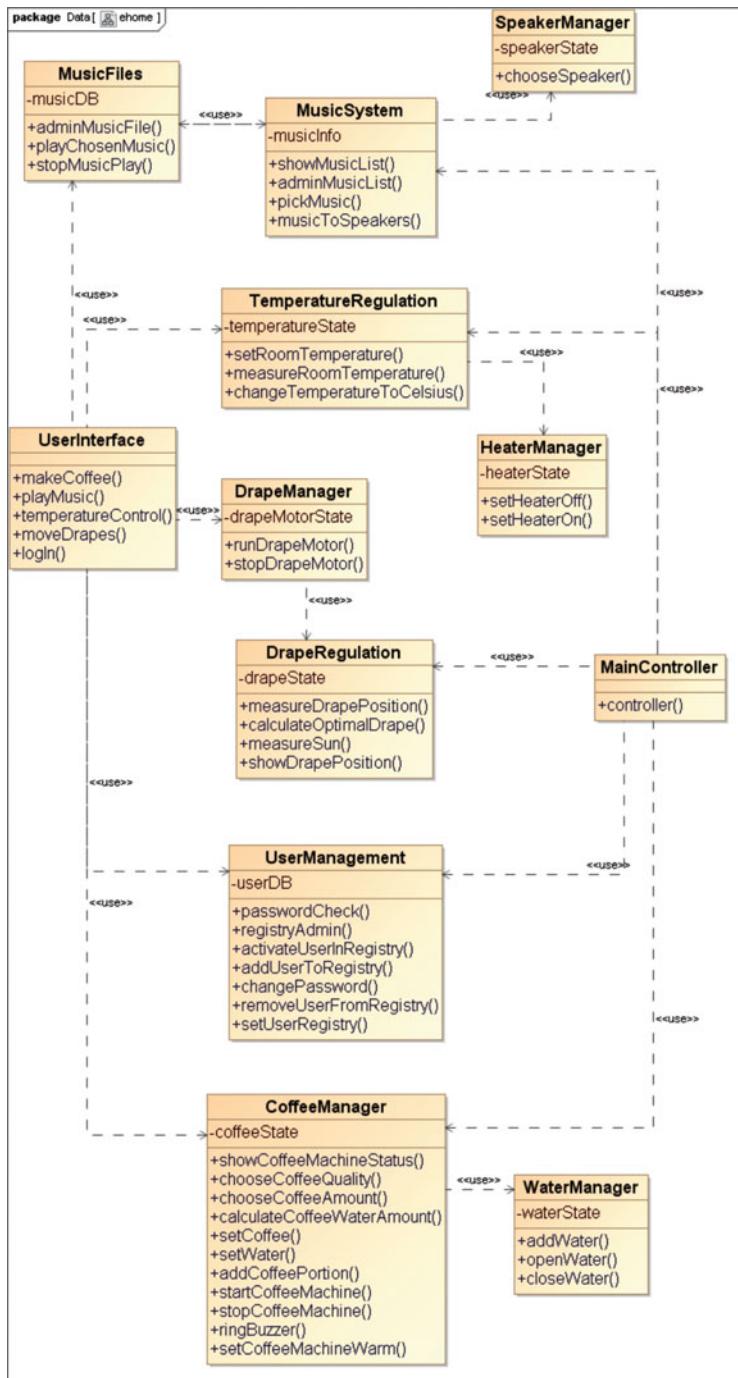


Fig. 18.5 Null architecture for ehome

After the operations are derived from the use cases, some properties of the operations can be estimated to support the genetic synthesis, regarding the amount of data an operation needs, frequency of calls, and sensitiveness for variation. For example, it is likely that the coffee machine status can be shown in several different ways, and thus it is more sensitive to variation than ringing the buzzer when the coffee is done. Measuring the position of drapes requires more information than running the drape motor, and playing music quite likely has a higher frequency than changing the password for the system. Relative values for the chosen properties can similarly be estimated for all operations. This optional information, together with operation call dependencies, is included in the information subjected to encoding.

Finally, different stakeholders' viewpoints are considered regarding how the system might evolve in the future, and modifiability scenarios are formulated accordingly. For example, change scenarios for the ehome system include:

- The user should be able to change the way the music list is showed (90%)
- The developer should be able to change the way water is connected to the coffee machine (50%)
- The developer should be able to add another way of showing the coffee machine status (60%).

A total of 15 scenarios were given for the ehome system.

18.5.2 Experiment

In our experiment, we used a population of 100 and 250 generations. The fitness curve presented is an average of 10 test runs, where the actual y-value is the average of 10 best individuals in a given population. The weights and probabilities for the tests were chosen based on previous experiments [34, 35, 40].

We first set all the weights to 1, i.e., did not favor any quality factor over another. The architecture achieved this way was quite simple. There were fairly well-placed instances of all low-level patterns (Adapter, Template Method and Strategy), and the client-server architecture style was also applied. Strikingly, however, the message dispatcher was not used as the general style, which we would have expected for this type of system. Consequently, we calibrated the weights by emphasizing positive modifiability over other quality attributes. Simultaneously negative efficiency was given a smaller than usual weight, to indicate that possible performance penalty of solutions increasing modifiability is not crucial. The fitness curve for this experiment is given in Fig. 18.6. As can be seen, the fitness curve develops steadily, and most improvement takes place between 1 and 100 generations, which is expected, as the architecture is still simple enough that applying the different mutations is easy.

An example solution with increased modifiability weight is depicted in Fig. 18.7. Now, the dispatcher architecture style is present, and there are also more Strategy patterns than in the solution where all quality factors were equally weighted. This is a natural consequence of the weighting: the dispatcher has a significant positive

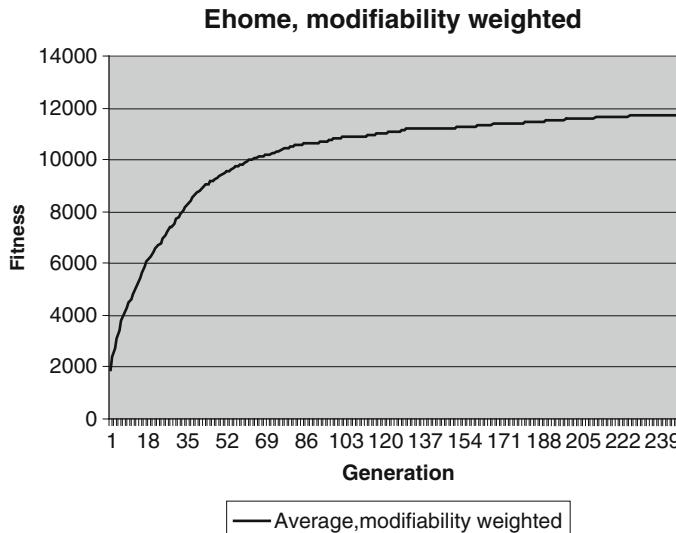


Fig. 18.6 Fitness development, modifiability weighted

effect on modifiability, and since it is not punished too much for inefficiency, it is fairly heavily used as a communication pattern. The same applies to Strategy, although in smaller scale.

18.6 Empirical Study on the Quality of Synthesized Architectures

As shown in the previous section, genetic software architecture synthesis appears to be able to produce reasonable architecture proposals, although obviously they still need some human polishing. However, since the method is not deterministic, it is essential to understand what is the goodness distribution of the proposals, that is, to what extent the architect can rely on the quality of the generated architecture. To study this, we carried out an experiment where we wanted to relate the quality of the generated architectures to the quality of the architectures produced by students. The setup and results of this experiment are discussed in the sequel.

18.6.1 Setup

18.6.1.1 Producing Architectures

First, a group of 38 students from an undergraduate software engineering class was asked to produce an architecture design for the ehome system. Most of the students

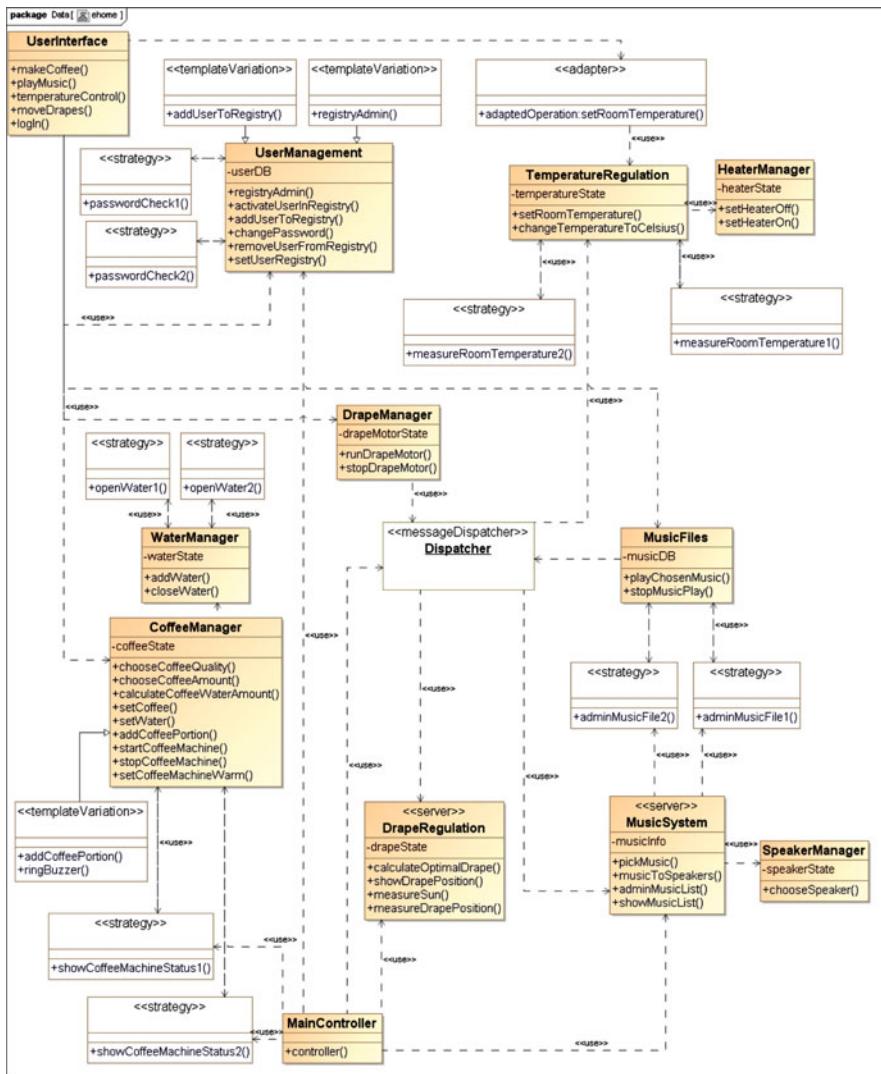


Fig. 18.7 Example architecture for ehome when modifiability is weighted over other quality factors

were third year Software Systems majors from Tampere University of Technology, having participated in a course on software architectures.

The students were given essentially the same information that is used as input for the GA, that is, the null architecture, the scenarios, and information about the expected frequencies of operations. In addition, students were given a brief explanation of the purpose and functionality of the system. They were asked to design the architecture for the system, using only the same architecture styles (message dispatcher and

client-server) and design patterns (Façade, Mediator, Strategy, Adapter, Template Method) that were available for GA. The students were instructed to consider efficiency, modifiability and simplicity in their designs, with an emphasis on modifiability. It took 90 min in the average for the students to produce a design.

In this experiment we wanted to evaluate genetically synthesized solutions against the student solutions in pairs. The synthesized solutions were achieved in 38 runs, out of which ten runs were randomly selected, resulting in ten architecture proposals. Each run took approximately 1 min (i.e., it took 1 min for the synthesizer to produce one solution). The setup for the synthesized architectures was the same as in the example given in Sect. 18.5.

18.6.1.2 Evaluating Architectures

After the students had returned their designs, the assistant teacher for the course (impartial to the GA research) was asked to grade the designs as test answers on a scale of 1–5, five being the highest. The solutions were then categorized according to the points they achieved. From the categories of 1, 3 and 5, one solution for each category was randomly selected. These architectures were presented as grading examples to four software engineering experts. The experts were researchers and teachers at the Department of Software Systems at Tampere University of Technology. They all had a M.Sc. or a Ph.D. degree in Software Systems or in a closely related discipline and several years of expertise from software architectures, gained by research or teaching.

In the actual experiment, the experts were given ten pairs of architectures. One solution in each pair was a student solution, selected randomly from the set of student solutions, and one was a synthesized solution. The solutions were edited in such a way that it was not possible for the experts to know which solution was synthesized. The experts were then asked to give each solution 1, 3 or 5 points. They were given the same information as the students regarding the requirements. The experts were not told how the solutions were achieved, i.e., that they were a combination of student and synthesized solutions. They were merely asked to help in evaluating how good solutions a synthesizer could make.

18.6.2 Results

The scores given by the experts ($e_1 - e_4$) to all the automatically synthesized architectures ($a_1 - a_{10}$) and architectures produced manually by the students ($m_1 - m_{10}$) are shown in Table 18.1. The points in Table 18.1 are organized so that the points given to the synthesized and human-made solutions of the same pair (a_i, m_i) are put next to each others so the pairwise points are easily seen. The result of each comparison is one of the following

Table 18.1 Points for synthesized solutions and solutions produced by the students

	a ₁	m ₁	a ₂	m ₂	a ₃	m ₃	a ₄	m ₄	a ₅	m ₅	a ₆	m ₆	a ₇	m ₇	a ₈	m ₈	a ₉	m ₉	a ₁₀	m ₁₀
e ₁	3	3	1	3	5	3	1	5	3	1	1	3	3	3	5	3	3	5	3	3
e ₂	5	1	3	3	5	1	1	1	3	3	3	5	1	1	3	1	1	1	5	1
e ₃	3	3	3	5	3	3	1	3	3	1	3	1	1	3	1	1	3	3	3	1
e ₄	3	1	5	3	3	5	3	1	5	1	5	3	3	3	3	1	3	3	5	1

- The synthesized solution is considered better ($a_i > m_i$, denoted later by +)
- The human-made solution is considered better ($m_i > a_i$, denoted later by -), or
- The solutions are considered equal ($a_i = m_i$, denoted latter by 0).

By doing so, we lose some information because one of the solutions is considered simply “better” even in the situation when it receives 5 points while the other receives 1 point. As can be seen in Table 18.1, this happens totally six times. In five of these six cases the synthesized solution is considered clearly better than the human-made solution, and only once vice versa. As our goal is to show that the synthesized solutions are at least as good as the human-made solutions, this lost of information does not bias the results.

The best synthesized solutions appear to be a_3 and a_{10} , with two 3’s and two 5’s. In solution a_3 the message dispatcher was used, and there were quite few patterns, so the design seemed easily understandable while still being modifiable. However, a_{10} was quite the opposite: the message dispatcher was not used, and there were especially as many as eight instances of the Strategy pattern, when a_3 had only two. There were also several Template Method and Adapter pattern instances. In this case the solution was highly modifiable, but not nearly as good in terms of simplicity. This demonstrates how very different solutions can be highly valued with the same evaluation criteria, when the criteria are conflicting: it is impossible to achieve a solution that is at the same time optimally efficient, modifiable and still understandable.

The worst synthesized solution was considered to be a_4 , with three 1’s and one 3. This solution used the message dispatcher but also the client-server style was eagerly applied. There were not very many patterns, and the ones that existed were quite poorly applied. Among the human-made solutions, there were three equally scored solutions (m_5 , m_8 , and m_{10}).

Table 18.2 shows the numbers of the preferences of the experts, with “+” indicating that the synthesized proposal was considered better than the student proposal, “-” indicating the opposite, and “0” indicating a tie. Only one (e_1) of the four experts preferred the human-made solutions slightly more often than the synthesized solution, while two experts (e_2 and e_4) preferred the synthesized solutions clearly more often than the human-made solutions. The fourth expert (e_3) preferred both types of solutions equally. There were totally 17 pairs of solutions with better score for the synthesized solution, nine pairs preferring the human-made solution, and 14 ties.

The above crude analysis clearly indicates that in our simple experiment, the synthesized solutions were ranked at least as high as student-made solutions. In order to get more exact information about the preferences and finding confirmation

Table 18.2 Numbers of preferences of the experts

	+	-	0
e ₁	3	4	3
e ₂	4	1	5
e ₃	3	3	4
e ₄	7	1	2
Total	17	9	14

even for the hypothesis that the synthesized solutions are significantly better than student-made solutions, it would be possible to use an appropriate statistical test (e.g., counting the Kendall coefficient of agreement). However, we omit such studies due to the small number of both experts and architecture proposals considered. At this stage, it is enough to notice that the synthesized solutions are competitive with those produced by third year software engineering students.

18.6.3 Threats and Limitations

We acknowledge that there are several threats and limitations in the presented experiment. Firstly, as the solutions for evaluations were selected randomly out of all the 38 student (and synthesized) solutions, it is theoretically possible that the solutions selected for the experiment do not give a true representation of the entire solution group. However, we argue that as all experts were able to find solutions they judged worth of 5 points as well as solutions only worth 1 point, and the majority of solutions were given 3 points, it is unlikely that the solutions subjected to evaluation would be so biased it would substantially affect the outcome of the experiment.

Secondly, the pairing of solutions could be questioned. A more diverse evaluation could have been if the experts were given the solutions in different pairs (e.g., for expert e₁ the solution a₁ would have been paired with m₅ instead of m₁). One might also ask if the outcome would be different with different pairing. We argue that as the overall points are better for the synthesized solutions, different pairing would not significantly change the outcome. Also, the experts were not actually told to evaluate the solutions as pairs – the pairing was simply done in order to ease the evaluation and analysis processes.

Thirdly, the actual evaluations made by the experts should be considered. Naturally, having more experts would have strengthened the results. However, the evaluations were quite uniform. There were very few cases where three experts considered the synthesized solution better or equal to the student solution (or the student solution better or equal to the synthesized one) and the fourth evaluation was completely contradicting. In fact, there were only three cases where such contradiction occurred (pairs 2, 3 and 4), and the contradicting expert was always the same (e₄). Thus we argue that the consensus between experts is sufficiently good, and increasing the number of evaluations would not substantially alter the outcome of the experiment in its current form.

Finally, the task setup was limited in the sense that architecture design was restricted to a given selection of patterns. Giving such a selection to the students may both improve the designs (as the students know that these patterns are potentially applicable) and worsen the designs (due to overuse of the patterns). Unfortunately, this limitation is due to the genetic synthesizer in its current stage, and could not be avoided.

18.7 Conclusions

We have presented a method for using genetic algorithms for producing software architectures, given a certain representation of functional and quality requirements. We have focused on three quality attributes: modifiability, efficiency and simplicity. The approach is evaluated with an empirical study, where the produced architectures were given for evaluation to experts alongside with student solutions for the same design problem.

The empirical study suggests that, with the assumptions given in Sect. 18.1, it is possible to synthesize software architectures that are roughly at the level of an undergraduate student. In addition to the automation aspect, major strengths of the presented approach are the versatility and options for expansion. Theoretically, an unlimited amount of patterns can be used in the solution library, while a human designer typically considers only a fairly limited set of standard solutions. The genetic synthesis is also not tied to prejudices, and is able to produce fresh, unbiased solutions that a human architect might not even think of. On the other hand, the current research setup and experiments are still quite limited. Obviously, the relatively simple architecture design task given in the experiment is still far from real-life software architecture design, with all its complications.

The main challenge in this approach is the specification of the fitness function. As it turned out in the experiment, even experts can disagree on what is a good architecture. Obviously, the fitness function can only approximate the idea of architectural quality. Also, tuning the parameters (fitness weights and mutation probabilities) is nontrivial and may require calibration for a particular type of a system. To alleviate the problem of tuning the weights of different quality attributes, we are currently exploring the use of Pareto optimality [41] to produce multiple architecture proposals with different emphasis of the quality attributes, instead of a single one.

In the future we will focus on potential applications of genetic software architecture synthesis. A particularly attractive application field of this technology is self-adapting systems (e.g., Cheng et al. [42]), where systems are really expected to “redesign” themselves without human interaction. Self-adaptation is required particularly in systems that are hard to maintain in a traditional way, like constantly running embedded systems or highly distributed web systems. We see the genetic technique proposed in this paper as a promising approach to give systems the ability to reconsider their architectural solutions based on some changes in their requirements or environment.

Acknowledgments We wish to thank the students and experts for participating in the experiment. The anonymous reviewers have significantly helped to improve the paper. This work has been funded by the Academy of Finland (project Darwin).

References

1. Denning P, Comer DE, Gries D, Mulder MC, Tucker A, Turner AJ, Young PR (1989) Computing as a discipline. *Commun. ACM* 32(1):9–23
2. Brown WJ, Malveau C, McCormick HW, Mowbray TJ (1998) Antipatterns – refactoring software, architectures, and projects in crisis. Wiley
3. S3 (2008) Proceedings of the 2008 workshop on self-sustaining systems. S3'2008, Potsdam, Germany, 15–16 May 2008. Lecture notes in computer science, vol 5146. Springer-Verlag, Heidelberg
4. Diaz-Pace A, Kim H, Bass L, Bianco P, Bachmann F (2008) Integrating quality-attribute reasoning frameworks in the ArchE design assistant. In: Becker S, Plasil F, Reussner R (eds) Proceedings of the 4th international conference on quality of software-architectures: models and architectures. Lecture notes in computer science, vol 5281. Springer, Karlsruhe, Germany, p 171
5. Buschmann F, Meunier R, Rohnert H, Sommerland P, Stal M (1996) A system of patterns – pattern-oriented software architecture. John Wiley & Sons, West Sussex, England
6. Holland JH (1975) Adaption in natural and artificial systems. MIT Press, Ann Arbor, Michigan, USA
7. Mitchell M (1996) An introduction to genetic algorithms. MIT Press, Cambridge
8. ISO (2001) Software engineering – product quality – part I: quality model. ISO/TEC 9126–1:2001
9. Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. *IEEE T Software Eng* 20(6):476–492
10. Clements P, Kazman R, Klein M (2002) Evaluating software architectures. Addison-Wesley, Reading
11. Clarke J, Dolado JJ, Harman M, Hierons R, Jones MB, Lumkin M, Mitchell B, Mancoridis S, Rees K, Roper M, Shepperd M (2003) Reformulating software engineering as a search problem. *IEE Proc – Softw* 150(3):161–175
12. Glover FW, Kochenberger GA (eds) (2003) Handbook of metaheuristics, vol 57, International series in operations research & management science. Springer, Heidelberg
13. Salomon R (1998) Short notes on the schema theorem and the building block hypothesis in genetic algorithms. In: Porto VW, Saravanan N, Waagen D, Eiben AE (eds) Evolutionary programming VII, 7th international conference, EP98, California, USA. Lecture notes in computer science, vol 1447. Springer, Berlin, p 113
14. Babar MA, Dingsoyr T, Lago P, van Vliet H (eds) (2009) Software architecture knowledge management – theory and practice establishing and managing knowledge sharing networks. Springer, Heidelberg
15. ISO (2010) Systems and software engineering – architecture description. ISO/IEC CD1 42010: 1–51
16. Kruchten P (1995) Architectural blueprints – the “4 + 1” view model of software architecture. *IEEE Softw* 12(6):42–50
17. Selonen P, Koskimies K, Systä T (2001) Generating structured implementation schemes from UML sequence diagrams. In: QiaYun L, Riehle R, Pour G, Meyer B (eds) Proceedings of TOOLS 2001. IEEE CS Press, California, USA, p 317
18. Harman M, Mansouri SA, Zhang Y (2009) Search based software engineering: a comprehensive review of trends, techniques and applications. Technical report TR-09-03, Kings College, London

19. Räihä O (2010) A survey on search-based software design. *Comput Sci Rev* 4(4):203–249
20. Bowman M, Brian, LC, Labiche Y (2007) Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms. Technical report SCE-07-02, Carleton University
21. Simons CL, Parmee IC (2007a) Single and multi-objective genetic operators in object-oriented conceptual software design. In: Proceedings of the genetic and evolutionary computation conference (GECCO'07). ACM Press, London, UK, p 1957
22. Simons CL, Parmee IC (2007) A cross-disciplinary technology transfer for search-based evolutionary computing: from engineering design to software engineering design. *Eng Optim* 39(5):631–648
23. Amoui M, Mirarab S, Ansari S, Lucas C (2006) A GA approach to design evolution using design pattern transformation. *Int J Inform Technol Intell Comput* 1:235–245
24. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns, elements of reusable object-oriented software. Addison-Wesley, Reading
25. Seng O, Bauyer M, Biehl M, Pache G (2005) Search-based improvement of subsystem decomposition. In: Proceedings of the genetic and evolutionary computation conference (GECCO'05). ACM Press, Mannheim, Germany, p 1045
26. Seng O, Stammel J, Burkhardt D (2006) Search-based determination of refactorings for improving the class structure of object-oriented systems. In: Proceedings of the genetic and evolutionary computation conference (GECCO'06). ACM Press, Washington, USA, p 1909
27. O'Keeffe M, Ó Cinnéide M (2004) Towards automated design improvements through combinatorial optimization. In: Workshop on directions in software engineering environments (WoDiSEE2004), Workshop at ICSE'04, 26th international conference on software engineering. Edinburgh, Scotland, p 75
28. O'Keeffe M, Ó Cinnéide M (2006) Search-based software maintenance. In: Proceedings of conference on software maintenance and reengineering. IEEE CS Press, Bari, Italy, p 249
29. O'Keeffe M, Ó Cinnéide M (2008) Search-based refactoring for software maintenance. *J Syst Software* 81(4):502–516
30. Mancoridis S, Mitchell BS, Rorres C, Chen YF, Gansner ER (1998) Using automatic clustering to produce high-level system organizations of source code. In: Proceedings of the international workshop on program comprehension (IWPC'98). Silver Spring, p 45
31. Di Penta M, Neteler M, Antoniol G, Merlo E (2005) A language-independent software renovation framework. *J Syst Software* 77:225–240
32. Menascé DA, Sousa JP, Malek S, Gomaa H (2010) QoS architectural patterns for self-architecting software systems. In: Proceedings of the 7th international conference on autonomic computing and communications. ACM Press, Washington DC, USA, p 195
33. Shaw M, Garlan D (1996) Software architecture – perspectives on an emerging discipline. Prentice Hall, Englewood Cliffs
34. Räihä O, Koskimies K, Mäkinen E (2008) Genetic synthesis of software architecture. In: Li X et al. (eds) Proceedings of the 7th international conference on simulated evolution and learning (SEAL'08). Lecture notes in computer science, vol 5361. Springer, Melbourne, Australia, p 565
35. Räihä O, Koskimies K, Mäkinen E, Systä T (2008) Pattern-based genetic model refinements in MDA. *Nordic J Comput* 14(4):338–355
36. Michalewicz Z (1992) Genetic algorithms + data structures = evolutionary programs. Springer, New York
37. Losavio F, Chirinos L, Matteo A, Lévy N, Ramdane-Cherif A (2004) ISO quality standards measuring architectures. *J Syst Software* 72:209–223
38. Mens T, Demeyer S (2001) Future trends in evolution metrics. In: Proceedings of international workshop on principles of software evolution. ACM Press, Vienna, Austria, p 83
39. Bass L, Clements P, Kazman R (1998) Software architecture in practice. Addison-Wesley, Boston

40. Räihä O, Koskimies K, Mäkinen E (2009) Scenario-based genetic synthesis of software architecture. In: Boness K, Fernandes JM, Hall JG, Machado RJ, Oberhauser R (eds) Proceedings of the 4th international conference on software engineering advances (ICSEA '09). IEEE, Porto, Portugal, p 437
41. Deb K (1999) Evolutionary algorithms for multicriterion optimization in engineering design. In: Miettinen K, Mäkelä MM, Neittaanmäki P, Periaux J (eds) Proceedings of the evolutionary algorithms in engineering and Computer Science (EUROGEN'99). University of Jyväskylä, Finland, p 135
42. Cheng B, de Lemos R, Giese H, Inverardi P, Magee J (2009) Software engineering for self-adaptive systems: a research roadmap. In: Cheng BHC, Lemos R de, Giese H, Inverardi P, Magee J (eds) Software engineering for self-adaptive systems. Lecture notes in computer science, vol 5525. Springer, Amsterdam, p 1

Chapter 19

How Software Architecture can Frame, Constrain and Inspire System Requirements

Eoin Woods and Nick Rozanski

Abstract Historically a system's requirements and its architectural design have been viewed as having a simple relationship where the requirements drove the architecture and the architecture was designed in order to meet the requirements. In contrast, our experience is that a much more dynamic relationship can be achieved between these key activities within the system design lifecycle, that allows the architecture to *constrain* the requirements to an achievable set of possibilities, *frame* the requirements making their implications clearer, and *inspire* new requirements from the capabilities of the system's architecture. In this article, we describe this relationship, illustrate it with a case study drawn from our experience and present some lessons learned that we believe will be valuable for other software architects.

19.1 Introduction

Historically, we have tended to view a system's requirements and its architectural design as having a fairly simple relationship; the requirements drove the architecture and the architecture was designed in order to meet the requirements. However this is a rather linear relationship for two such key elements of the design process and we have found that it is desirable to strive for a much richer interaction between them.

This chapter captures the results of our experience in designing systems, through which we have found that rather than just passively defining a system structure to meet a set of requirements, it is much more fruitful to use an iterative process that combines architecture and requirements definition. We have found that this allows the architectural design to *constrain* the requirements to an achievable set of possibilities, *frame* the requirements making their implications clearer, and *inspire* new requirements from its capabilities.

Our experience has led us to believe that the key to achieving this positive interplay between requirements and architecture is to focus on resolving the forces

inherent in the underlying *business drivers* that the system aims to meet. This process is part of a wider three-way interaction between requirements, architecture and project management. In this chapter we focus on the interaction between the requirements and architectural design processes, while touching on the relationship that both have with project management.

We start by examining the classical relationship between requirements and architectural design, before moving on to describe how to achieve a richer, more positive, interaction between requirements and architecture. We then illustrate the approach with a real case study from the retail sector. In so doing, we hope to show how architects need to look beyond the requirements that they are given and work creatively and collaboratively with requirements analysts and project managers in order to meet the system's business goals in the most effective way.

In this work, we focus on the architecture and requirements of *information systems*, as opposed to real-time or embedded systems. We have done this because that is the area in which we have both gained our architectural experience and applied the techniques we describe in the case study.

19.2 Requirements Versus Architecture

While both are very familiar concepts, given the number of interpretations that exist of the terms “system requirements” and “software architecture” it is worth briefly defining both in order to clearly set the scene for our discussion.

To avoid confusion over basic definitions, we use widely accepted standard definitions of both concepts, and they serve our purposes perfectly well.

Following the lead of the IEEE [7] we define systems requirements as being “(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 document; or a documented representation of a condition or capability as in (1) or (2)*.”

For software architecture, we use the standard definition from the ISO 42010 standard [8], which is “*the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution*.”

So for the purposes of our discussion we view requirements as being the definition of the capabilities that the system must deliver, and the architecture of the system being its structure and organization that should allow the system to provide the capabilities described by its requirements.

Gathering requirements is a complicated, subtle and varied task and for large systems the primary responsibility for this usually lies with a specialist requirements analyst (or “requirements engineer” depending on the domain and terminology in force). The task is usually divided into understanding the *functions* that the system must provide (its functional requirements) and the *qualities* that it must

exhibit while providing them (its non-functional requirements, such as security, scalability, usability and so on). There are many approaches for gathering requirements and the output of requirements-gathering activities varies widely [6, 17]. However, in our experience, requirements can be defined via formal written textual paragraphs, descriptions of scenarios that the system must be able to cope with, descriptions of typical usage of the system by one of its users, tables of measurements and limits that the system must meet or provide, user interface mock ups or descriptions, verbal discussions between individuals and so on. In reality the requirements of the system are usually found in a combination of sources –in, across and between the different requirements artefacts available.

For a large system, the design of the system’s key implementation structures (its “architecture”) is also a complicated and multi-faceted activity. It usually involves the design of a number of different but closely related aspects of the system including component structure, responsibility and interaction, concurrency design, information structure and flow, deployment approach and so on. One of the major difficulties with representing a system’s architecture is this multi-faceted nature, and in response to this, most architectural description approaches today are based on the idea of using multiple views, each capturing a different aspect of the architecture [4, 11, 19]. Hence the output of an architectural design exercise is usually a set of views of the system, with supporting information explaining how architecture allows the system to meet its key requirements, particularly its non-functional requirements (or quality properties as they are often known).

Finally, it is worth drawing a distinction between both the requirements and architecture of the system and their *documented representations*. In informal discussion we often merge the two concepts and refer to “requirements” to mean both the actual capabilities that the system must provide and the written form that we capture them in. Even more likely is confusion as to whether the term “the architecture” refers to the actual architecture of the system or the architectural documentation that explains it. In both cases, we would point out (as we and others have elsewhere, such as [2]) that whether or not they are clearly defined and captured, all systems have requirements and an architecture. We would also suggest that whether the latter meets the needs of the former is one of the significant determinants of the success for most systems. For this discussion, we focus on the real-world processes for requirements capture, architecture design and the interplay between them.

19.3 The Classical Relationship

Recognition of the importance of the relationship between the requirements and the design of a system is not a recent insight and software development methods have been relating the two for a long time.

The classical “Waterfall” method for software development [18] places requirements analysis quite clearly at the start of the development process and

then proceeds to system design, where design and architecture work would take place. There is a direct linkage between the activities, with the output of the requirements analysis processing being a (complete) set of specifications that the system must meet, which form the input to the design process. The design process is in turn a problem-solving activity to identify a complete design for the system that will allow it to meet that specification. The waterfall method is not really used in practice due to its rigidity and the resulting high cost of mistakes (as no validation of the system can be performed until it is all complete) but it does form a kind of cultural backdrop upon which other more sophisticated approaches are layered and compared. From our current perspective, the interesting thing about the waterfall approach is that although primitive, it does recognise the close relationship of requirements analysis and system architecture. The major limitation of the model is that it is a one-way relationship with the requirements being fed into the design process as its primary input, but with no feedback from the design to the requirements.

The well-known “spiral” model of software development [3] is one of the better-known early attempts at addressing the obvious limitations of the waterfall approach and has informed a number of later lifecycles such as Rational’s RUP method [12]. The spiral model recognises that systems cannot be successfully delivered using a simple set of forward-looking activities but that an iterative approach, with each iteration of the system involving some requirements analysis, design, implementation and review, is a much more effective and lower-risk way to deliver a complicated system. The approach reorganises the waterfall process into a series of risk-driven, linked iterations (or “spirals”), each of which attempts to identify and address one or more areas of the system’s design. The spiral model emphasises early feedback to the development team by way of reviews of all work products that are at the end of each iteration, including system prototypes that can be evaluated by the system’s main stakeholders. Fundamentally the spiral model focuses on managing risk in the development process by ensuring that the main risks facing the project are addressed in order of priority via an iterative prototyping process and this approach is used to prioritise and guide all of the system design activities.

A well-known development of the spiral model is the “Twin Peaks” model of software development, as defined by Bashar Nuseibeh [15] which attempts to address some limitations of the spiral model by organising the development process so that the system’s requirements and the system’s architecture are developed in parallel. Rather than each iteration defining the requirements and then defining (or refining) the architecture to meet them, the Twin Peaks model suggests that the two should be developed alongside each other because “*candidate architectures can constrain designers from meeting particular requirements, and the choice of requirements can influence the architecture that designers select or develop.*” While the spiral model’s approach to reducing risk is to feedback to the requirements process regularly, the Twin Peaks model’s refinement of this is to make the feedback immediate during the development of the two. By running concurrent, interlinked requirements and architecture processes in this way the approach aims to address some particular concerns in the development lifecycle, in particular

“I will know it when I see it” (users not knowing what their requirements are until something is built), using large COTS components within systems and rapid requirements change.

Most recently, the emergence of Agile software development approaches [14] has provided yet another perspective on the classical relationship between requirements and design. Agile approaches stress the importance of constant communication, working closely with the system’s end users (the “on-site customer”) throughout the development process, and regular delivery of valuable working software to allow it to be used, its value assessed and for the “velocity” (productivity) of the development team to be measured in a simple and tangible way. An important difference to note between the Agile and spiral approaches is that the spiral model assumes that the early deliveries will be prototype software whereas an Agile approach encourages the software to be fully developed for a minimal feature set and delivered to production and used (so maximising the value that people get from it, as early as possible). In an agile project, requirements and design artefacts tend to be informal and lightweight, with short “user stories” taking the place of detailed requirements documentation and informal (often short-lived) sketches taking the place of more rigorous and lengthy design documentation. The interplay between requirements and design is quite informal in the Agile approach, with requirements certainly driving design choices as in other approaches, and the emphasis being on the design emerging from the process of adding functions to the system, rather than “upfront” design. Feedback from the design to the requirements is often implicit: a designer may realize the difficulty of adding a new feature (and so a major piece of re-design – refactoring – is required), or spot the ability to extend the system in a new way, given the system’s potential capabilities, and suggest this to one of the customers who may decide to write a new “user story.”

In summary, the last 20 years have seen significant advances in the approach taken to relating requirements and design, with the emphasis on having design work inform the requirements process as early as possible, rather than leaving this until the system is nearly complete. However the remaining problem that we see with all of these approaches is that architecture is implicitly seen as the servant of the requirements process. Our experience suggests that in fact it is better to treat these two activities as equal parts of the system definition process, where architecture is not simply constrained and driven by the system’s requirements but has a more fundamental role in helping to scope, support and inspire them.

19.4 A Collaborative Relationship for Requirements and Architecture

We have found that the basis for defining a fruitful relationship between requirements and architecture needs to start with a consideration of the *business drivers* that cause the project to be undertaken in the first place. We consider the business drivers to be the underlying external forces acting on the project, and they capture

the fundamental motivations and rationale for creating the system. Business drivers answer the fundamental “why” questions that underpin the project: why is developing the system going to benefit the organization? why has it chosen to focus its energies and investment in this area rather than elsewhere? what is changing in the wider environment that makes this system necessary or useful? Requirements capture and architectural design, on the other hand, tend to answer (in different ways) the “what” and “how” questions about the system.

It is widely accepted that business drivers provide context, scope and focus for the requirements process, however we have also found them to be an important input into architectural design, by allowing design principles to be identified and justified by reference to them. Of course the requirements are also a key input to the architectural design, defining the capabilities that the architecture will need to support, but the business drivers provide another dimension, which helps the architect to understand the wider goals that the architecture will be expected to support. These relationships are illustrated by the informal diagram in Fig. 19.1.

Having a shared underlying set of drivers gives the requirements and architecture activities a common context and helps to ensure that the two are compatible and mutually supportive (after all, if they are both trying to support the same business drivers then they should be in broad alignment). However, it is important to understand that while the same set of drivers informs both processes, they may be used in quite different ways.

Some business drivers will tend to influence the requirements work more directly, while others will tend to influence architectural design more. For example, in a retail environment the need to respond to expansion from a single region where the organisation has a dense footprint of stores into new geographical areas is likely to have an effect on both the requirements and the architecture. It is clear that such drivers could lead to new requirements in the area of legislative flexibility, logistics and distribution, the ability to have multiple concurrent merchandising strategies, information availability, scalability with respect to stores and sales etc. However, while these requirements would certainly influence the architecture, what is maybe less obvious is that the underlying business driver could directly influence the architecture

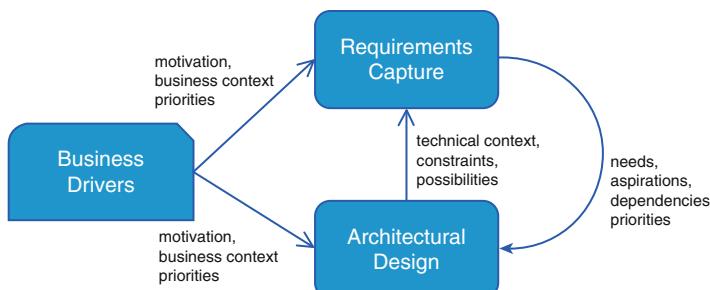
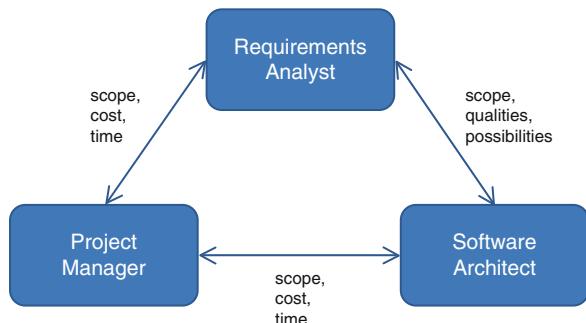


Fig. 19.1 Business drivers, requirements and architecture

Fig. 19.2 The three decision makers



in other ways, such as needing to ensure that the system is able to cope with relatively high network latency between its components or the need to provide automated and/or remote management of certain system components (e.g. in-store servers). These architectural decisions and constraints then in turn influence the requirements that the system can meet and may also suggest completely new possibilities. For example, the ability to cope with high network latencies could both limit requirements, perhaps constraining user interface options, and also open up new possibilities, such as the ability for stores to be able to operate in offline mode, disconnected from the data center, while continuing to sell goods and accept payments.

The other non-technical dimension to bear in mind is that this new relationship between requirements and architecture will also have an effect on the decision-making in the project. Whereas traditionally, the project manager and requirements engineer/analyst took many of the decisions with respect to system scope and function, this now becomes a more creative three-way tension between project manager, requirements engineer and software architect as illustrated by the informal diagram in Fig. 19.2.

All three members of the project team are involved in the key decisions for the project and so there should be a significant amount of detailed interaction between them. The project manager is particularly interested in the impact of the architect's and requirements analyst's decisions on scope, cost and time, and the requirements analyst and architect negotiate and challenge each other on the system's scope, qualities and the possibilities offered by the emerging architecture.

19.5 The Interplay of Architecture and Requirements

As we said in the previous section, the relationship between requirements and architecture does not need to be a straightforward flow from requirements “down” to architecture. Of course, there is a definite flow from the requirements

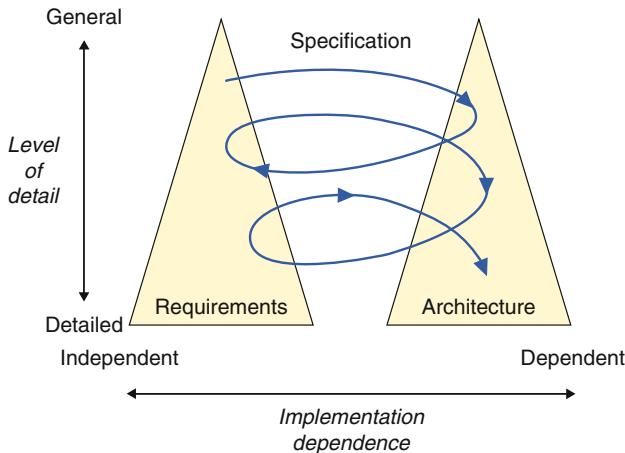


Fig. 19.3 Twin peaks (from [15])

analysis activity into the architectural design, and the requirements are one of the architect's primary inputs. But rather than being a simple one-way relationship, we would suggest that it is better to aim for a more intertwined relationship in the spirit of Twin Peaks, but developing this theme to the point where architecture frames, constrains and inspires the requirements as both are being developed. So we need to consider how this process works in more detail.

Bashar Nuseibeh's "Twin Peaks" model, as shown in Fig. 19.3, shows how the requirements definition and architectural design activities can be intertwined so that the two are developed in parallel. This allows the requirements to inform the architecture as they are gathered and the architecture to guide the requirements elicitation process as it is developed. The process of requirements analysis informing architectural design is widely accepted and well understood, as the requirements are a primary input to the architectural design process and an important part of architectural design is making sure that the requirements can be met. What is of interest to us here is how the architectural design process influences the requirements-gathering activity and we have found that there are three main ways in which this influence manifests itself.

19.5.1 The “Framing” Relationship

Starting with the simplest case, we can consider the situation where the architecture *frames* one or more requirements. This can be considered to be the classical case, where the requirements are identified and defined by the requirements analysis process and then addressed by the architecture. However, when the two are being developed in parallel then this has the advantage that the architecture provides

context and boundaries for the requirements during their development rather than waiting for them to be completed. Such context provides the requirements analyst with insights into the difficulty, risk and cost of implementing the requirements and so helps them to balance these factors against the likely benefits that implementing the requirement would provide. If this sort of contextual information is not provided when eliciting requirements for a system then there is the distinct danger that “blue sky” requirements are specified without any reference to the difficulty of providing them. When the architecture is developed in parallel with the requirements, this allows the architect to challenge requirements that would be expensive to implement. In cases where they are found to be of high value, consider early modifications or extensions to the system’s architecture to allow them to be achieved at lower cost or risk.

For example, while a “surface” style user interface might well allow new and innovative functions to be specified for a system, such devices are relatively immature, complicated to support, difficult to deploy and expensive to buy, so it would be reasonable for any requirements that require such interfaces to be challenged on the grounds of cost effectiveness and technical risk. The architecture doesn’t prevent this requirement from being met, but investigating its implementation feasibility in parallel with defining the requirement allows its true costs and risks to be understood.

19.5.2 The “Constraining” Relationship

In other situations, the architect may realize that the implementation of a requirement is likely to be very expensive, risky or time-consuming to implement using any credible architecture that they can identify. In such cases, we say that the architecture *constrains* the requirements, forcing the requirements analyst to focus on addressing the underlying business drivers in a more practical way.

To take an extreme example, while it is certainly true that instant visibility of totally consistent information across the globe would provide a new set of capabilities for many systems, it is not possible to achieve this using today’s information systems technology. It is therefore important that a requirements analyst respects this constraint and specifies a set of requirements that do not require such a facility in order to operate. In this case, understanding the implementation possibilities while the requirements are being developed allows a requirement to be highlighted as impossible to meet with a credible architecture, so allowing it to be removed or reworked early in the process.

19.5.3 The “Inspiring” Relationship

Finally, there are those situations where the architectural design process actually *inspires* new aspects of the emerging requirements, or “the solution drives the

problem” as David Garlan observed [5]. However while Garlan was commenting on this being possible in the case of product families (where the possible solutions are already understood from the existing architecture of the product family), we have also seen this happen when designing new systems from scratch. As the architecture is developed, both from the requirements and the underlying business drivers, it is often the case that architectural mechanisms need to be introduced which can have many potential uses and could support many types of system function. While they have been introduced to meet one requirement, there is often no reason why they cannot then also be used to support another, which perhaps had not been considered by the requirements analyst or the system’s users.

An example of this is an architectural design decision to deliver the user interface of the system via a web browser, which might be motivated by business drivers around geographical location, ease of access or low administrative overhead for the client devices. However, once this decision has been made, it opens up a number of new possibilities including user interface layer integration with other systems (e.g. via portals and mash ups), the delivery of the interface onto a much wider variety of devices than was previously envisaged (e.g. home computers as well as organisational ones) and accessing the interface from a wider variety of locations (e.g. Internet cafes when employees are travelling as well as office and home based locations). These possibilities can be fed back into the requirements process and can inspire new and exciting applications of the system. This effectively “creates” new requirements by extending the system’s possible usage into new areas.

So as can be seen there is great potential for a rich and fruitful set of interactions between requirements analysis and architectural design, leading to a lot of design synergy, if the two can be performed in parallel, based on a set of underlying business principles.

In practice, achieving these valuable interactions between requirements and architectural design means that the requirements analysts and the architects must work closely together in order to make sure that each has good visibility and understanding of the other’s work and that there is a constant flow of information and ideas between them.

As we said in the previous section, it is also important that the project manager is involved in this process. While we do not have space here to discuss the interaction with the project manager in detail, it is important to remember that the project manager is ultimately responsible for the cost, risk and schedule for the project. It is easy for the interplay between requirements and architecture to suggest many possibilities that the current project timescales, budget and risk appetite do not allow for, so it is important that the project manager is involved in order to ensure that sound prioritisation is used to decide what is achievable at the current time, and what needs to be recorded and deferred for future consideration.

19.6 Case Study

19.6.1 The Problem

A major clothing retailer was experiencing problems with stock accuracy of size-complex items in its stores, leading to lost sales and a negative customer perception.

A *size-complex* such as a men's suit is an expensive item which is sold in many different size permutations. A store will typically only have a small stock of each size permutation (for example, 44-Regular) on display or in its stockroom, since larger stock levels take up valuable store space and drive up costs in the supply chain.

Manual counting of size-complex items is laborious and error-prone. Even a small counting error can also lead to a *critical stock inaccuracy*, where the store believes it has some items of a particular size in stock but in fact has none. Critical inaccuracies lead to lost sales when customers cannot find the right size of an item they want to buy.

According to inventory management systems, the retailer's stock availability was around 85% for size-complex lines (that is, only 15% were sold out at any one time). However stock sampling indicated that real availability was as low as 45% for some lines, and that critical inaccuracy (where the line is sold out but the stock management system reports that there is stock available in store) was running as high as 15%. This was costing millions of pounds in lost sales, and also driving customer dissatisfaction up and customer conversion down (so customers were leaving stores without buying anything).

19.6.2 Project Goals

The goal of the project was to drive a 3–5% upturn in sales of size-complex lines by replacing error-prone and time-consuming manual stock counting with a more accurate and efficient automated system. By reducing the time taken to do a stock count from hours to a few minutes, the retailer expected to:

- Increase the accuracy of the stock count;
- Reduce the level of critical inaccuracy to near zero;
- Drive more effective replenishment;
- Provide timely and accurate information to head office management.

19.6.3 Constraints and Obstacles

The new system was subject to some significant constraints because of the environment into which it was to be used and deployed.

- The in-store equipment had to be simple to use by relatively unskilled staff with only brief training.
- The in-store equipment had to be robust and highly reliable. The cost of repair or replacement of units in stores was high and would eat away at much of the expected profits.
- The system had to be compatible with the infrastructure currently installed in stores. This was highly standardised for all stores and comprised: a store server running a fairly old but very stable version of Microsoft Windows; a wireless network, with some bandwidth and signal availability constraints in older stores because of their physical layout; and a low-bandwidth private stores WAN carrying mainly HTTP and IBM MQ traffic.
- The system had to be compatible with the infrastructure and systems which ran in Head Office and in partner organisations.
- The solution had to minimise the impact on the existing distribution channels and minimise any changes that needed to be made by suppliers and distributors.
- The solution had to minimise any increase in material or production costs.

Some further constraints emerged during the early stages of the project as described below.

19.6.4 Solution Evolution

The eventual architectural solution emerged over a number of iterations.

19.6.4.1 Initial Design

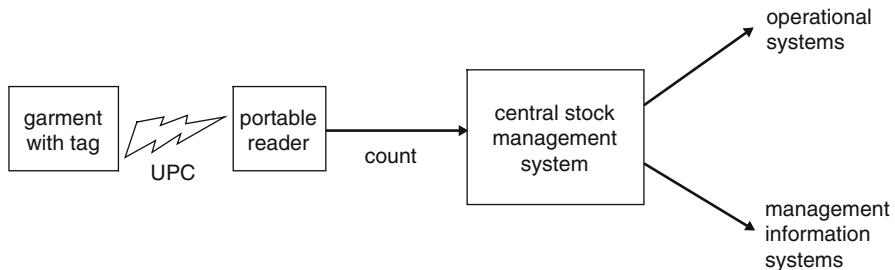
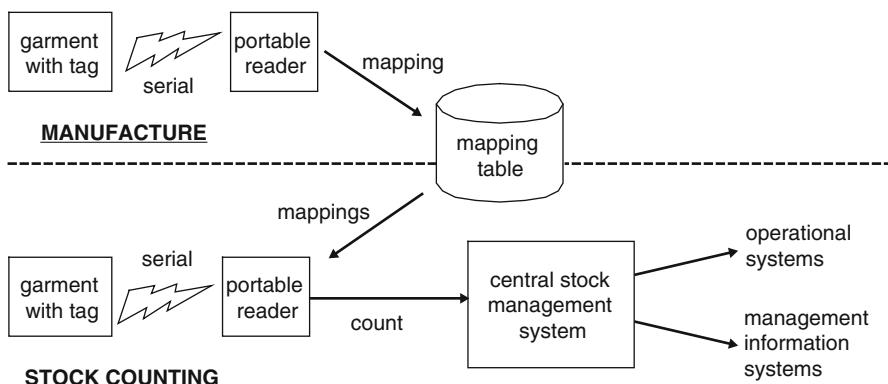
RFID (Radio Frequency Identification) was chosen as the underpinning technology for contactless data transfer. The initial concept was very simple: a passive (non-powered) RFID tag would be attached to each garment, and would store its UPC (universal product code, analogous to a barcode) which defined the garment's size, stroke etc.

A portable RFID reader would read the UPCs of all the garments in the area being counted, collate and filter the data, and send the resulting counts to the central stock management system in the form of a standard “stock correction” (Fig. 19.4).

This design illustrated the *framing* relationship between architecture and requirements. The retailer had some experience of RFID for stock-taking, but only at the pallet level, not for individual items.

19.6.4.2 First Iteration

Further investigation revealed that there were two types of RFID tag available for use, read-only (write once) and read-write (write many times). Read-write tags

**Fig. 19.4** Initial design**Fig. 19.5** First iteration

would be required for the solution above, since the UPC would need to be written to the tag when it was attached to the garment, rather than when the RFID tag was manufactured. However read-write tags were significantly more expensive and less reliable, so this approach was ruled out.

Ruling out read-write tags was a fairly significant change of direction, and was led primarily by cost and architectural concerns. However, since it had a significant impact on the production and logistics processes, the decision (which was led by architects) required the participation of a fairly wide range of business and IT stakeholders.

Since each RFID tag has a world-unique serial number, a second model was produced in which the serial number of a read-only tag would be used to derive the UPC of the garment. Once the tag was physically attached to the garment, the mapping between the tag's serial number and the garment's UPC would be written to a mapping table in a database in the retailer's data center (Fig 19.5).

There were again some significant implications to this approach, which required the architects to lead various discussions with store leaders, garment manufacturers, logistics partners and technology vendors. For example, it was necessary to develop a special scanning device for use by manufacturers. This would scan the RFID

serial number using an RFID scanner, capture the garment's UPC from its label using a barcode scanner, and transmit the data reliably to the mapping system at the retailer's data center. Since manufacturers were often located in the Far East or other distant locations, the device had to be simple, reliable and resilient to network connectivity failures.

This iteration illustrated the *constraining* relationship between architecture and requirements. The immaturity of the read-write RFID technology, and the potential cost implications of this approach led to a solution that was more complex, and imposed some significant new requirements on garment manufacturers, but would be significantly more reliable and cheaper to operate.

19.6.4.3 Second Iteration

It was initially planned to derive the UPC of the counted garment at the time that the tag serial numbers were captured by the in-store reader. However the reader was a relatively low-power device, and did not have the processing or storage capacity to do this. An application was therefore required to run on the store server, which maintained a copy of the mapping data and performed the required collation before the counts were sent off (Fig. 19.6).

This iteration also illustrated the constraining relationship between architecture and requirements.

19.6.4.4 Third Iteration

The next consideration was product returns, an important value proposition for this retailer. If a customer were to return a product for resale, then any existing tag

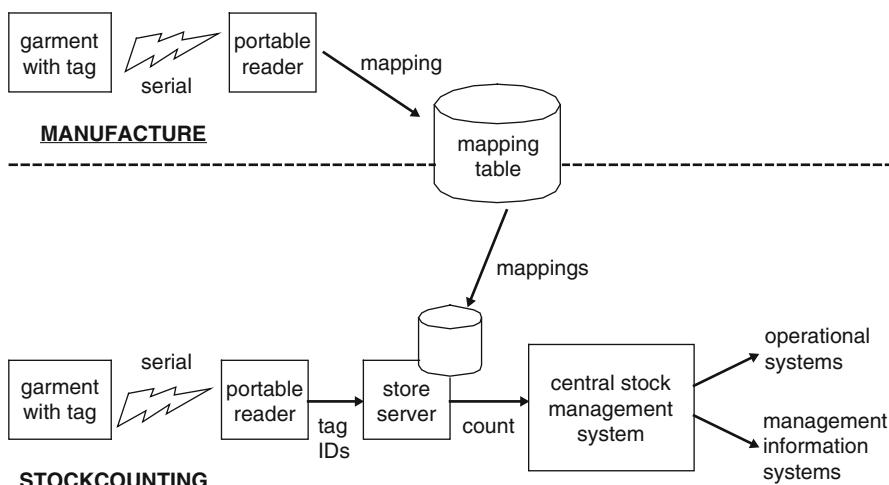


Fig. 19.6 Second iteration

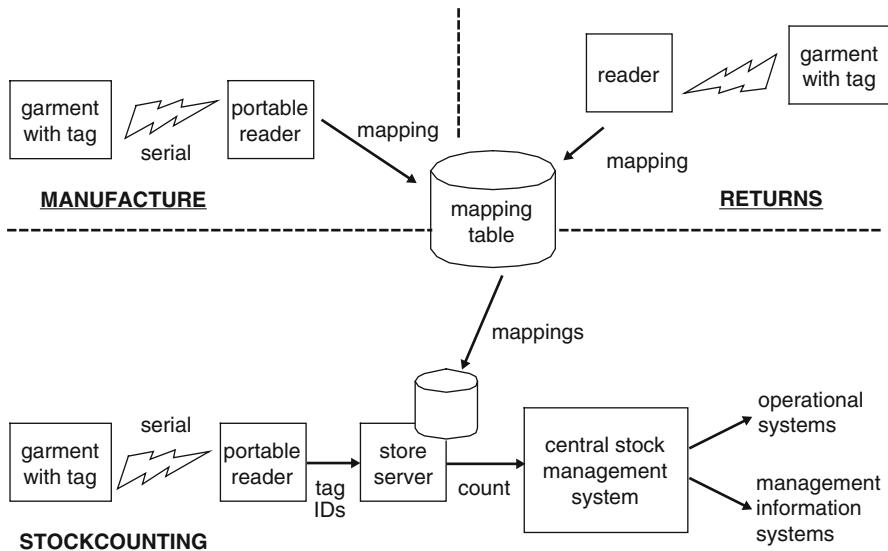


Fig. 19.7 Third iteration

would need to be removed, since it might have been damaged, a new tag attached, and the mapping table updated before the item was returned to the shop floor. This required a special tag reader on the shop floor, and also at the retailer's distribution centers.

This led to the third major iteration of the solution architecture as shown in Fig. 19.7.

This iteration illustrated the *inspiring* relationship between architecture and requirements. It was primarily the consideration of the architecture that prompted the addition of specific returns-processing capabilities to the solution, especially the provision of the specialized tag readers for this purpose.

19.6.4.5 Further Refinements

Discussions were also held with the team that managed the stock system. It already had the capability to enter stock corrections through a data entry screen, but an automated interface would need to be added and it was necessary to confirm that the system could deal with the expected volume of updates once the system was rolled out to all stores. This became an architectural and a scheduling dependency that was managed through discussions between the architects and project managers in both teams.

After surveying the marketplace it became clear that the reader would have to be custom-built. Manufacture was handed off to a third party but software architecture considerations drove aspects of the reader design. It needed to be portable, with its

own battery, and had to shut down gracefully, without losing data, if the battery ran out. It also had to present a simple touch-screen user interface.

Another constraint which emerged in the early stages of the project was around customer privacy. The retailer was very keen to protect its brand and its reputation, and the use of RFID to tag clothing was becoming controversial. In particular there was a concern amongst privacy groups that retailers would be able to scan clothes when a customer entered a store, and use the information to identify the customer and track their movements.

To allay any customer concerns, the tag was designed so that it could be removed from the garment and discarded after purchase. The retailer also met with the privacy groups to explain that their fears were unfounded, but needed to be careful not to reveal too much detail of the solution before its launch. Architects were involved in the technical aspects of this and also supported the retailer's Marketing Department who produced a leaflet for customers to explain the tags.

19.6.5 Project Outcome

The system was very successful. The initial pilot showed a consistent uplift in sales for counted lines, which exceeded the project goals. It was popular with staff and business users, and there was no customer resistance to the tags since they could be removed once the item had been paid for.

There were some further lessons from the pilot that were incorporated into the solution. For example, the readers proved to have a significantly larger operating range than expected and care needed to be taken not to count stock that was in the stock room rather than on the shop floor.

19.7 Evaluation of Approach

We have found the iterative approach to be an effective way of highlighting, early in the software development lifecycle, areas where requirements are missing or unclear, and which may otherwise only have become apparent much later in the project. By proposing and evaluating an initial architecture, architecturally significant problems and questions can also be made visible, and the right answers determined, before there has been costly investment in developing software that may later have to be changed.

Using iteration and refinement allows stakeholders to focus on key parts of the solution, rather than the whole, and to develop the overall architecture in stages. It also ensures that the proposed architecture can be considered in the light of real-world constraints, such as time, budget, skills or location, as well as just "architectural correctness."

There are some weaknesses to this approach however. There will be a significant amount of uncertainty and change in the early stages of the lifecycle, and if ongoing changes to the requirements or architecture are not communicated to everyone concerned, then there is a significant risk that the wrong solution will be developed.

Also, many stakeholders are uncomfortable with this level of uncertainty. Users may insist that the requirements are correct as initially specified, and object to changes they view as being IT-driven rather than user-driven. Developers, on the other hand, may struggle to design a system whose requirements are fluid or unclear.

Finally, an iterative approach can be viewed by senior management as extending the project duration. Explaining that this approach is likely to lead to an earlier successful delivery can be difficult.

All of these problems require careful management, oversight and discipline on the part of the project manager, the architect and the requirements analyst.

19.8 Lessons for Architects

The intertwined, parallel approach to relating system requirements and architectural design is the result of our experience working as information system architects, during which time we have attempted to use the ideas of the software architecture research community as they have emerged. As a result of this experience, we have not only refined our ideas about how an architect should work in the early stages of the definition of a new system, but have also learned some useful lessons which we try to capture here to guide others who are attempting the same type of work.

The lessons that we have found to be valuable during our attempts to work collaboratively with requirements analysts and project managers are as follows.

- *Early Involvement* – it is important for you to be involved during the definition and validation of requirements and plans, so aim to get involved as early in the system definition process as possible, even if this involves changing normal ways of working. Offering to perform early reviews and performing some of the requirements capture or planning yourself (particularly around non-functional requirements) can be a useful way into this process.
- *Understand the Drivers* – work hard to elicit the underlying business drivers that have motivated the commissioning of the system or project you are involved in. Often these drivers will not be well-understood or explicitly captured and so understanding and capturing them will be valuable to everyone involved in the project. Once you have the drivers you can understand the motivations for the project and start to think about how design solutions can meet them.
- *Intertwined Requirements and Architecture* – we have found that there are real benefits to the “twin peaks” approach of intertwining parallel requirements gathering and architectural design activity. You should build a working relationship with the requirements analyst(s) that allows this to happen and then work

together with them in order to develop the requirements and the architecture simultaneously with reference to each other.

- *Work Collaboratively* – as well as working in parallel and referencing each other's work, this approach needs a collaborative mindset on the parts of the requirements analyst and the architect, so aim to establish this early and try to work in this way. Of course, “it takes two to tango” so you may not always be successful if the requirements analyst is unused to working in this way, but in many cases if the architect starts to work in an open and collaborative way, others are happy to do so too.
- *Challenge Where Needed* – as requirements start to emerge, do not be afraid to challenge them if you feel that they will be prohibitively expensive or risky to implement, or you spot fundamental incompatibilities between different sets of requirements that will cause severe implementation difficulties. It is exactly this sort of early feedback that is one of the most valuable outputs of this style of working.
- *Understand Costs and Risks* – during the early stages of a large project you are in a unique position to understand the costs and risks of implementing proposed requirements, as it is unlikely that a project manager or a requirements analyst will have a deep knowledge of the design possibilities for the system. You should work with the project manager to understand the costs and risks inherent in the emerging requirements, and then explain them to the other decision makers on the project to allow more informed decisions to be made.
- *Look for Opportunities* – the other dimension that you can bring to the project in its early stages is an understanding of opportunities provided by each of the candidate architectural designs for the system. Your slightly different perspective on the business drivers, away from their purely functional implications, allows a broader view that often results in an architecture that has capabilities within it that can be used in many different ways. By pointing these capabilities out to the requirements analyst, you may well inspire valuable new system capabilities, even if they cannot be immediately implemented.

In short, our experience suggests that rather than accepting a set of completed system requirements, architects need to get involved in projects early and working positively and collaboratively with the requirements analysts and project managers to shape the project in order to increase its chances of successful delivery, while making the most of the potential that its underlying architecture can offer.

19.9 Related Work

As already noted, many other people working in the software architecture field have explored the close relationship between requirements and software architecture.

The SEI's Architecture Trade-off Analysis Method – or ATAM – is a structured approach to assessing how well a system is likely to meet its requirements, based on the characteristics of its architecture [10]. In the approach, a set of key scenarios are

identified and analysed to understand the risks and trade-offs inherent in the design and how the system will meet its requirements. When applied early in the lifecycle, ATAM can provide feedback into the requirements process.

A related SEI method is the Quality Attribute Workshop – or QAW – which is a method for identifying the critical quality attributes of a system (such as performance, security, availability and so on) from the business goals of the acquiring organisation [1]. The focus of QAW is identification of critical requirements rather than how the system will meet them, and so is often used as a precursor to ATAM reviews.

Global Analysis [16] is another technique used to relate requirements and software architecture by structuring the analysis of a range of factors that influence the form of software architectures (including organisational constraints, technical constraints and product requirements). The aim of the process is to identify a set of system-wide strategies that guide the software design to meet the constraints that it faces, so helping to bridge the gap between requirements and architectural design.

As well as architecture centric approaches, there have also been a number of novel attempts to relate the problem domain and the solution domain from the requirements engineering community.

One example is Michael Jackson's Problem Frames approach [9], which encourages the requirements analyst to consider the different *domains* of interest within the overall problem domain, how these domains are inter-related via *shared phenomena* and how they affect the system being built. We view techniques like problems frames as being very complimentary to the ideas we have developed here, as they encourage the requirements analyst to delve deeply into the problem, domain and uncover the key requirements that the architecture will need to meet.

Another technique from the requirements engineering community is the KAOS method, developed at the universities of Oregon and Louvain [13]. Like Problem Frames, KAOS encourages the requirements analyst to understand all of the problem domain, not just the part that interfaces with the system being built, and shows how to link requirements back to business goals. Again we view this as a complimentary approach as the use of a method like KAOS is likely to help the requirements analyst and architecture align their work more quickly than would otherwise be the case, as well as understand the problem domain more deeply.

19.10 Summary

In this chapter we have explained how our experience has led us to realise that a much richer relationship can exist between requirements gathering and architectural design than their classical relationship would suggest. Rather than passively accepting requirements into the design process, much better systems are created when the requirements analyst and architect work together to allow architecture to *constrain* the requirements to an achievable set of possibilities, *frame* the requirements making their implications clearer, and *inspire* new requirements from the capabilities of the system's architecture.

References

1. Barbacci M, Ellison R, Lattanze A, Stafford J, Weinstock C, Wood W (2003) Quality attribute workshops (QAWs) 3rd edn. Technical report, CMU/SEI-2003-TR-016, Software Engineering Institute, Carnegie Mellon University
2. Bass L, Clements P, Kazman R (2003) Software architecture in practice, 2nd edn. Prentice Hall, Englewood Cliffs
3. Boehm B (1986) A spiral model of software development and enhancement. ACM SIGSOFT Softw Eng Notes 21(5):61–72
4. Clements P, Bachmann F, Bass L, Garlan D, Ivers J, Little R, Nord R, Stafford J (2002) Documenting software architectures: views and beyond. Addison Wesley, Boston
5. Garlan D (1994) The role of software architecture in requirements engineering. In: Proceedings of the second international conference on requirements engineering. University of York, UK, 18–21 April 1994
6. Hull E, Jackson K, Dick J (2010) Requirements engineering. Springer Verlag, Berlin
7. Institution of Electrical and Electronic Engineers (1990) IEEE standard glossary of software engineering terminology. IEEE Standard 610.12-1990
8. International Standards Organisation (2007) Systems and software engineering – recommended practice for architectural description of software-intensive systems. ISO/IEC Standard 42010:2007
9. Jackson M (2001) Problem Frames. Addison Wesley, Wokingham
10. Kazman R, Klein M, Clements P (2000) ATAM: a method for architecture evaluation. Technical report, CMU/SEI-2000-TR-004, Software Engineering Institute, Carnegie Mellon University, Pittsburgh
11. Kruchten P (1995) The 4 + 1 view model of software architecture. IEEE Softw 12(6):42–50
12. Kruchten P (2003) The rational unified process: an introduction, 3rd edn. Addison-Wesley, Boston
13. van Lamsweerde A, Letier E (2002) From object orientation to goal orientation: a paradigm shift for requirements engineering. In: Proceedings of the radical innovations of software and systems engineering, Venice, Italy, 7–11 October 2002. Lecture notes in computer science, vol 2941. Springer, Heidelberg, pp 325–340
14. Larman C (2004) Agile and iterative development: a manager's guide. Addison-Wesley, Reading
15. Nuseibeh B (2001) Weaving together requirements and architectures. IEEE Comput 34 (3):115–119
16. Nord R, Soni D (2003) Experience with global analysis: a practical method for analyzing factors that influence software architectures. In: Proceedings of the second international software requirements to architectures workshop (STRAW). Portland, Oregon, 9 May 2003
17. Robertson S, Robertson J (2006) Mastering the requirements process, 2nd edn. Addison Wesley, Reading
18. Royce W (1970) Managing the development of large software systems. In: Proceedings of IEEE WESCON, vol 26. Los Alamitos, pp 1–9
19. Rozanski N, Woods E (2005) Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison Wesley, Boston

Chapter 20

Economics-Driven Architecting for Non Functional Requirements in the Presence of Middleware

Rami Bahsoon and Wolfgang Emmerich

Abstract The current trend is to build distributed software architectures with middleware, which provides the application developer with primitives for managing the complexity of distribution and for realizing many of the non-functional requirements like scalability, openness, heterogeneity, availability, reliability and fault-tolerance. In this chapter, we discuss the problem of evolving non-functional requirements, their stability implications and economics ramifications on the software architectures induced by middleware. We look at the role of middleware in architecting for non-functional requirements and their evolution trends. We advocate adjusting requirements elicitation and management techniques to elicit not just the current non-functional requirements, but also to assess the way in which they will develop over the lifetime of the architecture and their economics ramifications. These ranges of requirements may then inform the selection of distributed components technologies, and subsequently the selection of application server products. We describe an economics-driven approach, based on real options theory, which can assist in informing the selection on middleware to induce software architectures in relation to the evolving non-functional requirements. We review its application through a case study.

20.1 Middleware-Induced Architectures

The requirements that drive the decision towards building a distributed system architecture are usually of a non-functional nature. Scalability, openness, heterogeneity, and fault-tolerance are just examples. The current trend is to build distributed systems architectures with middleware technologies such as Java 2 Enterprise Edition (J2EE) [1] and the Common Object Request Broker Architecture (CORBA) [2]. COTS Middleware simplifies the construction of distributed systems by providing high-level primitives, which shield the application engineers from distribution complexities, managing systems resources, and implementing low-level details, such as concurrency control, transaction management, and network communication.

These primitives are often responsible for realizing many of the non-functional requirements in the architecture of the software system induced. Despite the fact that architectures and middleware address different phases of software development, the usage of middleware can influence the architecture of the system being developed. Conversely, specific architectural choices constrain the selection of the underlying middleware [3]. Once a particular middleware system has been chosen for a software architecture, it is extremely expensive to revert that choice and adopt a different middleware or a different architecture. The choice is influenced by the non-functional requirements. Unfortunately, these requirements tend to be unstable and evolve over time and threaten the stability of the architecture. Non-functional requirements often change with the setting in which the system is embedded, for example when new hardware or operating system platforms are added as a result of a merger, or when scalability requirements change due to sudden increase in users as it is the case of successful e-commerce systems.

Adopting a *flexible* COTS middleware induced-architecture that is capable of accommodating future changes in non-functional requirements, while leaving the architecture intact is important. Unfortunately, such viability and sustainability of the choice comes with a price. This is often a matter of how valuable this flexibility will be in the future relative to the likely changes in non-functional requirements. When such flexibility ceases to add a value and becomes a liability, watch out, the “beauty” is becoming wild and intolerable. Alternatively, a middleware with limited flexibility may still realize the change through “cosmetic” solutions of ad-hoc or proprietary nature, such as modifying part of the middleware; extending the middleware primitives; implementing additional interfaces; and so forth. These solutions could seem to be costly, problematic, and unacceptable; yet they may turn to be more cost-effective in the long-run.

As a motivating example, consider a distributed software architecture that is to be used for providing the back-end services of an organization. This architecture will be built on middleware. Depending on which COTS middleware is chosen, different architectures may be induced [3]. These architectures will have differences in how well the system is going to cope with changes. For example, a CORBA-based solution might meet the functional requirements of a system in the same way as a distributed component-based solution that is based on a J2EE application server. A notable difference between these two architectures will be that increasing scalability demands might be easily accommodated in the J2EE architecture because J2EE primitives for replication of Enterprise Java Beans can be used, while the CORBA-based architecture may not easily scale. The choice is not straightforward as the J2EE-based infrastructures usually incur significant upfront license costs. Thus, when selecting an architecture, the question arises whether an organization wants to invest into an J2EE application server and its implementation within an organization, or whether it would be better off implementing a CORBA solution. Answering this question without taking into account the *flexibility* that the J2EE solution provides and how *valuable* this flexibility will be in the future might lead to making the wrong choice.

The chapter is organised as follows: Sect. 20.2 reports on a case study, which demonstrates how economics-driven approaches can inform the selection of more stable middleware-induced architectures and discusses observation resulted from its application. Section 20.3 discusses closely related work. Section 20.4 concludes.

20.2 Case Study

20.2.1 *Rationale and Aims*

The case study demonstrates a novel application of real options theory and its fitness for informing the selection of a more “stable” middleware-induced architecture. The observations derived upon conducting the case aims at advancing our understanding to the architectural stability problem, when addressed in relation to middleware. The case simulates the selection process inspired by economics-driven approaches and highlights possible insights that could derive from the application of real options theory to the selection problem. In particular, the case study extends the confidence in the following specific claims: (1) the uncertainty, attributed to the likelihood of change(s), makes real options theory superior to other valuation techniques, which fall short in dealing with the value of architectural flexibility under uncertainty; (2) the flexibility of a middleware-induced architecture in face of likely changes in requirements creates values in the form of real options; (3) The problem of finding a potentially stable middleware-induced architecture requires finding a solution that maximizes the yield in the added value, relative to some likely future changes in requirements. If we assume that the added value is attributed to flexibility, the problem becomes maximizing the yield in the embedded or adapted flexibility provided by the selected middleware-induced architecture relative to these changes; and nevertheless (4) the decision of selecting a potentially stable architecture has to maximize the value added relative to some valuation points of view: we take savings in future maintainability as one dimension of value to illustrate the approach.

We note that case studies have been extensively used to empirically assess software engineering approaches. When performed in real situations, case studies provide practical and empirical evidence that a method is appropriate to solve a particular class of problems. According to Dawson et al. [4], conducting controlled and repeatable experiments in software engineering is quite difficult, if not impossible to accomplish. This is mainly because the way software engineering methods are applied varies across different contexts and involve variables that cannot be fully controlled. We note that the primary aim of this case study is to show how economics-driven approaches can provide an effective alternative to inform the selection of middleware-induced architectures. Under no considerations should the results be regarded as a definite distinction of the merit of one technology over the other as we have only used “flavors” of CORBA and J2EE.

20.2.2 Setting

Let us observe how the flexibility of a software architecture, when induced by different COTS middlewares, differs in coping with changes in non-functional requirements. We use the Duke's Bank application [1], an online banking application, which adequately represents a medium-size component-based distributed system. The architecture of the Duke's Bank application has a three-tier style, given in Fig. 20.1. The architecture has two clients: an application client used by administrators to manage customers and accounts, and a Web client used by customers to access account statements and perform transactions. The server-side components perform the business methods: these include managing: customers, accounts, and transactions. The clients access the customer, account, and transaction information maintained in a database.

We instantiate from the core architecture two versions, each induced by a different COTS middleware: one with CORBA and the other with J2EE. Assume that the Duke's Bank system needs to scale to accommodate the growing number of clients in 1-year time. Scalability denotes the ability to accommodate a growing future load, be it expected or not. We observe how a likely future change in scalability, a representative critical change in non-functional requirement, could impact the architectural structure of each version. The challenge of building a scalable system is to support changes in the allocation of components to hosts without breaking the architecture of the software system; changing the design and code of a component [5]; and/or rippling the change to impact other non-functionalities such as performance, reliability, and availability.

We use replication, an architectural mechanism, to achieve scalability. Both CORBA and J2EE do provide the primitives or guidelines for scaling a software system using replication, which make the comparison between the two versions feasible. In particular, the Object Management Group's CORBA specification defines a fault tolerance and a load balancing support, which provides the core capability for implementing scalability through replication. Similarly, J2EE provides clustering primitives for scaling the software system through replication. We adopt a goal-oriented approach to refining requirements [6, 7]. We refine the goal, using guidance on how it could be operationalised by the architecture, when

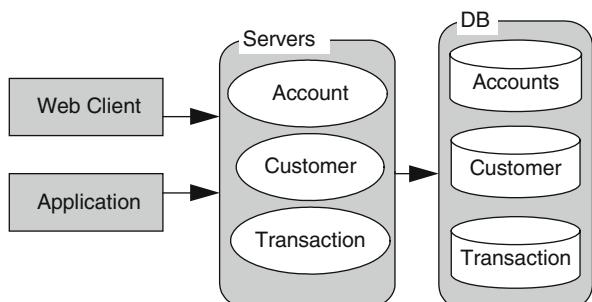


Fig. 20.1 The architecture of the duke's bank

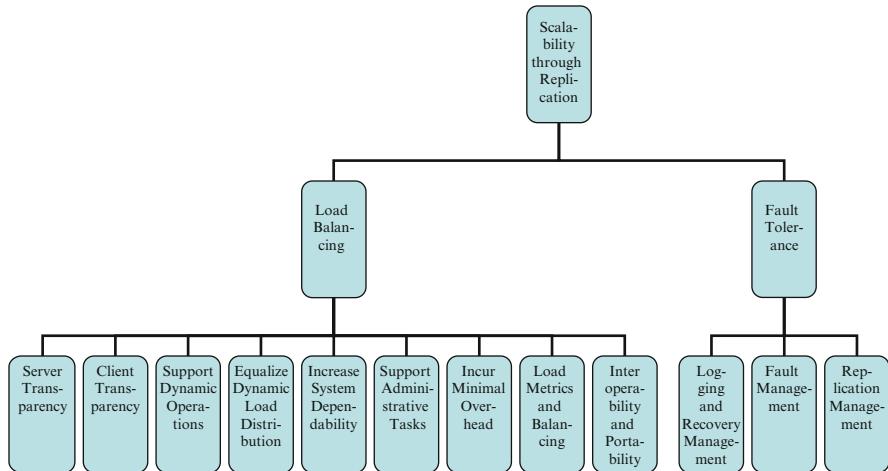


Fig. 20.2 The goal-oriented refinement for achieving scalability through replication

induced by a particular middleware. In more abstract terms, the guidance was given through the knowledge of the domain; vendor's specification [1, 2]; related design and implementation experience, mainly that of Othman et al. [8, 9]. We note that different architectural mechanisms may operationalise the scalability goal. As an operationalisation alternative, we use replication as way for achieving scalability. The reason is due to the fact that both CORBA and J2EE do provide the primitives or guidelines for scaling a software system using replication, which make the comparison between the two versions feasible. In particular, the Object Management Group's CORBA specification [2] defines a fault tolerance and a load balancing support, both when combined provide the core capability for implementing scalability through replication. Similarly, J2EE provides the primitives for scaling the software system through replication. Hence, the refinement and its corresponding operationalisation are guided by the solution domain (i.e., the middleware). Refinement of the scalability goal is depicted in Fig 20.2. We then estimate the structural impact and the SLOC to be added upon achieving scalability on both versions.

20.2.3 Scaling the J2EE Induced Architecture

An observable advantage of scaling the software architecture when induced by J2EE is that no development effort is required to realize the scalability requirements through replication, as when compared to the CORBA version. J2EE does provide clustering primitives for scaling the software system, which result in making the architecture of the software system more *flexible* in accommodating the change in scalability, as when compared to the CORBA version.

20.2.4 Scaling the CORBA Induced Architecture

Considering the CORBA-induced architecture of the Duke's Bank, supporting scalability through replication does not leave the middleware infrastructure and the application layer intact. Though the use of both CORBA specification and design patterns [10], has simplified the task of realizing the requirements for achieving fault tolerance and load balancing, implementation and integration overhead have not been abandoned. In particular, the fault tolerance and load balancing services need to be implemented and be integrated into the used middleware. The server and the client application need to be updated.

20.2.5 Valuing Flexibility with Options Theory

Middleware-induced software architectures differ in coping with changes in scalability. A question of interest is how valuable is the flexibility of either alternative, relative to likely change in scalability, will be in the long-run? How can we decide which COTS solution is better? The economic interplay between evolving requirements and the flexibility of the COTS middleware architecture in accommodating these changes in requirements is the governing factor that decides.

Let us assume that we are given the choice of two middleware M_0 and M_1 to induce the architecture of a particular system. Let us assume that S_0 , S_1 are the architectures obtained from inducing M_0 and M_1 respectively. Say, M_1 is an economical choice, if it adds value to S_1 relative to S_0 . We attribute the added value to the enhanced flexibility of S_1 over S_0 in responding in future changes in requirements. But the added value is uncertain, as the demand and the nature of the future changes are uncertain. Using options theory can inform the selection.

Real options analysis recognizes that the value of the capital investment lies not only in the amount of direct revenues that the investment is expected to generate, but also in the future opportunities that flexibility creates [11, 12]. These include growth, abandonment or exit, delay, and learning options. An option is an asset that provides its owner the right without a symmetric obligation to make an investment decision under given terms for a period of time into the future ending with an expiration date [13]. If conditions favourable to investing arise, the owner can exercise the option by investing the exercise price defined by the option. A call option gives the right to acquire an asset of uncertain future value for the strike price.

ArchOptions [10, 14, 15] values the *growth* options of an architecture relative to some future changes, as a way for understanding the architectural flexibility and its stability implications. A growth option is a real option to expand with strategic importance [13] and is common in infrastructure-based investments, as is the case with software architectures. Since the future changes are generally unanticipated, the value of the growth options lies in the enhanced flexibility of the architecture to cope with uncertainty.

Table 20.1 Financial/real options/archoptions analogy

Option on stock	Real option on a project	ArchOptions
Stock Price	Value of the expected cash flows	Value of the “architectural potential” relative to the change ($x_i V$)
Exercise Price	Investment cost	Estimate of the likely cost to accommodate the change (C_{ei})
Time-to-expiration	Time until opportunity disappears	Time indicating the decision to implement the change (t)
Volatility	Uncertainty of the project value	“Fluctuation” in the return of value of V over a specified period of time (σ)

Deciding on a particular middleware to induce the software system architecture can be seen as an investment to purchase future growth options that enhance the upside potentials of the structure, paying an upfront cost I_e , which corresponds to the cost of developing the architecture by the given middleware. A change in future requirement i , is assumed to “buy” $x_i\%$ of the “architectural potential” taking the form of embedded flexibility, paying C_{ei} , an estimate of the likely cost to accommodate the change in the software system. This is analogous to a *call option* to buy ($x_i\%$) of the base project, paying C_{ei} as exercise price. We view the investment opportunity in the system as a base investment plus call options on the future opportunities, where a future opportunity corresponds to the investment to accommodate some future requirement(s). The call options financial/real and their corresponding ArchOptions analogy is depicted in Table 20.1.

The payoff of the constructed call option gives an indication of how valuable the flexibility of an architecture is, when enduring some likely changes in requirements. The value of the architecture, is expressed in (1) accounting for both the expected value and exercise cost to accommodate future requirements i , for $i \leq n$. Valuing the expectation E of expression (1) uses the assumptions of Black and Scholes⁴ and detailed in previous work [15].

$$-I_e + \sum_{i=0}^n E[\max(x_i V - C_{ei}, 0)] \quad (20.1)$$

The selection has to be guided by the expected payoff in $(-I_e + \sum_{i=1\dots n} E[\max(x_i V - C_{ei}, 0)])_{S_1}$ relative to that of S_0 . That is, if $(-I_e + \sum_{i=1\dots n} E[\max(x_i V - C_{ei}, 0)])_{S_1} > \sum_{i=1\dots n} E[\max(x_i V - C_{ei}, 0)]_{S_0}$ for some likely changes, then it is worth investing in M_1 , as M_1 is likely to generate more growth options for S_1 than for S_0 . We use future savings in maintenance to quantify the value added due to a selection. Assume that $x_i V_{S_1/S_0}$ is the expected savings in S_1 over S_0 due to the selection: if $(-I_e + \sum_{i=1\dots n} E[\max(x_i V_{S_1/S_0} - C_{ei}, 0)])_{S_1} > 0$, then investing in M_1 is said to payoff. An optimal payoff could be when the option value approaches the maximum relative to some changes in non-functional requirements.

For a likely change in requirement k ,

Call option in-the-money. If $(E[\max(x_k V - C_{ek}, 0)])_{S_1} > 0$, then the flexibility of S_1 is likely to payoff, relative to S_0 , if k need to be exercised. Thus, inducing the

architecture with M_1 has more promise than M_0 , as the flexibility of S_1 in responding to the likely change is more valuable for S_1 than for S_0 .

Call option out-of-the-money: If $(E[\max(x_k V - C_{ek}, 0)])_{S1} = 0$, then M_1 is not likely to payoff relative to M_0 , as the flexibility of the architecture in response to k is not likely to add a value for S_1 , if k need to be exercised. Two interpretations are possible: (1) the architecture is overly flexible; its response to the change(s) has not “pulled” the options. This implies that the embedded flexibility or the resources invested in implementing flexibility are wasted to reveal the options relative to the change. (2) The architecture is inflexible relative to the change; the cost of accommodating the change on S_1 is relatively high.

Let CORBA and J2EE correspond to M_0 and M_1 respectively. S_0 and S_1 refer to the Duke’s Bank architecture when respectively induced by M_0 and M_1 . We use \$6,000 for man-month to cast the effort into cost. We construct a call option for the future scalability change, where the change is analogous to buying an “architectural potential”, paying an exercise price. We focus our attention on the payoff of the call options (i.e., $\sum_{i=1\dots n} E[\max(x_i V - C_{ei}, 0)]_{S1}$ relative to $\sum_{i=1\dots n} E[\max(x_i V - C_{ei}, 0)]_{S0}$), as they are revealing for the flexibility of the architecture-induced in responding to the likely future changes.

Estimating (C_{ei}): The exercise price corresponds to the cost of implementing replication to scale each structure, given by C_{ei} for requirement i . As the replicas may need to be run on different hosts, we devise a general model for calculating C_e as a function of the number of hosts, given by:

$$C_e = \sum_{h=1\dots k} (C_{dev}, C_{config}, C_{deploy}, C_{licesh}, C_{hardw})_h \quad (20.2)$$

h corresponds to the number of hosts. C_{dev} , C_{config} , and C_{deploy} , respectively correspond to the cost of development, configuration, and deployment for the replica on host h . C_{licesh} and C_{hardw} respectively correspond to licenses and hardware costs, if any, given in (\$). We provide three values: optimistic, likely, and pessimistic for each parameter, calculated using COCOMO II [16]. Upon varying the number of hosts, we report on pessimistic values, for they are more revealing. For simplification, we ignore any associated hardware costs.

Estimating ($x_i V = x_i V_{S1/S0}$): To value the “architectural potential” of S_1 relative to S_0 , we use the expected savings in development, configuration, and deployment efforts, when replication need to be accommodated on S_1 relative to S_0 , and respectively denoted as $\Delta_{S1/S0} C_{dev}$, $\Delta_{S1/S0} C_{config}$, $\Delta_{S1/S0} C_{deploy}$. Relative savings in licenses and hardware may also be considered and respectively denoted by ΔC_{licesh} , ΔC_{hardw} .

$$x_i V_{S1/S0} = \sum_{h=1\dots k} (\Delta_{S1/S0} C_{dev}, \Delta_{S1/S0} C_{config}, \Delta_{S1/S0} C_{deploy}, \Delta_{S1/S0} C_{licesh}, \Delta_{S1/S0} C_{hardw})_h \quad (20.3)$$

The savings, however, are uncertain and differ with the number of hosts, as the replicas may need to be run on different hosts. Such uncertainty makes it even more appealing to use “options thinking”. The valuation using ArchOptions is flexible to result in a comprehensive solution that incorporates multiple valuation techniques, some with subjective estimates, and others based on market data, when available. The problem associated with how to guide the estimation in this setting, we term as a *multiple perspectives valuation problem*. To introduce discipline into this setting and capture the value from different perspectives, we had suggested valuation points of view (i.e., market or subjective estimates) as a solution. The framework is comprehensive enough to account for the economic ramifications of the change, its global impact on the architecture, and on other architectural qualities. The solution aims to promote flexibility through incorporating both subjective estimates and/or explicit market value, when available. It is worth noting that for this case we only report on the structural maintainability value point of view. Nevertheless, the valuation could be extended to include other valuation dimensions.

Calculating the volatility (σ). Volatility is a quantitative expression of risk. Volatility is often measured by standard deviation of the rate of return on an asset price S (i.e., x_iV) over time. Unlike with financial options, in real options the volatility of the underlying asset’s value cannot be observed and must be estimated. During the evaluation of architectural stability, it is anticipated and even expected that stakeholders might undervalue or overvalue the architectural potential relative x_iV to the change in requirement(s). In other words, stakeholders tend to be uncertain about such value. For example, back to the motivating example of Sect. 20.4, suppose that the value of the architectural potential of inducing an architecture with J2EE and not CORBA (or perhaps vice versa) take the form of relative savings in development and configuration effort, if the future change in scalability need to be exercised on the induced structure: estimating such savings may vary from one architect to another within the firm. It differs with the architect’s experience, the novelty of the situation; consequently, it could be overvalued or undervalued. The variation in the future savings, hence, determines the “cone of uncertainty” in the future value of the architectural potential for embarking on a J2EE-induced architecture relative to the CORBA one. Thus, it is reasonable to consider the uncertainty of the architectural potential to correspond to the volatility of the stock price. In short, the volatility σ tends to provide a measure of how uncertain the stakeholders are about the value of the architectural potential relative to change; it tends to measure fluctuation in the said value. Volatility stands for the “fluctuation” in the value of the estimated x_iV . We take the percentage of the standard deviation of the three x_iVs estimates—the optimistic, likely, and pessimistic values to calculate σ .

Exercise time (t). The risk-free rate is a theoretical interest rate at which an investment may earn interest without incurring any risk. An increase in the risk-free interest rate leads to an increase in the value of the option. Finding the correspondence of this parameter is not straightforward, for the concept of interest in the architectural context does not hold strongly (as it is the case in the financial world) and is situation dependent. In our analogy, we set the risk-free interest rate to zero

assuming that value of the architectural potential is not affected by factors that could lead to either earning or depreciation in interest. That is, the value of architectural potential today is that of the time of exercising the flexibility option. However, we note that it is still possible for the analyst to account for this value, when applicable. For example, if the architectural platform is correlated in a way with the market, then the value of the architectural potential may increase or decrease with the market performance of the said platform. We set the exercise time to 1 year, assuming that the Duke's Bank needs to accommodate the change in 1 year time. We set the free risk interest rate to zero assuming the value of money today is the same as that of 1 year's time.

20.2.6 Tradeoffs Analysis

Scenario 1. We use JBoss, an open source, as M_1 with $C_{licesh} = 0$. Consider the case of running the replicas on one host. Table 20.2 shows that the overall expected savings $x_i V_{S1/S0}$ of inducing the structure with S_1 relative to S_0 are in the range of \$96,450(pessimistic) to \$150,704 (optimistic) for realizing the scalability requirements. As far as the development effort is concerned, expected savings are in the range of \$96,481(pessimistic) to \$ 150,753 (optimistic). As far as configuration effort is concerned, S_1 has not reported any expected savings relative to S_0 . However, the difference is insignificant. As for the effort of deployment, both are comparable when it comes to SLOC.

Let us use $x_i V_{S1/S0}$ to quantify the added value, taking the form of options due to the embedded flexibility on S_1 relative to S_0 . Table 20.2 shows that S_1 is in the money relative to the development, configuration, and the deployment and when compared to S_0 . Table 20.2 reads that inducing the architecture with M_1 is likely to enhance the option value by an excess of \$96,450 (pessimistic) to \$150,704 (optimistic) over S_0 , if the change in scalability need to be exercised in 1 year

Table 20.2 The options in (\$) on S_1 relative to S_0 for one host, with S_1 license cost ($C_{licesh} = 0$)

		Cei	$x_i V_{S1/S0}$	σ	t	Options
Over all	Optimistic	1,558	96,450			94,892
	Likely	1,948	120,563			118,615
	Pessimistic	2,435	150,704	22.7	1	148,269
	Opt	0	96,481			96,481
Development	Likely	0	120,602			120,602
	Pes.	0	150,753	22.7	1	150,753
	Opt	1,558	-31			0
Configuration	Likely	1,948	-39			0
	Pes.	2,435	-49	22.7	1	0
	Opt	0	0			0
Deployment	Likely	0	0			0
	Pes.	0	0	22.7	1	0

time. Thus, S_1 induced by M_1 is likely to add more value in the form of options relative to the change, when compared to S_0 . Note that, though S_1 is flexible relative to the scalability change, it might not necessarily mean that it might be flexible with respect to other changes. Obviously, J2EE does provide the primitives for scaling the software system, which result in making the architecture of the software system more flexible in accommodating the change in scalability, as when compared to the CORBA version. Calculating the options of S_0 relative to S_1 , S_0 is said to be out of the money for this change. The CORBA version has not added value, relative to J2EE as the cost of implementing the change was relatively significant to “pull” the options.

Scenario 2. We use WebLogic server [<http://www.bea.com/>] as M_1 with an average upfront payable license cost $C_{licesh} = \$25,000/\text{host}$. As an upfront license fee is incurred, increasing the number of hosts may carry unnecessary expenditures that could be avoided, if we adopt M_0 instead. However, M_0 does also incur costs to scale the system, due to the development of the load balancing and the fault tolerance services. Such a cost, however, maybe “diluted” as the number of hosts increases. The cost will be distributed across the hosts and incurred once, as the developed services can be reused across other hosts. An additional configuration and deployment cost materializes per host and sum up to C_e (2), when an additional host is needed to run a replica.

We calculate $x_i V_{S0/S1}$ using (3) and then the options of S_0 relative to S_1 . We adjust the options by subtracting the upfront expenditure of developing both services on M_0 , as reported in Table 20.3. The adjusted options reveal situations in which S_0 is likely to add value relative to S_1 , when the upfront cost is considered. These results may provide us with insights on the cost effectiveness of implementing fault tolerance and load balancing support to scale the software system relative to S_1 , where a licensing cost is incurred per host.

Therefore, a question of interest is: when is it cost effective to use M_0 instead of M_1 ? When does the flexibility of M_1 cease to create value relative to M_0 ? We assume that for any k hosts, S_0 and S_1 are said to support $U_k S_0$ and $U_k S_1$ concurrent users, respectively with $U_k S_0$ equal or different to $U_k S_1$. For the non-adjusted options results of Table 20.3, if we benchmark these options values against the cost of developing the load balancing and fault tolerance services (i.e., the upfront

Table 20.3 Options in (\$) on S_0 relative to S_1 with ($C_{licesh} = \$25,000$, $\sigma = 22.7$, and pessimistic C_{ei})

h	C_{ei}	$x_i V_{S0/S1}$	Options	Adjusted options	Conc. users
1	2,386	25,049	2,343	0	$U1S_0$ vs $U1S_1$
2	4,772	50,049	4,772	0	$U2S_0$ vs $U2S_1$
3	7,158	75,049	67,891	0	$U3S_0$ vs $U3S_1$
4	9,544	100,049	90,505	0	$U4S_0$ vs $U4S_1$
5	11,930	125,049	113,119	0	$U5S_0$ vs $U5S_1$
6	14,316	150,049	135,733	0	$U6S_0$ vs $U6S_1$
7	16,702	175,049	158,347	7,643	$U7S_0$ vs $U7S_1$

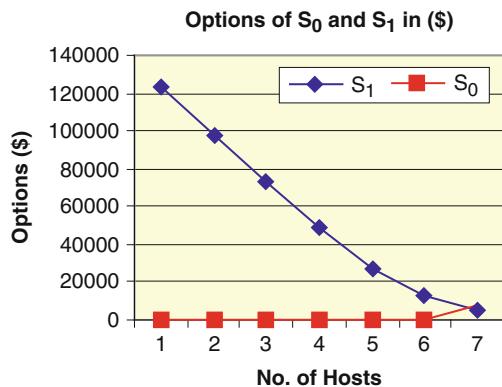
cost), we can see that payoff following developing these services is far from breaking even for less than 7 hosts. Once we adjust the options to take care of the upfront cost of investing to implement the both services, the adjusted options for S_0 relative to S_1 reports values in the money for the case of seven or more hosts, as shown in Table 20.3. For hosts ≥ 7 , M_0 appears to be a better choice under the condition that $U_nS_0 \geq U_nS_1$. This is due to the fact the expenditures in M_1 licenses increases with the number of hosts, henceforth, the savings in adopting M_1 cease to exist. For hosts < 7 , M_1 has better potentials and appears to be more cost-effective under the condition that $U_nS_1 \geq U_nS_0$.

20.3 Discussion

The change impact analysis has shown that the architectural structure of S_1 is left intact when the scalability change needs to be accommodated. However, the structure of S_0 has undergone some changes, mostly on the architectural infrastructure level to accommodate the scalability requirements. From a value-based perspective, the search for a potentially stable architecture requires finding an architecture that maximizes the yield in the added value, relative to some future changes in requirements. As we are assuming that the added value is attributed to flexibility, the problem becomes selecting an architecture that maximize the yield in the embedded or adapted flexibility in a software architecture relative to these changes. Even, if we accept the fact that modifying the architecture or the infrastructure is the only solution towards accommodating the change, analyzing the impact of the change and its economics becomes necessary to see how far we are expending to “re-maintain” or “re-achieve” architectural stability relative to the change. Though it might be appealing to the intuition that the “intactness” of the structure is the definitive criteria for selecting a “more” stable architectures, the practice reveals a different trend; it boils down to the potential added value upon exercising the change.

For the first scenario, the flexibility has yielded a better payoff for S_1 than for S_0 , while leaving S_1 intact. However, the situation and the analysis have differed upon varying the number of hosts and upon factoring a license costs for S_1 . Though S_0 has undergone some structural changes to accommodate the change, the case has shown that it is still acceptable to modify the architecture and to realize added value under the conditions that $U_nS_0 \geq U_nS_1$ for seven or more hosts (Table 20.3, Fig 20.3). Hence, what matters is the added value upon either embarking on a “more” flexible architecture, or investing to enhance flexibility which is the case for implementing load balancing and fault tolerance on S_0 . For the case of WebLogic, though M_1 is in principle more flexible, the flexibility comes with a price, where the flexibility turned to be a liability rather than a value for seven or more hosts, as when compared with the JacORB, under the condition that $U_nS_0 \geq U_nS_1$.

Fig. 20.3 Options on S_0 and S_1 upon varying the number of hosts



Observation 1. The “coupling” between middleware and architecture helps in understanding evolution trends of distributed software architectures and in connection with the solution domain.

Our hypothesis that middleware induced-software architectures differ in coping with changes is verified to be true for the given change. Based on the previous observations, we can see that the stability of S_1 and S_0 appears to be dependent on the flexibility of the middleware in accommodating the likely changes in the scalability requirements. For the category of distributed software systems that are built on top of middleware, the results of the case study affirm the belief that investigating the stability of the distributed software architecture could be fruitless, if done in isolation of the middleware, where the middleware constraints and dominate much of the solution that relate to the non-functionalities, managing system resources, and their ability to smoothly evolve over the life time of the software system. Hence, the development and the analysis for architectural stability and evolution shall consider the “coupling” between the architecture and the middleware. This addresses pragmatic needs and is feasible even at earlier stages of the software development life cycle: a considerable part of the distributed system implementation could be available, when the architecture is defined, for example, during the Elaboration phase of the Unified Process. We also note that the change in requirements could have been addressed by other architectural mechanisms. However, the middleware has guided the solution for evolving the software system. For instance, the choice of replication as an architectural mechanism for scaling the software system, with a given architectures S_1 and S_0 was respectively guided by the clustering primitives provided by M_1 and the core capabilities provided by M_0 to support load balancing and fault tolerance. Interestingly, Di Nitto and Rosenblum [3] state that “despite the fact that architectures and middleware address different phases of software development, the usage of middleware and predefined components can influence the architecture of the system being developed. Conversely, specific architectural choices constrain the selection of the underlying middleware used in the implementation phase”. Medvidovic, Dashofy and Taylors [17] take the idea of coupling the modeling power of software architectures with the

implementation support provided by middleware. They noted, “architectures and middleware address similar problems, that is large-scale component-based development, but at different stages of the development life cycle.” In more abstract terms, Rapanotti, Hall, Jackson, and Nuseibeh [18] advocate the use of information in the solution domain (e.g., the middleware-to be induced for our case) to inform the problem space:

Whereas Problem Frames are used only in the problem space, we observe that each of these competing views uses knowledge of the solution space: the first through the software engineer’s domain knowledge; the second through choice of domain-specific architectures, architectural styles, development patterns, *etc.*; the third through the reuse of past development experience. All solution space knowledge can and should be used to inform the problem analysis for new software developments within that domain [18].

The “coupling” between the middleware and the architecture becomes of higher interest in case of developing and analyzing software systems for evolution. This is because the solution domain can guide the development and evolution of the software system; provide more pragmatic and deterministic knowledge on the potential success (failure) of evolution, and consequently assist in understanding the stability of the software architectures from a pragmatic perspective.

Observation 2. Understanding architectural stability and evolution in relation to styles.

Following the definition of Di Nitto and Rosenblum [3], a *style* defines a set of general rules that describe or constrain the structure of architectures and the way their components interact. Styles are a mechanism for categorizing architectures and for defining their common characteristics. Though S_1 and S_0 have exhibited similar styles (i.e., three-tier), they have differed in the way they cope with the change in scalability. The difference was not only due to the architectural style, but also due to the primitives that are built-in in the middleware to facilitate scaling the software system. The governing factor, hence, appears to be to a large extent dependent on the flexibility of the middleware (e.g., through its built-in primitives) in supporting the change. The intuition and the preliminary observations, therefore, suggest that the style by itself is not revealing for the stability of the software architecture when the non-functional requirements evolve. It is, however, a factor of the extent to which the middleware primitives can support the change in non-functional requirements. Interestingly, Sullivan et al. [19] claims that for a system to be implemented in a straightforward manner on top of a middleware, the corresponding architecture has to be compliant with the architectural constraints imposed by the middleware. Sullivan et al. [19] support this claim by demonstrating that a style, that in principle seems to be easily implementable using the COM middleware, is actually incompatible with it. Following a similar argument, adopting an architectural style that in principle appear to be suitable for realizing the non-functionality and supporting its evolution, may not be compliant with the middleware in the first place. And if the architectural style happens to be compliant with the middleware, there are still uncertainties in the ability of the middleware primitives to support the change. In fact, the middleware primitives realize much of the

non-functional requirements. Hence, the architectural style by itself may not be revealing for potential threats that the architecture may face when the non-functional requirements evolve. The evolution of non-functionality maybe in principle easily supported by the style, but could be uneasily accommodated by the middleware. An observable advantage of scaling the software architecture induced by S_1 , for example, is that no development effort required to realize the scalability requirements through replication, as when compared to that of S_0 , knowing that in principle the style of S_1 and S_0 exhibit similar capabilities. Engineering for stability and evolution, requirements engineering has not only to be aware of the architecture (e.g., the style), but also of the underlying middleware. For example, if we take a goal-oriented approach to requirements engineering we advocate adjusting the non-functional requirements elicitation and their corresponding refinements to be aware of both the architectural style and the constraints imposed by middleware. The operationalisation of these requirements in the software architecture have to be guided by both the architectural style, the complaint middleware for the said style, and guided by previous experience. This, we believe, is a pragmatic need towards engineering requirements and developing “evolvable” software architectures that tend to be stable as the non-functional requirements evolve.

20.4 Closely Related Work

The use of real options in software engineering. Economics approaches to software design appeal to the concept of static Net Present Value (NPV) as a mechanism for estimating value. These techniques, however, are not readily suitable for strategic reasoning of software development as they fail to factor flexibility [20]. The use of strategic flexibility to value software design decisions has been explored in, for example [11, 12] and real options theory has been adopted to value the strategic flexibility: Baldwin and Clark [21] studied the flexibility created by modularity in design of components (of computer systems) connected through standard interfaces. Sullivan et al. [22] pioneered the use of real options in software engineering. Sullivan et al. suggested that real options analysis can provide insights concerning modularity, phased projects structures, delaying of decisions and other dynamic software design strategies. They formalized that option-based analysis, focusing in particular on the flexibility to delay decisions making. Sullivan et al. [23] extended Baldwin and Clark’s theory, which is developed to account for the influence of modularity on the evolution of the computer industry. They treat the “evolvability” of software design using the value of strategic flexibility. Specifically, they argued that the structure and value of modularity in software design creates value in the form of real options. A module creates an option to invest in a search for a superior replacement and to replace the currently selected module with the best alternative discovered, or to keep the current one if it is still the best choice. The value of such an option is the value that could be realized by the optimal experiment-and-replace policy. Knowing this value

can help a designer to reason about both investment in modularity and how much to spend searching for alternatives.

Architectural evaluation. Existing methods to architectural evaluation have ignored any economic considerations, with CBAM [24] being the notable exception. The evaluation decisions using these methods tend to be driven by ways that are not connected to, and usually not optimal for value creation. Factors such as flexibility, time to market, cost and risk reduction often have higher impacts on value creation. Hence, flexibility is in the essence. In our work, we link flexibility to value, as a way to make the value of stability tangible.

Relating CBAM to our work, the following distinctions can be made: with the motivation to analyse the cost and benefits of architectural strategies, where an architecture strategy is subset of changes gathered from stakeholders, CBAM does not address stability. Further, CBAM does not tend to capture the long-term and the strategic value of the specified strategy. ArchOptions, in contrast, views stability as a strategic architectural quality that adds to the architecture values in the form of *growth options*. When CBAM complements ATAM [25] to reason about qualities related to change such as modifiability, CBAM does not supply rigorous predictive basis for valuing such impact. Plausible improvements of the existing CBAM include the adoption of real options theory to reason about the value of postponing investment decisions. CBAM uses real options theory to calculate the value of option to defer the investment into an architectural strategy. The delay is based on cost and benefit information. In the context of the real options theory, CBAM tends to reason about the *option to delay* the investment in a specific strategy until more information becomes available as other strategies are met. ArchOptions, in contrast, uses real options to value the flexibility provided by the architecture to expand in the face of evolutionary requirements; henceforth, referred to as the options to expand or growth options.

20.5 Conclusion

Though the reported observations reveal a trend that agrees with the intuition, research, and the state-of-practice, confirming the validity of the observations are still subject to careful further empirical studies. These studies may need to consider other non-functional requirements, their concurrent evolution, and their corresponding change impact on different architectural styles and middleware. As a limitation, we have relaxed considering the change impact of scaling up the software system on other non-functional requirements like security, availability and reliability. However, we note that the analysis might get complex upon accounting for the impact of the change on other non-functional requirements and their interactions. Note the change could positively or negatively impact other non-functional requirements and understanding the cost implications is not straightforward and worth a separate empirical investigation. In this context, utilizing the NFR framework [26] could be promising to model the interaction of various non-functional requirements, their corresponding

architectural decisions, and the negative/positive contribution of the architectural decisions in satisfying these non-functionalities. The framework could be then complemented by means for measuring (1) the corresponding cost of implementing the change itself, and (2) the additional cost due to the impact of the change on other contributing or conflicting non-functionalities, as realized by either the CORBA or the J2EE middleware-induced architectures.

It is also worth noting that the investment decision in either CORBA or the J2EE might be influenced by other factors, such as the skills of the developers, the project maturity, and other organizational factors. The devised real options model does not explicitly take into account these factors. The treatment of these factors is left implicit and sufficiently addressed by our use of COCOMO II, where COCOMO II carries parameters to adjust the cost estimates based on these factors. It could be also argued that in iterative development, when estimations are continuously recalibrated (e.g., in the Unified Process), it is possible to come up with estimations that are more accurate than COCOMO II, as they will take into account the above mentioned factors.

We note that the flexibility of either solutions (i.e., the CORBA or the J2EE induced-architectures) is closely tied to the problem domain. In particular, domain-specific functional characteristics can also influence the flexibility of the solution and its behavior, as both the application component and the infrastructure are tightly coupled. The way the application components and the infrastructure are coupled varies across various middlewares. For this study, the functional characteristics are assumed to be stable for both the J2EE and the CORBA versions; that is, they have not undergone any changes that require from us understanding the impact of the functionality change on the flexibility of either solutions. It will be interesting, however, to investigate how changes in the domain functional characteristics can impact the flexibility and the stability of the middleware-induced architectures.

In short, the choice of the distributed software system architecture has to be guided by the choice of the underlying COTS middleware and its flexibility in responding to future changes in non-functional requirements. This is necessary to facilitate the evolution of the software system and to avoid unnecessary future investments. Unfortunately, non-functional requirements tend to be unstable and evolve over time to threaten the stability of the software system. Hence, there is a need for flexible COTS middleware architectures that tend to be stable as requirements evolve. But flexibility comes with a price. The economic interplay between evolving requirements and architectural flexibility is the governing factor that determines the cost-effectiveness of the middleware choice. No research effort has been devoted for understanding the evolution of non-functional requirements in relation to both the architecture and the COTS middleware when coupled, with our work [10] being the notable exception. Future research focus may need to consider other non-functional requirements, their concurrent evolution, and their corresponding structural and run-time change impact on different architectural styles and middleware, which we are investigating as part of our ongoing research agenda.

References

1. Sun MicroSystems Inc (2002) Enterprise javaBeans specification v2.1, Copyright© 2002 Sun Microsystems, Inc.
2. Object Management Group (2000) The common object request broker: architecture and specification, 24th edn. OMG, Framingham, MA, USA
3. Di Nitto E, Rosenblum D (1999) Exploiting ADLs to specify architectural styles induced by middleware infrastructures. In: Proceedings of the 21st International conference on software engineering. IEEE Computer Society Press, Los Angeles, pp 13–22
4. Dawson R, Bones P, Oates B, Brereton P, Azuma M, Jackson M (2003) Empirical methodologies in software engineering In: Eleventh annual International workshop on software technology and engineering practice, IEEE CS Press, pp 52–58
5. Emmerich W (2000) Software engineering and middleware: a road map. In: Finkelstein A (ed) Future of software engineering. Limerick, Ireland
6. Dardenne A, van Lamsweerde A, Fickas S (1993) Goal-directed requirements acquisition. *Sci Comput Program* 20:3–50
7. Anton A (1996) Goal-based Requirements Analysis. In: Proc. 2nd IEEE Int. Conf. Requirements Engineering. Orlando, USA
8. Othman O, O’Ryan C, Schmidt DC (2001) Designing an Adaptive CORBA Load Balancing Service Using TAO. *IEEE Distributed Systems Online* 2(4)
9. Othman O, O’Ryan C, Schmidt DC (2001) Strategies for CORBA Middleware-Based Load Balancing. *IEEE Distributed Systems Online* 2(3)
10. Bahsoon R, Emmerich W, Macke J (2005) Using real options to select stable middleware-induced software architectures. *IEE Proc Softw Spec* issue relating software requirements architectures 152(4):153–167, IEE press, ISSN 1462–5970
11. Erdoganmus H, Boehm B, Harrioss W, Reifer DJ, Sullivan KJ (2002) Software engineering economics: background, current practices, and future directions. In: Proceeding of 24th International conference on software engineering. ACM Press, Orlando
12. Erdoganmus H (2000) Value of commercial software development under technology risk. *Financier* 7(1–4):101–114
13. Schwartz S, Trigeorgis L (2000) Real options and investment under uncertainty: classical readings and recent contributions. MIT Press, Cambridge
14. Bahsoon R, Emmerich W (2004) Evaluating architectural stability with real options theory. In: Proceedings of the 20th IEEE International conference on software maintenance. IEEE CS Press, Chicago
15. Bahsoon R, Emmerich W (2003) ArchOptions: a real options-based model for predicting the stability of software architecture. In: Proceedings of the Fifth ICSE workshop on economics-driven software engineering research, Portland
16. Boehm B, Clark B, Horowitz E, Madachy R, Shelby R, Westland C (1995) The COCOMO 2.0 software cost estimation model. In: International society of parametric analysts
17. Medvidovic N, Dashofy E, Taylor R (2003) On the role of middleware in architecture-based software development. *Int J Software Engineer Knowledge Engineer* 13(4):367–393
18. Rapaport L, Hall J, Jackson M, Nuseibeh B (2004) Architecture driven problem decomposition. In: Proceedings of 12th IEEE International requirements engineering conference (RE’04). IEEE Computer Society Press, Kyoto
19. Sullivan KJ, Socha J, Marchukov M (1997) Using formal methods to reason about architectural standards. In: Proceedings of the 19th International conference on software engineering, Boston
20. Boehm B, Sullivan KJ (2000) Software economics: a roadmap. In: Finkelstein A (ed) The future of software engineering. ACM Press, Lemrick, Ireland
21. Baldwin CY, Clark KB (2001) Design rules – the power of modularity. MIT Press, Cambridge
22. Sullivan KJ, Chalasani P, Jha S, Sazawal V (1999) Software design as an investment activity: a real options perspective. In: Trigeorgis L (ed) Real options and business strategy: applications to decision-making. Risk Books, London

23. Sullivan KJ, Griswold W, Cai Y, Hallen B (2001) The structure and value of modularity in software design. In: The Proceedings of the ninth ESEC/FSE, Vienna, pp 99–108
24. Asundi J, Kazman R (2001) A Foundation for the Economic Analysis of Software Architectures. In: Proceedings of the Third Workshop on Economics-Driven Software Engineering Research
25. Kazman R, Klein M, Barbacci M, Lipson H, Longstaff T, Carrière SJ (1998) The architecture tradeoff analysis method. In: Proceedings of fourth International conference on engineering of complex computer systems (ICECCS '98). IEEE CS Press, Monterey, pp 68–78
26. Mylopoulos J, Chung L, Nixon B (1992) Representing and using nonfunctional requirements: a process-oriented approach. *IEEE Trans Software Eng* 18(6):483–497
27. Black F, Scholes M (1973) The pricing of options and corporate liabilities. *J Polit Econ* 81(3): 637–654
28. Emmerich W (2002) Distributed component technologies and their software engineering implications. In: Proceedings of the 24th International conference on software engineering. ACM Press, Orlando, pp 537–546
29. Nuseibeh B (2001) Weaving the software development process between requirements and architectures. In: Proceedings of STRAW 01 the First International workshop from software requirements to architectures, Toronto

Chapter 21

Conclusions

P. Avgeriou, P. Lago, J. Grundy, I. Mistrik, and J. Hall

21.1 Issues and Trends in Relating Requirements Engineering and Architecture

The preceding chapters in this book address a plethora of intriguing challenges, all directed towards bridging the gap between requirements engineering and architecture. These challenges, put together, capture a large part of the problem space currently faced by architects and requirements engineers alike. We review the various challenges that have been discussed in the preceding chapters. We present an abstracted form of the emerging trends that have been identified to deal with these challenges, referring back to individual chapters in order to exemplify them.

Requirements and architectures are, by and large, based on different metamodels which are problematic to semantically associate, e.g. use cases are semantically distant to object-oriented classes and objects, even though use cases are usually refined and elaborated through class models. One of the most popular approaches in bridging the gap between the metamodels of requirements and architecture is the **goal-oriented paradigm**. By expressing requirements as goals to be achieved and gradually refining them through tasks, agents and other elements, a transformation of problem space to solution elements can be achieved. Different ways have been proposed to relate goal elements to elements of the architecture. This gives a good indication of the semantic proximity between the concepts of goals, agents and architecture concepts like components.

This book contains two examples of goal-oriented approaches. Lawrence Chung, Sam Supakkul, Nary Subramanian, Jose Luis Garrido, Manuel Noguera, Maria V. Hurtado, Maria Luisa, Rodriguez, and Kawtar Benghazi (Chap. 7) propose an approach that transforms goal models into architecture models through a number of intermediate transformations. These transformations allow requirements engineers and architects to explicitly reason about the relationships between these problem space goals and objectives, and realizing components within the architecture. Luciano Baresi and Liliana Pasquale (Chap. 10) extend goal models with the notion of adaptive goals. They then use these augmented goal models to automatically create

the architecture of adaptive, service-centric applications. Tool support ensures the identified adaptive goals are indeed realized in appropriate architectural solution space components.

One of the ‘wholly grails’ of the software engineering community has been the establishment of **traceability** between the artifacts produced and consumed across the various software lifecycle activities. In this vein, traceability is one of the key tools in relating the requirements engineering and architecting processes as it links the products of these two activities. Of course traceability can also be established between the processes themselves (e.g. the V model), but most approaches focus on artifact traceability since the latter is more tangible and explicit. Traceability is usually applied among elements that belong to different models (or different views) by connecting the corresponding metamodels.

A promising approach in this direction is the one proposed by Jochen M. Kuster, Hagen Volzer and Olaf Zimmermann (Chap. 15). They suggest classifying all software engineering artifacts in a matrix with at least two suggested dimensions, the viewpoint and the realization level they belong to. This classification allows for the establishment of traceability links between artifacts and across dimensions. Another popular way to enhance traceability is through model-driven engineering. Huy Tran, Ta’id Holmes, Uwe Zdun and Schahram Dustdar (Chap. 14) use principles of model-driven architecture to explicitly trace requirements, design and code. Antony Tang, Peng Liang, Viktor Clerc and Hans van Vliet (Chap. 4) propose a general purpose ontology that captures the traceability links between co-evolving architectural requirements and design. By using semantic wiki technology they show how co-evolution can be managed in a number of typical scenarios relevant for both business analysts and designers. Inah Omoronyia, Guttorm Sindre, Stefan Biffl and Tor Stålhanne (Chap. 5) propose an approach to automatically harvest traceability networks. These then help engineers in making architectural knowledge explicit and facilitate understanding and reasoning. Finally, Rami Bahsoon and Wolfgang Emmerich (Chap. 20) describe an economics-driven approach based on “real options” theory. Their approach relates non-functional requirements to middleware for the selection of architectural decisions and allows economic decisions and architecture evolution to be considered in the architecting and requirements engineering process.

Another way to link the activities of requirements engineering and architecting is through **specific problem areas** that occur in certain systems or application domains, or in certain life cycle stages. A fine example of such a problem area is the volatility that occurs both in requirements specification (where requirements are uncertain and their values may range) and architecture design (where a product line may allow for multiple options). In such cases it can be very rewarding to combine approaches from both fields in order to provide a seamless transition from problems to solutions.

Zoë Stephenson, Katrina Attwood and John McDermid (Chap. 8) describe techniques for addressing requirements uncertainty. Uncertainty and product line engineering are combined in order to translate volatility of requirements into a choice of design alternatives. Soo Ling Lim and Anthony Finkelstein (Chap. 3)

investigate the problem of managing change by structuring requirements into layers depending on their expected change rate. In this way, the resulting design and implementation can better cope with evolution and the impact of changed requirements on architecture is minimized. Finally, Outi Räihä, Hadaytullah, Kai Koskimies and Erkki Mäkinen (Chap. 18) focus on modifiability, efficiency and simplicity. They present an innovative method based on genetic algorithms to generate a software architecture from functional and quality requirements.

One of the areas where good progress has been made in bringing closer the fields of requirements engineering and architecture was the design and use of **reference architectures, reference models and patterns**. In mature domains where numerous successful systems have been developed, the set of requirements is well-understood and can be communicated across practitioners. Similarly the design solutions that have been applied in order to satisfy those requirements are also documented and reused across different systems. This is one of the few cases where the match between the problem and the solution space is explicit, mature and offers fruitful design reusability. We find the documentation of reference architectures and architecture or design patterns a very promising approach in the topic of this book and encourage the community to mine for patterns across systems in different application domains and if possible compile the reusable design into reference architectures.

The approach proposed by Tim Trew, Goetz Botterweck and Bashar Nuseibeh (Chap. 13) is an excellent example of a process to develop with a reference architecture even in challenging domains. In their work the target domain is Consumer Electronics and their approach guides requirements engineers and architects in this area. Christine Choppy, Denis Hatebur and Maritta Heisel (Chap. 9) use problem frames as reusable patterns that express the problem. They then gradually transform these problem frame descriptions into architecture models that can be implemented to realize the requirements.

In software development practice, the need to relate requirements engineering and architecture compels practitioners to find **pragmatic ways** that can achieve the desired results. Such emerging approaches may not be accompanied by rigorous empirical evidence, but they do provide value in practice and they do help architects and requirements engineers to collaborate more efficiently. For example Michael Stal (Chap. 15) proposes an architecting method that is composed of a set of best practices within the industry based on the principle: requirements drive the architecture. Successive layers of an architecture – the Onion model – are used, along with a set of best-practice process steps, to develop the architecture from requirements. On a similar vein Bass and Clements (Chap. 11) emphasize the need to link architecture to a set of requirements that often do not become explicit: business goals. Similarly Eoin Woods and Nick Rozanski (Chap. 19) also propose starting from business drivers. However, they support requirements engineers, architects and project managers in working closely together in allowing the architecture to constrain, frame and inspire the requirements – the reverse of traditional requirements-to-architecture system engineering.

21.2 Looking Ahead

Software requirements and architecture are crucial software engineering artifacts that are tightly inter-dependent but have been historically researched in separate communities. In this publishing endeavor one of our aims has been to bring together experts from both communities and collect their work in bridging requirements and architecture, the associated knowledge, processes and tools. Despite the progress that had been achieved in relating requirements and architecture, the mindset of software engineering practitioners has not shifted much towards bridging the gap. There is still a persistent boundary in practice between requirements and architecture, which is often difficult to cross, possibly hindering innovation. In fact, many organizations still have requirements teams and architecture teams with sometimes very rigid boundaries between their spheres of influence and interaction. We find this boundary manifesting in many forms: in the software life cycle models (even the more agile ones); in the types of software development artifacts; in the competencies and job profiles used in organizations; in the supporting tools; in the education and training of practitioners; and in the separation of problem- and solution space within the practitioners' line of thinking.

In a way, this boundary is the heritage of the waterfall model, a limitation of which we still suffer. According to this model of software engineering, as discussed in Chap. 1, requirements and architecture have been always considered as two different types of animals assigned to distinct development phases and different spaces (problem and solution). Experience tells that to innovate we need to look at current approaches with a critical eye, question them, and look at the problem they solve from totally different angles. Following this line of thought, what if we take a totally new perspective, and look at the so-called “early phases of the life cycle” as a continuum where needs and their support co-evolve in a shared decision space? What if we forget about this requirements-architecture boundary and look at the evolution of software systems as- a-whole across time and space?

Some works are already following this path. Inspired by the evolution in the way architecture is defined (from mere solution structure to a set of design decisions), De Boer and Van Vliet claim that “architectural design decisions and architecturally significant requirements are really the same; they’re only being observed from different directions” [3]. In discussing the considerable overlap between the two fields, they argue that we should focus much more on the commonalities and how and to what purposes we should support different perspectives. They further observe that “architecture is not merely the domain of the architect. Each architecturally significant requirement is already a decision that shapes the architecture”. This duality is also discussed in terms of roles by Abrahamsson et al. who describe the *agile* architect as among others both a designer making choices and a communicator facilitating requirements elicitation with customers and mediating between decision makers, project managers, and developers [1].

Another area still needing further development concerns the representation of architectural requirements and design. In spite of the existing substantial research

and available solutions, industrial needs are still quite unfulfilled. Practitioners typically codify information about requirements in a fuzzy way: notations are informal; natural language (occasionally with some standardized guidelines) is the most common practice. This “information gap” makes it impossible to *precisely* relate requirements and design, the latter being much more detailed and formalized. Hence, approaches that assume availability of precise information do not fill industrial practices. We think that future efforts should thus be striving for ways to relate just the *core* pieces of information, such as only those requirements/design decisions that are more costly to change and that need to be monitored, or those that recur more frequently in one’s specific industrial domain. Secondly, links between requirements and architecture design decisions are multi-dimensional (one requirement influences multiple decisions, and the other way round), often indirect (one requirements might influence one decision that on its own might reveal a new requirement, etc.) and of different nature [2]. This complex network of dependencies, well known in both requirements engineering and software architecture fields, makes the problem addressed by this book difficult to solve in ways that are also applicable in industrial practice. Industrial applicability should hence become a *must have* for future research in both requirements engineering and software architecture fields, as well as when relating software requirements and architecture.

We further observe increasing attention dedicated to the notion and role of ‘context’ (or environment where a software system lives). In his paper on past, present and future of software architecture, Garlan [4] identified three major trends influencing the way architecture is perceived: the market shifting engineering practices from build to buy; pervasive-computing challenging architecture with an environment of diverse and dynamically changing devices; and the Internet transforming software from closed systems to open and dynamically configurable components and services. These trends, among many others, make software contexts always smarter and increasingly complex, hence posing challenging research questions on the co-evolution of contextual (external) and system (internal) requirements and architectures.

Last but not least, there is one research area that has focused since its inception on the relation between requirements and architecture: enterprise architecture has been evolving for three decades around the premise of aligning business processes with the technical world. Business processes are supported by software systems, which in turn pose constraints and generate new requirements for the former. However, in spite of the great deal of research carried out since the 1970s, Business-IT alignment remains a major issue [5]. The current trend is to tackle the problem through governance of the software development process, as a mechanism to guarantee meeting the business goals and mitigating associated risks through policies, controls, measures. Governance works at the interface between the structure of the business organization and the structure of the software and thus includes both requirements and architecture within its scope. Governance is particularly relevant in distributed development environments, which face increased challenges, as requirement and architecture, are often produced in different sites.

References

1. Abrahamsson P, Ali Babar M, Kruchten P (2010) Agility and architecture: can they coexist? *IEEE Softw* 27(2):16–22
2. Berg Mvd, Tang A, Farenhorst R (2009) A constraint-oriented approach to software architecture design. In: Proceedings of the quality software international conference (QSIC 2009), IEEE, Jeju, pp 396–405
3. de Boer RC, van Vliet H (2009) On the similarity between requirements and architecture. *J Syst Softw* 82(3):544–550
4. Garlan D (2000) Software architecture: a roadmap. In: Proceedings of the conference on the future of software engineering, limerick. ACM, Ireland, pp 91–101. doi:10.1145/336512.336537
5. Luftman J, Papp R, Brier T (1999) Enablers and inhibitors of business-IT alignment. *Commun AIS* 1(3es):1–32

Editor Biographies

Paris Avgeriou is Professor of Software Engineering in the Department of Mathematics and Computing Science, University of Groningen, the Netherlands where he has led the Software Engineering research group since September 2006. Before joining Groningen, he was a post-doctoral Fellow of the European Research Consortium for Informatics and Mathematics (ERCIM). He has participated in a number of national and European research projects directly related to the European industry of Software-intensive systems. He has co-organized several international workshops, mainly at the International Conference on Software Engineering. He sits on the editorial board of Springer Transactions on Pattern Languages of Programming. He has published more than 90 peer-reviewed articles in international journals, conference proceedings and books. His research interests lie in the area of software architecture, with strong emphasis on architecture modeling, knowledge, evolution and patterns.

John Grundy is Professor of Software Engineering at the Swinburne University of Technology, Australia. He has published over 200 refereed papers on software engineering tools and methods, automated software engineering, visual languages and environments, collaborative work systems and tools, aspect-oriented software development, user interfaces, software process technology and distributed systems. He has made numerous contributions to the field of collaborative software engineering including developing novel process modeling and enactment tools; collaborative editing tools, both synchronous and asynchronous; thin-client project management tools; software visualization tools for exploratory review; sketching-based UML and software design tools providing hand-drawn diagram support for collaborative review and annotation; and numerous event-based collaborative software architectures for building collaborative software engineering tools. He has been Program Chair of the IEEE/ACM Automated Software Engineering conference, the IEEE Visual Languages and Human-Centric Computing Conference, and has been a PC member for the International Conference on Software Engineering.

Jon G. Hall is a Senior Academic at The Open University, UK. He is Editor-in-Chief of Expert Systems: The Journal of Knowledge Engineering, a Wiley-Blackwell journal, of IARIA's International Journal of Software Advances, and is a member of the editorial board of Inderscience's International Journal of Learning and Intellectual Capital. He's a Fellow of the BCS and of IARIA, a Chartered Information Technology Professional, Chartered Engineer and Chartered Scientist.

Jon is membership secretary of the British Computer Society's ELITE (Effective Leadership in IT) group, manages the associated ELITE on LinkedIn forum, and is a member of the International Advisory Board of the 2011 IFIP World CIO Forum, China. Jon has been head of Product Research for Tarmin Solutions Limited since 2002, and co-leads 20first's solvemehappy problem solving consultancy (see <http://www.solvemehappy.com>). Jon has published widely, including on Problem Solving; Models of Concurrency; Logic; Formal Methods; Software, Systems, Knowledge, Educational, and Business Process Engineering and Re-engineering; Software Architectures and Requirements Engineering; and Multi-Agent Systems. He has edited numerous special issues and books, including those associated with the International Conference on Software Engineering workshop series on Applications and Advances of Problem Orientation and Problem Frames (IWAAPo & IWAAPF). His current research mission is to provide new thought tools to business for sophisticated problem solving based on a theory of engineering design into which computing and the traditional engineering disciplines fit, and within which their interrelatedness and complementarity can be explored. The vehicle for this work is Problem Oriented Engineering (POE), a validated framework for engineering that currently covers mission-critical software, socio-technical systems, business processes, and education and training design.

Patricia Lago is associate professor at the VU University Amsterdam, The Netherlands. Her research interests are in software- and service oriented architecture, architectural knowledge management, green IT and sustainable software engineering. Lago has a PhD in Control and Computer Engineering from Politecnico di Torino. She co-edited various special issues and co-organized workshops on topics related to service-oriented and software architecture, architectural knowledge management, and green IT. She has been Program Chair of the IEEE/IFIP Working Conference on Software Architecture, and is member of IEEE and ACM, and of the International Federation for Information Processing Working Group 2.10 on Software Architecture. She is Area Editor Service Orientation for the Elsevier Journal of Systems and Software. She published over 100 peer-reviewed articles in international journals, conference proceedings and books, and co-edited the book on Software Architecture Knowledge Management published by Springer in 2009.

Ivan Mistrík is an independent consultant and researcher in software-reliant systems engineering. He is a computer scientist who is interested in software engineering (SE) and software architecture (SA), in particular: life cycle software engineering, requirements engineering, aligning enterprise/system/software architectures, knowledge management in software development, global

software development, and collaborative software engineering. He has more than forty years' experience in the field of computer systems engineering as an information systems developer, R&D leader, SE/SA research analyst, educator in computer sciences, and ICT management consultant. In the past 40 years, he has been primarily working at R&D institutions in USA and Germany and has done consulting on a variety of large international projects sponsored by ESA, EU, NASA, NATO, and UN. He has also taught university-level computer sciences courses in software engineering, software architecture, distributed information systems, and human-computer interaction. He is the author or co-author of more than 80 articles and papers in international journals, conferences, books and workshops, most recently a chapter Capture of Software Requirements and Rationale through Collaborative Software Development, a paper Knowledge Management in the Global Software Engineering Environment, and a paper Architectural Knowledge Management in Global Software Development. He has also written over 90 technical reports and presented over 70 scientific/technical talks. He has served in many program committees and panels of reputable international conferences and organized a number of scientific workshops, most recently two workshops on Knowledge Engineering in Global Software Development at International Conferences on Global Software Engineering in 2009 and in 2010 (proceedings of these workshops have been published by IEEE Computer Society, Conference Publishing Services). He has been a guest-editor of IEE Proceedings Software: A special Issue on Relating Software Requirements and Architectures published by IEE in 2005 and a lead-editor of the book Rationale Management in Software Engineering published by Springer in 2006. He has been a co-author of the book Rationale-Based Software Engineering published by Springer in May 2008. He has been a lead-editor of the book Collaborative Software Engineering published in 2010. Currently he is a lead editor of the Expert Systems Special Issue on Knowledge Engineering in Global Software Development.

Index

A

- Accidental complexity, 281
- ActiveBPEL, 169
- Adaptation, 162, 172
- Adaptation goals, 167
- Adapter, 316–318, 322, 325, 326
- Agile development and requirements and architecture, 337
- Agile process, 287
- ArchE, 307
- Architect, 183–185, 187–194
- Architectural decisions, 207, 211, 219
- Architectural design, 35
- Architectural design activities
 - architectural analysis, 91
 - architectural evaluation, 91
 - architectural synthesis, 91
- Architectural information needs, 62, 65
- Architectural knowledge, 38, 61, 211, 212
 - codification, 38
 - decisions, 38
 - personalisation, 38
- Architecturally relevant requirement, 279, 282
- Architectural separation of concerns (ASC), 106
- Architectural structure, 216, 221
- Architectural style, 99, 138, 308, 317
- Architectural texture, 216, 219, 222
- Architecture, 111, 121, 123, 129
 - baseline, 281
 - design, 279
 - patterns, 283
 - smells, 297
- Architecture constraining requirements, 341
- Architecture framing requirements, 340
- Architecture inspiring requirements, 341

Architecture trade-off analysis method

(ATAM), 106, 312

Artifact and model transformation (AMT)

Artifact and model transformation (AMT) matrix, 260

Artifact management

Attribute-driven design (ADD), 106

Automated code generation, 113

Automatic teller machine (ATM), 143

B

BAPO/CAFCR, 106

Big design upfront, 280

Black box view, 289

Bottom-up requirements, 285

BPEL, 161

Bunch tool, 315

Business drivers relating requirements and architecture, 337

Business drivers, requirements and architecture relationships, 338

Business goals, 183–195

Business policies and rules, 23

C

Call graphs, 61, 70

Centrality, 67

Changeability, 308

Change-oriented requirements engineering (CoRE), 18

Change scenarios, 309, 319, 322

Changes to the requirements, 111, 113

Chromosome, 310

Client-server, 317, 318, 322, 324, 326

- Code quality management (CQM), 297
 Collaboration awareness, 268
 Commitment uncertainty, 111, 112, 114, 115, 120, 123, 126, 130
 Common object request broker architecture (CORBA), 353
 Component, 286
 interface component, 98
 process component, 98
 Component-based development (CBD), 260
 Component dependency
 process-interface component dependency, 99
 process-process component dependency, 98
 Composite structure diagram, 145
 Concrete architecture derivation, 99
 Connection, 142
 Consumer goal, 96
 Context diagram, 140, 290
 Continuous integration, 281
 CoRE. *See* Change-oriented requirements engineering
 COTS software. *See* Third-party software
 Crosscutting concerns, 294
 Crossover, 310
- D**
 Design, 279
 decisions, 218, 225, 228
 patterns, 210, 218, 224, 308, 314, 316, 317, 325
 pearls, 282
 process, 279
 policies (*see* Design decisions)
 rationale, 218, 225, 282
 tactics, 294
 Developmental, 283
 Developmental qualities, 284
 Development process, 205, 224, 228
 Dispatcher, 312, 318, 319, 322
 Domain, 135
 driven design, 281
 entities, 95
 model, 289
 Domain-specific language (DSL), 113, 121, 289
- E**
 Eclipse, 72
 Engagement model, 184, 190, 194, 195
 Enterprise Java Beans, 354
 Environment, 189
- Error handling, 207
 Event-based traceability, 61
 Expressiveness, 282
- F**
 Façade, 316, 317, 325
 Feature models, 113
 Fitness function, 310
 Flexibility with options theory, 358
 Force, 279
 Functional constraints, 22
 Functional requirements, 311, 312
 Function-oriented components, 96
 Fuzzy goals, 166
- G**
 Genetic algorithms, 309
 Global and multi-site collaboration, 269
 Goal-measure, 190
 Goal model, 162
 Goal modelling, 19
 Goal-object, 189
 Goal-oriented requirements analysis, 93
 Goal-oriented software architecting, 92
 Goals, 162
 hardgoals, 93
 softgoals, 93
 Goal-subject, 189
 Golden hammer syndrome, 307
 Guidelines, 282
- H**
 Hardgoal-entity relationships, 96
 Hill climbing algorithm, 315
- I**
 Implicit architectural insight, 64
 Incremental approach, 279
 Infrastructural, 283
 Initial population, 310
 Integration failures, 207, 208, 220
 Intellectual property, 209
 Iteration, 281
 Iterative approach, 280
 Iterative-incremental process, 280
- J**
 Java 2 enterprise edition (J2EE), 353

K

KAOS, 106, 163
Knowledge, 36

L

Label propagation procedure, 100
Latent properties, 76
Layered architecture, 152
Layer volatility, 28
Logical architecture derivation, 95

M

Machine, 135
Maintenance, 220
Market share, 186, 187
 as business goal, 187
Markov chain, 71
Mediator, 316, 317, 325
Message dispatcher, 316, 317, 322, 324, 326
Message dispatcher architecture style, 316
Meta-heuristics, 309
Meta-heuristic search techniques, 313
Method-specific artifact types, 263
Metrics, 308, 314, 318
Middleware-induced architectures, 353
Module dependency graph (MDG), 315
Multi-objective genetic algorithm (MOGA),
 314
Mutation, 310

N

Non-functional constraints, 23
Non-functional requirements, 91
Null architecture, 308, 311, 312, 314, 316,
 320, 324

O

Object-oriented components, 96
Onion model, 283
Ontology, 35
 class, 46
 relationship, 46
Operational, 283
Operational quality, 284

P

Pareto optimality, 328
Patterns, 99

Pedigree, of business goal, 190

Pedigree attribute eLicitation method
(PALM), 185, 192–195

Piecemeal growth, 280

Principles, 294

Problem description, 133

Problem diagram, 137, 140

Problem domain, 281

Problem frame, 134, 135, 140

Process centered collaboration, 266

Process-oriented approach, 107

Producer goal, 96

Product line, 111, 112, 117

 engineering, 111, 112, 114, 115, 291
 modelling, 111
 models, 111
 organization, 297

Product-oriented approach, 107

Project manager, requirements analyst and
 software architect relationships, 339

Pyramid model, 285

Q

Quality attribute, 183, 184, 188, 192, 193,
 195, 311

Quality attribute requirements, 184, 189

Quality requirements, 307, 308, 311, 312, 328

Quality trees, 281

R

Re-engineering, 315

Refactoring, 281

Reference architecture, 297

 The Open Group Application Framework
 (TOGAF), 210

 previous examples, 212

 requirements for, 209

 service-oriented architecture (SOA),

 210, 212

 structure of, 216, 225

 usage, 228

Replacement access, library and ID card
(RALIC), 26

Requirements, 35, 111–127, 129, 162,
 183–185, 188, 189, 192, 193,
 195, 279

 change, 17

 co-evolve, 36

 concurrent features, 208, 228

 documentation, 19

 elicitation, 18

Requirements (*cont.*)
 non-functional, 212, 225
 real-time, 205, 214
 relationship with architecture, 209, 228
 traceability, 37, 61
 uncertainty, 111–115, 135, 137
 volatility, 118
 Requirements change management, 19
 Requirements-driven design, 282
 Resource conflicts, 206, 208, 228
 Resource contention. *See* Resource conflicts
 Resource management, 207, 223
 Reusability, 314
 RFID retail case study, 343
 Risk, 113, 127
 management, 112, 114, 116, 129
 mitigation, 282
 Risk-driven development, 282
 RUP's 4+1 views, 106

S
 Satisficed, 93
 Scenario, 283, 312, 319
 Scenario-based architecture analysis method
 (SAAM), 106
 Scenario-based architecture reengineering
 (SBAR), 106
 Search-based software engineering, 313
 Selection, 310
 Self-adapting systems, 328
 Self-adaptive compositions, 169
 Self-sustaining systems, 307
 Semantic wiki, 49
 semantic annotation, 49
 semantic MediaWiki, 49
 semantic query, 50
 semantic traceability, 50
 Sequence diagrams, 311, 316, 320
 Service compositions, 170
 Service-oriented architecture (SoA), 161, 210,
 212, 286, 315
 Service qualities. *See* Requirements, non-functional
 Shared phenomena, 136
 Shearing layers, 20
 Shearing layers of requirements, 21
 Siemens's four-views (S4V), 106
 SOA. *See* Service-oriented architecture
 Social considerations, 270
 Software architect, 287
 Software architecture, 146, 279
 Software architecture generation, 311

Software components
 roles and responsibilities, 221, 227
 scope of functionality, 208, 220, 222, 227
 Software metrics, 308, 318
 Software patterns, 297
 Software renovation framework (SRF), 315
 Software system, 279
 Solution domain, 290
 Sphere of influence, 67
 Stakeholder, 183, 189–194, 279
 Stereotype, 134
 Strategy, 281, 317, 318, 322, 325, 326
 Subsystem, 286
 SWOT analysis, 281
 Systematic architecture design, 280
 Systems-of-systems, 107

T

Tactical, 281
 Technical context diagram, 139
 Technology roadmap, 280
 Template method, 317, 322, 325, 326
 Test-driven development, 282
 The classical relationship between
 requirements and architecture, 335
 The i* framework, 106
 The NFR framework, 106
 The Open Group Application Framework, 210
 The Open Group Architecture Framework
 (TOGAF), 106
 The spiral model and requirements and
 architecture, 336
 Third-party software, 206, 219, 227
 Threats to validity, 103
 Time behavior, 308
 Traceability, 36, 61, 280
 Traceability links, 66
 Transition probability, 71
 Twin peaks model of software development,
 336

U

UML class diagram, 311
 UML4PF, 153
 Uncertainty analysis, 111, 113, 114, 117, 121,
 122, 126, 129, 130
 Understandability, 308
 Unified modeling language (UML), 259, 290,
 311, 314
 Use case, 18, 283, 312
 User interface consistency, 206, 214, 228

V

- Validation condition, 134, 148, 151, 153
- Viewpoints, 262
- 4+1 views, 311
- Volatility, 111, 119, 135, 145

W

- Waterfall model, 287
- White box view, 289
- Work contexts, 67