universität
wien

D I S S E R T A T I O N

Titel der Dissertation

Supporting Reusable Architectural Design Decisions

Decision Making, Resolving Uncertainty, and Establishing Consistency

verfasst von

Ioanna Lytra

angestrebter akademischer Grad

Doktorin der technischen Wissenschaften (Dr. techn.)

Wien, 2015

Studienkennzahl lt. Studienblatt:	A 786 880
Dissertationsgebiet lt. Studienblatt:	Informatik
Betreuer:	Univ.-Prof. Dr. Uwe Zdun

Declaration of Authorship

I, Ioanna Lytra, declare that this thesis with the title, "**Supporting Reusable Architectural Design Decisions:** Decision Making, Resolving Uncertainty, and Establishing Consistency" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

Abstract

Every software architecture entails a set of architectural decisions. Architectural decision documentations capture not only the resulting design structures but also the justification, the strengths and weaknesses, as well as alternatives for the selected design solutions. Therefore, software architects capture architectural design decisions for analyzing and understanding, as well as sharing and communicating the rationale and implications of these decisions. Capturing architectural decisions, thus, prevents the potential loss of architectural knowledge, a phenomenon known as architectural knowledge vaporization. The development of various tools and approaches for architectural decision making and documentation in the past ten years and the adoption of architectural decision methods by the industry proves the increasing interest in capturing and managing architectural decisions in the software design and development processes. In addition, a few works have set the focus on gathering, organizing, and effectively leveraging reusable architectural decisions. Reusable architectural decisions are documented, for instance, in the form of reusable decision models, in order to provide architectural guidance in software projects. This thesis is inspired by this trend and motivated by the challenges that emerge in this area.

Although the research in tools and methods by the software community for supporting reusable architectural knowledge has increased, there are still very few empirical studies and limited empirical evidence on whether and how the leveraging of reusable architectural decisions may increase the effectiveness and efficiency of software architects. Furthermore, some aspects that need to be considered during decision making including the use of quality attributes, the uncertainty of architectural decisions, the relationships between application-generic and application-specific knowledge, have not been addressed for reusable architectural decisions adequately. Another challenge we identified is related to the inconsistency management between architectural decisions and designs, so that decision making and documentation based on reusable knowledge gets integrated with other software design, development, and evolution processes.

In this thesis, we observed the modeling and leveraging of reusable architectural decision models in an industrial case study and two controlled experiments. For this, we have developed a prototype to support architectural decision making and documentation based on reusable decision models. For the needs of the case study, we created a reusable decision model on service-based platform integration which was afterwards validated by platform experts. These experts also participated in a case study for integrating heterogeneous platforms operating in a software system for controlling a warehouse. The two controlled experiments we conducted with novice architects provided empirical evidence that the use of reusable decision models increases both efficiency and effectiveness of software architects. In addition, by conducting a literature survey on the use of quality attributes in architectural decision tools and methods, we found out that there is only limited support on quality-driven architectural decision making based on reusable architectural knowledge. This finding led

us to introduce an extension of reusable decision models enriched with quality attributes and to consider the uncertainty of quality attributes in decision making, in order to support better reusable decisions entangled with quality attributes. We also demonstrated our proposal in a case study from an industrial project on smart cities software ecosystems. Some other issues like the integration of variability and architectural decisions have emerged as a need for supporting decision making in product line engineering. Finally, our proposal for consistency management between architectural decisions and designs and the domain-specific language for transforming reusable architectural knowledge into designs can be seen as an improvement to the current practice of maintaining decisions and designs manually and a step towards automating the integration of reusable decisions in software design and evolution processes. Our proposed method and corresponding tooling have been evaluated regarding the efficiency, scalability, modeling effort and reusability in the context of an industrial case study. In summary, this thesis has contributed considerably to a better understanding and support for reusable architectural decisions, whose adoption by the industrial practice is expected to increase in the next years.

Zusammenfassung

Jede Softwarearchitektur beinhaltet eine Reihe von Architekturentscheidungen. Die Dokumentation von Architekturentscheidungen erfasst nicht nur die sich daraus ergebenden Designstrukturen, sondern auch die Begründung, die Stärken und Schwächen, sowie die Alternativen zu den ausgewählten Designlösungen. Softwarearchitekten erfassen daher Architekturentscheidungen, um die Begründung und die Implikationen dieser Entscheidungen zu analysieren, zu verstehen, zu teilen und zu kommunizieren. Architekturentscheidungen zu erfassen verhindert folglich potentiellen Verlust von Architekturwissen, ein Phänomen das auch als "Knowledge Vaporization" bekannt ist. Die Entwicklung von verschiedenen Tools und Ansätzen für das Treffen und Dokumentieren von Architekturentscheidungen sowie die Einführung von Architekturentscheidungsmethoden in die Industrie in den letzten zehn Jahren belegen das steigende Interesse an der Erfassung und dem Management von Architekturentscheidungen beim Softwaredesign- und Softwareentwicklungsprozess. Manche Arbeiten stellen das Sammeln, Organisieren und das effektive Verwenden von wiederverwendbaren Architekturentscheidungen in den Fokus. Wiederverwendbare Architekturentscheidungen sind zum Beispiel in Form von Architekturentscheidungsmodellen dokumentiert, die das Architekturdesign von Softwareprojekten unterstützen. Diese Doktorarbeit ist durch die Herausforderungen, die in diesem Bereich entstehen, motiviert.

Trotz intensiver Forschung an Tools und Methoden für die Unterstützung von wiederverwendbarem Architekturwissen in der Software Community in letzter Zeit, gibt es immer noch sehr wenige empirische Studien und beschränkte empirische Evidenz über die Effektivität und Effizienz von Softwarearchitekten, die wiederverwendbare Architekturentscheidungen verwenden. Außerdem sind bis jetzt manche Aspekte, die während des Treffens von Architekturentscheidungen berücksichtigt werden müssen, wie zum Beispiel die Verwendung von Qualitätsattributen, die Ungewissheit von Architekturentscheidungen oder die Beziehungen zwischen applikationsgenerischem und applikationsspezifischem Wissen, für wiederverwendbare Architekturentscheidungen wenig untersucht worden. Eine andere Herausforderung, die wir identifiziert haben, bezieht sich auf das Konsistenzmanagement zwischen Architekturentscheidungen und Architekturdesigns, sodass das Treffen und die Dokumentation von Architekturentscheidungen, die auf wiederverwendbarem Wissen basieren, in anderen Softwaredesign-, Softwareentwicklungs- und Softwareevolutionsprozessen integriert werden.

In dieser Doktorarbeit haben wir die Modellierung und die Verwendung von wiederverwendbaren Architekturentscheidungsmodellen in einer Fallstudie in der Industrie sowie in zwei kontrollierten Experimenten beobachtet. Dafür haben wir einen Prototypen entwickelt, um das Treffen und das Dokumentieren von Architekturentscheidungen, die auf wiederverwendbaren Entscheidungsmodellen basieren, zu unterstützen. Für den Zweck der Fallstudie haben wir ein wiederverwendbares

Architekturentscheidungsmodell über Service-basierte Plattformintegration entwickelt; dieses Modell wurde danach von Plattformexperten validiert. Diese Experten haben sich auch bei einer Fallstudie über die Integration von heterogenen Plattformen, die im Rahmen eines Softwaresystems für die Kontrolle eines Warehouse funktionieren, beteiligt. Weiters haben wir zwei Experimente mit angehenden Softwarearchitekten durchgeführt. Diese lieferten uns empirische Evidenz, dass die Verwendung von wiederverwendbaren Entscheidungsmodellen sowohl die Effizienz als auch die Effektivität von Softwarearchitekten erhöht. Wir haben auch eine strukturierte Literaturrecherche über die Verwendung von Qualitätsattributen in Tools und Methoden für Architekturentscheidungen durchgeführt und herausgefunden, dass nur wenige Ansätze das Treffen von Architekturentscheidungen, basierend auf Qualitäten und wiederverwendbarem Architekturwissen, unterstützen. Diese Erkenntnisse haben uns dazu veranlasst, wiederverwendbare Architekturentscheidungsmodelle mit Qualitätsattributen zu erweitern und die Ungewissheit von Qualitätsattributen beim Treffen von Architekturentscheidungen zu untersuchen. Das Ziel war es, wiederverwendbare Architekturentscheidungen zusammen mit Qualitätsattributen besser zu unterstützen. Wir haben diesen Ansatz in einer Fallstudie im Rahmen eines industriellen Projekts über Softwareökosysteme für Smart Cities demonstriert. Einige andere Themen wie die Integration von Variabilitäts- und Architekturentscheidungen sind als Anforderungen im Rahmen der Unterstützung von Architekturentscheidungen im Product Line Engineering entstanden. Unser Ansatz für das Konsistenzmanagement zwischen Architekturentscheidungen und Architekturdesigns und die domänenspezifische Sprache zur Umwandlung wiederverwendbaren Architekturwissens in Designs ist als eine Verbesserung zur aktuellen Praxis der manuellen Verwaltung von Architekturentscheidungen und Architekturdesigns und als ein Schritt in die Richtung der Automatisierung der Integration von wiederverwendbaren Entscheidungen in Softwaredesign- und Softwareevolutionsprozessen zu sehen. Unsere Methode und entsprechendes Tooling wurden für ihre Effizienz, die Skalierbarkeit, den Modellierungsaufwand und die Wiederverwendbarkeit im Rahmen einer industriellen Fallstudie evaluiert. Zusammenfassend hat diese Doktorarbeit zum besseren Verständnis und der besseren Unterstützung von wiederverwendbaren Architekturentscheidungen – deren Anwendung in der industriellen Praxis in den kommenden Jahren steigen sollte – erheblich beigetragen.

Acknowledgements

First of all, I would like to acknowledge my advisor Prof. Uwe Zdun for his valuable supervision, expert advice, optimism, and encouragement throughout my dissertation research. Without his guidance and support it would not be possible to go from the beginning to the end of this dissertation research successfully and in time. I would also like to thank Prof. Matthias Riebisch who agreed to be reviewer for my thesis. I would like to specially thank Christian Mader for being patient to do the boring proof reading of my thesis. To all my partners in research projects and co-authors in scientific papers I would like to say it was an honor and pleasure to work with you and our cooperation contributed much to making a step forward to finishing my thesis. A big thank you also goes to my colleagues for the fruitful and inspiring discussions and the cups of coffee we had together during our brakes. Working with them and next to them made me always think optimistic and of course it was a lot of fun. Last but not least, I would like to thank my family for their unlimited support.

List of Publications

This doctoral dissertation is mainly based on research work which has been either already published in scientific workshops, conferences, and journals or submitted to a research venue (currently under review). In particular, content of the following publications has been used in this thesis:

- Ioanna Lytra et al. "Harmonizing architectural decisions with component view models using reusable architectural knowledge transformations and constraints." In: *Future Generation Computer Systems* [2014]. DOI: 10.1016/j.future.2014.11.010
- Ioanna Lytra and Uwe Zdun. "Inconsistency Management Between Architectural Decisions and Designs Using Constraints and Model Fixes." In: *Proceedings of the 23rd Australian Software Engineering Conference*. ASWEC'14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 230–239. DOI: 10.1109/ASWEC.2014.33
- Ioanna Lytra et al. "On the Interdependence and Integration of Variability and Architectural Decisions." In: *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*. VaMoS'14. Sophia Antipolis, France: ACM, 2014, 19:1–19:8. DOI: 10.1145/2556624.2556634
- Ioanna Lytra et al. "A Survey on Quality Attributes Use in Architectural Design Decisions." submitted to: *ACM Computing Surveys* [submitted in December 2014]
- Patrick Gaubatz et al. "Automatic Enforcement of Constraints in Real-time Collaborative Architectural Decision Making." In: *Journal of Systems and Software* [2015]. accepted for publication
- Ioanna Lytra et al. "Two Controlled Experiments on Model-based Architectural Decision Making." submitted to: *Information and Software Technology* [revised in January 2015]
- Dan Tofan et al. "Empirical Evaluation of GADGET – a Process to Increase Consensus in Group Architectural Decision Making." to be submitted
- Ioanna Lytra et al. "Reusable Architectural Decision Models for Quality-driven Decision Support: A Case Study from a Smart Cities Software Ecosystem." accepted for publication at: *ICSE 3rd International Workshop on Software Engineering for Systems-of-Systems (SESoS)*. 2015
- Ioanna Lytra et al. "Supporting Consistency Between Architectural Design Decisions and Component Models Through Reusable Architectural Knowledge Transformations." In: *Proceedings of the 7th European Conference on Software Architecture*. Vol. 7957. ECSA'13. Montpellier, France: Springer-Verlag, 2013, pp. 224–239. DOI: 10.1007/978-3-642-39031-9_20

- Ioanna Lytra and Uwe Zdun. "Supporting Architectural Decision Making for Systems-of-systems Design Under Uncertainty." In: *Proceedings of the First International Workshop on Software Engineering for Systems-of-Systems*. SESoS'13. Montpellier, France: ACM, 2013, pp. 43–46. DOI: 10.1145/2489850.2489859

- Christian Inzinger et al. "Decisions, Models, and Monitoring – A Lifecycle Model for the Evolution of Service-Based Systems." In: *Proceedings of the 17th IEEE International Enterprise Distributed Object Computing Conference*. EDOC'13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 185–194. DOI: 10.1109/EDOC.2013.29

- Huy Tran et al. "Derivation of Domain-specific Architectural Knowledge Views from Governance and Security Compliance Metadata." In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. SAC'13. Coimbra, Portugal: ACM, 2013, pp. 1728–1733. DOI: 10.1145/2480362.2480689

- Ioanna Lytra et al. "Architectural Decision Making for Service-Based Platform Integration: A Qualitative Multi-Method Study." In: *Proceedings of the 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*. WICSA-ECSA'12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 111–120. DOI: 10.1109/WICSA-ECSA.212.19

- Ioanna Lytra et al. "A Pattern Language for Service-based Platform Integration and Adaptation." In: *Proceedings of the 17th European Conference on Pattern Languages of Programs*. EuroPLoP'12. Irsee, Germany: ACM, 2012, 4:1–4:27. DOI: 10.1145/2602928.2603080

- Ioanna Lytra et al. "Constraint-Based Consistency Checking Between Design Decisions and Component Models for Supporting Software Architecture Evolution." In: *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*. CSMR'12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 287–296. DOI: 10.1109/CSMR.2012.36

Vita

Ioanna Lytra is a PhD student at the Research Group Software Architecture (SWA), Faculty of Computer Science, University of Vienna since 2011. Before that, she worked as a software engineer in the area of telecommunications. She received her Diploma in Electrical and Computer Engineering with specialization in Computer Networks and Telecommunications from the National Technical University of Athens in 2006. Her research interests focus on software architectures, architectural decisions, software patterns, service-oriented architectures, and model-driven development.

Contents

List of Figures

List of Tables

Abbreviations

ADD	**A**rchitectural **D**esign **D**ecision
ADvISE	**A**rchitectural **D**esign Dec**I**sion **S**upport Fram**E**work
AK	**A**rchitectural **K**nowledge
C&C	**C**omponent & **C**onnector
CoCoADvISE	**C**onstrainable **C**ollaborative **A**rchitectural **D**esign Dec**I**sion **S**upport Fram**E**work
DSL	**D**omain-**s**pecific **L**anguage
FR	**F**unctional **R**equirement
NFR	**N**on-**F**unctional **R**equirement
QA	**Q**uality **A**ttribute
QOC	**Q**uestions **O**ptions **C**riteria
RMS	**R**emote **M**aintenance **S**ystem
SOA	**S**ervice-**o**riented **A**rchitecture
VbMF	**V**iew-**b**ased **M**odeling **F**ramework
VSP	**V**irtual **S**ervice **P**latform
WMS	**W**arehouse **M**anagement **S**ystem
YMS	**Y**ard **M**anagement **S**ystem

Part I

Foundations and Research Overview

1 Introduction

All software systems, either small and simple or more complex, bad-designed or sophisticated, documented or not have an architecture. The architecture of a system describes what the individual system parts do, how they interact with each other or with the world around them. The Software Engineering Institute at Carnegie-Mellon University gives the following definition [Cle+02] for software architecture: *"The architecture of a software-intensive system is the structure of structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them"*. In the literature many view models defining a coherent set of views for the construction of a software architecture exist. According to the IEEE Standard 1471 [ISO11] *"An architectural view represents the architecture of the (whole) system with respect to a specified set of related architectural concerns"*. As a software system evolves the architectural views change, new requirements come or get modified, and designers' decisions are revised.

Jansen and Bosch proposed ten years ago a new perspective of software architecture by using a set of architectural design decisions (ADDs) for its description. The knowledge and information about the design decisions are implicitly embedded in the architecture, and in this sense, the ADDs can be seen as an explicit part of the software architecture. The aim of documenting ADDs is to capture architectural knowledge (AK) and rationale about the design that otherwise get lost and forgotten. In recent years, software architecture is less seen as only the components and connectors constituting a system's principal design, and more and more as a set of principal design decisions governing a system. This claim is supported by the various tools and techniques addressing architectural decision making and documentation [Sha+09], for instance, with the aid of templates [TA05], ontologies [LK07], and meta-models [Zim+07].

A few researchers have set the focus on gathering, organization, and effective use of reusable ADDs [Zim+08; Har+07]. Architectural decision models [Zim+07; ZS09; May+09; Lyt+12b] and the documentation of ADDs in such models or similar forms for providing architectural guidance in software projects has lately gained much attention in industrial practice [GB14; Zim+08]. The idea of capturing reusable ADDs is similar to leveraging design patterns, which give proven solutions to recurring design problems, for architectural decision making. Software architects can benefit much from the collaboration and knowledge exchange between different AK repositories, for instance, repositories containing reusable architectural decision models [Now+10].

This trend in architectural knowledge management and various challenges and open questions that we identified in this area have motivated the research performed in this thesis. In particular, we studied empirically the architects' effectiveness and efficiency during decision making based on reusable architectural decision models. In addition, we investigated issues like the integration of quality attributes (QAs) with ADDs, the uncertainty of architectural decision making, and the inconsistency management between architectural decisions and designs during software systems evolution. This PhD thesis has contributed substantially to understanding, extending, and improving the support of reusable ADDs in architectural decision making and documentation.

Thesis Structure

This PhD thesis is structured as follows:

In Part I, we discuss the state of the art that constitutes the basis of the thesis and point out the novelty of our research work in comparison to existing approaches (see Chapter 2). In addition, we analyze the main research problems addressed and the research methods applied in the context of this thesis (see Chapter 3).

In Part II, we propose an architectural decision meta-model and a corresponding tool to support decision making and documentation for reusable ADDs (see Chapter 4). Through a qualitative multi-method study we consolidate and create a reusable decision model on service-based platform integration and validate it in the context of an industrial case study (see Chapter 5). Furthermore, we perform an empirical evaluation of our tool to test the efficiency and effectiveness of software architecture students during architectural decision making and documentation (see Chapter 6).

In Part III, we focus on the integration of reusable ADDs with QAs. First, we conduct a literature survey to study the relationships between QAs and ADDs in existing methods and tools for architectural decision making and documentation and define research gaps and challenges in that area (see Chapter 7). We propose to enrich reusable architectural decision models with QAs in order to provide quality-driven decision support for software architects (see Chapter 8). In addition, we deal with the problem of uncertainty of QAs in architectural decision making (see Chapter 9).

In Part IV, we address the problem of inconsistency management between reusable ADDs and design views and propose a transformation language to translate reusable ADDs into designs (see Chapter 10). Apart from that, we study the interdependence between architectural and variability decisions and propose to systematically integrate them in product line engineering (see Chapter 11).

Finally, in Part V, we summarize the main contributions of the thesis, discuss the limitations, and present open challenges to be addressed in our future work (see Chapter 12).

2 State of the Art

In this chapter, we give the context of our research by discussing the related work. In this discussion, we also define some challenges and motivate our research work. First, we briefly introduce the definitions of the key concepts of this PhD thesis like *architectural design decision* and *design patterns and pattern languages* (see Section 2.1). In addition, we present the related work on the existing approaches and tools for architectural decision making and documentation, we study the role of QAs in architectural decision approaches, and finally discuss the maintenance of consistency between architectural designs and ADDs in Section 2.2, Section 2.3, and Section 2.4 respectively. We provide, thus, a first comparison of our proposal to the state of the art. The discussion of the related work, however, does not end here. We come back to some topics like the uncertainty of ADDs and the interdependence between variability and architectural decisions in the corresponding chapters that follow.

2.1 Key Concepts and Definitions

Architectural Design Decision (ADD) According to the ISO/IEC/IEEE 42010 standard [ISO11] an ADD affects various architectural elements, pertains to or raises concerns, and is justified by architecture rationale. Ven et al. understand design decisions as a first-class artifact that couples rationale with software architecture [Ven+06]. ADDs capture knowledge that may concern a software system as a whole, or one or more components of a software architecture. Rather than documenting the structure of software systems (e.g., components and connectors) ADDs entail the design rationale that led to that structure (e.g., justification about the ADDs that were made and the architectural alternatives not chosen). Capturing ADDs is important for analyzing and understanding the rationale and implications of these decisions and for gathering (reusable) AK.

Design Patterns and Pattern Languages According to Buschmann et al. a pattern documents a recurring problem-solution pair within a given context [Bus+07b]. It includes both the problem and the solution, along with the rationale that binds them together. The problem is considered with respect to conflicting forces and the proposed solution is described in terms of

its structures including a clear presentation of its consequences. Since the GoF book on patterns [Gam+94], the design pattern literature is becoming more and more extensive (i.e., [Fow03; HW04; Völ+05]). To systematically explain how to apply a number of patterns in combination, many pattern authors document patterns as part of larger pattern languages [Ale+77; Cop96], containing rich pattern relationships and extensive examples and known uses sections.

Reusable ADDs Reusing ADDs can contribute to simplifying architecting [Fal+11]. Thus, addressing systematic documentation of ADDs and providing guidance during decision making for recurring design issues, the use of reusable ADD models has been proposed in the literature [Zim+07]. Similar to leveraging patterns for architectural decision making [Har+07], reusable ADD models provide proven solutions – both application generic and technology-specific – to various design issues along with their forces and consequences.

Quality Attribute (QA) Software quality is defined as the degree to which software possesses a desired combination of attributes (such as performance, reliability, interoperability, or adaptability) [IEE98]. In software architecture, these attributes are typically called quality attributes [Bas+03][1]. QAs are orthogonal to functionality [Bas+03]. That is, the choice for a functionality in a design decision does not dictate, for instance, the level of performance or interoperability. Instead, many different design solutions for a functionality can be chosen, leading to different levels for QAs like performance or interoperability, and also to the trade-offs between them. As a consequence, ADDs drive the selection of qualities of the system and the other way around where QAs are seen as decision drivers [Zim+07; Zim+08; Bas+03; Lyt+12b].

2.2 Existing Approaches for Architectural Decision Support

The documentation of the design rationale as well as the gathering of AK have promoted ADDs to first class citizens in software architecture [JB05]. There are numerous attempts on documentation and leveraging of design rationales. For instance, Clements et al. suggest a general outline for documenting architectures and guidelines for justifying design decisions [Cle+02]. Tyree and Akerman present a rich template for capturing and documenting several aspects of ADDs [TA05]. Another technique proposed by Lee and Kruchten aims at establishing a formalized ontological description of architectural decisions and their relationships [LK07] whilst Harrison et al. use patterns for capturing recurring decisions [Har+07]. Zimmermann et al. proposed decision meta-models and guidelines for formulating and documenting design decisions and illustrated their methods in the context of the SOA (Service-oriented Architecture) design space [Zim+07].

[1] Please note that in the requirements engineering context, the term non-functional requirements (NFRs) is commonly used when referring to quality attribute requirements [CP09].

A substantial amount of work has been done in the direction of documenting AK using *architectural decision modeling* (refer to [Sha+09] for a comparison of existing architectural decision models and tools). For instance, Jansen and Bosch propose a meta-model for capturing decisions that consist of problems, solutions and attributes of the AK [JB05]. Zimmermann et al.'s meta-model for capturing ADDs [Zim+08] consists of three core domain entities: *Architectural Decision (AD)* related to one or more *ADTopics* organized in *ADLevels*, entailing *ADAlternatives*, the selection of which leads to an *ADOutcome*. The advantage of such ADD models is that they are reusable and can be used as guidance for architectural decision making activities, whenever recurring design issues emerge. Reusable ADD models share common concepts with patterns (see [Har+07]), that is, they both provide proven solutions for specific design issues along with their motivation and rationale. The main difference is that reusable ADD models provide the means for formally defining more complex relationships for ADDs (e.g., the selection of a design option may exclude a design solution). Furthermore, they allow us to capture except for generic knowledge – usually addressed by patterns – also domain and technology-specific AK. Yet, the relationship between architectural patterns and reusable ADD models can be eventually synergetic [Zim+07], for instance, reusable decision models can be integrated with patterns and guide the selection of patterns. Various reusable ADD models have been documented in the literature, covering SOA-related solutions [Zim+07], service-based platform integration [Lyt+12b], the design of domain specific languages [ZS09], and model and meta-data repositories [May+09].

2.2.1 Application-generic and Application-specific AK

In the design decision literature, application-generic and application-specific knowledge have been distinguished. Farenhorst and Boer [FB09] define this distinction and the conversion from one type to another. Application-generic knowledge can be used in many different projects. Examples of generic knowledge are software patterns [Gam+94; Bus+07b], architectural styles [SG96] and reusable architectural decisions [Zim+08]. Application-specific knowledge deals with the knowledge of a specific system, documented e.g., during its development or evolution. Decision models [TA05; JB05; Kru+06] and models created using architecture description languages [MT00] are examples of application-specific knowledge. Both levels of design knowledge though should be considered during decision making.

A number of approaches (e.g., [Zim+08; TA05]) capture technology-specific knowledge in the fields of decision meta-models or decision templates. These approaches consider many levels of design knowledge at decision making, but usually the knowledge levels are only depicted as fields in decision meta-models and templates. So far, however, the integration of multiple levels of application-generic and application-specific architectural decisions has not been studied systematically. In addition, none of the approaches documents and systematizes the whole solution space

for a design problem at hand considering pattern variants and pattern implementations for various technologies. We discuss and address these two challenges in more detail in Chapter 9.

2.2.2 Comparison of Tools Supporting ADDs

As mentioned before, several tools have been developed to ease capturing, managing and sharing of ADDs [Tan+10; Sha+09; Tof+14]. In most of the cases, the focus is set on the manipulation of architectural decision artifacts and their relationships, and the capturing and reuse of architectural knowledge, as well as collaboration aspects. In this section, we discuss existing tools for architectural decision making and documentation and compare these to ADvISE and CoCoADvISE, the tools we have developed to provide semi-automated decision support for reusable ADDs.

PAKME [BG07] and ADDSS [Cap+07] are some of the first attempts to develop tools for decision making and documentation. PAKME aims at providing AK management support for knowledge acquisition, maintenance, retrieval, and presentation, and contains features for capturing and classifying ADDs, integrating pattern-based knowledge, and relating ADDs with quality-based scenarios and requirements. Similar to PAKME, ADDSS can be used for storing, managing and documenting ADDs. ADvISE and CoCoADvISE target both decision making and documentation. For decision making, reusable ADD models are leveraged [Zim+07] and for documentation, a text template like the one introduced by Tyree and Akerman [TA05] is used. Unlike tools that focus on the visualization and understandability of ADDs (such as [Zal+11; Sha+10]), we mainly target decision guidance and automation of steps in decision making and documentation.

Many existing tools support the integration of ADDs in the software architecture and development processes. For instance, Archium aims primarily at capturing architectural decisions and weaving them into the development process, that is, binding them with models and system implementation [Jan+07]. Also, the ADVERT approach targets the capturing of pattern-specific ADDs with their rationale along with decision-relevant requirements, QAs, and architectural elements [Kon+13].

Although automated support is an important aspect in decision making, it is addressed very little by existing approaches. Ameller and Franch propose ArchiTech to support software architects during the architectural decision-making process by suggesting alternative decisions for satisfying non-functional requirements [AF14]. For this, optimization techniques are applied on an AK ontology. Also, ArchDesigner provides a quantitative quality-driven approach to find the best possible fit between various quality goals, competing architectural concerns, and constraints [AN+05]. It uses Multiple Attribute Decision Making (MADM) and optimization techniques to find a best-fitting architecture composed of interdependent ADDs that satisfies prioritized quality goals. In Section 4.2 we discuss in detail how ADvISE and CoCoADvISE provide semi-automated guidance for decision making and documentation. Similar to our tool, Zimmermann et al. provide

an architectural design method to combine pattern languages with reusable architectural decision models [Zim+08]. The goal of their approach is to provide domain-specific pattern selection advice and traceability links from platform independent patterns to platform-specific decisions. However, the reusable ADD models have to be used manually by software architects while in CoCoADvISE semi-automated decision guidance based on reusable decision models is provided.

The sharing of architectural knowledge is one of the main concerns of the architectural decision management tools AD_{kwik} [Sch+07], Knowledge Architect [Lia+09], and PAKME [BG07]. In addition, Compendium [Sha+10] supports a visual environment for documenting and visualizing design rationale behind ADDs for multiple users. In CoCoADvISE, we have extended the concepts of collaborative architectural decision making by introducing automatic enforcement of constraints during group decision making (e.g., permissions or role-based constraints). Automatic support is thus an advantage of CoCoADvISE in comparison to the aforementioned tools, which do not target any automation in collaborative decision making (see Chapter 4.5).

We notice that the majority of the proposals have been evaluated either in case studies (e.g., [Zim+08; Zal+11]), in industrial projects [Cap+07], or focus groups [NP13], and in a few cases no evaluation is reported. None of the tools has been empirically validated and only little feedback has been gathered by the users[2]. We have conducted two controlled experiments in order to test the supportive effect of the main concepts of ADvISE and CoCoADvISE, namely the reusable ADD models, on capturing ADDs.

Existing methods and techniques, as we discussed above, are grounded on the main idea of considering ADDs first-class citizens in software architecture design. Therefore, these approaches lay a solid foundation on capturing and managing design decisions for several successive studies, including our approach presented in this paper. For instance, the common ADD model supported by ADvISE and CoCoADvISE inherits common concepts and elements of existing meta-models and templates for recording design decisions. Therefore, we did not intend to develop "yet another tool" for ADD management but rather to implement existing concepts in architectural decision support such as reusable architectural decision models [Zim+07] and the Questions, Options, and Criteria (QOC) approach [Mac+91] and provide semi-automated tool support integrating these concepts. Nevertheless, the implementation of the prototypes ADvISE and CoCoADvISE (see Section 4.2) also serves the empirical validation of reusable architectural decision models.

Table 2.1 provides a comparison overview of the tools discussed in this section. In particular, we report on the type of the tool support (decision making and/or documentation), whether it provides automation and collaboration support, and what research methods have been used for its evaluation.

[2]Except for two studies [Sha+11; NP13] which contain only a very brief summary of the results and do not study their statistical significance.

Name	Decision Making	Documentation	Automation Support	Collaboration Support	Approach Evaluation
ArchPad [Zim+08]	+	–	–	–	Case study from the finance industry. SOA-related decisions (300) are documented.
ArchiTech [AF14]	+	–	Suggests alternative decisions based on NFRs	+	No evaluation reported
AD_{kwik} [Sch+07]	–	+	–	+	SOA-related decisions (300) are documented
Archium [Jan+07]	+	+	–	–	Motivating example
MAD 2.0 [Zal+11]	+	+	–	–	Case study (CRM system)
PAKME [BG07]	+	+	–	+	No evaluation reported
Compendium [Sha+10]	+	+	–	+	SOA-related decisions are documented
ADDSS [Cap+07]	+	+	–	–	Two industrial projects (multimedia system and virtual reality application)
ADVERT [Kon+13]	+	–	–	–	Evaluation using the Common Component Modeling Example (CoCoME)[1]
ArchDesigner [AN+05]	+	–	Making trade-offs between stakeholders	+	Industrial project (Glass Box)
SAW [NP13]	+	–	–	+	In a focus group. 20 participants considered 100 issues with 5 alternatives each.
ADvISE/CoCoADvISE [Lyt+b; Gau+15]	+	+	Predefined questionnaires used during decision making	+	Two controlled experiments with 49 and 122 students respectively

[1] CoCoME is an example system which provides a benchmark for component-based modeling approaches [Her+07].

TABLE 2.1: Comparison overview of ADD tools

2.3 Quality Attributes in ADD Approaches

In the process of architectural decision making, QAs constitute key drivers for designing software systems and therefore, it is important to document QAs along with the decisions captured. QAs, such as performance or interoperability, are commonly used in software architecture to describe the non-functional aspects of the architecture [Bas+03]. That is, many different design solutions for a functionality can be chosen, leading to different levels of the QAs and also to different trade-offs between them. For example, choosing a solution with a better adaptability often leads to trade-offs in terms of performance, as the indirections necessary for enabling adaptations slow down the system. As a consequence, even though many ADDs concern functionalities of the system, the QAs are often among the most important decision drivers [Zim+07; Zim+08; Bas+03].

The complexity of AK methods and tools is increased by the subjectivity of quality concerns that can be interpreted differently for different stakeholders and in different contexts, the importance of some qualities that are evaluated above others, and the impact of QAs on the decisions made. For these reasons, supporting QAs entangled with ADDs is challenging.

Several approaches described in the literature, such as the Architecture Tradeoff Analysis Method (ATAM) [Bas+03], the Cost Benefit Analysis Method (CBAM) [Kaz+01], and Attribute Driven Design (ADD) [Bas+02], can be used for architectural evaluation purposes. However, the majority of architectural decision support methods and tools focus on other aspects, such as reducing AK vaporization [Har+07], reusability of ADDs [Zim+07], knowledge sharing [Far+08], group decision making (such as [NP13]), or the definition of traceability links between decisions and other software artifacts (such as [Zim+09]). In Chapter 7, we perform a literature survey with focus on QA support of existing tools and methods as well as main QA-related challenges. In this chapter, we discuss and compare the use of QAs in approaches for architectural decision making and documentation in detail.

2.4 Relating ADDs to Software Architectures and Maintaining Consistency

There are a few works in the literature that discuss the need to relate ADDs to software architectures. The problem of not documenting and not maintaining ADDs that cause design knowledge vaporization has been discussed before [JB05; Cho+09]. For instance, Choi et al. suggest to make ADDs more explicit by introducing a meta-model for relating decisions with architectural elements and a decision constraint graph for representing decision relationships and studying decision change impact analysis [Cho+09]. Compared to our proposal (see Chapter 10), the aforementioned approach demands that most of the work is done manually: decision making, architectural design and change propagation during software evolution.

STREAM-ADD [Der+12] also relates architectural decisions documented in decision templates with requirements and architectural models generated from these requirements. This approach focuses rather on the integration of systematic documentation of structural and technological decisions with requirements and architectural models than on the consistency checking between decisions and designs. Küster has proposed to intertwine ADDs with standard architecture documentation processes by defining architecture-specific decision types along with OCL constraints used for decision conformance checking of component and deployment views [Küs13]. The aforementioned approach uses similar concepts for checking the conformance of ADDs to architectural views, however, comparing to our proposal, it does not support the transformation of ADDs into components and connectors.

Another popular technique for relating ADDs and software architecture aims at establishing trace links between design decisions and architectural elements. Capilla et al. introduce fine-grained traceability links between design decisions and other software artifacts [Cap+11]. Könemann and Zimmermann establish links between design decisions and design models in model-based software development in order to support architectural knowledge documentation and reuse [KZ10], as well as to check consistency. Mirakhorli and Cleland-Huang introduce the TTIM approach that provides a reusable infrastructure for tracing architecture tactics to designs used to trace from tactic-related design decisions to architecture components in which a decision is realized [MCH11]. Malavolta et al. present an approach that enables analysis of change impact caused by changes in ADDs [Mal+11]. This is mainly achieved by manually defining trace links between design decisions and other artifacts. None of the aforementioned approaches targets the reusability of these links between ADDs and architectural views, therefore, these approaches would not be able to tackle the complexity of large numbers of reusable ADDs. In order to address this gap, we propose to automatically transform architectural design intentions and knowledge into concrete architectural elements and configurations (see Chapter 10).

2.4.1 Mapping Specifications to Architectural Views

The generation of architectural design views from specifications or other architectural views has been studied extensively in the literature. Pérez-Martínez and Sierra-Alonso use semi-automated model-to-model transformations to generate component-and-connector (C&C) architecture models from classes and packages analysis models by using OCL-based mapping rules [PMSA06]. In this approach, architects can use the analysis model, which is a UML class diagram extended with additional profiles for the Unified Development Process [Jac+99], to describe architectural design intentions. After that, the elements of the analysis model will be coerced to the concepts of the C&C model according to the mapping rules.

In a different approach by Loughran et al., variability elements from the problem space are connected to architectural elements in the solution space using a Variability Modeling Language (VML) [Lou+08]. That language provides primitives for referencing and invoking decisions which result in fine-grained or coarse-grained compositions of variable and common core architectural elements. Different from the approach we introduce in this thesis the aforementioned approach supports rather the composition than the generation of software architectures, as it requires that all architectural elements are prescribed.

A considerable amount of research has been conducted in relating requirements with software architectures. For example, Kaindl and Falb show that model-driven development techniques can help in mapping requirements to architectural design [KF08]. Grünbacher et al. introduce the mapping from requirements to intermediate models that are closer to software architecture [Grü+03]. A

different approach presented by van Lamsweerde derives software architectures from the formal specifications of a system goal model (KAOS) using transformation rules and refines the architectures incrementally using patterns that satisfy quality of service goals such as availability and fault-tolerance [Lam03]. Sochos et al. present an approach, namely Feature-Architecture Mapping (FArM), that focuses on providing a mapping between features and software product line architectures [Soc+06]. The drawback of this approach is that it assumes a one-to-one mapping, that is, each component of the resulting product line architecture encapsulates the business logic of a feature. In reality, a high-level abstraction concept (such as a requirement or feature) might relate to more than one architectural elements and vice versa.

In the aforementioned approaches, in which requirements are related to software designs, although the transformations are done automatically, the mapping mostly has to be performed manually and is not reusable. In our approach (see Chapter 10), such mappings are also reusable. We note that, in the aforementioned approaches, the rationales that led from the requirements to the architectural views are neither considered nor documented. That is, because the requirements belong to the problem space while ADDs belong to the solution space. In our work, we assume that architectural decision making follows the collection of requirements and precedes the design of software architectures.

2.4.2 Inconsistency Management in Software Architecture

As models are often used for describing ADDs and architectural views, the problem of keeping ADDs and design views synchronized can be seen as an *inconsistency management problem*. Spanoudakis and Zisman see inconsistency management as a multi-step process composed of (1) the detection of inconsistencies, (2) the diagnosis of the cause of inconsistencies, and (3) the handling of inconsistencies [SZ01]. The consistency checking of software designs and the resolution of inconsistencies have been considered as central activities in software evolution and maintenance, and many approaches have been proposed for inconsistency management.

Silva et al. use Prolog-based inconsistency rules to detect inconsistencies in UML models and specify their causes [Sil+10]. In their proposal, a search-based algorithm finds the best plan from a subset of predefined plans expressed as a set of actions applied on the model elements. Egyed proposes to fix inconsistencies in UML design models by executing inconsistency rules, whose execution returns references to the affected model elements [Egy07]. The main idea of this approach is to identify all possible changes and provide to the user the ones which resolve the inconsistency. Nentwich et al. introduce a framework for detecting and fixing inconsistencies in distributed XML documents [Nen+03]. The constraints are described as first order logic formulae and the repairs are generated automatically from predefined mappings. In a different approach, Dam and Winikoff describe software designs as agent systems and use Belief-Desire-Intention (BDI) plans derived

from OCL rules and from a library of repair plan types to express alternative repair scenarios when the constraints are violated [DW11]. Other approaches use graph transformation rules to detect and propose resolutions for various types of inconsistencies in UML models [Men+06] or formalize classified inconsistencies, resolution rules conditions and conclusions for UML models using equational logics [BM11] and description logics [SD06]. In all cases, the consistency checking rules and the repair plans for resolving inconsistencies are predefined. The aforementioned approaches concentrate mainly on the well-formedness of UML models.

Puissant et al. use an artificial intelligence technique based on automated planning which does not require any manually written resolution rules [Pui+12]. In this approach, all repair actions are calculated automatically given the initial state, a set of possible actions, and the desired goal. In our case, the handling of inconsistencies is more complex, as potential inconsistencies have to be resolved in two directions: by changing the architectural view models or by reconsidering the ADDs. The inconsistency problem can be solved also by propagating the changes from one model to the other and by synchronizing the models using bidirectional model transformations. For example, Chechik et al. introduce an algorithm that locates changes for activity and sequence diagrams and use relationships between their elements to propagate changes [Che+09]. For model synchronization using bidirectional transformations the technique of triple graph grammars is widely used (see, for example, [KS06]). Such predefined mappings at meta-model level are not possible in our case, as ADDs are reflected in different ways in the architectural design.

The existing approaches can not deal with the handling of inconsistencies between ADDs and architectural views, as each and every decision leads to decision-specific inconsistencies, different to the inconsistencies that might be caused by another decision. In addition, for each caused inconsistency the intention of the software architect has to be considered. In contrast, the majority of the approaches discussed here aim to generally resolve inconsistency management problems (e.g., for all models based on the UML meta-model) providing automated fixes. We address this challenge as follows: unique constraints are generated for each ADD that is reflected in an architectural view, thus, detected inconsistencies have to be handled in each case separately, dependent on the reusable ADD model and its mapping to the corresponding design view (see Chapter 10 for details).

3 Problem Analysis and Research Approach

In this chapter, we formulate the main research problems that are addressed in the context of this PhD thesis. Each research problem leads us to consider one or more research questions. Furthermore, we discuss the research methodology as well as the research methods that have been applied to evaluate the proposed approaches that address the research questions under study.

3.1 Problem Statement

This dissertation deals with the problem of understanding and supporting the use of reusable architectural decision models in architectural decision making and documentation. That involves among others the consideration and integration of QAs in reusable AK. In addition, the inconsistency management between architectural decisions and designs and the interdependence between variability and architectural decisions are being addressed in the current thesis. In the following, we will discuss the research problems and corresponding research questions in detail.

Capturing, organizing, and effectively reusing AK has gained a lot of attention in recent years by both the academia and industry. However, there is *little empirical evidence* regarding the effectiveness and efficiency of software architects at architectural decision making based on reusable ADDs. Therefore, we would like to investigate how and whether established methods for leveraging reusable AK such as reusable (pattern-based) architectural decision models contribute to more effective and efficient decision making by software architects. The aforementioned problem leads us to consider the following research question:

Research Question 1
Does the use of reusable AK in the form of reusable architectural decision models have a positive effect on the effectiveness and efficiency of architects during decision making and documentation?

To address this research question, in Chapter 5, we describe the consolidation of a reusable decision model on service-based platform integration and observe its use in a case study. In addition, in

Chapter 6, we investigate in an empirical study the supportive effect of reusable decision models in architectural decision making and documentation.

Although QAs are among the most important decision drivers in architectural decision making existing methods and tools supporting ADDs do not address or resolve the *relationships between QAs and ADDs* adequately. Especially for reusable AK captured in reusable architectural decision models, the relationships between QAs and ADDs are often not explicit and in most cases not supported in architectural decision making. Therefore, supporting QAs entangled with ADDs is challenging and more research is needed to provide methods for capturing and documenting the quality properties along with the ADDs and the relationships among them. This research gap leads us to consider the following research question:

Research Question 2

To what extent do existing AK management methods and tools deal with the relationships between ADDs and QAs and how can reusable ADDs be integrated with QAs in order to provide quality-driven decision making support based on reusable AK?

We deal with the first part of this research question in Chapter 7, in which we perform a literature survey on the relationships between QAs and ADDs in existing AK methods and tools. We address the question "how to integrate reusable ADDs with QAs" in Chapter 8.

For a design problem at hand software architects need to consider competing and imprecise requirements, various design alternatives and technology implementations, informal documentations and domain expertise; all these factors introduce many sources of *uncertainty*. Existing approaches do not consider this inherent uncertainty of architectural decision making, which has been until now largely ad hoc and informal, without explicit, automated support. Apart from that, during decision making, software architects need to consider not only the general ADDs but also system and technology options. A further challenge related to making decisions under uncertainty is therefore to consider in this process both application-generic and application-specific knowledge. Thus, we investigate the following research question:

Research Question 3

How can the inherent uncertainty in architectural decision making be resolved – for both application-generic and application-specific knowledge?

We deal with the problem of uncertainty of QAs in architectural decision making in Chapter 9.

Architectural decisions and designs become *inconsistent* as software systems evolve. The main reason for that is that ADDs do not get maintained nor synchronized with the corresponding design views over time. So far there is no formal mapping or automated translation between ADDs

and design views, thus, keeping decisions and designs consistent to each other remains mainly ad hoc and tedious. This leads us to consider the following research question:

Research Question 4

How can reusable ADDs be mapped to design views and how can consistency between reusable ADDs and design models be ensured over time?

We address this research question in Chapter 10.

Keeping ADDs consistent to the corresponding designs becomes more complex in product line engineering where architects have to deal with both the product line and product architectures. In that case, variability decisions need also to keep in line with the architectural decisions and designs. Thus, in this PhD thesis, we have also studied the interdependencies between architectural and variability decisions in the context of product line engineering. Variability management approaches entailing variability decisions focus on describing variabilities in product lines and products. On the other hand, existing architectural design and decision support tools cover various architectural aspects, views, and reasoning of ADDs but lack adequate support for related variability decisions. However, when designing product lines as well as at product line derivation both types of decisions need to be considered. Thus, the research question that emerges is the following:

Research Question 5

How can architectural and variability decisions be systematically integrated in the context of product line engineering?

We propose how to systematically integrate architectural and variability decisions in Chapter 11.

3.2 Research Methods

The research conducted in this doctoral dissertation uses principles of design science research, an iterative methodology including the analysis of a problem, a proposal, implementation of a solution, and its evaluation. For evaluation purposes, mainly primary but also secondary research methods have been applied. Primary research involves the collection and analysis of data, through experimentation, surveys, case studies, etc. while secondary research is based on data from previously published studies for the purpose of research synthesis [Sjo+07]. We discuss here the research methods that have been used and explain briefly their application in the context of the current thesis.

3.2.1 Design Science Research

Design science research produces rigorous, meaningful results for information systems and gives the potential to investigate new technologies and to advance accepted practice – in the absence of a strong theory base – through the construction and evaluation of these systems and their components [VK07]. Design science comprises two central activities, *build* and *evaluate*, which are parallel to discovery and justification in the natural sciences [MS95].

A few frameworks for conducting design science research in Information Systems have been proposed in the literature, including very similar steps and processes though. The framework applied in this thesis can be described as a synthesis of the frameworks proposed by Vaishnavi and Kuechler [VK07] and Peffers et al. [Pef+07]. It includes the following steps: (1) Identify problem & motivate, (2) Design & development, (3) Demonstration & evaluation, (4) Conclusion, and (5) Communication. The last two steps (Conclusion and Communication) may lead to further iteration loops. Figure 3.1 illustrates the application (in simplified form) of design science research for the *Research Question 4*.

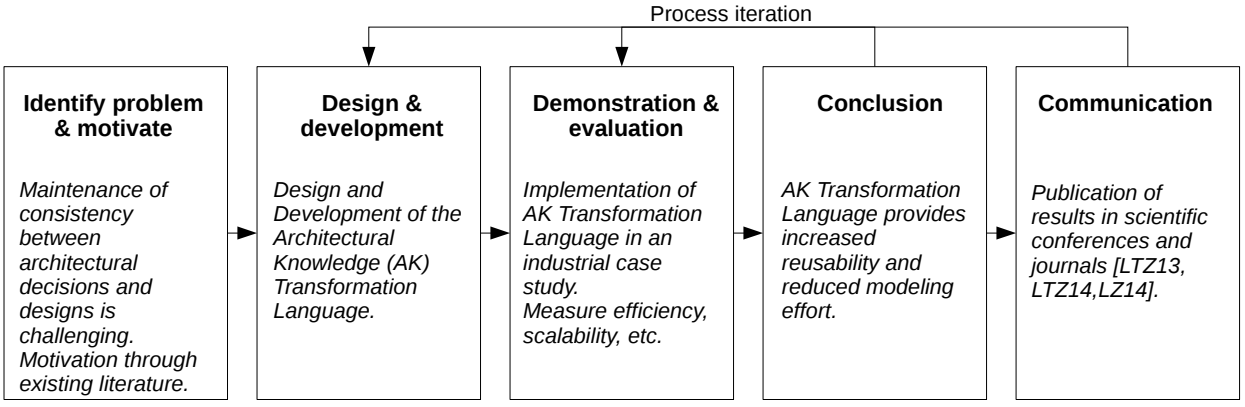


FIGURE 3.1: Application of design science research in *Research Question 4* of the PhD thesis

3.2.2 Case Study

Case studies are characterized as "research-in-the-typical" as they are looking at what is happening on a real project (e.g., through monitoring activities, processes, etc.) [Kit+95]. According to Yin [Yin02] a case study is *"an empirical inquiry that investigates a contemporary phenomenon within its real-life context, especially when the boundaries between phenomenon and context are not clearly evident"*. The aim of performing case studies is to understand how and why some phenomena occur within a specific time space, as well as the mechanisms by which various cause-effect relationships are established [Eas+08; Woh+03]. Two types of case studies exist: *exploratory* case studies that are used to derive new hypotheses and build theories from the investigation of some phenomena, and *confirmatory* case studies that test existing theories [Eas+08].

According to Wohlin et al. case studies are suitable for industrial evaluation of software engineering methods and tools as they can help avoiding scale-up problems [Woh+03]. We have, thus, chosen to evaluate the methods and tools developed in this thesis using industrial case studies that were performed in the context of research projects. During the execution of the case studies we observed, for instance, the architectural decision making by software architects and/or collected data to evaluate e.g., the efficiency and scalability of the introduced tools. Table 3.1 gives an overview of the elaborated case studies. In particular we document on the type of the case study, its purpose, and its reference to the corresponding chapter. The case study "Warehouse Management System" (INDENICA Case Study) was developed in the context of the EU research project INDENICA while the case study "Smart City Ecosystems" was designed in collaboration with Siemens.

Case Study	**Type**	**Used to study...**	**Reference**
Warehouse Management System	exploratory	decision making levels and stages in service-based platform integration	Chapter 5
Warehouse Management System	confirmatory	applicability/appropriateness of pattern language/reusable decision model for service-based platform integration	Chapter 5
Smart City Ecosystems	confirmatory	applicability of integration of QAs and ADDs	Chapter 8
Warehouse Management System	confirmatory	applicability and modeling effort of fuzzy logic based decision models	Chapter 9
Warehouse Management System	confirmatory	reusability, modeling effort, efficiency, and scalability of AK Transformation Language	Chapter 10

TABLE 3.1: Case studies in the PhD thesis

3.2.3 Controlled Experiment

A controlled experiment is an investigation of a testable hypothesis where one or more independent variables are manipulated to measure their effect on one or more dependent variables. The main difference between a controlled experiment and a case study is that an experiment samples over the variables that are being manipulated (controlled study), while a case study samples from the variables representing the typical situation (observational study) [Woh+03].

The independent variables represent the treatments in the experiment (e.g., control vs experiment group) while dependent variables are the variables which are measured to investigate whether they are affected by the dependent variables [Woh+03]. This research method allows us to determine if and how the variables are related in order to find out whether we have a cause-effect relationship between them [Eas+08]. We have performed two controlled experiments in order to find out whether the use of reusable decision models at architectural decision making and documentation can improve the efficiency and effectiveness of software architecture students (potential novice software architects). The controlled experiments and their results are presented in detail in Chapter 6.

3.2.4 Grounded Theory

Grounded theory is a qualitative strategy in which the researcher derives a general, abstract theory of a process, action, or interaction grounded in the views of participants in a study [Cre08]. Although the research conducted in the context of this dissertation is not based on grounded theory, we have used principles of the grounded theory method in order to observe and analyze architectural decision making in service-based platform integration performed by architects (see Chapter 5 for details).

3.2.5 Literature Review/Literature Survey

Literature reviews and surveys are methods that provide research synthesis [CH94]. The goal of a literature review, systematic literature review, and literature survey is to summarize and integrate findings of studies on a topic or research question. They are important for informing research and practice but also for identifying crucial areas and questions that have not been studied adequately in the literature [Sjo+07], as well as defining future trends and challenges. In the context of this dissertation, a literature survey on the use of QAs in architectural decision tools has been performed (see Chapter 7). In addition, a systematic literature review of pattern literature related to service-based platform integration has been conducted in the context of a quality multi-method study, with the aim to consolidate a pattern language on service-based platform integration (see Chapter 5).

Part II

Decision-making Support for Reusable Decisions

4 Modeling & Using Reusable Architectural Design Decisions

4.1 Introduction

As discussed in detail in the previous chapter, ADDs have gained much attention and importance in both research and industry. In addition, the documentation of ADDs for providing architectural guidance in software projects has recently found application in industrial practice [GB14; Zim+08]. We claim that capturing design solutions and their rationale is important not only for the experienced software architects but also for novice software designers who usually need to be educated on the existing AK and the systematic reasoning on ADDs. Reusable decision models address the systematic documentation of ADDs and provide guidance during decision making for recurring design issues. In Section 2.2, we provided a few examples of reusable decision models that have been documented in the literature (e.g., for designing SOAs) [Zim+07]). Although many tools have been developed to support architectural decision making and documentation (refer to Section 2.2 for a tool comparison list), little guidance exists on the systematic modeling and using of reusable decision models and very few empirical evidence can be found about whether the use of reusable decision models has a positive effect on architectural decision making.

In this chapter, we introduce the concepts constituting the basis of the tools ADvISE and CoCoADvISE, which have been developed in the context of this thesis. In particular, we present the principles of the Eclipse-based tool ADvISE and Web-based tool CoCoADvISE for reusable ADDs, namely we introduce the common ADvISE/CoCoADvISE meta-model and explain how they provide decision support. CoCoADvISE provides, additionally to ADvISE, support for collaborative decision making. We discuss, thus, in detail how CoCoADvISE allows the definition, resolution, and enforcement of constraints in collaborative decision making. Our main contribution here is not the tools themselves but the evaluation of the concepts behind them. The tools ADvISE and CoCoADvISE are afterwards evaluated in various empirical studies in the following chapters. Therefore, the current chapter provides the basis of our research work on supporting reusable ADDs presented in this dissertation.

4.2 ADvISE/CoCoADvISE Approach

In this section, we introduce the main concepts of ADvISE and CoCoADvISE and give an overview of the related tool support. In particular, we discuss their decision meta-model (see Section 4.2) and core functionalities (see Section 4.3), and provide tool implementation details (see Section 4.4). As mentioned before, CoCoADvISE is the Web-based version of the Eclipse-based tool ADvISE and was developed at a later time, mainly for the needs of the prototype's evaluation in controlled experiments and in order to provide collaboration support. Both tools, however, share the same concepts and common functionalities.

Reusable Decision Meta-model

ADvISE and CoCoADvISE provide tool support for modeling reusable ADDs inspired by the Questions, Options, and Criteria (QOC) approach [Mac+91]. In the QOC approach, the *Questions* represent design issues structuring the space of alternatives, *Options* constitute the potential alternatives and *Criteria* correspond to the design drivers for choosing among the alternatives. ADvISE/CoCoADvISE decision meta-model extends the traditional QOC approach with – among others – additional dependencies between options, such as enforcement or inclusion, and between options and questions or decisions (i.e., an option triggers a next question or decision). In addition, it supports the categorization of reusable solutions (i.e., options in QOC) such as design patterns, technology-related solutions, and so on. It also shares common concepts with other reusable decision meta-models that have been introduced in the literature like Zimmermann et al.'s meta-model [Zim+07] such as the relationships between ADDs (e.g., an ADD enforces another ADD, or an ADD is incompatible with another ADD) [Zim+09].

ADvISE and CoCoADvISE introduce a reusable decision meta-model (see Figure 4.1) for the design space of certain application domains, consisting of Decisions, Questions, Options, and Solutions, as well as various relationships among them. For each design issue, a set of Questions (including one or more first Questions) providing multiple Options has to be modeled. Examples of relationships are that a selection of an Option triggers a follow-up Decision or an Option is incompatible with or enforces another Option. A selection of an Option may lead to a Solution to be suggested to the software architect. This Solution will be applied using Architecture Guidance, that is Technology-related Architectural Knowledge (AK) (potentially related to a Technology), a Design Pattern, an Architectural Pattern, or a Composite Reusable Solution combining any of the aforementioned Architecture Guidances. Of course, the meta-model can be extended to include any other kind of reusable Architecture Guidance such as reference architectures, best practices, etc.

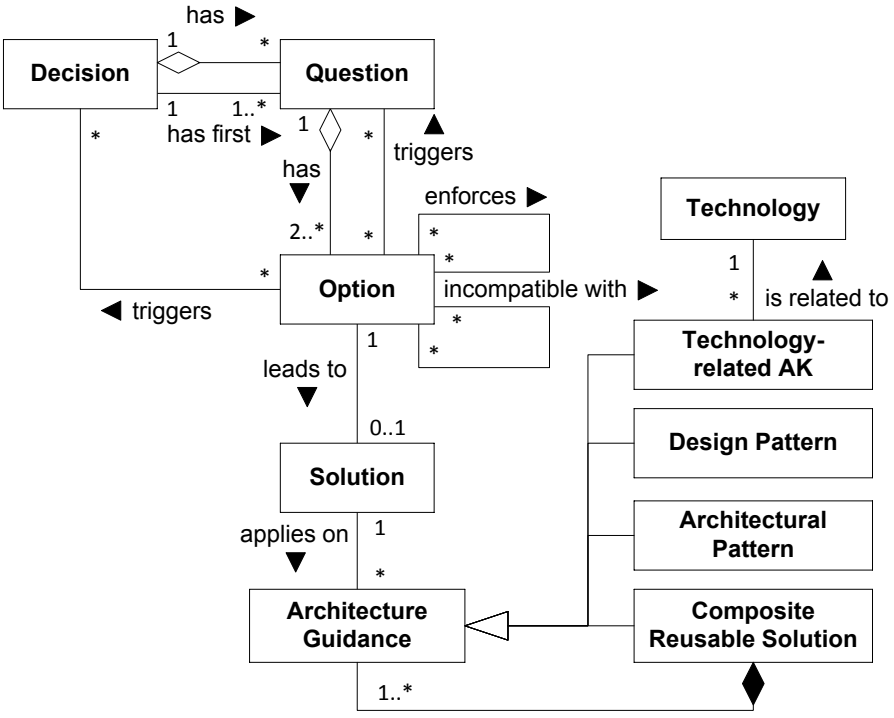


FIGURE 4.1: Reusable decision meta-model

4.3 Decision Support Based on Reusable Decision Models

The advantage of ADvISE/CoCoADvISE's reusable decision models is that they are created only once for a recurring design situation and can be reused multiple times following the model-driven paradigm. In similar application contexts, corresponding questionnaires can be generated and used for making concrete decisions. Based on the outcomes of the questionnaires answered by software architects through the decision making process, ADvISE/CoCoADvISE can automatically resolve potential constraints and dependencies (e.g., reveal follow-on questions and decisions, deactivate incompatible options, etc.), recommend best-fitting design solutions, and eventually, generate half-filled ADD documentations.

Figure 4.2 depicts the main elements present, as well as the stakeholder roles involved in the ADvISE/CoCoADvISE tool. At design time, the modeling of the reusable decision models is supposed to be performed by experienced software architects, while at execution time, the software architects (i.e., members of the same development team) are guided through questionnaires to make and document ADDs. During the editing of questionnaires by the architects, ADvISE and CoCoADvISE guarantee the consistency of the decision model instances, i.e., by automatically hiding, showing, disabling, etc. specific parts of the questionnaires like questions and available options.

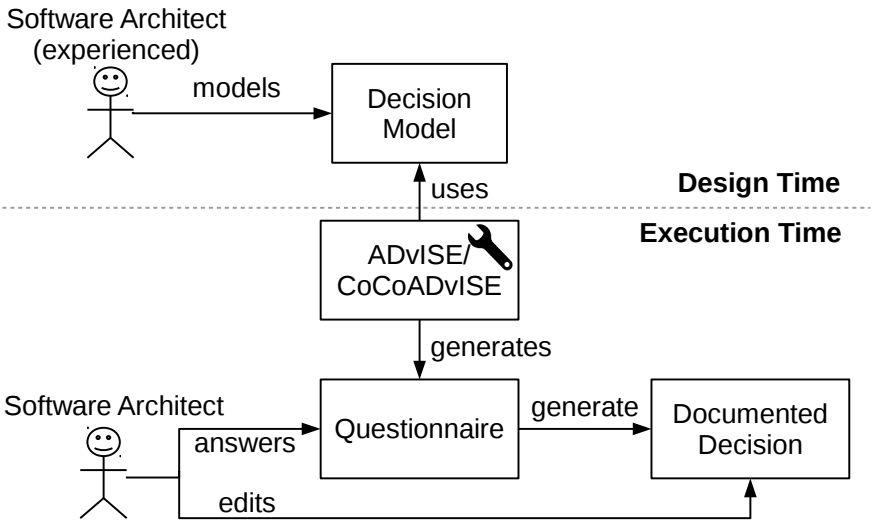


FIGURE 4.2: General overview of ADvISE and CoCoADvISE tools

4.4 Prototype Implementation Details

While ADvISE is implemented as a set of Eclipse plug-ins, CoCoADvISE has been developed as a Web-based application. The Eclipse Modeling Framework (EMF)[1] project has been used to create the architectural decision meta-model and the code for editing architectural decision models in the ADvISE tool. CoCoADvISE is a mostly client-side executed Single-page Web application that is implemented using Google's AngularJS[2] framework. The back-end of this Thin Server Architecture is based on the Node.js[3] framework and runtime environment. For the needs of the collaboration support, it utilizes the real-time synchronization engine Racer[4].

ADvISE is based on Eclipse plugins and allows the modeling (see ⓐ of Figure 4.3) and leveraging (see ⓑ of Figure 4.3) of reusable architectural decision models.

CoCoADvISE has a similar user interface with ADvISE. Figure 4.4 contains some screenshots of the use of CoCoADvISE. Software architects can choose from a list of reusable decision models (see ①) and generate one or more questionnaires based on these models. The questionnaire provides architectural decision guidance through single-choice questions which lead to follow-on questions, decisions, and recommendations as indicated in ②. Finally, software architects may generate semi-complete ADD documentations such as the one shown in ③ based on design solution recommendations. CoCoADvISE fills in automatically some of the fields of the ADD documentations in template form [TA05]; architects need to fill in afterwards the rest of the required information.

[1] http://www.eclipse.org/modeling/emf
[2] http://angularjs.org
[3] http://nodejs.org
[4] http://github.com/codeparty/racer

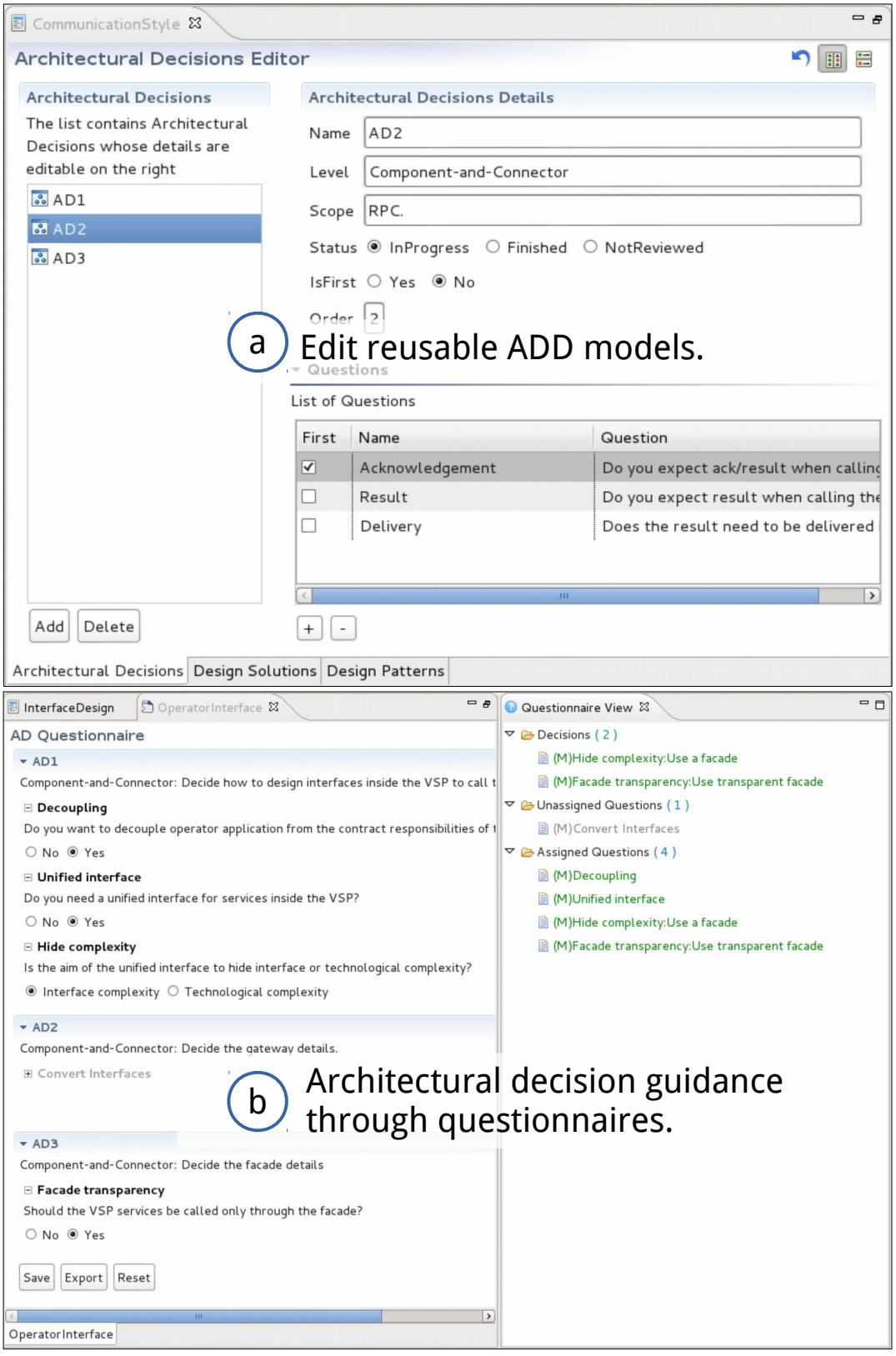


FIGURE 4.3: Modeling reusable decision models and decision making in ADvISE

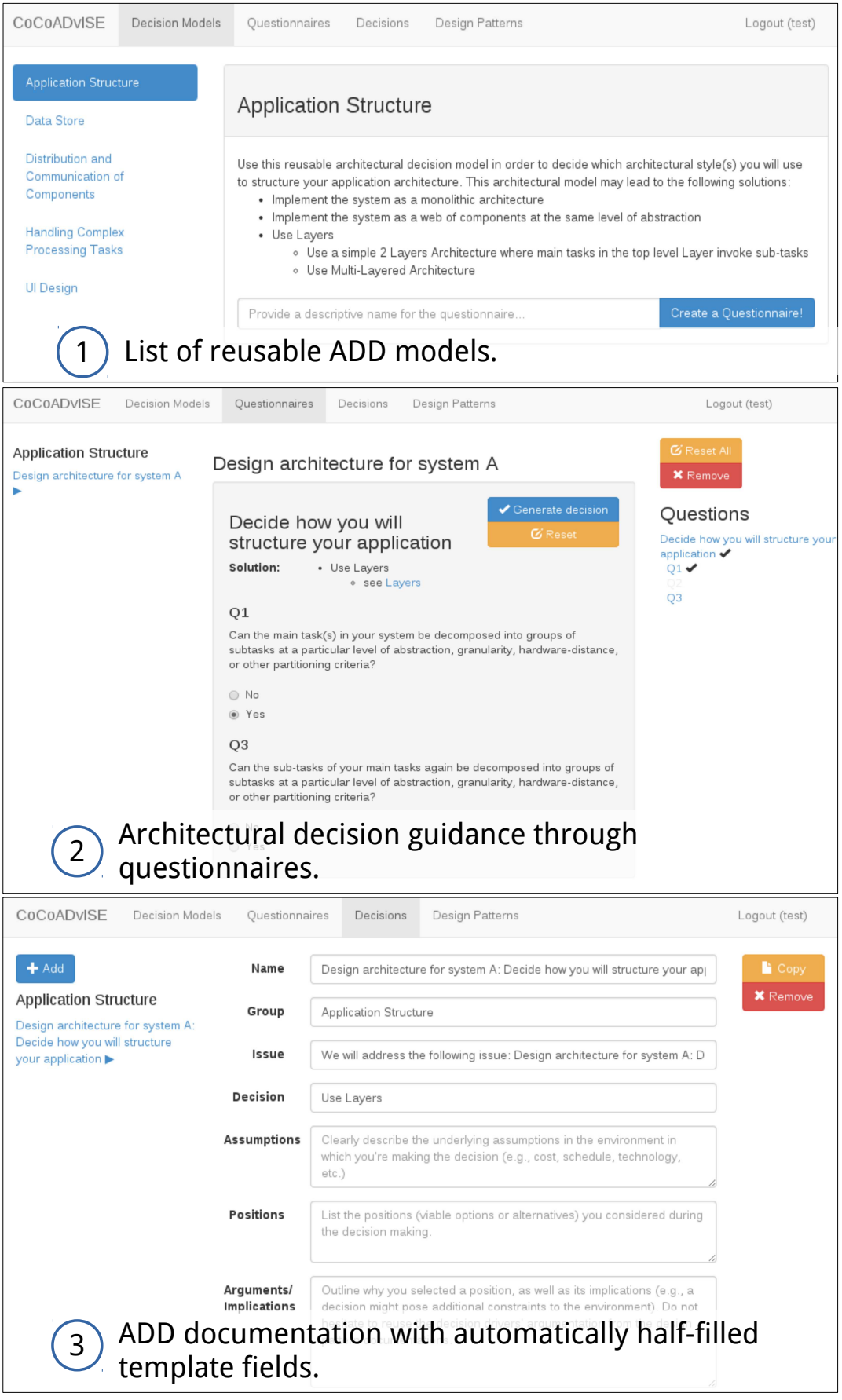


FIGURE 4.4: Decision making and documentation in CoCoADvISE

Although both tools share common functionalities ADvISE and CoCoADvISE still have different goals and shall be used by different stakeholders and for different purposes. The similarities and differences between the two tools can be viewed in the comparison Table 4.1.

ADvISE	**CoCoADvISE**
Started development in January 2012	Started development in October 2013
Shall be used by experienced architects (for creating reusable decision models)	Can be used by (novice) architects
Eclipse-based	Web-based
Supports modeling of reusable decision models	Does not support modeling of reusable decision models
Decision making support based on questionnaires	Decision making support based on questionnaires
Does not support constrainable collaborative decision making	Supports constrainable collaborative decision making

TABLE 4.1: Comparison between ADvISE and CoCoADvISE

4.5 Resolving Constraints in Collaborative Architectural Decisions

In this section, we motivate the collaboration support in CoCoADvISE, present the definition and enforcement of constraints in decision making and documentation, and summarize the results of an empirical study performed with the tool. As collaborative architectural decision making is not the focus of this thesis, we provide here only a brief presentation of the collaboration support in CoCoADvISE and its empirical evaluation – a formalized definition of the collaboration constraints and a detailed presentation of the empirical validation of CoCoADvISE can be viewed in [Gau+15].

4.5.1 Collaboration in Architectural Decision Making

Although making and documenting ADDs become increasingly important in the process of software architecting, the remoteness of different decision stakeholders, ranging from local distribution in an office environment to globally distributed teams, as well as the different domain knowledge, expertise, and responsibilities of the stakeholders hinder effective and efficient collaboration. The trend of globally distributed projects in software and IT industries makes collaboration and coordination within dispersed teams challenging [HG99]. Unlike co-located project teams, geographically distributed partners need to overcome collaboration problems caused by the distance, different concerns of stakeholders, and different development processes. While these challenges are particularly problematic in distributed project settings, even in a locally distributed environment like offices on multiple floors, efficient and effective collaboration is an issue.

Existing tools and methods that target collaborative architectural decision support (see, e.g., [NP13; Sch+07; Tse12; Sha+10; Lia+09; BG07; Cle+10]) mainly focus on team building, achieving

consensus, making of trade-offs, and sharing of architectural knowledge, but do not consider the various stakeholders' decision making constraints, for instance, due to their roles in the development process. This is despite the fact that such roles (see, e.g., [Nor+09; SP02; Eck10]), their potentially diverging rights and duties, and their potentially conflicting objectives (see, e.g., [Raa+08; Nak+13; Ova+03]) within a possibly constrained (see, e.g., [CS07; Kof+09; JS10]) decision making process actually have been documented in related literature.

In addition, in recent years, software development governance has been recognized as a key component for steering software development projects. Kofman et al. frame the ideal software development governance environment [Kof+09]: *"Every member of a team would know at any given point of time, what needs to be done, who is responsible for which task, and how to perform the tasks for which he or she is responsible"*. In the course of developing the IBM Software Development Governor prototype they also observed and documented the need for decision making constraints. They mention decision making policies, such as framing the boundaries of the decision (i.e., who should participate and when), or specifying how the decision is to be made (i.e., consensus or voting). Jensen and Scacchi noted that *"there are many issues critical to governing software development, including decision rights, responsibilities, roles, accountability, policies and guidelines, and processes"* [JS10].

In summary, architectural decisions are key decisions in software development processes, that can – in certain contexts – be subject to software development governance including the entailing of decision making policies.

4.5.2 Motivating Example

To illustrate the concept of collaborative architectural decision making and to motivate the need for automatic enforcement of decision making constraints, let us consider a fictive online retailer company that wants to provide third-party online stores with a Web service for purchasing books and electronic gadgets. Based on the functional as well as non-functional requirements, mainly concerning secure and encrypted transactions, the company has to decide on the kind of Web services that eventually will be deployed. For decisions that have such far-reaching implications throughout the system, company policies require that the involved software development teams, which are spread across several, geographically distributed offices, participate in the decision making process collaboratively. In particular, the following decision making constraints can be derived from the company policies: First, all decision stakeholders with the role *Software Architect* shall *unanimously* decide on the type of Web service to be exposed. *Application Developers* shall decide on technical details like a concrete transport protocol, and *Software Architects* have to eventually confirm and approve these decisions. *Security Experts* shall propose a solution for the security and encryption related decisions. Finally, representatives from selected third-party companies shall be

allowed to participate in the decision making process. However, their votes can be overruled at any time by internal stakeholders of the retailer company. The company might now rely on guidelines that tell each stakeholder to exactly know their roles and duties within this decision making process and to stay compliant to these constraints in any decision process they participate in. However, with a growing number of stakeholders, stakeholder roles, responsibilities, duties and other forms of constraints, it becomes nearly impossible for a single participant to not (unintentionally) violate some of these policies. In the following sections, we will show how CoCoADvISE automatically enforces such constraints.

4.5.3 Collaboration in CoCoADvISE

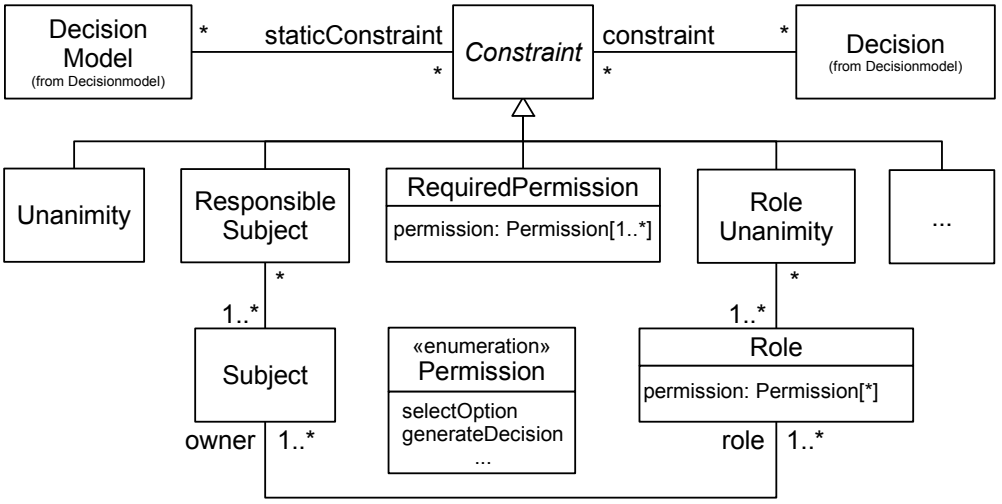
In order to support collaborative decision making in CoCoADvISE, we propose a meta-model for a set of decision making constraints which we integrate with the reusable architectural decision meta-model introduced in Section 4.2. In addition, the real-time collaboration features of CoCoADvISE enable multiple, possibly geographically dispersed software architects and stakeholders to participate in the group decision making and documentation process. CoCoADvISE employs a model-driven approach in which reusable decision models are made subject to constraints at design time. At runtime, the collaboration related constraints are automatically enforced.

4.5.4 Definition and Enforcement of Decision Making Constraints for Collaboration

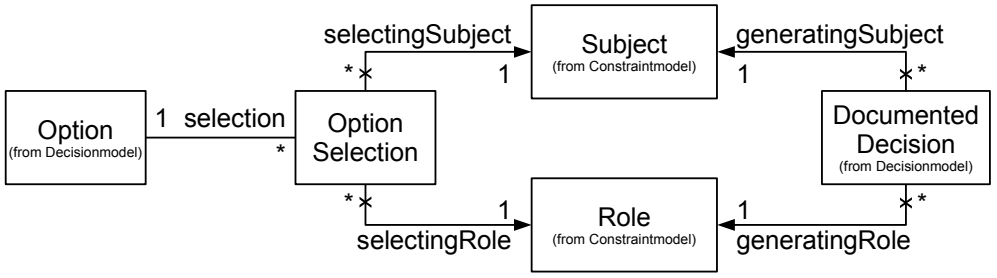
In CoCoADvISE, reusable decision models are augmented with additional constraint elements. This section gives an overview of the generic meta-model for the specification of decision making constraints for reusable architectural decision models and explains how these constraints are enforced at runtime.

The class diagrams depicted in Figure 4.5 provide a conceptual overview of the essential concepts of a Constrainable Decision Meta-model including Subjects, Roles, Permissions, and various types of Constraints. Technically, the Constrainable Decision Meta-model extends the previously presented Decision Meta-model (see Figure 4.1) with additional elements, relevant at design time (i.e., Figure 4.5(a)) and execution time (i.e., Figure 4.5(b)).

CoCoADvISE provides two separate mechanisms for defining decision making constraints. First of all, we can make particular reusable architectural decisions subject to decision making constraints by assigning Constraints to Decisions. In addition, all reusable architectural decisions of a decision model can be made subject to decision making constraints at once by assigning the corresponding Constraints to the Decision Model, instead of a single decision. These two mechanisms are inherited by all subclasses of the abstract Constraint class (e.g., Unanimity or Responsible Subject).



(a) Constraint model extension



(b) Runtime model extension

FIGURE 4.5: Conceptual overview of CoCoADvISE's constrainable decision meta-model

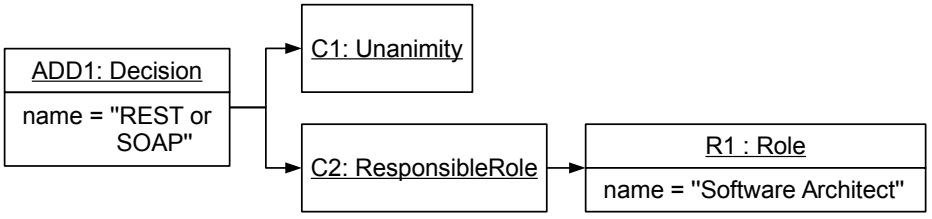


FIGURE 4.6: Exemplary constraint model

As an example, let us consider the decision about what kind of Web services will be exposed: "REST or SOAP". Figure 4.6 depicts how the first set of decision making constraints can be implemented in CoCoADvISE. By defining both a Responsible Role and a Unanimity constraint we can formalize the requirement that stakeholders with the role *Software Architect* have to *unanimously* decide on the type of Web service. Note that we have to parametrize the Responsible Role constraint by assigning the Role object, which represents the *Software Architect* role, to it. On the contrary, a Unanimity constraint does not need any further parametrization. According to Figure 4.5(a) the three additional classes that are used for parametrization of some constraints are: Subject, Role, and Permission. A Subject represents a human person, i.e., a user/stakeholder

of the system (see, e.g., [SM11]). Roles are assigned to subjects, and through their roles the subjects are granted Permissions. A Permission models a right to perform a certain action within the system. In CoCoADvISE we mainly focus on constraining two fundamental actions, i.e., selecting options and generating decisions. Figure 4.5 reflects this circumstance by explicitly listing the corresponding permissions selectOption and generateDecision. Other permissions could be added to the tool as extensions. Figure 4.5(b) shows how the original CoCoADvISE decision meta-model has been extended with additional runtime-specific elements and relations. For the purpose of enforcing certain decision making constraints we have to introduce – among others – the concept of an Option Selection. At execution time, the existence of an Option Selection object means that the corresponding Option has been selected by a user (i.e., by executing a selectOption action). More precisely, the relations selectingSubject and selectingRole of this Option Selection object are supposed to point to the user's corresponding Subject respective Role objects. Analogously, the generatingSubject and generatingRole relations of a Documented Decision object are supposed to refer to the Subject and Role objects of the user who has generated a decision (i.e., executing a generateDecision action).

We omit here the formal definition of the meta-model's elements and mappings, as well as the semantics of the introduced constraints, as these do not belong to the main contributions of this thesis. For a complete reference to the formal constraints and their semantics the reader can refer to [Gau+15].

The actual constraint enforcement logic is mainly embedded in the user interface of the collaborative Web application. CoCoADvISE also guarantees the consistency of the decision model instances, i.e., by automatically hiding, showing, disabling, etc. specific parts of the questionnaires and the user interfaces in a way that users simply can not violate any defined constraints at all. Figure 4.7 visualizes a possible runtime situation of the previously defined decision (see Figure 4.6). We can see that the specified responsible role constraint is enforced by showing the "generate" button only to those users that own the required role (i.e., User 1, which owns the role *Software Architect*), while hiding it for all other users (i.e., User 2). Given that User 1 chooses option 2 and User 2 chooses another option 1, the system correctly enforces the unanimity constraint by disabling the "generate" button. As soon as all users have unanimously agreed on the same set of options, the system will eventually enable the button again, allowing User 1 to generate and edit ADD documentations.

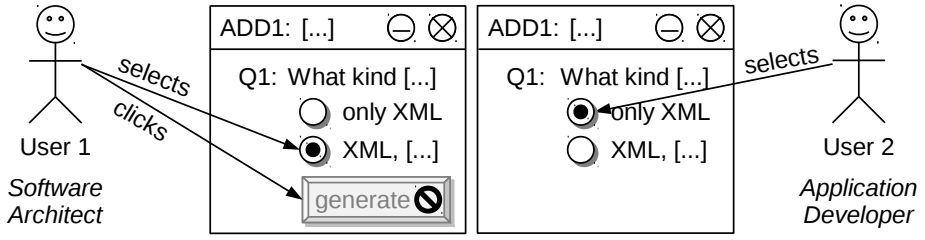


FIGURE 4.7: Enforcing exemplary constraints at execution time

4.5.5 Empirical Evaluation

The collaborative part of CoCoADvISE has been validated for the efficiency and effectiveness of supporting automatic enforcement of constraints for humans in the context of a controlled experiment with 48 participants. In this experiment, teams consisting of two different stakeholders had to make design decisions (for a software system) subject to various constraints collaboratively. The experiment group used a version of CoCoADvISE where the constraints were integrated and automatically enforced (i.e., they could not be violated in any way), while this feature was deactivated for the control group. The experiment provided evidence that automatic enforcement of decision making constraints significantly increases both the time and effort related efficiency and effectiveness of users working with decision making tools.

In particular, we found that:

- The experiment group was able to perform more work tasks than the control group, i.e., its participants were *more effective* than the participants of the other group.
- The experiment group needed less time than the control group, i.e., its members were *more efficient* than the members of the other group.
- The experiment group performed less actions than the control group, i.e., its participants were *more efficient* than the participants of the other group.

In order to better understand the adverse effects of constraint violations, we also performed a linear regression analysis. In particular, we derived three different linear regression models that can be used to quantify the effect of constraint violations on a user's effectiveness and efficiency. These regression models fortified and complemented our main findings. While our hypotheses dealt with finding evidence that automatic enforcement of constraints is beneficiary in terms of effectivity and efficiency in general, the regression models provided further insights into (1) what exactly are the adverse effects of constraint violations and (2) to which extent they influence the effectivity and efficiency of users. The design, execution, and detailed results of the controlled experiment can be viewed in [Gau+15].

4.6 Conclusions

To summarize, in this chapter, we introduced a reusable decision meta-model that has been used as a basis for building the Eclipse-based tool ADvISE and the Web-based tool CoCoADvISE for supporting architectural decision making and documentation. Aim of the introduced tools and accompanying meta-model is to guide software architects in making and documenting recurring

ADDs based on reusable AK. In addition, in the context of CoCoADvISE, we have presented the definition and automatic enforcement of constraints in real-time collaborative architectural decision making based on reusable architectural decision models. ADvISE and CoCoADvISE have been used as basis in the remaining chapters for performing empirical studies and investigating issues related to reusable ADDs and their support.

5 Architectural Decision Making for Service-Based Platform Integration: A Qualitative Multi-method Study

5.1 Introduction

The initial goals of the qualitative multi-method study we present in this chapter were to consolidate and validate a reusable decision model, and afterwards to study the process of architectural decision making based on reusable – mainly pattern-based – ADDs modeled in such a reusable model. Furthermore, the steps performed in this study can also be seen as a systematical guidance for consolidating and modeling reusable architectural decision models for software architects.

The context in which we performed a series of qualitative studies following a multi-method approach is the service-based integration of heterogeneous platforms and the development of applications on top of those integration services. A *software platform* is a collection of software sub-systems, like communication middleware and databases, and interfaces which together form a reusable infrastructure for developing a set of related software applications. To build a concrete application by reusing software artifacts in a platform, the platform lays out a customization and configuration process on top of its interfaces [Gha+12]. A software platform, therefore, abstracts from details inside and underneath the platform and thereby facilitates developing, maintaining, and deploying domain-specific software applications. Today, many software systems are based on software platforms. Examples of software systems based on software platforms include SAP R/3 for enterprise resource planning, the Facebook Platform[1] for social networks, Amazon's S3[2] for data storage, etc.

The selection of the specific domain was related to the scope of the EU research project INDENICA[3], in the course of which the current study was performed. In the following sections, we first discuss our main research questions and objectives of our qualitative multi-method study, then present the design and execution of the study, and finally report on the main findings.

[1] https://developers.facebook.com
[2] http://aws.amazon.com/s3
[3] http://www.indenica.eu

5.2 Research Questions of the Qualitative Multi-method Study

Certain design decisions related to a given technical domain, such as SOA, are found to be taken repetitively. This is due to certain design decisions reflecting established design knowledge in the field or certain technology or development process choices being firm requirements in the field. In this context, we address the following research question:

What are the recurring ADDs on service-based platform integration documented by existing patterns and pattern collections?

In the discussion about the types of ADDs in Section 2.2, we have seen that early works on architectural decision modeling (such as [TA05]) have stressed application-specific AK, while in more recent works (such as [Zim+07]) also decision models for application-generic AK have been proposed. When integrating heterogeneous service platforms for supporting domain-specific software applications, decisions ranging from high-level architectural decisions to implementation-, technology-, and application-dependent decisions must be tackled to specify the system's architecture. Motivated by these findings on multi-level decision-making and based on our reusable decision model on service-based platform integration, we investigated:

What are the levels of decision making when designing an architecture for service-based platform integration?

To address these two research questions, we conducted a series of *three* qualitatively-driven studies on architectural decision making. To identify patterns relevant for service-based platform integration solutions, as well as their potential relationships and the relevant forces and consequences of applying the patterns, we performed a *systematic literature review* [Kit+09]. We reviewed in depth 402 patterns from 11 pattern collections. The result was a candidate pattern language featuring 40 patterns (31 plus 9 referenced patterns). From this pattern language we derived a pattern-based architectural decision model. Next, we performed *interviews* with platform experts [Sea99; Rob02] to validate the AK documented in the pattern language and the architectural decision model. Finally, we participated in an industrial *case study* [Sea99] on service-based platform integration.

By integrating the results of the three studies to answer the two research questions, we arrived at two major contributions: First, we documented a pattern language and a pattern-based architectural decision model for service-based platform integration [Lyt+12b], validated and refined through expert interviews and a case study. Second, the analysis of the decision-making process has led to a general model identifying the levels and stages of architectural decision making in service-based platform integration.

5.3 Research Study Design

While designing our study, we faced the problem that the two research questions are substantially different in terms of their views on architectural decision making; while pattern descriptions relate to architectural decisions and decision drivers content-wise, the issue of decision-making levels requires a process view. A single research method could not accommodate both views. Also, the research questions required both *exploratory* and *confirmatory* approaches. For example, after having identified the candidate pattern material and architectural design decisions as initial results, their *recurring* character should be explained based on practitioners' experience. At the same time, the research questions are closely related, given that different patterns can address different decision-making levels. Consequently, we adopted a multi-method design [TT10], following a sequential timing for three qualitative study projects.

Figure 5.1 illustrates the research study design: The first method applied was a systematic literature review from which we distilled an initial pattern language and a preliminary architectural decision model according to the procedures in [Har+07; Zim+08]. The systematic review step was performed by four reviewers. The second instrument was intended (1) to validate and to improve our pattern language and decision model, as well as (2) to explore how decision making in platform integration architectures is performed. For this, we carried out semi-structured interviews [Rob02] with experts on three platforms used in industry. The interviews' data were synthesized using the constant comparison method [Sea99] and used to revise the pattern language and the decision model. Based on the integrated interview findings, we designed a case study which was then performed as the third and final inquiry step. With this, we were able to document specific ADDs in the decision-making process and design the architecture of the (case study's) software system. In the subsequent sections, we elaborate on the individual study parts in greater detail.

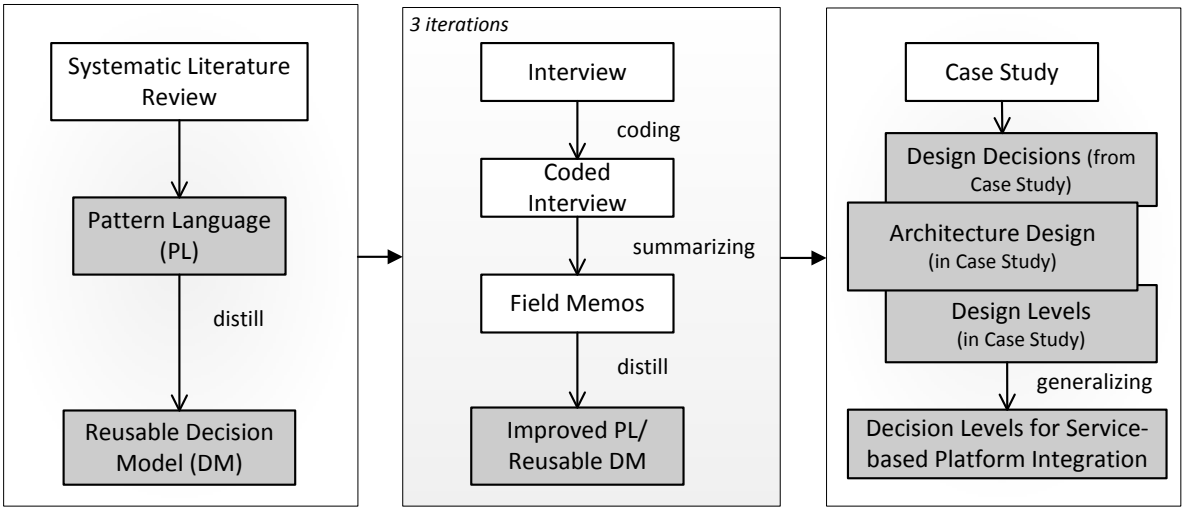


FIGURE 5.1: An exploratory, sequential, qualitative multi-method design

5.4 Systematic Literature Review

As an initial step, a systematic review of the pattern literature was conducted to identify, gather, and filter relevant pattern descriptions from pattern sources on distributed system design [Bus+07a], enterprise application architecture [Fow03], messaging [HW04], remoting middleware [Völ+05], service-oriented systems [HZ09], service design [Dai12], and process-driven SOA [HZ12]. In addition, software design [Gam+94] and software architecture [Bus+00; HZ09] patterns have been consulted. The goal of this systematic review was to derive a solution space for the technical domain of service-based platform integration, by integrating selected subsets of these pattern collections into a dedicated pattern language.

The practice of systematically selecting and integrating parts of existing pattern languages while adding new, context-specific relations between the integrated patterns has been reported in [Bus+07b; HZ09]. While pattern catalogs, pattern collections, and pattern languages have been used before to extract and populate decision models [Har+07; Zim+08], our approach based on a systematic pattern literature review (following the guidelines by [Kit+09]) is novel. The steps performed in the systematic review process are documented in the following subsections.

5.4.1 Review Questions of Literature Review

The systematic review was guided by the following questions:

Review Question 1: *Which patterns or pattern collections exist that support designing service-based software systems?*
Motivation: Platform clients and the underlying platforms require and expose their functionality in terms of services. With this, their architectures and designs incorporate service-specific design decisions to a certain extent (e.g., a layer for service handling). In addition, the intermediate integration platform is built for integrating heterogeneous service endpoints (e.g., bridging different invocation styles). Any pattern collection, and even solitary pattern description, documenting service-based system design from various perspectives (e.g., communication styles, adaptation, configuration, transport handling) is therefore considered relevant.

Review Question 2: *Which patterns or pattern collections document the integration of service-based software systems?*
Motivation: The requirements for service integration are substantially different to alternative application integration strategies (e.g., shared repositories) and demand different design decisions. Looking at service-based systems (platform client, integration platform, and platforms), critical paths for integrating exposed and required services, as well as the underlying component interactions must be identified. The pattern collections should lay out design practices and decision drivers at all levels (e.g., service interface design, process synchronization, communication primitives).

Review Question 3: *Which patterns and pattern collections describe the architecture and design of an integration platform?*

Motivation: The respective pattern collections and pattern descriptions should characterize the properties of software platforms, in general. More specifically, we require patterns which structure an integration platform internally (e.g., its layers, central components and connectors) and in the overall architecture between platform clients and integrated platforms (e.g., as a specific integration layer). Also, pattern material is sought which describes how existing components (e.g., process engines) can be adopted to play the architectural role of an integration platform.

5.4.2 Pattern Search and Selection

The process of searching relevant pattern descriptions and pattern collections was performed manually by four reviewers in a coordinated manner. In a first step, the reviewers held a brainstorming session and identified initial pattern candidates based on their experience in the field. Second, conferences, journals, and book series specific to the pattern community were selected. This selection corresponds to the established stages of publishing pattern material, with pattern descriptions and pattern collections first being disclosed to a pattern audience at a PLoP conference. The PLoP conference series guarantees that the pattern material has been reviewed and edited under the guidance of a shepherd, reviewed by a program committee, and discussed in a writer's workshop. The pattern material may be afterwards submitted to a pattern journal, such as LNCS Transactions on Pattern Languages of Programming (TPLoP), which subjects the material to the additional scientific review process of an academic, archival journal. Alternatively, pattern material may grow into a book publication. We considered papers originating from 33 conference proceedings of PLoP and EuroPLoP, two issues of an archival journal (TPLoP), five pattern collection books, and 25 pattern books.

Inclusion and Exclusion Criteria Patterns and pattern collections published till February 20th, 2012[4] were included, provided that the articles met certain minimum requirements: Firstly, the pattern or pattern collection concerns one of the review questions reported before. Secondly, the article presents one or more patterns, in one of the shapes established in the various pattern communities. In particular, the patterns are described in an established pattern form [Cop96] as single patterns, pattern compounds, pattern stories, pattern catalogs, or pattern languages (see [Bus+07b]). In our in-depth analysis we studied all patterns with regard to the review questions and excluded those that are not addressing one of the review questions. In total, we selected 11 sources with 402 patterns for further in-depth analysis. For duplicate reports of entire pattern collections, the most recent version was included in our review. The reason is that all occurrences of multiple versions in our review qualified as improvements of the pattern collection over time.

[4]The underlying study was performed in the first quarter of 2012.

Quality Assessment Each pattern publication identified was evaluated according to the following criteria: (1) At least a single version of the pattern collection must have been reviewed by a pattern audience (e.g., a writer's workshop at a PLoP conference or a TPLoP journal review); (2) At least three known uses are reported; (3) The pattern or pattern collection was considered by the related work [Zim+07] on documenting ADDs in the SOA technical domain.

Pattern Extraction The data extracted from the search phase were the publication sources, a categorization of the pattern material (e.g., single pattern, pattern story, etc.), short summarizing descriptions, pattern forces and consequences, and other referenced patterns. In accordance with the review guidelines in [Kit+09], the extraction was performed by the reviewers independently from each other.

Synthesis We selected 31 out of the 402 patterns in our in-depth analysis for inclusion in the pattern language, plus 9 patterns that are referenced by the pattern language. Please refer to Table A.1 (Appendix A) for a summary of the pattern sources along with the selected publications per source. Also, refer to Appendix B for a complete reference of the derived pattern language on service-based platform integration.

5.5 Interview Data Collection and Analysis

The interview instrument was carefully designed using the guidelines by Hove and Anda [HA05]. We performed three interviews with nine experts from three different companies that offer three different software platforms supposed to be integrated. The platform experience of the interviewees varied between one to four years, and most of them had more than three years of industry experience. Almost all of them had experience in service technology and were either software designers or developers for the platforms. The roles as well as the years of experience of the experts in the industry and with the specific platforms[5] are given in Table 5.1.

The interview instrument was mainly based on the reusable decision model and consisted of five categories (questions about *adaptation and integration*, *interface design*, *communication style* and *communication flow*, as well as *technical details*) with 29 questions in total. This questionnaire comprised open-ended, as well as closed-ended questions. Out of the 29 questions, 24 questions were based on general AK and five concerned technical platform details needed for preparing the case study. The complete interview guide can be viewed in A.2 of Appendix A.

The interviews varied between 100 and 150 minutes in length. The interviews were recorded on site, using field notes which were then transformed into a structured format through a coding

[5] A detailed description of the platforms and the integration scenario will be given in Section 5.8.

Interviewee	Role type	Platform experience[1]	Industry experience	Experience with services	Platform experience
1	design	WMS	10	4	yes
2	implementation	WMS	8	0	no
3	variability	WMS	10+	2	no
4	variability	WMS	1.5	0	yes
5	design/implementation	RMS	5+	1.5	yes
6	design/implementation	RMS	4	1.5	yes
7	design/implementation	RMS	10	1.5	yes
8	design/implementation	YMS	7	4	yes
9	design/implementation	YMS	1	1	yes

[1] (WMS) Warehouse Management System, (RMS) Remote Maintenance System, (YMS) Yard Management System

TABLE 5.1: Experience of interviewees

process. That is, we used a coding scheme to categorize decisions, decision alternatives, and levels/stages of decision making to map the interviewees' answers to concepts such as patterns in our pattern language, relationships between patterns, and correlations between application-generic and application-specific ADDs. For example, an interviewee's statement that *"the interface can be accessed remotely without any changes"* implies the pattern REMOTE PROXY as a code.

Having analyzed each interview's data, the pattern language and the decision model were reviewed by applying a process of *data saturation* [GS67]. In particular, new patterns and new connections between the patterns were added to the pattern language. Decision categories and levels/stages of decision making were documented. Apart from that, patterns and pattern connections that were considered either irrelevant or superfluous were selected as candidates for removal during subsequent iterations.

5.6 Derivation of Reusable Architectural Decision Model

As mentioned before, a pattern language was distilled from the extracted set of 31 patterns (plus 9 referenced patterns) divided in four categories (*Adaptation and Integration* patterns, *Interface Design*, *Communication Style*, and *Communication Flow*). The pattern language documents relations between the patterns, within and between the four categories. In Figure 5.2, the *Integration and Adaptation* patterns and their relationships are depicted. The patterns are related in four ways: First, a pattern can represent a *variant* of a more generic pattern description (e.g., REMOTE PROXY as a variant of PROXY). Second, patterns can play the role of alternative (exclusive-or) or complementary (inclusive-or) design practices when applied at the same design level (e.g., for integration and adaptation, or denoting communication styles). For instance, in Figure 5.2, the REMOTE PROXY and the INTEGRATION ADAPTER pattern are alternatives to each other; each one realizes an integration platform with different capabilities (i.e., direct proxy vs interface adaptation). Third, adopting one particular pattern (i.e., INVOCATION ADAPTER) *leads to* evaluating

another pattern at the same level of abstraction under certain conditions or requirements (e.g., the COMPONENT CONFIGURATOR if runtime adaptability is needed). Finally, some patterns (e.g., PROTOCOL PLUGIN) are used as part of the solution of a higher-level pattern (e.g., REMOTE PROXY) to realize a certain aspect of the solution (i.e., multi-protocol support).

In the drafting phase of our architectural decision model, we considered such identified pattern relations as indicators of architectural decision points to be documented for a single concrete (or even multiple) integration platform architectures. For capturing such recurring, pattern-based architectural design decisions, we adopted the following description scheme:

- *Decision context*: Arriving at a decision point is motivated by previous decisions. Also, the same decision point may be reached several times while constructing an architecture; yet, depending on the given context, different decision options and drivers become relevant. For instance, while Decision 1 (D1) in Figure 5.2 for a given architecture refers to the REMOTE PROXY pattern in the context of the overall integration platform design, the REMOTE PROXY is also relevant in the context of designing the communication flow (e.g., to bridge to a backend platform service).
- *Decision point*: The point of decision making documents the essence of the decision problem. Depending on the decision context (e.g., proxying with or without adaptation), the decision description can refer to the problem and solution statements of decision-related patterns (e.g., REMOTE PROXY and INTEGRATION ADAPTER).
- *Options*: In our pattern-based decision model, the range of adoptable patterns represent the options space at a given decision point in a decision context. For Decision 1 in Figure 5.2, applying either the REMOTE PROXY or the INTEGRATION ADAPTER are alternative options.
- *Decision drivers*: The actual drivers for selecting one of the available options can partly be mined from the forces and consequences documented for the decision-related patterns.

Figure 5.2 illustrates how a draft of the reusable pattern-based decision model was derived from the pattern language. This was afterwards used as basis for the final version of the reusable decision model edited with the aid of the ADvISE tool. The various decision points and alternative options included in the derived reusable decision model are documented in detail in Table A.3.

5.7 Interviews and Improvement Iterations

The results from performing the interviews with the platform experts were manifold. The importance of the key patterns of our pattern language has been confirmed from the point of view of the independent experts. For example, either PROXIES or ADAPTERS are needed in all cases for invoking

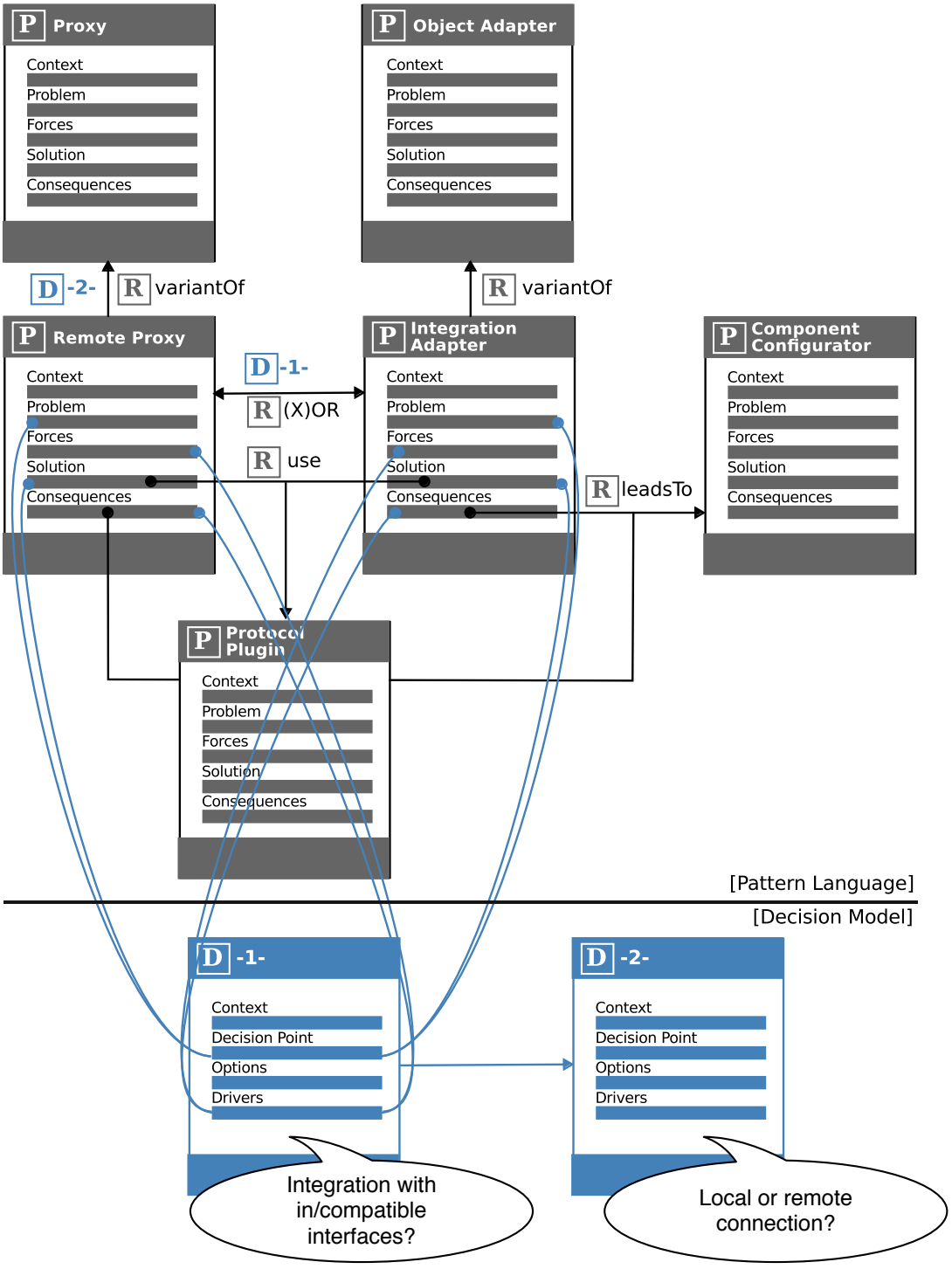


FIGURE 5.2: Example excerpt: pattern language and two derived decision model fragments

the platform services. Some patterns, like SERVICE ABSTRACTION LAYER and EXTENSION INTERFACE were regarded as being useful, but not critical for integrating the platforms at hand. The industry experts agreed that those patterns are rather needed in the field of enterprise applications. In addition, we found that patterns initially omitted in the first version of the pattern language (e.g., PUBLISH-SUBSCRIBER) were used very often by the platform experts. These patterns were

added to the pattern language for the next iterations. Apart from that, new connections between the patterns were introduced. Along with the improvements to our pattern language we updated our decision model accordingly.

These interviews were also used as a preparation for the design of the case study. During this process we gathered existing platform services that were combined in various integration scenarios. We noticed that during the interviews the interviewees could not avoid getting into technical details that mainly focused on the technologies used by the platforms and for their integration with other platforms. These technical details introduce constraints that exclude patterns from the pattern language and narrow down the design space. Hence, another important finding of our interviews was that, in the domain of platform integration, decision making has to be performed at multiple application-generic and application-specific levels, and at multiple stages (see Section 5.9).

5.8 Case Study

5.8.1 Objective

Our integration case study was both confirmatory and exploratory in nature. Our task was to discuss and to design architectural views together with the platform experts the integration architecture based on various integration scenarios. First of all, we studied to which extent the pattern language corresponds to the platform integration domain and the architectural decision model helps navigate the design space. We evaluated the applicability and the appropriateness of the pattern language and the reusable decision model in the context of the case study using these two assets as our main guidance for decision making. Apart from that, we analyzed the case study design to gain insights into the decision-making process. The design of a common case study on integrating the three platforms allowed us to define and to explore new decision levels, correlations between application-generic and application-specific design decisions, and different stages of the decision-making process.

5.8.2 INDENICA Case Study

The case study we present in this section deals with the integration of three heterogeneous platforms in the industry automation area.

In the INDENICA case study, a **Warehouse Management System (WMS)**, a **Yard Management System (YMS)** and a **Remote Maintenance System (RMS)** are integrated to allow an operator application to utilize the services provided by these platforms. The YMS manages the scheduling and coordination of trucks in a yard, as well as the loading and unloading of the

goods from these trucks. The WMS handles the storage of the goods (or storage bins) into racks via conveyor systems. The RMS system is connected to the warehouse to monitor every incident occurring in the warehouse and the yard. It also supports remote communication of operators and workers in the warehouse and the yard. An operator application uses the services of the three platforms via a domain-specific **Virtual Service Platform (VSP)** which is responsible for performing service-based platform integration. The overall high-level architecture is schematically illustrated in Figure 5.3. The VSP must handle various integration aspects including interface adaptation between the platforms, integration of service-based and non-service-based solutions, routing, enriching, aggregation, splitting, etc. of messages and events, handling synchronization and concurrency issues, and so on.

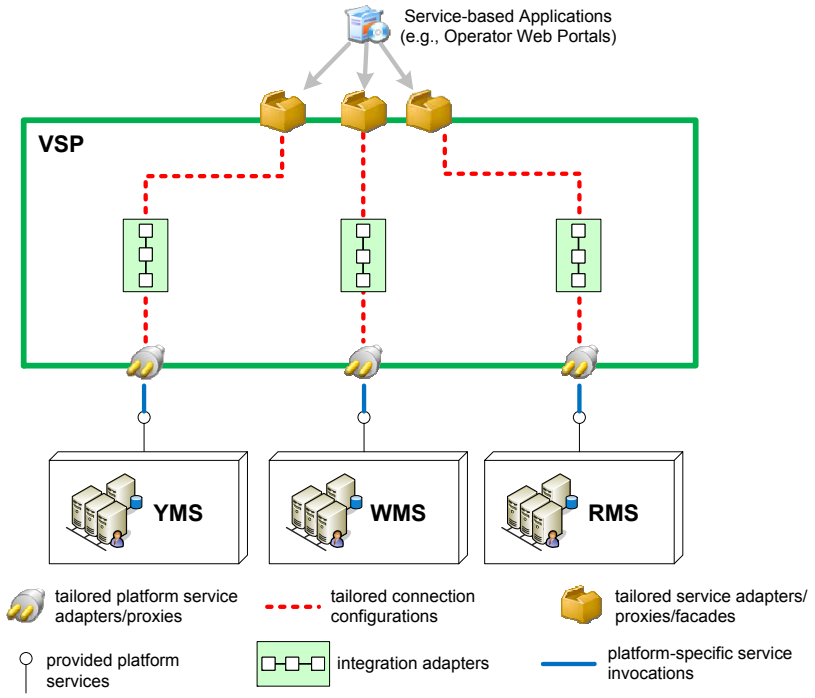


FIGURE 5.3: INDENICA case study

One of the scenarios considered during the design of our case study is presented in Figure 5.4 as a sequence diagram. This scenario addresses unloading storage bins onto the racks in the warehouse. When a loaded truck arrives at the yard, the operator gets notified (*truckArrived*) and requests a free dock (*getFreeDock*) for the truck. After receiving a free dock (a *dockID*), the operator communicates with the personal (*initiateVoiceCall* and *endCall*) and coordinates the redirection of the truck to the assigned dock (*moveTruckToDock*) for its unloading. When the operator gets notified that the truck is ready for unloading (*truckReady*), she sends a command to the personal to start unloading (*startUnloading*). For each product a storage unit is registered (*registerStorageUnit*) and a suitable bin location is reserved (*searchAndReserveSuitableBinLocation*). The product is stored in the reserved bin location (*transportStorageToReservedBinLocation*). Upon finishing the

unloading, a notification (*unloadingFinished*) is sent to the operator. Finally, the operator gets notified when the truck leaves the dock (*truckLeft*).

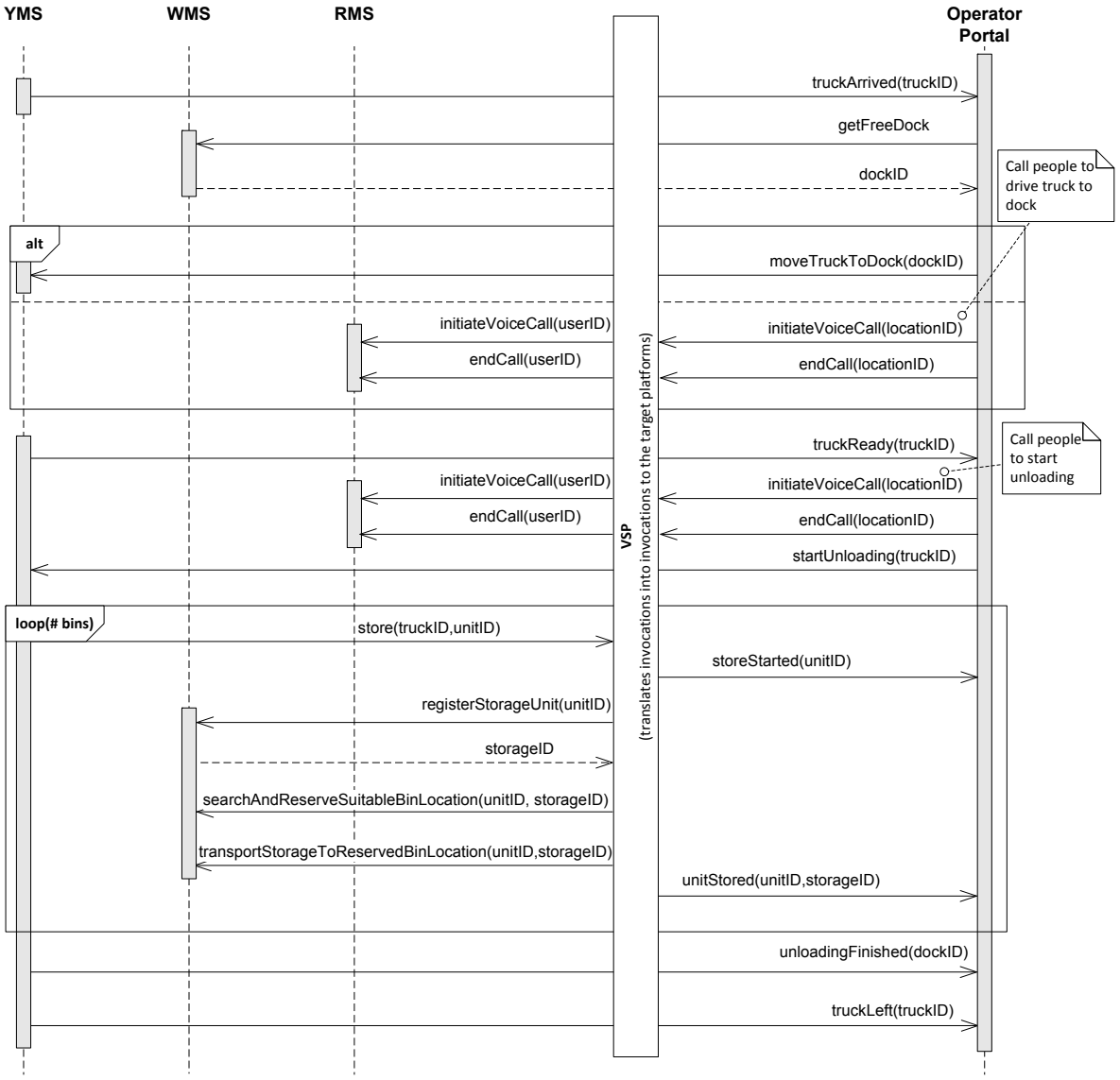


FIGURE 5.4: Integration scenario in the warehouse

During the design of the case study, the general ADDs were refined during multiple stages reflecting the single platform technologies, the integration solutions, and the operator application requirements. The ADDs were initially documented as instances of the reusable architectural decision model (and later using the ADvISE tool). The outcome of this process was an initial design of the integration solution, for which we used different architectural views to describe the adaptation and the communication levels. The names and the annotations of the design elements signal the use of a specific design pattern in many cases. Table 5.2 summarizes the reusable ADDs that were made in the course of this case study and how often they were made. These ADDs reflect of course parts of the integration solution that was designed in the context of the research project INDENICA.

Reusable ADD	Num. of ADDs	Reusable ADD	Num. of ADDs
Proxy	3	Gateway	2
Adapter	1	Publish-Subscriber	7
Result Callback	7	Data Mapper	2
Facade	1	Content Enricher	1

TABLE 5.2: Documented reusable ADDs

In Figure 5.5, we present an excerpt of the high-level integration architecture containing the three backend platforms (i.e., YMS, WMS, and RMS), the Virtual Service Platform (VSP), and the operator application. It also contains an Enterprise Resource Planning (ERP) system, which is not actual part of the scenarios but is necessary for modeling a real-world process in the warehouse (e.g., ERP provides data about trailer's goods).

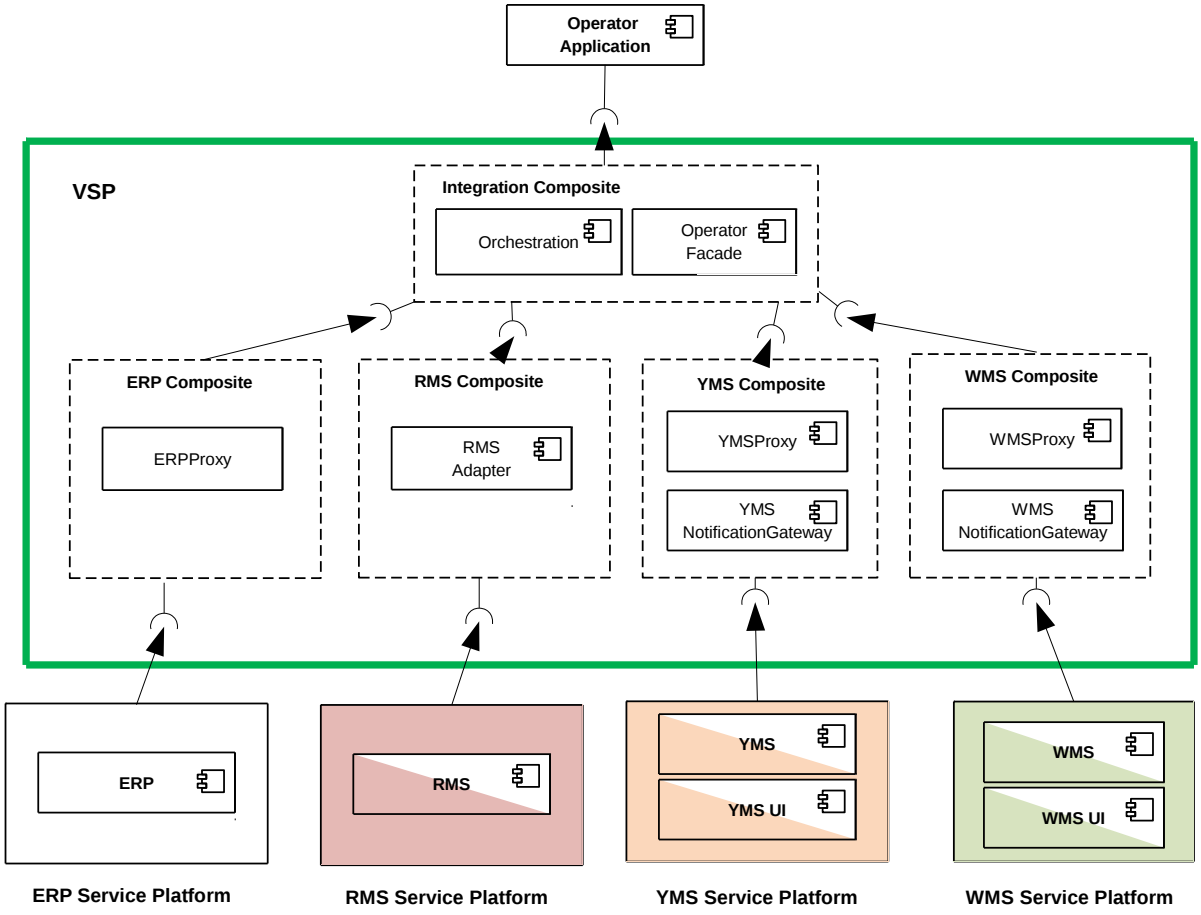


FIGURE 5.5: Excerpt from the high-level integration architecture

The major advantage of introducing the VSP in the integration scenario is that it seals the application built on top (i.e., the *OperatorApplication*) from the complexity of the underlying technologies of various service platforms and provide unified development interfaces via the *OperatorFacade*. In case the access to the remote services does not require any interface adaptations, a PROXY or

GATEWAY component is used (e.g., *WMSProxy*, *WMSNotificationGateway*, etc.). Otherwise, an ADAPTER component is deployed to resolve interface incompatibilities (e.g., *RMSAdapter*).

Other recurring ADDs documented include the use of PUBLISH-SUBSCRIBERS (e.g., to receive the notifications from the VSP – the operator is expected to subscribe to the appropriate notification channel) and the use of RESULT CALLBACKS (e.g., for communicating with the WMS to create orders).

5.9 Decision-making Levels and Stages

Approaches for modeling architectural decision making usually focus only on one or two levels of decision making and knowledge acquisition (i.e., the application-generic and application-specific levels [HA09]). From analyzing our interviews, we found that in platform-based software development multiple *levels* of decisions and multiple decision times (i.e., *stages*) must be considered. Each decision level is commonly performed by different stakeholders or stakeholder groups (e.g., platform supplier, system integrator), often working in different organizations. At each stage, the decision space derives from the decisions taken at prior stages. In addition, at each stage the results of decision making at different levels must be taken into account. That is, starting from generic knowledge about a specific form of software platform integration (*Level-1*), architectural and technical knowledge about each of the platforms integrated (*Level-2*), about each of the integration technologies and solutions used (*Level-3*), and about the applications developed based on top of the platforms (*Level-4*) must be considered.

Inspired by the guidelines on multi-stage, multi-level transformations of feature models by [Cza+05], we documented the observed decision-making process. In contrast to Czarnecki et al.'s approach, we support decision making on the architectural level rather than on the concrete software characteristics. Figure 5.6 depicts an abbreviated example of ADDs related to the communication style in our case study. Here, an architectural decision is illustrated using a transformation between two feature models. In our case study and in the example in Figure 5.6, the four levels represent decisions to be taken by different stakeholder roles (i.e., integration architect, platform supplier, system integrator, and application engineer). The reusable, pattern-based ADDs form the basis of *Level-1* decisions. The original decision space for each role is modeled as an initial feature model at *Stage-0*. Each subsequent stage sets a discrete decision time in which decisions at different levels of abstraction are addressed at *Level-1* by the integration architect(s) through reviews of historical decisions taken at the other levels. Levels of abstraction are represented by, e.g., appending increasingly specific branches to the decision outcomes (here: the feature trees) for each level.

The flow of decisions in Figure 5.6 illustrates the process of narrowing down the communication style to be adopted by the service-based integration platform. For instance, at *Stage-1* (in *Level-2*)

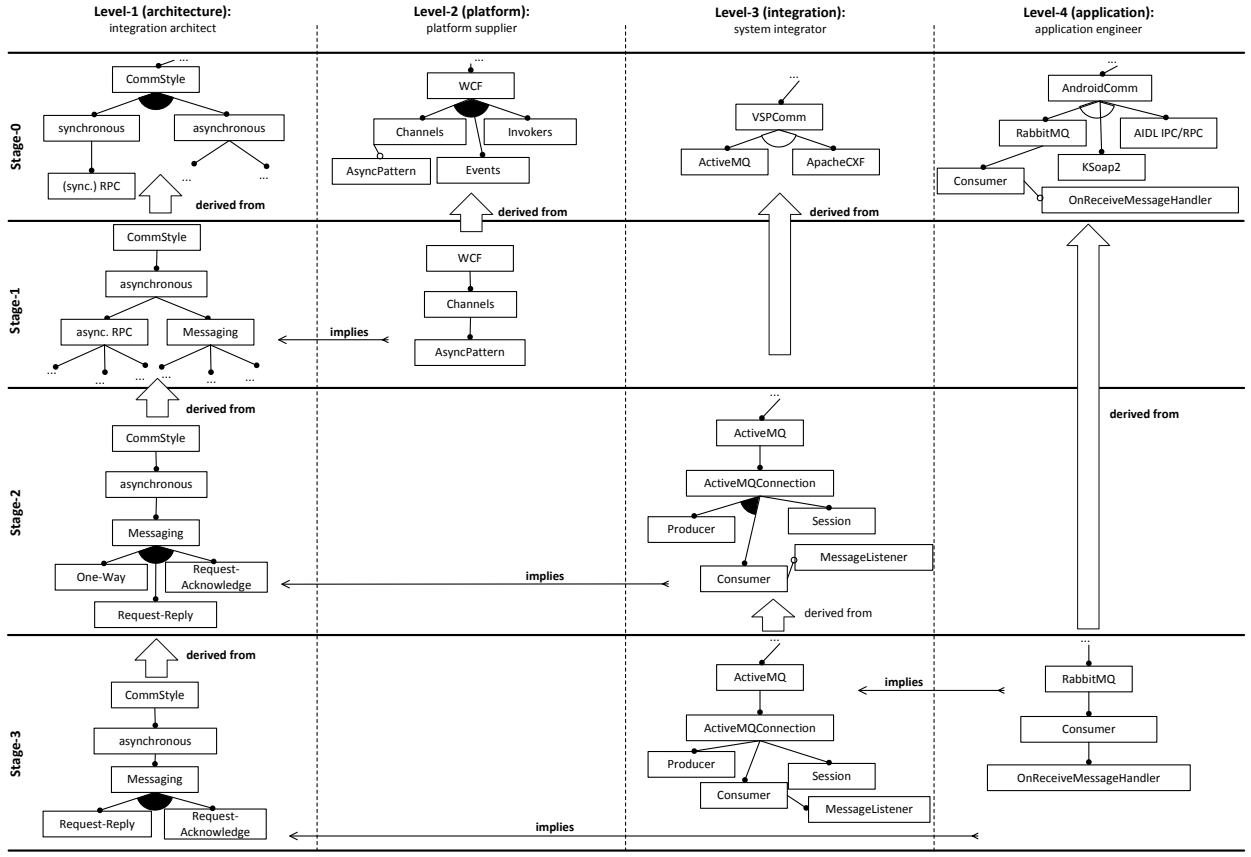


FIGURE 5.6: Exemplary levels and stages of decision making in service-based platform integration

the architects of the WMS platform supplier have opted for the *Windows Communication Foundation (WCF)* technology for building asynchronous services (i.e., *AsyncPattern*). As a result, the integration architects at *Level-1* can refute all the synchronous communication alternatives from the initial architectural decision model. Next, at *Stage-2*, the decisions about the integration middleware (*Level-3*) were considered. Given the asynchronous communication style adopted by the integrated platforms, the platform integrator chose a message-oriented middleware (*ActiveMQ*) to drive the integration platform. For the *Level-1* decision space, this meant to refute all the communication style patterns except those related to MESSAGING. Reviewing the decisions taken for the operator application (*Level-4*) at the final stage (*Stage-3*), it turned out that the application engineers had decided for the *RabbitMQ* message-oriented middlware and that the application had been built around required notifications about the status of its requests (i.e., using *OnReceiveMessageHandler*). Reviewing this decision step meant to specialize the decisions available to the integration architects for the operator application connection even further. That is, any one-way MESSAGING can be excluded for subsequent decision steps. Also, this application-level decision demanded a particular configuration of the message connection in the VSP (*Level-3*), i.e., message producers and message consumers, to deliver the notifications to the client application.

We have observed this decision scheme in our case study. The dimensions (levels and stages) result from the nature of distributed decision making (e.g., given that the developers of platforms,

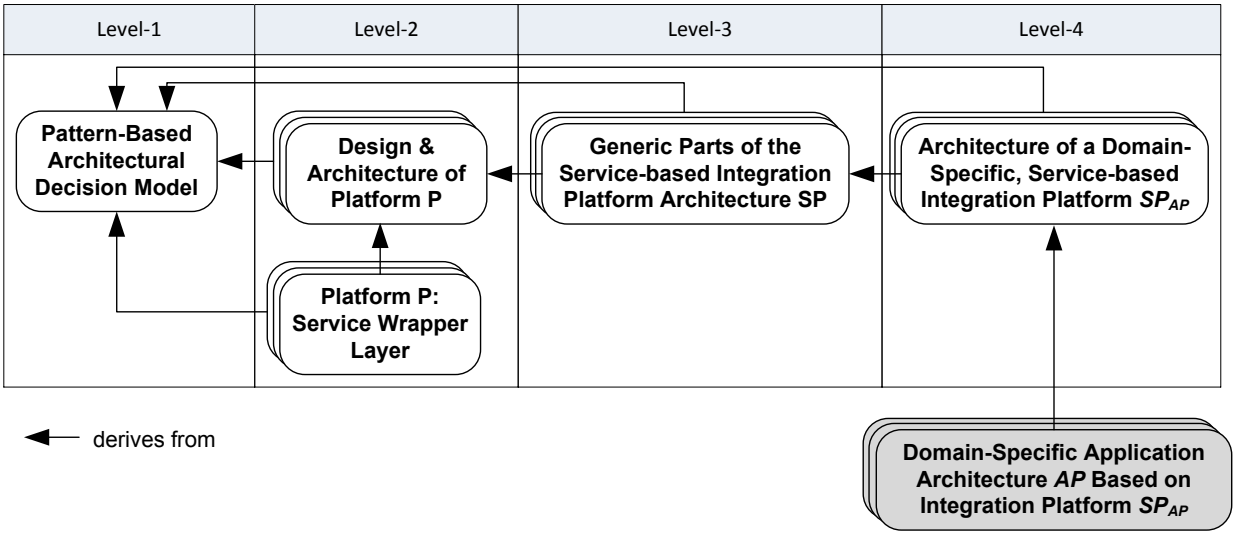


FIGURE 5.7: Artifacts and their AK derivation relationships in different levels of decision making

applications, and integration solutions are often not working in the same organization). However, the decisions taken influence the design space left open for later decisions and each additional stage excludes or refines decisions that must be made in the subsequent steps. Hence, we propose to extend the existing architectural decision making concepts with support for multiple levels and for multiple stages of decision making along those lines. Our general model of the artifacts at the different decision-making levels and their AK derivation relationships are shown in Figure 5.7.

The evidence presented in this section that the notion of decision levels and decision stages applies to architectural decision making in general. As for platform-driven software development, the four decision levels (e.g., architecture, platform, integration, application) could be characteristic and are likely to apply to other, platform-like software development approaches (e.g., software product lines). Our finding about multiple levels and multiple stages in architectural decision making also relates to the way architectural decision-making techniques are designed and how these techniques are classified [Fal+11]. For example, our finding converges with the decision-making technique by Zimmermann et al. [Zim+09], also indicating the need for integrating leveled decision making.

5.10 Limitations and Threats

Systematic Literature Review In conducting the systematic review of pattern literature, we deviated from the original guidelines [Kit+09] in significant ways. Most importantly, we conducted a purely manual search over a confined selection of conference proceedings, journals, and book series rather than a semi-automated search process supported by (scientific) search engines. While this practice is consistent with closely related work on pattern-based architecture decision-modeling [Zim+08; HA09], there is the risk that we have missed relevant pattern material. It simply might be the case that relevant AK about service-based platform integration has not yet

been documented in pattern form. We consider this threat a minor one because we have found pattern material covering all aspects of the considered cases and we have not found any evidence in our interviews that relevant AK is missing. As for the manual search, we explicitly excluded pattern materials which had not been published in any of the pattern community venues.

To include only pattern materials of accepted quality, we relied on the established quality assurance procedures in the pattern community (e.g., shepherding, writer's workshops). With this, threats to validity with regard to the quality of the included material are minimized: mature pattern material that has been reviewed multiple times by pattern and domain experts is likely to contain relevant AK. This conjecture was confirmed by the follow-up interviews. However, our results are certainly limited with respect to the selected pattern pool of documented design knowledge. A systematic literature review risks missing relevant primary material during the initial search step because of the high number of candidates, the extensiveness of the material (e.g., content complexity), and the inappropriateness of content proxies. Most importantly, incomplete abstracts are weak substitutions for the full document. Although our initial search yielded a considerable amount of publications (including extensive pattern books) referring to 402 individual patterns, the established practice of presenting patterns using structured descriptions (e.g., pattern templates, pattern thumbnails, structure diagrams for pattern language) provided important content proxies to us, beyond the abstracts. Similar to structured abstracts [Bud+11], the structured descriptions reduced the likeliness of missing relevant material, by mitigating misjudgments when interpreting titles, abstracts, and conclusions. The content proxies also facilitated the quality assessment.

To reduce the overall bias of personal judgment (e.g., due to individual research interests, experiences, time constraints), the search step, the quality-assessment step, and the extraction step were performed by each of the reviewers independently from each other. The results were then synthesized in joint re-iterations. In addition, in the overall sequential research design, we validated our resulting pattern language and decision model through interviews with nine independent industry experts. Despite these efforts, an author's bias can not be completely excluded.

Interviews To ensure that the conclusions we took away from the interviews are valid, we discuss how we dealt with the threats to external and to internal validity [Rob02; Woh+00]. As with all qualitative studies, there is the threat to validity with regard to the generalizability of results (*external validity*) that the small size of considered cases is not enough to generalize the results to other cases of service-based platform integration. We have tried to limit this threat by focusing on patterns as sources of AK. In our point of view, it is more likely that similar established best practices are used for other cases of service-based platform integration than if we would have analyzed novel, innovative approaches. Apart from that, our interviewees come from three broad domains (warehouse management, yard management, telecommunications) which of course does

not permit us to do any statistical generalization but it offers a broad span of domains in our sample.

As with all qualitative designs, there is the threat to validity that the interviewers themselves have been an instrument in the study (*internal validity*) and might have accidentally influenced the study's outcomes or made misinterpretations because of, e.g., limited knowledge, wrong assumptions, and personal bias. We tried to avoid this threat by keeping an open mind, by encouraging the interview participants to talk and by observing rather than steering the discussion. Also, we included both open-ended and closed-ended questions to get the widest range of feedback. However, it can not be fully excluded that we influenced the results by our participation in the studies or by our interpretation. Regarding the threats to construct and interpretation validity, we employed *observer triangulation* [Rob02] so that three different researchers were involved as observers and interviewers.

Generalizability The limitations of the literature search in our systematic review setup imply that our findings, namely the pattern language and the derived reusable decision model, can not be reused for similar architecting activities in related SOA settings with different emphasis. With the focus on the integration and adaptation view, our review questions and the subsequent pattern search did not cover related SOA concerns such as monitoring or middleware framework design. Pattern material and patterns, even if identified partly, have not been incorporated. Due to the literature filtering and quality assessment, possible connection points to such concerns might have been missed.

The different levels of knowledge among the researchers involved poses another threat. As for the systematic review and pattern extraction, the reviewer group consisted of a pattern expert and senior researcher, two experienced postdoc researchers and a doctoral student focusing on pattern-related research. This heterogeneous distribution of pattern knowledge among the researchers certainly affected the preparation of the review protocol, the quality assessment of the pattern material, and the extraction of pattern data for drafting the pattern language. By continuously switching the roles of extractor and checker [Bre+07], and multiple iterations over the selected pattern material, this effect might have been substantially lowered, yet not neutralized. The expert opinions collected from the interviews are directly dependent on the nine experts' experience with the platforms, service-oriented architecting, middleware technologies and software patterns. However, we benefited from a relatively mature expert pool with seven out of nine interviewees having more than three years of industrial experience.

While the aforementioned limitations and risks threaten the generalizability of our results, our methodological approach and the sequential multi-method research design [Fal+10] can be applied to comparable research activities on pattern-based architecting. We also consider this systematic

approach to be a guidance for modeling reusable ADDs in the form of reusable architectural decision models.

5.11 Lessons Learned

Empirical methods in software engineering are marked by a high level of investigation costs [Fal+10]. The time effort required for preparing and conducting the systematic review was considerable, caused by face-to-face coordination meetings, numerous phone conferences, etc. over a period of two months. Identifying and approaching experienced subjects for the interview and case study steps was facilitated by our involvement in a large-scale European research project with major industry partners. In our study, we have learned in practice that using software patterns facilitates iterative architectural decision making [Har+07].

The actual forces and consequences in pattern descriptions document abstracted choices at a given decision point. While our pattern language does not document ADDs for a concrete integration platform architecture, it identifies observed designs for a family of these architectures. In the case study, the pattern language was reused by the architects for sketching a concrete architecture by referencing pattern descriptions from within ADD descriptions. In addition, the pattern form complemented the qualitative research methods used in our study. The structured presentation of patterns and pattern collections facilitated the systematic review. Patterns proved to be an important communication vehicle between the interviewers and the interviewees to bridge the gap between their different technology backgrounds.

At the same time, when preparing our study, we became aware of documented limitations of software patterns as empirical research objects. Especially, forcing subjects in experiments into using and applying architectural styles and design patterns resulted in observations indicating an increased level of development and maintenance effort. The complexity of learning and comprehending pattern material affected the observations. Such effects have more systematically been reported for research designs involving patterns and artificial software artifacts of lower complexity (toy applications). While the earlier studies adopted experimental designs, we learned two lessons from these. First, our research design should not impose design decisions (in terms of patterns) onto our subjects [Fal+10]. Second, the architecting process must be observed in the context of a real, not artificially constructed development project. The first risk was mitigated by not confronting the experts directly with the selected patterns, but rather with design decisions derived from our pattern language. With this, patterns played only an indirect role for the observed architecting process. As for the second risk, the architecture designed in our case study applies to a large-scale software prototype in the context of a European research project; as such, the architecture is not specific to or was not created for the purpose of our study alone.

5.12 Conclusions

To summarize, in this chapter, we conducted an empirical research, in which we employed multiple qualitative methods to explore the practice of architectural decision making in service-based platform integration using reusable ADDs. First, we performed a systematic literature review from which we distilled a pattern language and a pattern-based architectural decision model. In the next stage, we performed interviews with platform experts and conducted a case study. The results of these research steps were a refined pattern language and a reusable architectural decision model for service-based platform integration. In addition, we proposed a model comprising four levels of decision making for platform integration, that was identified in our case study. At a later stage of the aforementioned research project, the derived reusable decision model was modeled using the ADvISE tool and was used by the industrial partners to model ADDs employed in the case study.

In the next chapter, we will present a further empirical study on the use of reusable decision models created with the ADvISE/CoCoADvISE meta-model and leveraged using the Web-based CoCoADvISE tooling.

6 Empirical Evaluation of Reusable Architectural Decision Models

6.1 Introduction

As we saw in Section 2.2, few reusable decision models and related tools that support their management (such as [Abr+06]) have been evaluated in real-life contexts. For instance, Zimmermann et al.'s decision model consisting of 300 ADDs from the SOA domain which covers various aspects, such as Web service integration and transaction management, has been used by practitioner communities and in industrial projects [Zim+07]. However, no feedback or empirical evidence has been gathered on whether and to which extent reusable decision models are beneficial (i.e., they support effectiveness and efficiency of software architects) in the architectural decision making process. While a few studies have investigated how reusable AK management and sharing is practiced in industrial contexts [GB14; Hoo+11] and have validated the supportive effect of pattern-based reusable AK in the decision making process [Hee+12b], the software architecture community lacks empirical evidence on the positive impact of reusable AK on decision making and documentation. Such empirically-grounded findings are important not only for validating the benefits of reusable AK in practice, but also for understanding, improving, and supporting the management and leveraging of reusable ADDs.

Therefore, we conducted two controlled experiments with software architecture students – potentially novice software architects – to test whether the use of reusable decision models increases the efficiency and effectiveness of architects in the decision making and documentation process. We explicitly considered novice architects in our evaluation, as reusable decision models are supposed to be used as guidance models by trainers for systematically teaching patterns and technology best practices to new or inexperienced members in a software development team [Zim11]. In the two controlled experiments, 49 and 122 students, respectively, with background in software architecture and design patterns, were asked to make and document ADDs while designing the architecture of two different software systems. The Web-based tool CoCoADvISE was provided to the experiment and control groups. Both the experiment and control group received material with related patterns

and technology documentations and could use the tool to make and document decisions individually. Contrary to the control group, the tool provided additional semi-automated decision making guidance based on reusable decision models for the participants of the experiment group.

In the following sections, we describe our experimental settings, as well as the analysis of the results for the two controlled experiments we conducted. We also report on the findings, implications, and validity threats of our empirical study.

6.2 Two Controlled Experiments

To measure the effect of using reusable architectural decision models on the efficiency and effectiveness of software architects, we have conducted two separate controlled experiments with software architecture students. For the design and execution of the two experiments, we followed the guidelines of Kitchenham et al. [Kit+02] while for the analysis, evaluation, and presentation of results we have consulted the advice of Wohlin et al. [Woh+00]. In the following subsections, we discuss the common goals and hypotheses of the controlled experiments and present their design and execution in detail.

6.2.1 Goals and Hypotheses

The goal of the experiments is to study and quantify the benefits of using CoCoADvISE and in consequence reusable decision models for making and documenting ADDs in terms of quantity and quality related effectiveness, as well as time efficiency. For this, we compare two participant groups, one using reusable decision models and one using an ad-hoc process based on pattern documentations to make and document ADDs in template form.

We postulate the following *three* null hypotheses and corresponding alternative hypotheses: making and documenting ADDs using reusable decision models...

H_{01} leads to lower or equal *quantity related effectiveness* of software architecture students compared to an ad-hoc process for architectural decision making.

H_1 leads to increased *quantity related effectiveness* of software architecture students compared to an ad-hoc process for architectural decision making.

H_{02} leads to lower or equal *quality related effectiveness* of software architecture students compared to an ad-hoc process for architectural decision making.

H_2 leads to increased *quality related effectiveness* of software architecture students compared to an ad-hoc process for architectural decision making.

H_{03} leads to lower or equal *time related efficiency* of software architecture students compared to an ad-hoc process for architectural decision making.

H_3 leads to increased *time related efficiency* of software architecture students compared to an ad-hoc process for architectural decision making.

We perform statistical tests to find out whether the corresponding null hypotheses can be rejected. We expect that the users of CoCoADvISE will deliver ADDs of better quality and will manage to document more ADDs and in less time.

6.2.2 Parameters and Values

Several variables have been observed during the two experiments. We present here all dependent, independent, and derived variables we used for testing our hypotheses. A detailed list of variables (type, description, scale type, unit, and range) is provided in Table 6.1.

Type	Name	Description	Scale Type	Unit	Range
Dependent	*time*	Overall time needed to make and document decisions	Ratio	Minutes	Natural numbers including 0
	numOfDecisions	Number of documented decisions	Ratio	–	Natural numbers including 0
	quality	Quality of documented decision	Interval	–	1 (lowest) to 10 (highest)
Derived	*timeNorm* $(= \frac{time}{numOfDecisions})$	Time needed per decision	Ratio	Minutes	Natural numbers including 0
	qualityPerStudent $(= \frac{\sum_{i=1}^{numOfDecisions} quality}{numOfDecisions})$	Average quality of decisions	Interval	–	1 (lowest) to 10 (highest)
Independent	*group*	Treatment group	Nominal	–	Either "experiment" or "control"
	exp	Programming experience	Ordinal	Years	4 classes: 0-1, 1-3, 3-6, >6
	commExp	Commercial programming experience in industry	Ordinal	Years	4 classes: 0-1, 1-3, 3-6, >6

TABLE 6.1: Observed and derived variables

Dependent Variables All dependent variables have been captured by and extracted automatically from CoCoADvISE's database. In particular, we instrumented its source code in such a way that we could precisely record all user activities within the system. The variable *time* indicates a single user's total time spent logged in in the application. The number of decisions that were

documented by each student is indicated with the variable $numOfDecisions$. Finally, the variable *quality* refers to the quality of each ADD that was evaluated in a scale from 1 to 10 by two independent experts.

Derived Variables To allow for a meaningful comparison of the *time* spent by the students to document the decisions we decided to introduce the variable $timeNorm$ which expresses the time needed per decision. This way, we exclude the possibility that one treatment group needed less time to perform the tasks because they just delivered less work. In addition, we calculate the average points per student for the total of documented decisions ($qualityPerStudent$).

Independent Variables The independent variables *group*, *exp* and *commExp* can potentially influence the dependent variables. In particular, *group* indicates the participant's treatment group (i.e., control or experiment), and *exp* and *commExp* refer to their programming experience in general and in the industry, respectively.

6.2.3 Experiment Design

The controlled experiments were conducted in the context of Information System Technologies and Software Architecture lectures respectively at the Faculty of Computer Science, University of Vienna, Austria. The first controlled experiment took place in Winter Semester 2013/2014 (January 2014) and the second in Summer Semester 2014 (June 2014). All participants were at the final semesters of their bachelor studies and had background in software architecture and design patterns. The readers can refer to Table 6.2 for a summary of the experimental design in the two cases. We will refer to the first and second controlled experiment as *ORExp* (Online Retailer Experiment) and *UniISExp* (University Information System Experiment) respectively from the corresponding case studies that were used in each case.

	ORExp	**UniISExp**
Location	University of Vienna, Austria	University of Vienna, Austria
Time	January 2014	June 2014
Software System	Online Retailer	University Information System
# Participants	49 (27 control / 22 exp.)	122 (56 control / 66 exp.)
# ADDs	319	762
# Decision Models	5 (16 patterns)	5 (40 patterns)

TABLE 6.2: Experimental design data

Participants In the first controlled experiment conducted in January 2014, from the 49 students of the lecture, 27 participated in the control group and 22 in the experiment group. In the second

experiment in June 2014, a bigger sample of 122 students (56 in the control group and 66 in the experiment group) participated.

The experiments have been conducted in the course of mandatory practical exercises on architectural decisions for service-based software systems and distributed software systems respectively. The experiments took place in separate exercise groups (4 in the first and 6 in the second experiment) and in different computer labs at different times, which also explains the unequal number of participants in the corresponding control and experiment groups. The students in each group were assigned to the treatment groups randomly. In summary, 319 ADDs were documented in ORExp and 762 in UniISExp. All participants had experience with Java programming and design patterns and were familiar with the concepts of software architecture and architectural design decisions, as these topics had been already discussed in the corresponding lectures or were prerequisites to accomplish the previous exercises of the corresponding subjects.

Objects A list of documented architectural design patterns and technology-related solutions was integrated in the CoCoADvISE tool for both groups in wiki format. The design patterns and architectural decision models were selected based on the lecture materials known to the students as well as the students' experiences from the previous programming exercises. The experiment group was provided additionally with a set of reusable architectural decision models and instructions how to use them for getting decision support with the aid of the CoCoADvISE tool.

Instrumentation In the preparation phase, all students were asked to study the catalog of architectural design patterns and related technologies (afterwards integrated in the CoCoADvISE tool). Before starting with the experiment tasks, all participants had to fill in a short questionnaire regarding their programming experience. Afterwards, the participants of both the control and experiment groups were provided with a description and requirements of the system to be designed. That was "An Online Retailer for Selling Books and Gadgets" and "A University Information System (IS)" for the ORExp and UniISExp experiments respectively. In Table 6.3 and Table 6.4 we give examples of the aforementioned software systems' requirements.

Name	Description
Take Orders	Customers can place orders via two different channels: Web site and call center. Each of these systems is based on a different technology and stores incoming orders in a different data format. The call center system is a packaged application, while the Web site is a custom J2EE application. We want to treat all orders equally, regardless of their source. For example, a customer should be able to place an order via the call center and check the order status on the Web site. *Decide* how to deal with the different channels and feed the retailer with unified messages.

TABLE 6.3: Excerpt from the "Online retailer for selling books and gadgets" requirements

The students had to consider six sets of requirements in ORExp (*Take Orders*, *Process Orders*, *Logging*, *Track Orders*, *Announcements*, and *Customer Login*) and also six sets of requirements in

Name	Description
Research Network	The University is collaborating with a central research system for establishing networks between researchers, dissemination of research results, access to publications, etc. The University IS should be able to send updates of new publications and research projects of its research staff. In this case, reliability and performance are not very important. The Research Network uses only the XML-RPC protocol for communication with external systems. *Decide* how you will design the communication between the University IS and the Research Network.

TABLE 6.4: Excerpt from the "University Information System (IS)" requirements

UniISExp (*General Requirements*, *Student Services*, *Services for Research and Administrative Staff*, *Library Service*, *Stream Service*, and *Research Network*) – concerning different parts of the software system – given in descriptive form. Additionally, some hints were provided with information about the concrete decisions that were expected (e.g., *"Decide how to deal with the different channels and feed the retailer with unified messages"* from Table 6.3). For each requirement, one or more ADDs were expected to be documented by the students.

Eventually, all participants were given access to the CoCoADvISE tool. The participants of the experiment groups could reuse five decision models which were different for the two controlled experiments. The names and descriptions of the provided decision models are documented in Table 6.5. In contrast, CoCoADvISE was modified for the participants of the control groups, so that the decision model based support was hidden.

Blinding In order to reduce the subjective bias of the participants and the reviewers we have applied double-blinding in both experiments. The participants were not aware of the two different treatment groups and different CoCoADvISE settings, therefore, they were not able to guess the goal of the experiments and whether they belong to an experiment or control group. In addition, the participants' documented ADDs were scrambled and given anonymized to the reviewers for evaluation. It was, thus, not possible to find out whether an ADD comes from the control or experiment group, which ADDs belong to which participants, and which ADDs belong together (i.e., were documented by the same person). Also, the reviewers did not get any other information about the goals and the different settings of the experiments.

6.2.4 Execution

As described in the previous section, the controlled experiments were executed in the context of the Information System Technologies and Software Architecture lectures (Faculty of Computer Science, University of Vienna) with students in the end of their bachelor studies, in Winter Semester 2013/2014 and Summer Semester 2014 respectively.

OrExp	
External Interface Design	Use this reusable architectural decision model in order to decide how the service-based system will expose its interfaces to the external world.
Lifecycle Management	Use this reusable architectural decision model to decide how the lifecycle management of the remote objects (e.g., by creating per-request instances) and the resource consumption of the service-based system will be designed.
Message Routing	Use this reusable architectural decision model in order to make decisions on strategies for routing the messages in the service-based system.
Message Transformations	Use this reusable architectural decision model in order to make decisions on methods for transforming the messages inside the service-based system.
Service-based Invocations	Use this reusable architectural decision model to decide on the type of remote asynchronous invocations that will be used as well as the type of communication channels required for the exchange of messages.
UniISExp	
Application Structure	Use this reusable architectural decision model in order to decide which architectural style(s) you will use to structure your application architecture.
Data Store	Use this reusable architectural decision model in order to decide if and how you will need to store your data.
Distribute Components	Use this reusable architectural decision model in order to decide how you will distribute the different components of your system and how to design the communication between components.
Handling Processing Tasks	Use this reusable architectural decision model in order to decide how you will handle complex processing tasks required in your application.
UI Design	Use this reusable architectural decision model in order to decide how you will design the UI of your application.

Table 6.5: Reusable architectural decision models overview

The practical course groups were randomly assigned to experiment and control groups. That is the reason why we have unequal groups of participants, namely 27 (control group) and 22 (experiment group) participants in OrExp and 56 (control group) and 66 (experiment group) participants in UniISExp. Nine students that participated in both experiments were excluded from the second experiment – although the topic and the tasks required in OrExp and UniISExp were different – in order to reduce the result bias. That led to a reduced sample of 50 and 63 participants for the control and experiment group of UniISExp respectively.

In addition, before the execution of the controlled experiments, we decided to exclude students that did not manage to achieve more than 50% of maximum points at the practical course. However, the final grade of the students at the practical course was not known at the time of execution, therefore, we excluded participants in the end of the semester and before we started with the analysis of the results. No students were excluded in OrExp while we had to exclude one student from the control group and two students from the experiment group in UniISExp. Thus, our sample for UniISExp was further reduced to 49 and 61 participants for the control and experiment group respectively.

Information about the programming experience of the students has been gathered with questionnaires that were given to the students in the beginning of the experiments. As we can see in

Figure 6.1 and Figure 6.2 the programming experience, as well as the industry programming experience of the participants are quite similar for both groups. In OORExp, the control group has slightly more programming experience while in UniISExp we observe the opposite. The majority of the students in both experiments has 1 to 3 years of programming experience while some of the participants are quite experienced in programming (i.e., have more than 3 years of programming experience). However, only very few participants of both experiments have experience in industrial projects[1].

The exact same materials were handed out to all participants in the beginning of the exercise. As mentioned before, the experiment and control groups used different versions of CoCoADvISE, that is, the experiment group could use the reusable ADD models as shown in Figure 4.3 while this feature was deactivated for the control group. The participants had 90 minutes[2] time for reading, understanding, and performing the exercise. They were allowed to finish with the tasks earlier though. Access to the Web-based tool was given only during this session to avoid any offline work or discussion among the students. Apart from that, in order to avoid communication between the students or use of other materials the participants were prevented (i.e., using Web browser policies) from opening Web pages other than that of CoCoADvISE.

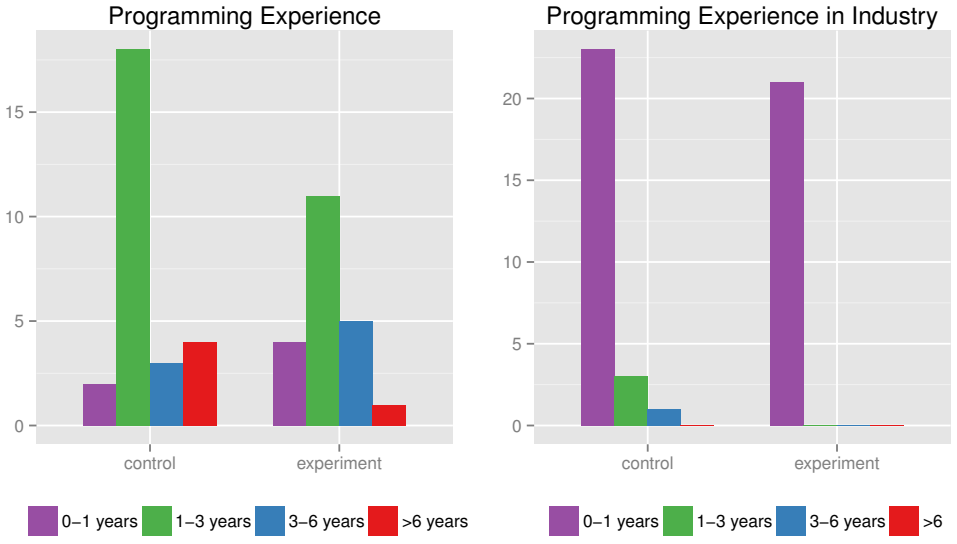


FIGURE 6.1: OORExp: participants' programming experience

The collection of the participants' data was performed automatically during the experiment. In particular, all relevant information, such as created questionnaires, documented decisions, as well as all relevant events, such as deletions or modifications of architectural decisions, were saved in a database. The evaluation of the documented ADDs with the use of a Likert scale from 1 (lowest)

[1]Most of the students with industrial experience have been working from a couple of months to maximum one year at a company.

[2]This is a typical duration for a lab exercise in the two computer science courses. The experiments' tasks were thus designed given the limited time that the students would spend in the controlled environment.

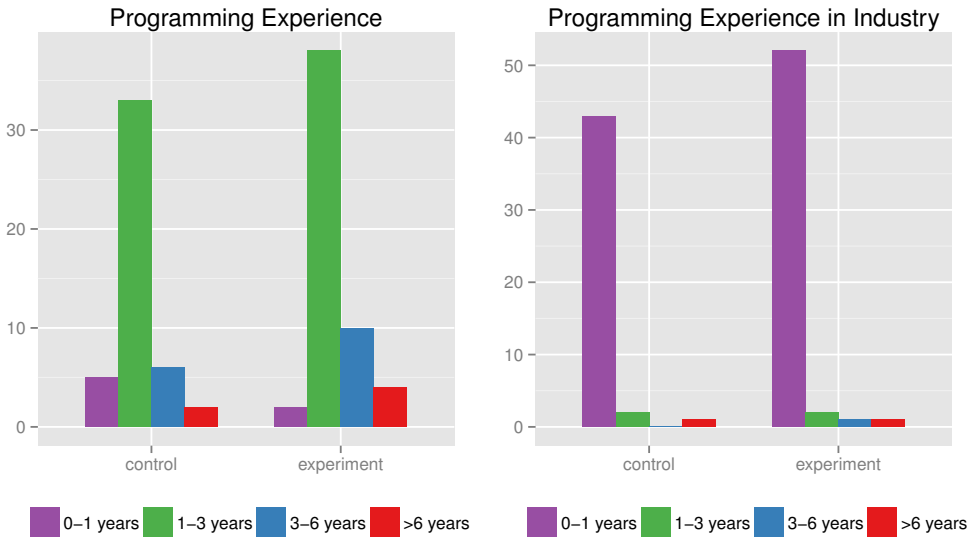


FIGURE 6.2: UniISExp: participants' programming experience

to 10 (highest) was performed afterwards by two independent experts. The experts were instructed to consider the distances between the points of the 1–10 Likert scale to be constant. Hence, we argue that the corresponding variables *quality* and *qualityPerStudent* qualify as interval scale type. In case of high distances between the evaluations (i.e., 4 points or more between the ratings) the two experts discussed their evaluations until they reached a consensus. In total, 47 and 122 ADDs in ORExp and UniISExp respectively had to be revised. For the evaluation of the ADDs the following criteria were applied by both experts:

- The decision is stated clearly.
- The rationale of the decision is stated clearly.
- The decision corresponds to the described issue.
- The documented decision is a viable solution with regard to the described issue.
- Potential alternatives are documented.

Regarding the execution of the two controlled experiments, we did not have any deviations from the initial study design or situations in which participants behaved unexpectedly.

6.3 Analysis of Results

In order to analyze the data collected during the two controlled experiments and test our hypotheses (see 6.2.1) we used the R language and environment for statistical computing [Tea+05].

6.3.1 Descriptive Statistics

As a first step in the analysis of the results we used descriptive statistics to compare the means and medians of the observed variables. In particular, Table 6.6 and Table 6.7 along with Figure 6.3 and Figure 6.4 display the mean and median values of the number of documented ADDs (*numOfDecisions*), the quality per ADD (*quality*), as well as the average quality of ADDs per student (*qualityPerStudent*), the time needed for completing the required tasks (*time*) and for documenting one decision (*timePerDecision*), for both control and experiment groups in the OR-Exp and UniISExp experiments respectively.

It is noticeable that the participants of the experiment group documented one ADD more on average than the control group: 7 against 6 and 6 against 5 for ORExp and UniISExp respectively. However, this is not necessarily an indicator of "higher efficiency" of the students as the experiment group may have needed more time for performing the tasks and thus delivered more ADDs. We notice, though, that the control groups also spent more time on documenting a single decision, that is, 3.91 and 2.64 more minutes than the corresponding experiment groups on average. Therefore, the experiment groups needed *less time* to document *more decisions* although they have dedicated some of their time to understand and use the reusable ADD models. The reason is that CoCoADvISE with ADD model support generated semi-complete documentations which saved some time for the experiment group. We also calculated the average number of characters used for each ADD to find out whether the smaller number of ADDs and the increased time spent on their documentation was due to the smaller "size" of decisions. However, both treatment groups documented decisions of almost the same size in ORExp (489 characters for the control group and 431 for the experiment group). We notice also a small difference in the length of the ADDs in UniISExp – 890 characters for the control group and 588 for the experiment group. This can be explained by the fact that often the participants of the control group reused parts of the pattern documentations to fill in some ADD template fields (e.g., Argumentation/Implications) which led to longer ADD texts – unlike for the experiment group these documentations provided the main source for decision support.

As mentioned in Section 6.2, each documented ADD was evaluated by two independent experts using a 10-point Likert scale ranging from very low quality (1) to very high quality (10). Likert scales are treated in the literature as both ordinal (e.g., [Vig77]) and interval (e.g., [Mau98]) scales. Ordinal scales show us the order, but not the distances between the ranking, that means, that in a 10-point Likert scale 2 means better quality than 1, 3 better quality than 2, and so on.

Interval scales, on the other hand, show the order of things with equal intervals between the scale points. In order to be able to use descriptive statistics and statistical tests to analyze the quality of decisions we must treat the 10-point Likert scale as an interval scale. We assume, therefore, that this holds true in our case.

Variable	Means		Medians	
	control	exp.	control	exp.
numOfDecisions	5.59	7.57	6	7
quality	4.82	5.46	4.85	5.29
qualityPerStudent	4.9	5.35	4.5	5.5
time (min)	44.71	35.75	48.11	36.69
timePerDecision (min)	9.06	5.15	8.12	4.79

TABLE 6.6: Means and medians of observed and derived variables for OORExp

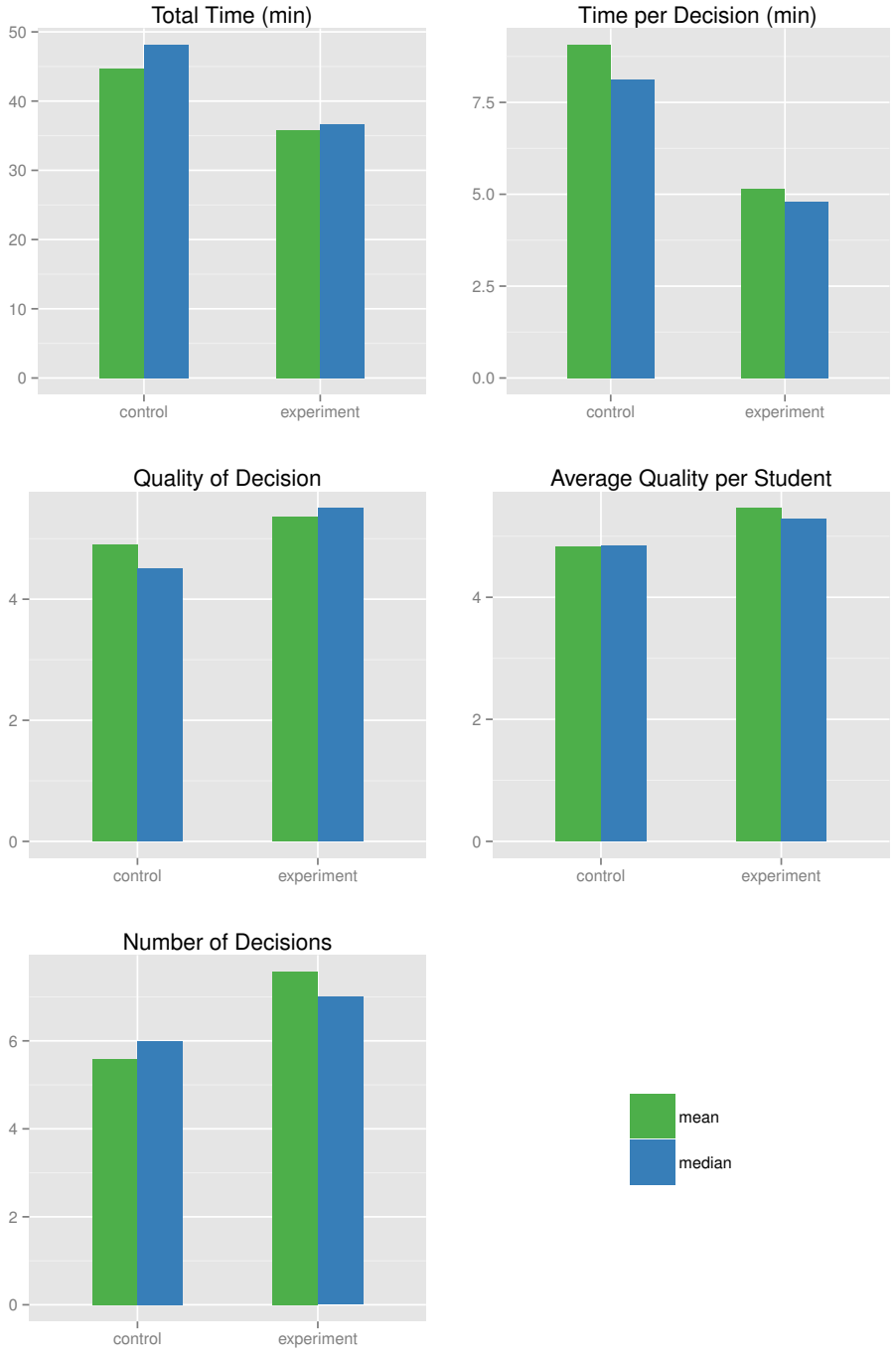


FIGURE 6.3: OORExp: comparison of means and medians for the observed and derived variables

Variable	Means		Medians	
	control	exp.	control	exp.
numOfDecisions	5.70	6.72	5	6
quality	6.34	6.96	6.79	7.04
qualityPerStudent	6.37	6.92	7	7
time (min)	57.50	44.99	58.37	45.74
timePerDecision (min)	10.72	8.08	11.52	7.20

TABLE 6.7: Means and medians of observed and derived variables for UniISExp

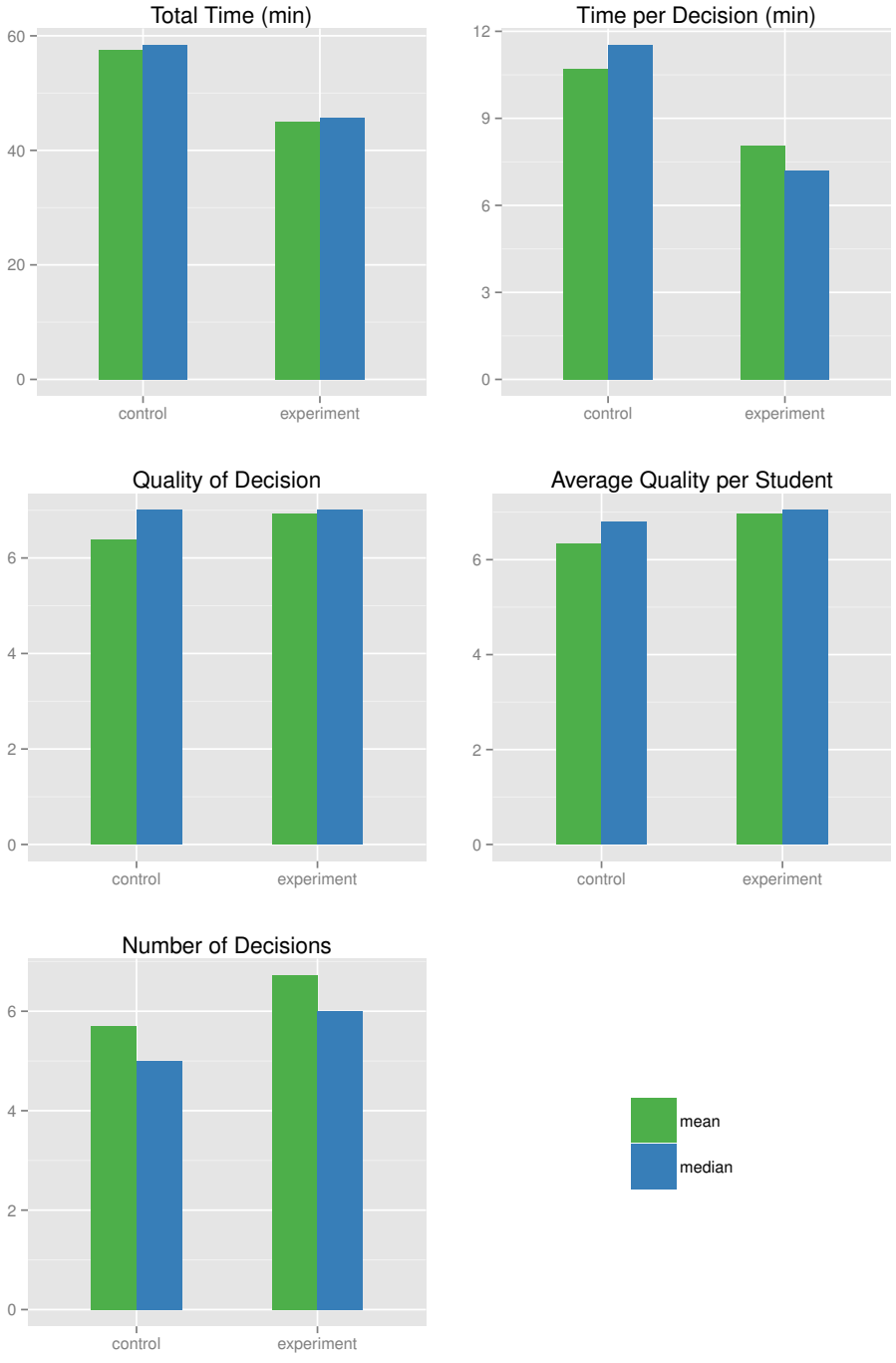


FIGURE 6.4: UniISExp: comparison of means and medians for the observed and derived variables

The average quality of the experiment group's ADDs is 5.46 and 6.96 compared to 4.82 and 6.34 in the control group for the ORExp and UniISExp respectively. In addition, the students in the experiment group delivered ADDs of better quality on average. To summarize, we can say that the treatment group that used the ADD models support *documented more ADDs* and achieved results of *better quality* and in *less time*.

On average, the participants of the experiment groups used 7 (ORExp) and 6 reusable decision models. In addition, we calculated the time needed for filling in the questionnaires and the ADD templates separately. In ORExp 25% and in UniISExp 16% of the total time (1.07 and 0.81 minutes per questionnaire respectively) was spent on answering the questionnaires. The rest of the time was spent on filling in the semi-complete ADD documentations. Therefore, the effort for using the decision support capabilities of CoCoADvISE is very small in comparison with the effort needed to make and document decisions without automated or semi-automated guidance.

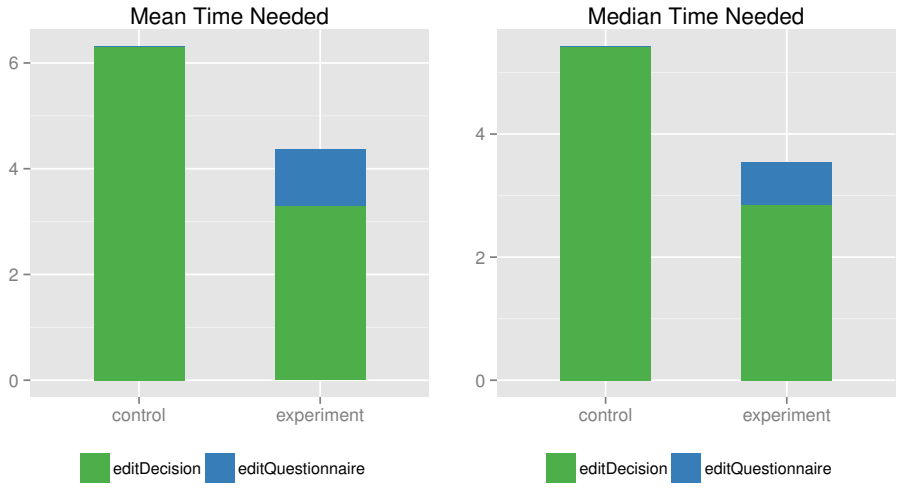


FIGURE 6.5: ORExp: comparison of time spent by participants for making/documenting ADDs

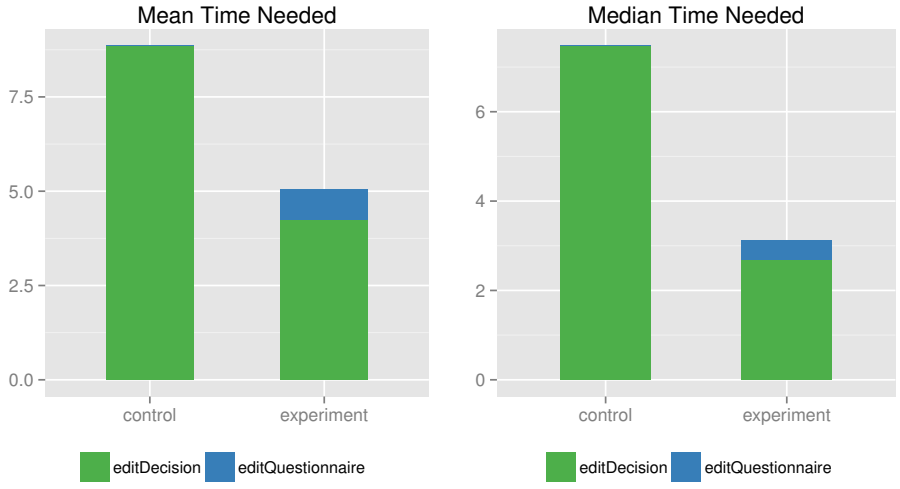


FIGURE 6.6: UniISExp: comparison of time spent by participants for making/documenting ADDs

6.3.2 Data Set Reduction

The few outliers that we noticed by studying the deviations from the means for each of the observed variables correspond to different participants, that is, these outliers correspond to ADDs documented by different students. Thus, the fact that these points have much higher or much lower values than the means (e.g., a student documented only a few decisions or needed much time for one decision) does not necessarily make the participant an outlier. Therefore, we excluded only students from the second controlled experiment (UniISExp) that had participated in the first as well and students who did not complete the practical course successfully (i.e., scoring less than 50% of the maximum points), and who therefore would make the study results vulnerable. This was done, however, before the data analysis (see explanation in Section 6.2.4); at this stage, we did not perform any further data set reduction.

6.3.3 Hypotheses Testing

6.3.3.1 Testing for Normal Distribution

Parametric tests like the *t*-test assume the normal distribution of the analyzed data. In order to decide whether to use parametric or non-parametric tests for the analysis of the data, we applied the Shapiro-Wilk [SW65] normality test. The null hypothesis of the Shapiro-Wilk test states that the input data is normally distributed. We test the normality at a level of confidence of 95%. That means that if the calculated p-value is lower than 0.05 the null hypothesis is rejected and the input data is not normally distributed. Conversely, if the p-value is higher than 0.05, we can not reject the null hypothesis that the data is normally distributed. Table 6.8 lists the p-values of the Shapiro-Wilk normality test for each observed variable and treatment group (all values are rounded to 4 decimal digits) for both controlled experiments. P-values that indicate normal distribution are emphasized (i.e., using **bold** font). We can see that only *numOfDecisions* and *qualityPerStudent* for OORExp and *time* for UniISExp controlled experiment exhibits a tendency of being normally distributed, while for the other variables we can not assume that they are normally distributed. As a result, we decided to pursue a non-parametric statistical test for analyzing the data.

Variable	*OORExp* **p-Value**		*UniISExp* **p-Value**	
	control	**exp.**	**control**	**exp.**
numOfDecisions	**0.493**	**0.0941**	0.0009	0.0042
quality	$0(10^{-5})$	$0(10^{-5})$	$0(10^{-11})$	$0(10^{-9})$
qualityPerStudent	**0.7323**	**0.7145**	0.0046	**0.0643**
time (min)	0.0107	**0.8832**	**0.3593**	**0.2717**
timePerDecision (min)	0.002	**0.918**	**0.5181**	$0(10^{-10})$

TABLE 6.8: Shapiro-Wilk normality test

Hypothesis	**Variable (μ)**	**p-Value**	
(**Assumption**)		**ORExp**	**UniISExp**
$\mathbf{H_{01}}$ ($\mu_{exp} \geq \mu_{control}$)	$numOfDecisions$	0.0027	0.0080
$\mathbf{H_{02}}$ ($\mu_{exp} \geq \mu_{control}$)	$quality$	0.0332	0.0459
	$qualityPerDecision$	0.0550	0.0173
$\mathbf{H_{03}}$ ($\mu_{exp} \leq \mu_{control}$)	$time$	0.0045	$0(10^{-5})$
	$timePerDecision$	0.0001	$0(10^{-5})$

TABLE 6.9: Hypotheses testing results (Wilcoxon rank-sum test)

Hypothesis	**Variable (μ)**	**p-Value**	
(**Assumption**)		**ORExp**	**UniISExp**
$\mathbf{H_{01}}$ ($\mu_{exp} \geq \mu_{control}$)	$numOfDecisions$	0.0021	
$\mathbf{H_{02}}$ ($\mu_{exp} \geq \mu_{control}$)	$qualityPerDecision$	0.0330	
$\mathbf{H_{03}}$ ($\mu_{exp} \leq \mu_{control}$)	$time$		$0(10^{-6})$

TABLE 6.10: Hypotheses testing results (t-test)

6.3.3.2 Comparing the Means of Variables

To compare the means of the observed variables for the control and experiment groups, we applied the one-tailed Wilcoxon rank-sum test [MR47], a non-parametric test for assessing whether one of two data samples of independent observations is stochastically greater than the other. Its null hypothesis, which is appropriate for the hypotheses in our experiments, is that the means of the first variable's distribution is less than or equal to the means of the second variable's distribution, so that we can write $H_0 : A \leq B$. The Wilcoxon rank-sum test tries to find a location shift in the distributions, i.e., the difference in means of two distributions. The corresponding alternative hypothesis H_A could be written as $H_A : A > B$. If a p-value for the test is smaller than 0.05 (i.e., the level of confidence is 95%), the null hypothesis is rejected and the distributions are shifted. If a p-value is larger than 0.05, the null hypothesis can not be rejected, and we can not claim that there is a shift between the two distributions. Table 6.9 contains the p-values of five Wilcoxon rank-sum tests that were performed to find out whether we can reject the null hypotheses presented in Section 6.2.1. Note that only the first four decimal places of the results are reported. Based on the obtained p-values, we can assess that almost all distributions show a statistically significant shift between each other and that most of the null hypotheses can be rejected. Analogously, Table 6.10 contains the p-values of t-Tests for those variables, where the assumption of normal distribution could not be rejected (see Table 6.8).

Testing Hypothesis $\mathbf{H_1}$ The experiment group documented significantly more ADDs than the control group. We reject the null hypothesis H_{01} (i.e., the use of reusable ADD models has no effect on the quantity related effectiveness of software architecture students) since the calculated p-value is 0.0027 (t-test: 0.0021) and 0.008 for the ORExp and UniISExp experiments respectively. Hence,

there is evidence that the use of reusable ADD models for decision making and documentation increases the *quantity related effectiveness* of software architecture students.

Testing Hypothesis H_2 In our experiments, we observed that the participants of the experiment groups delivered ADDs of better quality than the participants of the control groups. As this observation holds for both variables *quality* and *qualityPerStudent* we tested the null hypothesis H_{02} (i.e., the use of reusable ADD models has no effect on the quality related effectiveness of software architecture students) for these two variables. In the experiment ORExp, the p-values 0.0332 and 0.055 (> 0.05) do not allow us to reject H_{02} completely (however, t-test: 0.0330). We can reject this hypothesis for the quality of single ADDs but not for the average quality of documented ADDs per student. For UniISExp though, in which we tested a bigger sample of ADDs and participants, H_{02} could be rejected given the p-values 0.0459 and 0.0173 for the variables *quality* and *qualityPerStudent* respectively. Therefore, there is evidence for supporting H_2, that is, the ADDs of the experiment group were in total of better quality – according to the reviewers' evaluations – than those of the control group and the single participants' performance was also significantly "better". Hence, we can report evidence that using reusable ADD models for making and documenting decisions also increases the *quality related effectiveness* of its users.

Testing Hypothesis H_3 Finally, we discovered that the experiment group needed significantly less time to document a decision. This holds also for the total time this group spent on the assigned tasks. With a p-value of 0.0045 and 0.0001 for *time* and *timePerDecision* in ORExp and corresponding values close to 0 (10^{-5}) in UniISExp we can reject the null hypothesis H_{03} for both experiments (the same holds for the corresponding t-test for the variable *time*), that is, the use of reusable ADD models has no effect on the time related efficiency of software architecture students. Thus, we can conclude that there is evidence that reusable ADD models lead to increased *time related efficiency* of software architecture students as well.

6.4 Main Findings and Limitations

The following subsections discuss our main findings and their implications and inferences for software architects. We also report on the threats to validity.

6.4.1 Evaluation of Results and Implications

Increased Effectiveness The first two hypotheses, i.e., H_1 and H_2, are related to the effectiveness of software architecture students which we study separately with regard to the quantity and the quality of the documented ADDs. As reported in Section 6.3, we have provided strong

evidence for rejecting the corresponding null hypotheses H_{01} and H_{02}. Thus, reusable decision models contribute both to the completeness and the quality of ADD documentations.

The semi-automated guidance using questionnaires provided by the CoCoADvISE tool allowed software architecture students (representative for novice software architects) to (1) identify the ADDs that had to be made, (2) understand the design space in the corresponding case study, (3) find out the alternatives for each design issue, and finally (4) document the appropriate design solution according to the requirements. Some of the participants of the experiment groups stated afterwards that CoCoADvISE's reusable models helped them find quickly the answer to their problem without needing to read all design pattern documentations. That turned out to be especially useful in cases where the design solutions were not so obvious from the requirements and required better research and analysis of the design situation. For instance, for handling some complex processing tasks in the UniISExp case study, "single or multi threaded implementation of the tasks" would be viable solutions inferred by the reusable ADD models. Yet, most of the participants of the control group opted for a "pipes and filters" solution, which would clearly be "overkill" in this specific situation. Another phenomenon that we observed was that participants of the control groups were often not confident about what they should document in the ADD template fields. On the contrary, the reusable ADD models provided support (i.e., some fields are filled in automatically) and guidance (i.e., the reusable ADD models already contain design rationale) for filling in the decision related fields. We decided not to fill in further fields automatically (e.g., positions, arguments, etc.) as this would give a big advantage to the experiment groups and would make the comparison between the treatment groups difficult.

The observed phenomenon of experiment groups documenting more decisions can possibly be explained as follows. The reusable decision models guided the students to follow-on decisions that were not considered by the participants of the control groups at all.

Systematically capturing and leveraging reusable AK in the form of reusable decision models, therefore, may lead to increased effectiveness of novice software architects. Our findings also validate Zimmermann et al.'s claim about some of the benefits of architecting with reusable decision models: reusable decision models (1) provide means for the semi-automatic decision identification and (2) improve the quality of decision making [Zim+07].

Increased Efficiency We showed in the previous section that our last alternative hypothesis H_3 could also be accepted, that is, using CoCoADvISE's support on reusable ADD models reduces time and effort for software architecture students. This finding was highly expected though, as much "manual" work for making and documenting ADDs (e.g., reading documentations, filling in ADD templates repeatedly) is made redundant due to the semi-automated guidance by the questionnaires and the semi-complete documentations generated from decision recommendations. Thus, architects may need less time to document ADDs when guided by reusable ADD models.

6.4.2 Threats to Validity

To ensure the validity of our results, we consider the categorization of validity threats of Wohlin et al. [Woh+00] and discuss each of them separately in the context of our controlled experiments.

Conclusion Validity The conclusion validity focuses on the relationship between the treatment we used in the experiment and the actual outcome, i.e., on the existence of a significant statistical relationship. The way we measured the working time of the students automatically from the database entries may pose a threat to conclusion validity, as the users might have spent some working time idle or with pen and paper. Apart from that, to measure the actual time spent on working with CoCoADvISE for performing the assigned tasks is very difficult, if not impossible, as the participants may have spent some time reading the tasks or familiarizing with the tool. However, we think that idle working times, times spent on other tasks, or offline work can largely be excluded due to the limited experiment time of 90 minutes in which the participants needed to work in a concentrated manner in order to get the work done.

In addition, the interpretation of the 10-point Likert scale that was used to rate the ADDs may pose a threat to the conclusion validity. In Section 6.3, we argued that we consider the Likert scale an interval scale, and thus the descriptive statistics and statistical tests that we applied are, making this assumption, valid. Another potential threat to validity is the subjectivity of the quality ratings and of the reviewers. It could have happened as well that the one reviewer evaluated more strictly than the other. To reduce these risks, we asked two independent experts in the field of software architecture to evaluate all ADDs for both experiments and calculated afterwards the quality of each ADD on average. In case of disagreements in the evaluations the two reviewers needed to discuss their ratings until they reached a consensus. Nevertheless, this does not erase the subjectivity of the evaluations completely as some aspects related to the quality of ADDs like the evaluation of ADDs after the implementation of the software system are not taken into consideration in our case.

Internal Validity The internal validity refers to the extent to which treatment or independent variables caused the effects seen on the dependent variables. In order to reduce this kind of validity threats, we made sure that the participants of both groups had at least medium experience in programming and design – with slight differences – and that they were aware of the architectural design patterns they had to use for making and documenting architectural decisions. For this, the participants were asked to study all related patterns in advance. Apart from this, a few patterns were applied in previous practical exercises (i.e., programming exercises concerning the implementation of various software systems) of the lecture by the students.

The experiments were carried out in controlled environment and also the treatment group members were in different rooms. Thus, they could not know the goals of the experiment, as they were not aware of the existence of different groups and/or the different CoCoADvISE settings for the two treatment groups. During the experiment, the students were allowed to use only the CoCoADvISE tool and all other applications and Web pages were blocked. This way, we prevented access to other Internet materials as well as the participants' communication through chat applications. Also, an observer in the room ensured that no interactions between the participants of the same room occurred to ensure that the students worked individually.

We also prevented any access to CoCoADvISE and the accompanying materials outside the dedicated session or outside the labs where the controlled experiments were carried out.

Construct Validity The construct validity focuses on the suitability of the experiment design for the theory behind the experiment and the observations.

The variables that have been observed in the experiment are regarded accurate and objective as they are related to the actual use of tools and were automatically extracted from the database. The students did not have any training or experience with the tools. A potential threat to validity may be posed by improper use of CoCoADvISE by participants who did either not understand the purpose of the reusable ADD models and the questionnaires or did not comprehend the tasks. We also need to take into account language difficulties in some cases (CoCoADvISE texts were in English, all other materials were both in English and German). We tried to overcome such problems by encouraging the students to ask the instructors for further explanations. Another potential threat to construct validity is the fact that only one measure was used to evaluate the quality of the documented ADDs, which does not allow us to cross-check the experiments' results.

Finally, there is a potential threat to validity imposed by the participating subjects, knowing the research topics of our research group and therefore being able to "guess" results that we hoped to obtain. We tried to minimize the impact of this threat by choosing independent and external experts that were not aware of our research agenda.

External Validity The external validity is concerned with whether the results are generalizable outside the scope of our study. The subjects of the experiments had medium programming experience and were familiar with the ADDs they were asked to make. However, only few students had experience in the industry. We hence consider the participants of both controlled experiments to be representative for novice software developers or architects. A threat to validity of the relevance of our study's findings in practice is the use of students instead of practitioners in both experiments. We tried to mitigate the threat by educating the students on the patterns and architectural decisions used in the experiment. At the time of the experiments, all participants had theoretical

knowledge of, as well as programming experience with the majority of the design solutions that had to be considered for the required tasks. Therefore, our results can be likely to a certain extent transferred to more experienced architects. However, we will need to conduct similar experiments with practitioners in order to be able to consider our experiment results generalizable – applicable to professional novice (or more experienced) software architects. In addition, Kitchenham et al. regard students close to practitioners, as they are considered to be the next generation of software professionals [Kit+02], which strengthens our claim that our findings could apply to professional architects as well.

Also, the fact that the treatment groups exclusively used the CoCoADvISE tool, poses the threat that the results are specific only for this tool. However, in CoCoADvISE, we tried to implement general concepts and practices from reusable decision models [Zim+07] and ADD documentations [TA05]. Hence, we expect the same results in case the participants worked with tools based on similar concepts as well.

As mentioned before, the measurements were extracted from the tool database, avoiding any bias by the experimenters.

6.4.3 Inferences

Van Heesch et al. found out that the quality of ADDs increases when focus is set on the use of design patterns for documentation [Hee+12a]. Our findings confirm and supplement these previous results for students of software architecture. That means that the use of design patterns in the form of reusable ADD models further increases the effectiveness of students leading to less effort and better quality for ADD documentations. We claim that our results can be transferable to novice/trainee software architects, which needs to be proven, however, in an industrial context with practitioners.

Although we have conducted our experiments with students with software architecture background and with little commercial experience, we believe that our results can apply similarly to novice or more experienced architects in the industry. To validate these claims we will need more empirical evidence and feedback from the use of reusable architectural decision models in practice and in the context of industrial projects.

In practice, ADDs remain either undocumented or partially documented and the documentations are often dispersed in different places like wikis, meeting minutes, and issue tracking systems [MW13]. In addition, documenting ADDs that correspond to recurring design issues is tedious and time-consuming. Providing the means for semi-automated decision making and documentation based on reusable ADD models may encourage software architects to capture AK regularly and systematically. That will contribute to reducing AK vaporization [Har+07].

It is of course important that reusable ADD models that have been documented in the literature (e.g., [Zim+07; Lyt+12b; ZS09; May+09]) and have been tested in a very limited scale (i.e., by practitioners of the same company, etc.) get validated and enriched by practitioners from different domains. That will increase the acceptance of such reusable AK and confirm Nowak et al.'s vision of collaboration and knowledge exchange between different AK repositories (i.e., repositories containing reusable architectural decision models) [Now+10]. Therefore, the impact of tools like CoCoADvISE which provide semi-automated decision support based on such ADD models will get more significant.

6.5 Other Related Empirical Studies

There are various literature surveys and reviews on the approaches and tools for architectural decision making and documentation. For instance, Falessi et al. provide a comparison of various techniques for selecting architectural alternatives during decision making [Fal+11]. Shahin et al. provide a survey on existing decision models and tools [Sha+09] while Bu et al. provide an analysis of decision-centric approaches for design support [Bu+09]. The mapping study by Tofan et al. provides an overview and taxonomy of the existing literature on ADDs [Tof+14]. Furthermore, Weinreich and Groher compare software architecture management approaches with focus on the main aims of documenting AK as well as the elements used to represent AK [WG14]. Except for these literature surveys and reviews, little empirical evidence (especially quantitative results) exists on the use and benefits of ADDs in the industry and by practitioners. Heesch et al. conducted a multiple case study with four teams of software engineering students working in industrial projects in order to find out whether decision documentation supports novice architects in following a rational design process [Hee+13]. Shahin et al. use a controlled experiment with 10 participants to test their hypothesis that the visualization of ADDs and their rationale improves the understanding of architecture designs [Sha+11]. Also, the supportive effect of pattern use in recovering of ADDs in terms of quality and quantity of documented ADDs has been validated in a controlled experiment with 33 software architecture experts and practitioners in a different study [Hee+12b]. A few other qualitative studies such as [ZM05; MW13] focus on how software architects make and document ADDs and which of them are eventually being documented. To the best of our knowledge, no empirical-grounded evidence yet exists on the impact of reusable AK and in particular reusable decision models on the efficiency and effectiveness of software architects.

6.6 Conclusions

In this chapter, we reported the results of two separate controlled experiments on architectural decision making and documentation based on reusable decision models. Due to the current lack of

empirical evidence, the experiments were designed to determine the supportive effect of reusable ADD models for software architects. We were particularly interested in quantifying their impact on software architecture students' efficiency and effectiveness while making and documenting decisions. Our experiments provided strong evidence that reusable decision models can potentially increase both the effectiveness and the efficiency of novice software architects while making and documenting ADDs. We can further report, that our findings are in line with similar studies (see, e.g., [Hee+12b]) and support other claims about the benefits reusable ADDs (see, e.g., [Zim+07; Lyt+12b; May+09]) in principle.

Part III

Quality-driven Reusable Decisions

7 A Survey on Quality Attributes Use in Architectural Design Decisions

7.1 Introduction

During the architecting activity, quality attributes are considered major drivers of the design process in order to achieve high quality systems. Consequently, QAs must play a role for decision-making processes and be documented alongside the decisions captured. Although most AK management approaches focus on capturing and sharing the design decisions made, only a few of them address the relationships between QAs and ADDs. To the best of our knowledge, the use and integration of QAs in existing AK management methods and tools have been under-investigated and there is a need to provide an accurate description of the impact and relationships between both types of artifacts as a research path to improve existing methods. In this chapter, we summarize the results of a literature survey that studies the importance and impact of the relationships between QAs and ADDs and to what extent existing AK management methods and tools deal with the decisions that affect the quality of the system. We also report on the implications of the relationship between qualities and decisions and we state the related challenges and future research paths for AK methods and tools.

7.2 Selection Method and Selected Publications

Unlike systematic literature reviews (SLRs) in software engineering which entail well-defined steps for identifying, analyzing and interpreting all evidence related to specific research questions [KC07] and conduct an extensive and exhaustive search and evaluation of results, literature surveys focus on analyzing and summarizing a non-complete publication list with narrow scope. We decided to conduct a literature survey for the following reasons: (a) we wanted mainly to provide an overview and qualitatively summarize the literature about methods and tools for AK management which provide an updated view of the current AK management research on software architecture, and (b) the related tools and methods supporting AK management cover the majority of the spectrum of AK management research since 2004.

We collected and analyzed publications from the period 2004-2014, as the topic of ADDs started to become relevant for the software architecture community since 2004[1]. Since Bosch [Bos04] pointed out that a software architecture should be seen as the result of a set of design decisions rather than a set of components and connectors, the software architecture community started to discuss lively about the importance of knowledge vaporization and how ADDs should be captured and represented. Therefore, a significant body of research on the topic has been produced along the past ten years. Consequently, we selected papers from related conferences and workshops, such as Working IEEE/IFIP Conference on Software Architecture (WICSA), European Conference on Software Architecture (ECSA), Conference on the Quality of Software Architectures (QoSA), and Workshop on Sharing and Reusing Architectural Knowledge (SHARK). We also considered articles from journals like the Journal of Systems and Software (JSS), Information and Software Technology (IST), and IEEE magazines among others, as these venues encompass the main body of knowledge in the topic under study.

The search and selection of the related papers was done according to the involved researchers' experience in the domain of ADDs research and with the aid of search tools of digital libraries (SCOPUS, ACM Digital Library, IEEE Xplore, SpringerLink, CiteSeer, ScienceDirect, etc.). The inclusion/exclusion criteria were used to select all those papers focusing on methods and tools for ADDs support and documentation and to reject papers that focused on other forms of AK management or did not refer explicitly to ADDs. During a first pass, we selected, after multiple iterations, 58 papers based on the title, abstract, and keywords. In a second pass, we filtered out the remaining ones performing a reading of the contents and excluded works which were not explicitly focusing on ADDs or were proposing general methodology and/or guidance for ADDs. The minimum requirement was that the publication should be related to at least one of the three research questions we address in the survey. In addition, we excluded works that referred to the same approaches and/or tools and kept only the most recent or most detailed publications (e.g., journal or conference papers instead of workshop papers). As a result, we consolidated a list of 27 publications (5 journal, 17 conference, and 5 workshop publications) and we classified the selected publications into three focus areas according to our research goals. Our analysis concentrates on the most relevant papers for practitioners and experienced software architects that contain explicit relationships between QAs and ADDs and the underpinning methods and approaches (supporting them) and covers the majority of existing research related to our research questions in the period 2004-2014.

Table 7.1 lists the selected papers along with the publication year and venue as well as the name of the method or tool used for architectural decision making and documentation. The majority of the publications (21) are from the years 2007-2014 which gives an indication of the increasing interest of the software architecture community in ADDs in recent years. In addition, in most of the cases,

[1]With the exception of [Mac+91] which was published in 1991.

the proposals are accompanied by tools for software architects. More specifically, we analyzed 13 tools for decision making, 5 for decision documentation, and 9 for both tasks (see also Table 7.2).

Publication	**Tool/Approach Name**	**Venue**	**Year**
Zimmermann et al. [Zim+08]	ArchPad (RADM)	WICSA	2008
Jansen et al. [Jan+07]	Archium	WICSA	2007
Cui et al. [Cui+08]	-	WICSA	2008
Tyree and Akerman [TA05]	Text templates	IEEE Software	2005
Lytra et al. [Lyt+13]	ADvISE	ECSA	2013
Zalewski et al. [Zal+11]	MAD 2.0	ECSA	2011
Makki et al. [Mak+08]	ADD+	ECSA	2008
Boer et al. [Boe+09]	QuOnt	WICSA/ECSA	2009
Babar and Gorton [BG07]	PAKME	SHARK	2007
Zimmermann et al. [Zim+07]	RADM	QoSA	2007
Shahin et al. [Sha+10]	Compendium	SHARK	2010
Capilla et al. [Cap+07]	ADDSS	SHARK	2007
Kruchten et al. [Kru+06]	Ontology (with Aduna Cluster Map)	QoSA	2006
Ameller and Franch [AF14]	ArchiTech (Quark)	CLEI electronic journal	2014
Harrison et al. [Har+07]	-	IEEE Software	2007
Zdun [Zdu07]	-	Software Practice & Experience	2007
MacLean et al. [Mac+91]	QOC	Human-Computer Interaction	1991
Babar and Capilla [BC08]	PAKME	MARK	2008
Lytra and Zdun [LZ13]	FuzzDS	SESoS	2013
Konersmann et al. [Kon+13]	ADMD3	QoSA	2013
Choi et al. [Cho+06]	AQUA	FMOODS	2006
Al-Naeem et al. [AN+05]	ArchDesigner	ICSE	2005
Heesch et al. [Hee+12a]	-	WICSA/ECSA	2012
Harrison and Avgeriou [HA07]	-	ECSA	2007
Nowak and Pautasso [NP13]	Software Architecture Warehouse	ECSA	2013
Tibermacine et al. [Tib+06]	AURES	CBSE	2006
Dermeval et al. [Der+12]	STREAM-ADD	COMPSAC	2012

TABLE 7.1: Selected papers overview

In the following sections, we introduce the research questions we investigated in this literature survey, and we present the survey results for the publications that have been reviewed (see Table 7.1).

7.3 Research Questions

Because the main goal of our survey was to investigate the explicit interactions and relationships between ADD approaches using QAs, from the overall list of papers examined we focused on those that clearly address the research goals that motivated our study. The selected papers provide solid evidence and supportive methods able to manage such relationships from a practical perspective. We summarize below the main objectives of our study:

1. Review QA-related topics with regard to ADDs to understand the relationships between both artifacts.

2. Review evidence for the relationship between QAs and ADDs to find out how these are modeled and documented.
3. Review those ADD methods that use QAs in the decision making activity and vice-versa.
4. Review approaches that use incomplete knowledge to make decisions using QAs.
5. Identify research gaps for improving existing ADD methods and tools with regard to supporting QAs.

Therefore, we used these drivers to derive the following research questions that we used in our survey to analyze the role and importance of QAs in ADDs in software architecture research works.

RQ1 What is the role/importance of QAs in existing ADD methods and tools?

RQ2 Which QA topics are addressed, to what extend, and by what types of ADD methods and tools?

RQ3 What is the scope and what research methods are used to evaluate the role of QAs in existing ADD approaches?

The first research question (RQ1) looks for evidence for QA support in the existing ADD methods. For answering the second research question (RQ2), we are interested in extracting QA-related topics that are challenging at decision making. Finally, for research question RQ3, we address the maturity of ADD methods and real experiences that deal with QAs. In order to answer each of the research questions we defined several criteria to evaluate the selected publications according to these criteria. We iterated several times to refine the criteria as we discovered new aspects for consideration while evaluating the selected papers. In the following subsections we present the categorization criteria used for the three RQs under investigation in detail.

7.3.1 RQ1: Criteria to Evaluate the Scope and Use of QAs in ADD Methods

1. *Focus on QAs*: Some approaches target explicitly QAs in architectural decision making and/or documentation, while other proposals contain only implicit or very limited information related to QAs.
2. *Appearance of QAs*: We distinguish between tools that integrate QAs in decision making, in ADD documentation, or in both.

3. *Support for Implicit/Explicit QA-ADD Relationships*: We investigate the relationships between QAs and ADDs with a positive or negative impact on specific QAs, and whether these are explicitly or implicitly documented.

4. *Evaluating the Interactions between ADDs and QAs*: Some approaches use QAs to evaluate the best or the optimal decision choices using QAs in order to evaluate alternative decisions. However, other research works focus on how to make design decisions in order to select one or several QAs that are more important for the architecture (ADDs impact on the selection of QAs). Consequently, the interactions between ADDs and QAs are bidirectional.

7.3.2 RQ2: Criteria to Evaluate QA-related Topics Used in ADD Methods

1. *QA Uncertainty*: Uncertainty is caused by vague, incomplete, or imprecise information about QAs of design solutions and requirements [LZ13]. That uncertainty is supported by an approach means that the approach provides means for expressing and/or resolving QA uncertainty.

2. *QA Interdependencies*: QAs may have positive or negative impact on other QAs. Apart from that, prioritization of QAs is often considered in architectural decision making.

3. *QA Trade-offs*: Making ADDs is essentially the result of making trade-offs between competing requirements and stakeholders' concerns. In case QA trade-offs are addressed, we are further interested in whether their resolution is performed manually or semi-automatically by software architects.

7.3.3 RQ3: Criteria to Evaluate the Maturity of ADD Methods with Regard to the Design Space

1. *Design Space*: We analyze the size and scope of the design space or spaces in which each evaluated method or tool has been applied. The rationale here is that we consider methods and tools which have been applied for one or more substantial or realistic design spaces to be likely more mature than one that is only applied for toy examples.

2. *Evaluation Method*: We study the methods used to evaluate the methods and tools under study.

3. *QAs*: We finally analyze which QAs are used during the evaluation of design decisions in the publications under study, as well as their frequency.

7.4 Role and Importance of QAs in ADD Tools and Methods

The importance of QAs during decision making in software architecture and the fact that good decisions affect the selection of the right QAs during architecture evaluation and reviews are widely recognized. In order to answer RQ1, in this section we discuss the role of QAs in AK approaches. More specifically, we will address in the following subsections (1) when the QAs appear in the decision making process or are just documented, (2) the explicit and implicit relationships between ADDs and QAs, (3) how decisions are evaluated using QAs and how QAs are used to decide on the best alternative, and (4) if the AKM approaches examined mainly focus on QAs. Table 7.2 summarizes our findings with regard to RQ1.

7.4.1 Documenting Decisions and QAs

As mentioned before, it is of key importance to document the QAs that will drive the shape of the architecture and the decisions made during the architecting activity in order to prevent AK vaporization. From the approaches analyzed for this RQ1, we observed three major trends. The first trend is to document QAs in approaches using AK. The second trend is documenting QAs in the decision making process, and the third set of research works document QAs in both documentation and decision making.

In the first case, QAs are described explicitly in the documentation alongside with the design decisions. This approach is taken by three of the examined papers, excluding those that appear in both documentation and decision making (i.e., [Sha+10; Kru+06; Tib+06]). While some works [TA05] document QAs using templates for describing the design decisions, others use tool support to document QAs alongside with the decisions made using, for instance, a matrix to describe the relationships between both artifacts [Sha+10].

The second trend describes how the QAs are documented and used in the decision making process. At least 12 works highlight the importance of QAs in the reasoning activity and how QAs act as major decision drivers (e.g., [Zim+08; AF14; BC08; LZ13]). In some of these works, QAs play a key role for architectural trade-off decision making that sometimes is carried out using multiple attributes [Mak+08]. Different criteria and methods can be used to evaluate the role of QAs in the decision process such as the Questions, Options, and Criteria (QOC) method [Mac+91] or utility trees [BC08]. Corresponding tool support can assist software architects in the decision making process using QAs for evaluating alternative decisions. Other tools provide automated support for managing the interactions between QAs and decisions such as the ArchiTech tool, an application of the Quark approach [AF14]. Finally, at least nine approaches document both the QAs in ADD documentations while at the same time they describe how QAs influence the reasoning activity for ADDs. For instance, the STREAM-ADD approach [Der+12] suggests to explicitly document

ADDs alongside NFRs and to use QAs during architectural refinements in structural and technology decisions. In this approach, NFRs are used with soft-goals to evaluate the set of alternatives using architectural styles and the rationale for them. In [LZ13] we have proposed a fuzzy model to represent AK and make decisions under uncertainty in which QAs are linked to design patterns. In this case, a semi-automated support for decision making and documentation of reusable ADDs under uncertainty using a fuzzy logic expert system is used in combination with a requirements editor, used to provide input to the fuzzy reasoning system.

7.4.2 Explicit and Implicit Relationships between ADDs and QAs

Regarding explicit and implicit relationships between ADDs and QAs, we explored the type of the relationships between decisions and quality requirements. This is important for the software architecture documentation because it helps software maintainers to identify the trace links between both artifacts when decisions change and to re-evaluate the decisions if QAs are modified. From our results we identified two types of links between both artifacts: (a) fully explicit and (b) support for it but not fully explicit relationships. From the approaches that model and define explicit ADD-QA links we can highlight the one introduced by Cui et al. [Cui+08]. The authors describe such explicit connections in a meta-model, linking decisions to design artifacts and requirements, which serves supporting an automatic synthesis method of candidate architecture solutions. The decision-centric architecture design process uses the QAs to identify the issues that will lead to the issue solutions. These issue solutions will be used to synthesize the solution architecture from the decisions captured with other software engineering artifacts and the relationships among them. Other approaches [Zim+07; Cap+07; Hee+12a] provide different but similar ways to relate several types of ADDs with QAs and document such relationships explicitly with the use of meta-models.

Complementarily, a certain support for ADD-QA links that is not fully explicitly described can be found in other approaches [Har+07; Zal+11; NP13]. In these approaches, the authors highlight mainly the relationships between ADDs and QAs. Such links are leveraged by tools like MAD 2.0 to support decision making during architecture evolution [Zal+11] or the Software Architecture Warehouse (SAW) to handle collaborative decisions with voting support [NP13]. However, documenting such links explicitly is not a major concern of the aforementioned approaches.

Finally, we did not find evidence for approaches describing only implicit relationships between both artifacts.

7.4.3 Evaluation of ADDs with QAs and Vice Versa

Decisions can be used to evaluate which QAs are more suitable to improve the quality of the architecture. Vice versa, alternative decisions can be evaluated based on qualities as the pros and

Publication	Focus on QAs	Appearance of QAs	Relationship ADDs-QAs	Evaluation of ADDs with QAs and vice versa
Zimmermann et al. [Zim+08]	no	decision making	explicit	not supported
Jansen et al. [Jan+07]	no	both	explicit	partially used
Cui et al. [Cui+08]	no	decision making	explicit	fully used for evaluation
Tyree and Akerman [TA05]	no	documentation	not fully explicit	partially used
Lytra et al. [Lyt+13]	no	both	not fully explicit	not indicated
Zalewski et al. [Zal+11]	no	both	not fully explicit	not indicated
Makki et al. [Mak+08]	yes	decision making	fully explicit	fully used for evaluation
Boer et al. [Boe+09]	yes	decision making	explicit	fully used for evaluation
Babar and Gorton [BG07]	no	both	not fully explicit	captured but not used
Zimmermann et al. [Zim+07]	no	decision making	explicit	partially used
Shahin et al. [Sha+10]	no	documentation	explicit	not supported
Capilla et al. [Cap+07]	no	both	explicit	captured but not used
Kruchten et al. [Kru+06]	no	documentation	explicit	captured but not used
Harrison et al. [Har+07]	no	documentation	not fully explicit	not supported
Zdun [Zdu07]	yes	decision making	fully explicit	fully used for evaluation
MacLean et al. [Mac+91]	yes	decision making	explicit	fully used for evaluation
Babar and Capilla [BC08]	yes	decision making	explicit	fully used for evaluation
Lytra and Zdun [LZ13]	yes	both	explicit	fully used for evaluation
Konersmann et al. [Kon+13]	no	decision making	not fully explicit	not supported[1]
Choi et al. [Cho+06]	yes	both	explicit	fully used for evaluation
Al-Naeem et al. [AN+05]	yes	decision making	explicit	fully used for evaluation
Heesch et al. [Hee+12a]	yes	both	explicit	fully used for evaluation
Harrison and Avgeriou [HA07]	yes	decision making	not fully explicit	fully used for evaluation
Ameller and Franch [AF14]	yes	decision making	explicit	fully used for evaluation
Nowak and Pautasso [NP13]	no	decision making	not fully explicit	not supported
Tibermacine et al. [Tib+06]	yes	documentation	explicit	fully used for evaluation
Dermeval et al. [Der+12]	no	both	explicit	captured but not used

[1] Their presentation from http://is.uni-paderborn.de/fileadmin/Informatik/architekturen2012/dokumente/Architekturen2012_Reussner_Architecture_SPP1593_ADVERT.pdf indicates partial support.

TABLE 7.2: Results summary for RQ1

the cons of each decision. In the first case, we decide on the best or worst QAs that must be present in a design. In the second case, the pros and cons are used to decide for the right or the optimal decisions (e.g., add a middleware or protocol to implement security) for a desired quality (e.g., security).

Other works suggest a multi-attribute decision making approach to make decision trade-offs, evaluate the most suitable QAs implemented in the system [Mak+08], and estimate the impact of a decision on the software quality [AF14]. Sometimes, in contrast, qualities are used to make technology decisions among several alternatives (e.g., in [Hee+12a]). In other cases, decisions and QAs are partially used or they are captured but the explicit usage is not indicated [TA05; Zal+11; LZ13]. Finally, there are some research works that do not support or do not indicate any of the aforementioned types of evaluations using ADDs and QAs (e.g., [Har+07; Zim+08; Sha+10; Zal+11; NP13; LZ13]).

7.4.4 AK Approaches Focus on QAs

A bit less than 50% of the approaches using design decisions and other forms of AK examined in this literature survey had a main focus on QAs. From our observations we can infer that QAs are relevant enough for AK management practices but some research works do not address QAs explicitly or the evaluation using QAs and ADDs is not the main goal.

7.5 QA-related Topics in ADD Tools and Methods

In this section, we discuss the QA-related topics that are addressed in existing tools and methods for architectural decision making and documentation. In particular, we focus on making of QA trade-offs, expressing and resolving QA uncertainty, describing QA interdependencies, as well as other QA-related topics like the formalization of QAs and their relationships to ADDs that are discussed in a few cases. Table 7.3 summarizes our findings with respect to RQ2.

Publication	QA Trade-offs	QA Uncertainty	QA Interdependencies
Zimmermann et al. [Zim+08]	Yes	No	No
Jansen et al. [Jan+07]	No	No	No
Cui et al. [Cui+08]	Yes	No	No
Tyree and Akerman [TA05]	Yes	No	No
Lytra et al. [Lyt+13]	No	No	No
Zalewski et al. [Zal+11]	No	No	No
Makki et al. [Mak+08]	Yes	No	Yes
Boer et al. [Boe+09]	Yes	No	Yes
Babar and Gorton [BG07]	Yes	No	No
Zimmermann et al. [Zim+07]	Yes	No	No
Shahin et al. [Sha+10]	No	No	No
Capilla et al. [Cap+07]	No	No	No
Kruchten et al. [Kru+06]	No	No	No
Harrison et al. [Har+07]	No	No	No
Zdun [Zdu07]	Yes	Yes	No
MacLean et al. [Mac+91]	Yes	No	No
Babar and Capilla [BC08]	Yes	No	Yes
Lytra and Zdun [LZ13]	Yes	Yes	No
Konersmann et al. [Kon+13]	No	No	No
Choi et al. [Cho+06]	No	No	No
Al-Naeem et al. [AN+05]	Yes	No	Yes
Heesch et al. [Hee+12a]	No	Yes	No
Harrison and Avgeriou [HA07]	No	No	No
Ameller and Franch [AF14]	Yes	No	Yes
Nowak and Pautasso [NP13]	Yes	No	No
Tibermacine et al. [Tib+06]	No	No	No
Dermeval et al. [Der+12]	Yes	No	No

TABLE 7.3: Results summary for RQ2

7.5.1 QA uncertainty

The task of making rational ADDs contains many sources of uncertainty. Uncertainty is caused, among others, by imprecise and unknown QAs of design solutions. Also, software architects often need to consider competing and imprecise requirements, various design alternatives and technology implementations, informal documentations and domain expertise which introduce additional sources of uncertainty. The task of resolving this kind of uncertainty during decision making is on the one hand complex and on the other hand time-consuming [LZ13], thus, it is challenging to make uncertainty explicit when considering alternative ADDs and their QAs and to resolve this uncertainty at decision making.

Although in a few cases this inherent uncertainty of QAs has been made explicit, the resolution of this kind of uncertainty is not supported by the majority of the existing tools and methods.

For instance, Zdun expresses uncertainty of quality goals using approximated scores (++ for very positive influence, + for positive influence, o for no influence, - for a negative influence, and -- for a very negative influence expected) [Zdu07]. Also, the aforementioned approach allows combination of scores in order to express a bigger degree of uncertainty (e.g., a design solution has either no or negative influence on maintainability). A similar notation is used by van Heesch et al. for evaluating decision forces (mainly QAs) of alternative design solutions for both decision making and documentation [Hee+12a].

Fuzzy Logic in [LZ13] allows not only for expressing but also for resolving uncertainty. This is achieved by modeling QAs as fuzzy sets and their membership functions (e.g., the QA performance can be described as high, medium, or low and these linguistic values can be mapped to overlapping membership functions) – see also Chapter 9.

7.5.2 QA Interdependencies

The interaction between QAs is widely recognized in software architecture evaluation. In the literature analyzed various kinds of interdependencies are being discussed. First of all, a QA may have *positive or negative impact* on other QAs [EG04]. For instance, increasing system security will come at the cost of availability [Bas+03]. Apart from that, QAs often have *overlapping* definitions; for instance, portability can be a subset of modifiability. Also, *prioritization* is a way of expressing relationships between qualities, that is, QAs are considered to be more or less important than other QAs at decision making. Considering all kinds of QA interdependencies at decision making is complex and can hardly be addressed manually.

Not all kinds of QA interdependencies, however, have been studied extensively in the existing literature. Resolving priorities for QAs has been mainly addressed in the related approaches.

Preferences on QAs are given by comparing them pairwise and giving quantitative weights in the quality-driven approach for decision making by Al-Naeem et al. [AN+05]. In the Quark approach, prioritization of QAs can be used by a decision inference system to provide a prioritized list of ADDs [AF14]. de Boer et al. use also prioritization of QAs [Boe+09]. Priorities are quantified on an ordinal scale and affect the ranking of design solutions with respect to the a list of QAs of interest. Apart from that, de Boer et al. define two other interesting QA interdependencies. The first is, "a QA *hasSubCharacteristic* QA". The second one is expressed by the reciprocal relation between effect of quality criteria on QAs related to relative strength (stronger than, weaker than, or comparable to other effect).

Dependencies among QAs are indirectly expressed through prioritized and conflicting quality scenarios by Makki et al. [Mak+08]. Finally, Babar and Capilla define subsets of QAs in a form of utility trees by distinguishing between quality attributes and quality factors [BC08]. For example, links are established between the QA performance and its related quality factors latency and throughput. Quality factors are afterwards related to concrete quality scenarios describing situations regarding the expected quality of a system.

7.5.3 QA Trade-offs

First of all, ADDs are the result of making trade-offs for requirements related to QAs, often competing and imprecise [Bas+03]. Many of the approaches share this common view for architectural decision making and propose various manual or semi-automated methods for resolving QA trade-offs. In this section, we focus on the approaches that either discuss or imply making of trade-offs. In particular, 15 of the 27 approaches under study support making trade-offs at decision making – mainly manually. Table 7.4 gives an overview of the approaches that support trade-offs along with the level of automation they provide (manual, semi-automatic) and the method used to achieve these trade-offs.

From the approaches which focus on trade-offs at architectural decision making very, few propose to resolve these trade-offs automatically or semi-automatically. For instance, the ADD+ method [Mak+08] supports automatic trade-offs between conflicting and incomplete quality scenarios and various stakeholders' concerns and preferences. Makki et al. formalize the process of stakeholders' preference elicitation and architectural decision making as a multi-attribute decision problem, in which the attributes represent the degree of satisfaction of conflicting quality scenarios. Another approach that supports automatic trade-offs uses fuzzy logic to automatically infer best-fitting solutions from competing and vague requirements [LZ13]. In particular, fuzzy rules expressing the relationship between design alternatives and QAs along with provided requirements are processed by a fuzzy inference system to produce an ordered list of design solutions. ArchDesigner considers making of trade-offs a multi-attribute decision making problem and leverage the

Analytic Hierarchy Process (AHP) to calculate value scores for design alternatives given their relative impact on QAs and relative stakeholders' preferences for QAs [AN+05]. The tooling introduced by de Boer et al. can be used for automated trade-off analysis as well, that is, the software architects select the QAs they will include in their analysis, set QA priorities and finally receive rankings of the alternative solutions under study, according to the information stored in QuOnt, an ontology for reuse of quality criteria [Boe+09]. Furthermore, Ameller and Franch formalize quality requirements and design constraints with a context free grammar and use constraint satisfaction algorithms to provide a prioritized list of ADDs [AF14].

Method/Tool Name	**Automation Level**	**Type of Trade-off**
RADM [Zim+08]	Manual	Reusable decision models are mainly based on patterns and patterns are considered to provide trade-offs between QAs. In addition, trade-offs are supported by SWOT tables and QOC diagrams.
Decision-centric architecture design method [Cui+08]	Manual	Architects select from synthesized architecture solutions according to their pros and cons with respect to FRs and NFRs.
ADD Templates [TA05]	Manual	Trade-offs are supported by tables including quality attribute related questions and their answers for each design alternative.
ADD+ [Mak+08]	Semi-automatic	The process of stakeholders' preference elicitation and architectural decision making are formalized as a multi-attribute decision problem.
QuOnt [Boe+09]	Semi-automatic	Partial ordering is used to calculate scores for QAs based on QA prioritization and QA dependencies.
PAKME [BG07]	Manual	Trade-offs are achieved by reusing pattern-based AK in elicited scenarios.
ArchPad [Zim+07]	Manual	See RADM.
[Zdu07]	Manual	Visual structures similar to QOC structures are used.
QOC [Mac+91]	Manual	QOC diagrams provide a kind of trade-off structure for decision making support.
PAKME integrated with QAs [BC08]	Manual	Trade-offs are supported with the aid of utility trees.
Fuzzy-based models [LZ13]	Semi-automatic	A fuzzy inference system is used to infer best fitting solutions based on fuzzy rules involving QAs and requirements.
ArchDesigner [AN+05]	Semi-automatic	The Analytic Hierarchy Process (AHP) is used to calculate value scores for design alternatives considering their impact on QAs and the stakeholders' preferences.
ArchiTech [AF14]	Semi-automatic	Trade-offs are made by solving a constraint satisfaction problem based on constraints posed by required qualities and QA priorities.
Software Architecture Warehouse [NP13]	Manual	Trade-offs are made in a group discussion after voting on the advantages/disadvantages of alternative design solutions.
STREAM-ADD [Der+12]	Manual	QA trade-offs are made by considering the fulfillment and the priorities of softgoals and NFRs with regard to the design options.

Table 7.4: Summary of QA trade-offs

Currently, many approaches integrate trade-offs in the architectural decision making process and

discuss methods to facilitate balancing of requirements and architects' preferences. Quality scenarios are used in some cases to assist making of trade-offs. In PAKME, for instance, these scenarios can help the architect identify the patterns that can be used to satisfy the requirements [BG07; BC08]. ADDs are assigned a ranking based on the architects' opinion about each ADD's capability of achieving a certain level of a required quality factor related to one or more QAs. The ArchPad (RADM) method of Zimmermann et al. provides reusable mainly pattern-based decision models which entail the information for resolving requirements at different levels and stages [Zim+08; Zim+07]. Pattern-based decisions are considered to be trade-offs between QAs. Trade-offs with the RADM technique are made using techniques like semi-structured SWOT (Strengths, Weaknesses, Opportunities, and Threats) tables, QOC diagrams, etc. Cui et al. propose a method for assisting the selection of and reasoning on architectural solutions [Cui+08]. First, requirements are listed as FRs and NFRs, second, the related design issues are elicited, and finally, discovered issue solutions with their advantages and disadvantages are related to each design issue. Afterwards, the issue solutions are connected to each other with inclusive, generalized, and conflictive relations, based on which candidate combinations of solutions are extracted. Architects use the summarized advantages/disadvantages of the issue solutions as well as additional evaluations of each architecture solution to make trade-offs. In addition, STREAM-ADD includes a manual trade-off analysis of alternatives considering the fulfillment and the priorities of softgoals and NFRs [Der+12].

Tables with QA-related questions for the software architects such as the ones presented by Tyree and Akerman [TA05] can serve as visualization aid for the evaluation of alternative solutions against QAs, as well as stakeholders' preferences or rankings, and eventually support architects in making trade-offs. Criteria in the Questions, Options, and Criteria (QOC) method play an important role in understanding trade-offs in the design space [Mac+91]. MacLean et al. claim that the connecting lines between alternative options and criteria – solid for positive evaluation, dashed for negative evaluation – express a simple trade-off structure: Options against conflicting Criteria. A similar visual structure for understanding trade-offs is introduced in [Zdu07] although trade-offs are not explicitly discussed in this work.

In Software Architecture Warehouse trade-offs are made in a collaborative context [NP13]. The stakeholders discuss the advantages and disadvantages of the design alternatives until they agree on common positions and a common outcome (approval or rejection of a specific ADD).

7.6 Maturity and Evaluation for QAs in ADD Tools and Methods

As we discussed in the previous section, dealing with QAs in architectural decision making is complex because of the nature of QAs: they are often contradicting, interdependent, imprecise and various trade-offs need to be made based on vague and incomplete requirements. Many approaches have addressed these issues proposing different methods and tools. Yet, the demonstration and

evaluation of these approaches in real life scenarios and empirical studies provides validation of the respective proposals and guidance for practitioners who need to integrate QAs in architectural decision making and documentation processes. For this, we report in this section on the evaluation methods performed, as well as the size and scope of the design space of study. The rationale for including the design space size and scope relies on the importance for each industrial and academic case study about issues such as decision capturing effort, number of dependencies between decisions and other software artifacts, type and nature of the decisions, number of alternatives considered, and the relevant QAs for each application domain.

The big majority of the proposals has been demonstrated only for a small number of decisions (<20). However, Zimmermann et al. [Zim+07; Zim+08] have created an impressive design space consisting of 300 reusable SOA-related decisions. We do not have any information though regarding the results or experiences of using these reusable ADDs along with their QAs in practice. The big majority of the proposals has been demonstrated in a case study (nine approaches) or using motivating examples (nine approaches). Only in two cases, the authors describe that the proposed methods and/or tools have been used in real industrial projects [Zim+07; Cap+07], but no specific empirical evaluation or lessons learned from these projects are reported. A few approaches (such as [Har+07]) do not perform any kind of evaluation or they just document types of decisions ranging from those based on concrete scenarios [BC08] to structural, behavioral, and technology decisions [Der+12]. Except for Zimmermann et al.'s work [Zim+08; Zim+07], the majority of the approaches lack empirical evaluation of the introduced decision making and documentation techniques in the industrial practice. Software Architecture Warehouse [NP13] seems also to have been validated in practice: the introduced tool has been used in various design workshops with students, however, only the experience of these workshops is briefly reported, without any quantitative or qualitative evaluation of the results.

Let us have a closer look at the QAs documented in the related works. In Table 7.5 we summarize the QAs reported per publication, only for the publications that refer to one or more concrete QAs. The majority of the papers reference at least three QAs, either in the evaluation part or in the demonstration of the corresponding approach, that are related to specific ADDs at decision making and/or documentation. Only in six publications no QAs are reported explicitly. Some QAs have been reported more often than others. For instance, performance (13x), security (10x), maintainability (8x), cost (7x), reliability (7x), and usability (6x) are referenced six or more times while other QAs like efficiency, accuracy, complexity, and availability are reported only once. Furthermore, other QAs like modifiability (4x), portability (3x), and scalability (3x) are also relevant but not key decision drivers for making the early design decisions during QA evaluation.

Regarding RQ3 we can conclude that the maturity of QAs in ADD tools and methods is rather low with regard to the evaluation in terms of evaluation methods used, design space size and scope, and number of QAs that are systematically studied. Although, ADDs are discussed along with their

QAs we could not gather much evidence about, e.g., the effectiveness, usefulness, and efficiency of the collected approaches with regard to the QA-related topics (e.g., interdependencies of QAs, QA uncertainty, etc.) we examine in this work.

Publication	**QAs Reported**
Zimmermann et al. [Zim+08]	Maintainability, Performance, Scalability, Interoperability, Cost
Jansen et al. [Jan+07]	Reusability, Maintainability, Completeness
Cui et al. [Cui+08]	Performance (real-time, non-overload), Maintainability
Tyree and Akerman [TA05]	Cost, Maintainability, Performance, Reusability, Capacity, Reliability, Modifiability, Security, Agility
Lytra et al. [Lyt+13]	N/A
Zalewski et al. [Zal+11]	Migrability, Integrability
Makki et al. [Mak+08]	Modifiability, Cost
Boer et al. [Boe+09]	Security, Usability, Changeability
Babar and Gorton [BG07]	N/A
Zimmermann et al. [Zim+07]	Interoperability, Performance
Shahin et al. [Sha+10]	N/A
Capilla et al. [Cap+07]	Maintainability, Evolvability (there are more QAs but not indicated)
Kruchten et al. [Kru+06]	Usability, Security
Harrison et al. [Har+07]	N/A
Zdun [Zdu07]	Performance, Reliability, Efficiency, Modifiability, Understandability
MacLean et al. [Mac+91]	Cost, Performance, Usability, Security
Babar and Capilla [BC08]	Latency, Accuracy, Capacity, Maintainability, Security
Lytra and Zdun [LZ13]	Reliability, Performance
Konersmann et al. [Kon+13]	Performance
Choi et al. [Cho+06]	Performance, Concurrency, Fault tolerance
Al-Naeem et al. [AN+05]	Modifiability, Scalability, Performance, Cost, Portability, Reliability, Complexity, Security, Usability, Effort
Heesch et al. [Hee+12a]	Extendability, Reliability, Scalability, Security, Portability, Usability, Cost
Harrison and Avgeriou [HA07]	N/A
Ameller and Franch [AF14]	Performance, Maintainability, Security, Reliability
Nowak and Pautasso [NP13]	Security, Flexibility, Reliability
Tibermacine et al. [Tib+06]	Maintainability, Portability, Availability, Performance
Dermeval et al. [Der+12]	Usability, Performance, Cost, Security

TABLE 7.5: Reported QAs per publication

7.7 Discussion

The findings from this literature survey have revealed an increasing interest over the past 10 years on the influence of QAs in ADD methods and tools. Most of the examined papers investigate the role of QAs in architectural decision making approaches and how design decisions are used to select better or the optimal QAs during architecture evaluation and design. The results also revealed that in the majority of cases, QAs are documented or used in a decision making process. Other findings show that a large majority of studies (63%) describe fully explicit relationships between QAs and ADDs which highlights the importance of such trace links for documentation and knowledge capturing and use purposes. With respect to the evaluation of QAs based on design decisions and

vice versa, a significant set of studies (26%) do not indicate or support such an evaluation. This fact leads us to assume the difficulty of carrying out such an evaluation activity. The QA-related topics used in existing ADD methods show that only the 56% of the research works examined deal with QA trade-offs, which matches with a similar number of works using decisions to select the most appropriate QAs or those that use the QAs to make better decisions. However, only 11% and 19% of the research works address QA uncertainty and QA interdependencies respectively, an indication that this research areas are still immature or lack enough interest from researchers. Another interesting aspect we can mention is the automation level of approaches using QAs: the results reported in half of the approaches show that ten of them are "manual" and only five use semi-automatic processes to calculate the priority of the QAs for decision making and trade-off evaluation.

The findings from this literature survey have revealed that the status of AK methods and tools using QAs is still a green field and challenging for researchers and practitioners. The evidence and results shown reflect the importance for companies to evaluate the quality of their systems and architectures for making better decisions. We also expect that empirical evaluation and assessment will play a vital role in a more rigorous systematic evaluation process and where the interplay between ADDs and QAs should facilitate the decision making and trade-off evaluation processes during the development phases.

7.7.1 Implications for Researchers and Practitioners

With respect to the publication channels, the reviewed primary studies are found in software architecture conference proceedings. The implication of this finding for researchers is that limiting the scope of search of primary studies to a short list of well-known publication channels is enough to derive important results. About the paper selection process (see Section 7.2), we paid attention to those papers excluded and to the potential impact of the number of citations reported in Google Scholar in order not to miss highly influential papers in the area.

The results also reveal that industry practitioners provided a certain evidence for successful use of ADDs entangled with QAs in specific domains or application cases (e.g., SOA decisions, control systems, smart systems, financial IT systems) where real case studies support this evidence, from high-level business decisions to lower-level technical decisions. The number of decisions examined in relationship with QAs vary from big sets of 300 decisions to smaller sets of e.g., 10 decisions. Therefore, the size and scope of the decisions analyzed is of key importance for the evaluation of the effort of the knowledge captured and the number of trace links to other software artifacts that have to be maintained. A lack of industrial evaluation of the reported approaches is an important issue for practitioners that want to enhance the processes to confront decisions in QA evaluation processes.

Our analysis has also revealed that a large majority of the studies (85%) report evaluation of the presented approaches using a few industrial cases and in most of the cases scientifically less rigorous evaluation approaches such as "non-industrial case studies", "example applications", and "motivating examples". This finding reflects the importance for researchers and industry software engineers of real examples, where significant sets of decisions are made, based on stringent quality requirements. However, the size of the decision sets in some of these industrial cases is sometimes low, which does not necessarily diminish the importance of the evaluation approach adopted, as such decisions may belong to a subset of the architecture or system under study. Finally, one interesting finding for academics and professional software engineers is the result based on the frequency of the appearance of the QAs detected in AK management approaches. From the 29 QAs identified only six (21%) show frequencies six or more, while only six (21%) vary in a range between two and four times. The rest of identified QAs appear only once. These results show that companies or the analyzed case studies tend to focus on a reduced set of QAs which seem to be more important for the scope of the analyzed applications than others. Just as an example, performance, security, maintainability, cost, usability, and reliability exhibit higher frequencies of appearance than the rest.

7.7.2 Challenges and Future Trends

While the findings of this literature survey have several implications for software architects, researchers, and practitioners, they also identify promising areas where the software engineering community can be encouraged to collaborate to improve the state of practice more rigorously. However, there are still a number of factors that challenge a broader adoption of ADD methods using QAs. While the knowledge capturing problem and the relationships to other software artifacts like requirements seems solved, the majority of the examined approaches can not provide more automatic procedures to evaluate QAs using decisions and make decisions for the selection of the required QAs. Only some prioritization mechanisms for QAs and alternative decisions, for instance, can be partially automated. Group decision making is another challenging area for knowledge sharing and collaborative approaches where decisions are not made by one single person, as group decision making may also influence certain software development approaches like agile models. In addition, fuzzy decisions, often made under a certain degree of uncertainty, become challenging and constitute another green field where more evaluation methods (sometimes from other disciplines) are needed. Finally, future research should provide better support and tools for QA evaluation for making better decisions and assist software architects in the QA trade-off process and when decisions can or should be made under uncertainty. Having methods and tools supporting sustainable decisions is another future research field where the longevity of the decisions is influenced by the quality of previous decisions that tend to endure over time.

7.7.3 Limitations of the Literature Survey

Unfortunately, QAs are not the main focus of many of the primary studies we have investigated. That means, in many cases we needed to interpret the implicit use of QAs in the ADD related approaches, and thus, some of the results reflect our understanding given the missing or blurry specification. In addition, we often needed to read between the lines in order to evaluate the works against our research questions. It is also noticeable that very different definitions of QAs (such as [Bac+05]) or even terms for QAs [CP09] exist in the literature. We tried, however, to establish a common understanding of the topics we studied with the three RQs. In addition, many of the tools we discuss here are not available online. Therefore, we needed to base our findings only on the reported results in the publications under study. Our study is also less systematic than systematic literature reviews in the sense that we did not follow a research strategy that performs an exhaustive literature search with precisely predefined steps [KC07]. We rather opted for a literature survey instead, as we wanted mainly to focus on methods and tools for AK management and the related tools and methods supporting AK management cover the majority of the spectrum of AK management research. Hence, we believe that it is likely that our literature survey covers the relevant literature to a large extent.

7.8 Conclusions

To summarize this literature survey, our findings are that QAs, though not the main focus of many of the approaches and respective studies in the literature, are supported by many of the decision making and documentation approaches. In many cases there is explicit support for relationships between ADDs and QAs. We found evidence for three major QA-related topics in the primary studies on ADDs: support for QA trade-offs is provided in more than the half of the primary studies, while QA interdependencies and uncertainty are more seldom supported. Finally, the maturity of the approaches with regard to evaluation is rather low. First of all, real-life case studies and empirical studies are only used in rare cases, none with an explicit focus on the QA support in the approaches. Secondly, both the design space sizes and the number of evaluated QAs are usually low. Hence, in addition to more research on the future trends we pointed out above, such as in the direction of automation support or group decision making, more empirical evaluations of the approaches and explicit focus on the QAs is needed to improve the state-of-the-art.

8 Quality-driven Decision Support with Reusable Architectural Decision Models

8.1 Introduction

The approach we introduce in this chapter was motivated by and applied in an industrial case study on a large-scale software ecosystem for smart cities, containing numerous scenarios in which the consideration of QAs is required to model reusable AK adequately. In the context of software ecosystems, the design choices the software architects make have been studied with respect to their quality attributes [Jan13]. Some important quality properties of software ecosystems, like portability, openness, and scalability (see [Jan13; Amo+14]) as well as design decisions [CP14] have been investigated for the ecosystem architecture as a whole. That means, that these existing works have documented design decisions and QAs in a form of best practices, but have not introduced guidance for concrete (reusable or not) ADDs. Also, the concept of reusable decision models, although it has been used in other domains (e.g., SOA solutions [Zim+07]), has never been considered in the context of software ecosystems.

Quality-driven decision making support is especially important in a software ecosystem context, as ecosystems focus on a set of businesses functioning as a unit and interacting with a shared market for software and services, together with relationships among them [MS03]. Consequently, not only a single application, but many possible applications based on a common platform must be considered when making ecosystem decisions. In addition, for ecosystem decisions not only a single main development organization but many interacting players in the ecosystem must be taken into account. To address these problems and raise the quality of recurring decisions in the ecosystem context, we propose to use a reusable architectural decision making approach. The goal is to base the ecosystem decisions on established AK, such as existing software patterns [Gam+94; Bus+00] or other well-documented AK, in order to address the broad nature of ecosystem decisions and support reuse of knowledge. We further propose, as our main contribution, to integrate reusable ADDs with QAs to enable quality-driven decision making support and provide corresponding tool support (based on CoCoADvISE). In addition to the approach itself, we present its application in the smart cities ecosystem case study [Sea99] and discuss the lessons learned from that case.

8.2 Smart Cities Ecosystem Case Study

We motivate our approach by introducing a case study on software ecosystems in the smart cities domain. Software ecosystems are defined as a set of businesses functioning as a unit and interacting with a shared market for software and services, together with relationships among them [MS03]. The relationships are mainly realized through the exchange of information, resources, and artifacts. Software ecosystems entail multiple product developments relying on a common architecture platform. A software ecosystem allows a company to make a platform available to other parties outside its organizational boundary. This is where the transition from software product lines to software ecosystems occurs, allowing third parties to realize functionality and customization of software applications [Bos09].

A smart city is an *instrumented*, *interconnected*, and *intelligent* city [Har+10]. It is *instrumented* because it provides capturing of data from various sensors (e.g., for temperature, voltage, etc.), devices, like cameras, smart phones, and medical devices, as well as the Web. In many cases, data, e.g., from electricity, gas, water, cooling, and telecommunication networks, need to be monitored periodically. *Interconnected* refers to the integration of such heterogeneous data in an enterprise platform in order to provide information to smart city services for various stakeholders (e.g., operators, citizens, etc.). *Intelligent* means, in this context, the use of analytics, reports, visualizations, and so on, to support operational decisions.

Our case study concerns a system that is being developed at Siemens with the purpose to provide a platform for a large-scale software ecosystem for various smart cities projects with partner organizations inside and outside of Siemens. Recently, much interest has been given by both the academia and industry on developing smart city intelligence systems as software ecosystems (see, for instance, the iCity Platform[1]). In such ecosystems, the software architects are faced with many design options which have to be considered along with the QAs that are used as decision drivers during architectural decision making.

From an industry point of view, one motivation for the smart city movement is the need to run urban infrastructures such as the electrical grid or water distribution systems more efficiently, effectively, cleaner and more secure. Siemens has various initiatives in the context of smart cities. To name two initiatives, the sustainable cities program developing the city intelligence platform and the smart city living lab in Vienna, Seestadt Aspern, have both their headquarters in Vienna, Austria. At the heart of each of those two initiatives, smart city ICT platforms enable the realization of various novel applications. The idea of these platforms is to provide a software ecosystem for smart city applications (so-called apps). Apps can be implemented by Siemens internal developers or by external partners using Web APIs. From an architecture point of view, the platform layers (see Figure 8.1) include:

[1] http://www.icityproject.eu/

- *Data Integration.* Extract Transform Load (ETL) is performed to obtain data from external systems and to load data into an integrated data schema. External systems include meter data management, water asset management, and building energy monitoring and controlling. These systems deliver data in various formats and demand for flexible and robust integration techniques. Data integration is important to get a complete view of the problem context.

- *Messaging.* The messaging layer enables the integration of real-time data. Whereas ETL performs batch import of data in mostly fixed time intervals, the messaging layer follows an event driven approach following a publish-subscriber mechanism. The data integration choice (ETL vs message driven) depends on the capabilities and interfaces of the external system and the application requirements.

- *Data Storage.* Within the database/data-warehouse layer, data is stored in a persistent manner for querying and analysis. Both, SQL-based and NoSQL database technologies are supported. SQL is the preferred choice for structured data when ACID properties are critical. The typical application areas for NoSQL are unstructured data or time series data with relaxed consistency constraints.

- *Business Logic.* On top of the data storage layer, a business logic layer provides APIs to access data structures. In addition, an analytics runtime and modules provide capabilities to implement data mining algorithms. Algorithms must be efficient in computing results.

- *Services.* Web APIs provide access to data and functionality in a standardized manner (RESTful services with JSON/XML data support). Web APIs help to establish a software ecosystem based on data and services wherein a community of developers can implement new smart city applications.

- *GUI.* Web dashboards provide the means for visualization and presentation of results to a variety of users. A dashboard is configurable to provide visualizations depending on the information need of a user. Web dashboards are not restricted to a specific platform and can therefore be viewed on large or small screen devices by following the responsive design paradigm.

The platform offers various points for extensibility in each layer so that new applications and new smart city deployments can be realized. The different design strategies and tactics are guided by both functional requirements and QAs. The QAs depend on the specific applications to be realized and on the various smart city stakeholders. Some possible extension scenarios and their potential impact on QAs are:

- New ETL jobs can be implemented and automated to load data into the platform. Data quality, data completeness, and reliability need to be considered when designing new jobs.

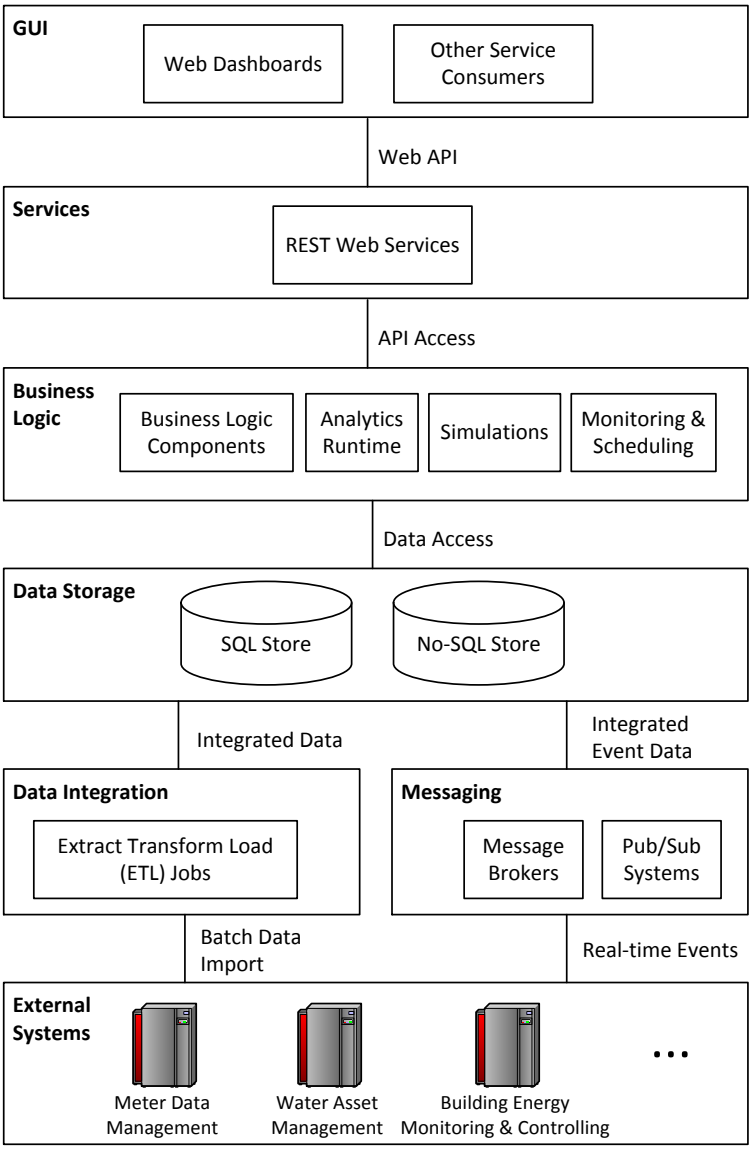


FIGURE 8.1: Smart city case study

- Subscriptions to event driven data sources can be added within the messaging layer. Here performance and scalability play an important role.
- The data model can be extended by new entities to accommodate new data sources. Maintainability of the overall data model needs to be ensured.
- New APIs in the business layer and new data mining algorithms can be added. Maintainability of the APIs is important, as well as performance of the data mining algorithms.
- New Web APIs and services can be implemented. Here, security, performance, and reliability are important.
- Configuration of new dashboards can be performed by the dashboard users. Thus, usability and also privacy issues play a role.

The question of which QAs are important and their priorities depend on the needs of the stakeholders. Various stakeholders are involved in different parts, activities, and processes of a smart city ecosystem. For instance, *Data Providers* provide data stores of information that is being monitored (i.e., organization, structuring, and delivery of information), *Facility Operators* are the main users of the smart city system that processes inputs from heterogeneous sources (e.g., smart grid networks, mobile devices, etc.), and *Application Developers* are responsible for the visualization, analysis, and generation of reports summarizing information from different resources for which *Smart City Stakeholders* and *Smart Citizens* are the main users. According to their needs, the various stakeholders will access data at different times, from different data sources, and using different interfaces or protocols.

In addition, policies, security issues, and business rules must be considered during data access and processing. For instance, *Facility Operators* may have JDBC access directly to the database(s), while internal *Application Developers* are allowed to use Object-Relational Mapping (ORM) APIs, and external *Application Developers* have access through a REST API to the corresponding applications. These three technology options, however, impose quite different performance characteristics on the applications that use them and the analyses they are able to perform. The *Facility Operators* may also have access to building control systems (e.g., heating, air-conditioning) using BACnet, a data communication protocol for building automation. Sensitive data should not be provided to any of the stakeholders unless it is anonymized. For the *Smart City Stakeholders* and *Smart Citizens* only aggregated data is provided based on privacy regulations. While some data is publicly available (such as traffic information or aggregated information offered by the city), encryption is required for personal data such as a citizen's energy consumption.

As a result, a software architect is typically confronted with a set of complex design decisions on how to realize application requirements without violating QAs. This is a nontrivial task given the multitude of extension points, the number of different stakeholders, and the range of QAs to be considered. Many QA trade-offs with numerous interdependencies must be considered for virtually every ADD that is being made in the context of the smart city ecosystem. This would already be a severe challenge during normal application development. However, in the software ecosystem context, it is significantly more problematic, as the software ecosystem decisions consider not only a single application but a whole range of applications and not only a single main development organization but the many players in the ecosystem. Our main contributions are to propose reusable architectural decision making support in this context, in order to base the ecosystem decisions on established AK, such as existing software patterns or other well-documented knowledge. This enables us to tackle the broad nature of ecosystem decisions and support reuse of AK. We further propose to integrate reusable ADDs with QAs in order to provide quality-driven decision making support that is based on established AK for smart cities software ecosystems design.

8.3 Extension of ADvISE/CoCoADvISE Meta-model to Support Quality Attributes

As we discussed in Section 4.2, the ADvISE/CoCoADvISE meta-model is inspired by the Questions, Options, and Criteria (QOC) approach [Mac+91] and additionally includes among others dependencies between options, options and decisions, and options and questions. To support, additionally, QAs during decision making, we have extended the underlying meta-model to include relationships between design solutions and QAs, as well as interrelationships among QAs. In Figure 8.2, the new meta-model elements are indicated with black color, while the existing ones (see also 4.1) are grayed-out.

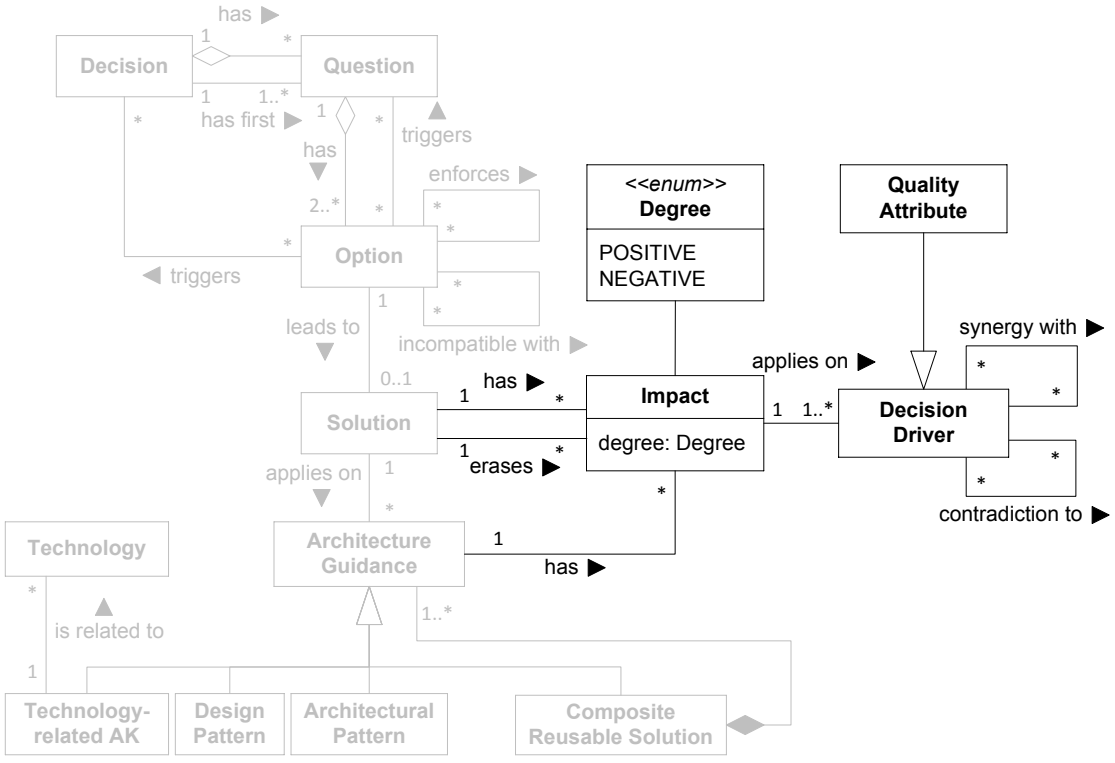


FIGURE 8.2: Extension of reusable architectural decision meta-model

The extended meta-model contains explicit links from Solutions to Decision Drivers, an abstraction for Quality Attributes. Thus, a Solution can have or erase an Impact (e.g., positive or negative impact) on a Decision Driver. The same relationship applies for an Architecture Guidance. The enumeration Degree can be extended to include more levels of impact (e.g., very positive, very negative). A Decision Driver can finally affect other Decision Drivers positively (be in synergy with) or negatively (be in contradiction with).

Let us consider an excerpt of a simplistic reusable decision model for data management (see Figure 8.3), consisting of one Decision (i.e., "Data Extraction"), related to one Question (i.e., "How often will you extract data from device?"), providing three Options: (a) once, (b) based on events,

and (c) based on schedule. The selection of each option leads to a different solution, that is, if we need to extract data only once or periodically we may use an ETL job, and consider the polling mechanism if we extract data based on a schedule; if data is extracted from a device once it is available, we will implement publish-subscriber and need to subscribe for new events at the corresponding device. ETL jobs are rather complex to program and introduce maintenance costs, especially when business rules change over time and data quality varies. One of the advantages of publish-subscriber is that it offers high scalability when the number of subscribers increases. Thus, modeling these relationships means connecting the two solutions "subscribe for events" and "use ETL job (polling)" to the QAs "Maintainability" and "Scalability" with NEGATIVE and POSITIVE impact respectively.

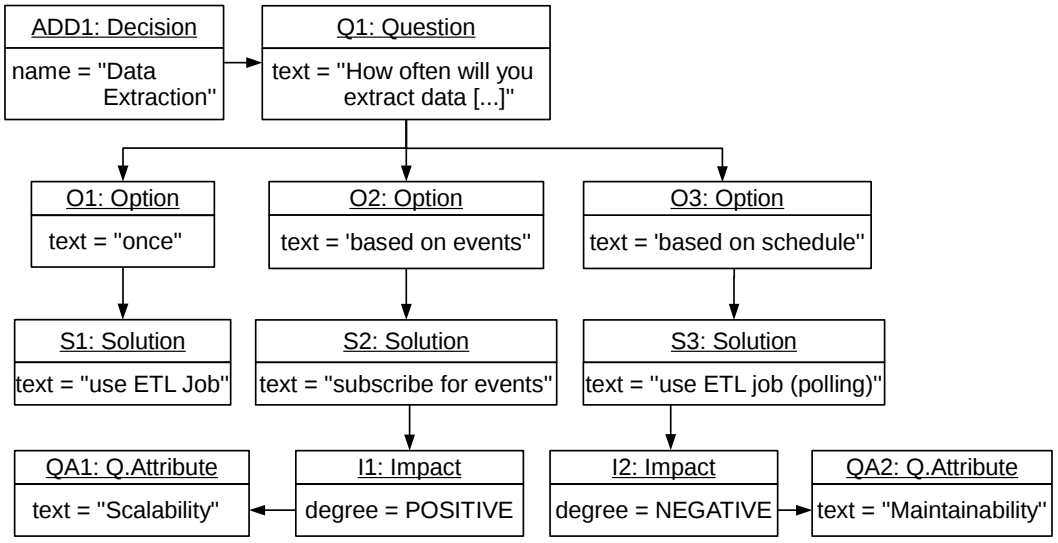


FIGURE 8.3: Exemplary architectural decision model

From such a reusable decision model, software architects can instantiate questionnaires many times in similar design situations. By selecting options the recommended solutions are indicated for a concrete decision as soon as they are applicable. For instance, the selection of the third option of the question of Figure 8.4 will reveal the recommended solution "use ETL job (polling)". In addition, other decisions, questions and options are activated (shown) when they are valid and deactivated (hidden) when they are not applicable (according to some predefined constraints and dependencies between them).

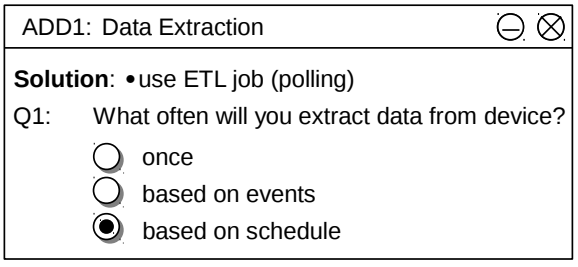


FIGURE 8.4: Exemplary questionnaire

At the same time, the QAs of interest are evaluated according to the recommended solutions and based on predefined relationships between the solutions, design patterns, etc. and the QAs (such as the ones discussed before). Plus (+) indicates positive effect, while minus (-) indicates negative effect of a specific decision on the QA. Thus, software architects receive guidance with respect to two concerns: (a) how are the QAs affected by their (recommended) design solutions and (b) what is the rationale behind positively or negatively evaluated QAs. The first is supported in the tool by the visualization of the impact of the design options on the QAs with plus (+) and minus (-) "votes" while the second is achieved by providing related tooltip information (e.g., *Decision "use ETL job" for ADD1: Data extraction affects Maintainability negatively*).

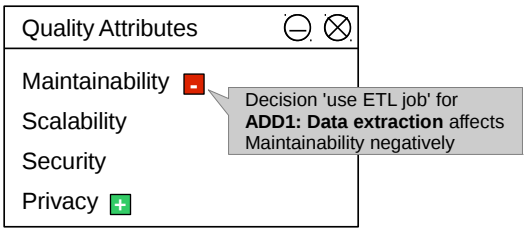


FIGURE 8.5: QAs evaluated against design solutions

8.4 Case Study

In this section, we elaborate on using our approach in the context of the case study from the smart cities domain. In particular, we demonstrate how to model data management related ADDs for the design of software ecosystems for smart cities in a reusable decision model. Afterwards, we model the relationships between the various design solutions and QAs, that is, for a set of QAs of interest we investigate and capture the positive or negative effect of the corresponding ADDs on them. Finally, we use the reusable model and the CoCoADvISE tool support to make and document concrete decisions for the Aspern Smart City Project[2].

The steps we performed in the context of this case study are illustrated in Figure 8.6. The tasks we performed in each step are indicated with white color while the outcomes of each task are indicated with gray color. Two researchers and two domain experts were involved in the design of the case study, in three phases/steps and multiple refinements for each phase.

First of all, a set of use cases with respect to data management in software ecosystems for smart cities were collected and analyzed. Based on these use cases, we defined related decision points, options, and design alternatives, as well as related reusable AK (e.g., design patterns, technology-related solutions, etc.) and organized them in categories of ADDs that need to be made in the context of data management. Afterwards, this information was organized in a reusable decision model according to the CoCoADvISE meta-model.

[2] http://www.ascr.at/

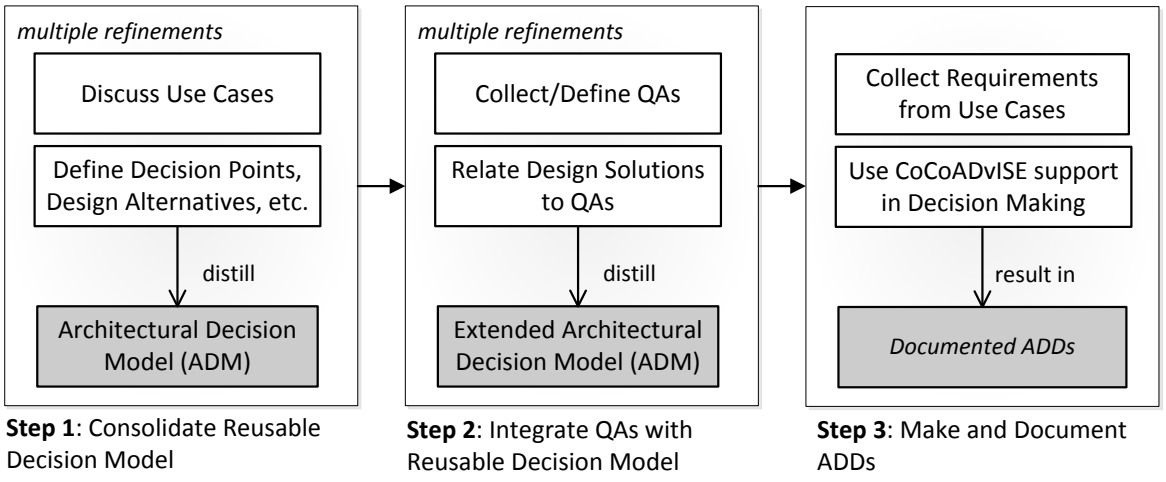


FIGURE 8.6: Steps performed in the case study design

Table 8.1 summarizes an excerpt of the reusable decision model consolidated in the first step of the case study design, consisting of data management related decisions, divided in six categories (*Data Extraction*, *Data Processing*, *Data Routing*, *Data Storage*, *Data Presentation*, and *Data Privacy*). For simplification reasons, we do not document the relationships between options and decisions, nor do we include details about the related AK artifacts (e.g., design patterns).

Category	Question	Options	Design Solutions
Data Extraction	How often will you extract data from device?	• Once • Based on events • Based on schedule	→ Use ETL Job → Subscribe to device for events (publish-subscriber) → Use ETL Job (with polling)
	Do you need to transform unstructured data to structured data?	• No • Yes	→ Consider *Data Storage* category (processing or routing not possible) → Consider *Data Processing* category
	Will the extracted data need processing before it will be loaded in the application?	• No • Yes	→ - → Consider *Data Processing* category
	Will the extracted data need routing/filtering before it will be loaded in the application?	• No • Yes	→ - → Consider *Data Routing* category
Data Processing	Do you need to translate extracted data (from different sources) in a common format?	• No • Yes	- → Use a normalizer/canonical model to define a common format
	Do the extracted data contain multiple elements to be processed independently?	• No • Yes	- → Extract elements using a splitter
	Do you need to aggregate multiple data that share a common format?	• No • Yes	- → Aggregate elements using an aggregator

Continued on next page

Table 8.1 – continued from previous page

Category	Question	Options	Design Solutions
	Do you need to attach extra information to extracted/processed data?	• No • Yes	- → Use a content enricher
	Do you need to remove information from data or simplify the structure of data?	• No • Yes	- → Use a content filter
	Does any other data processing take place?	• No • Yes • Yes - time-consuming algorithms	→ - → Prefer online algorithms → Prefer offline algorithms
Data Routing	Will the data be sent to one or multiple receivers?	• One • Multiple	→ Use point-to-point connection → Use publish-subscriber
	How will the routing of that data be decided?	• Based on the content • Based on published topics • Recipient-based	→ Use content-based routing → Use topic-based routing → Use a recipient list
Data Storage	How would you characterize the size of data that needs to be stored?	• Small/medium • Big	→ Scaling not required → Use vertical or horizontal scaling
	How long will the data be available after storage/persistence?	• Forever • For a long period of time • For a short period of time	→ Never delete data → Delete periodically/schedule delete job → Cache data/delete data after a timeout
	Shall old data be archived?	• No • Yes	- → Perform manual/automatic archiving
	How to decide which data will be deleted?	• FIFO • Based on priority	→ Ring Buffer → Priority buffer
	Where will the data be persisted?	• File • Database	→ Store in file system → Use a database
	What kind of database will be used for the data storage?	• Relational • Non-relational	→ Use SQL database → Use NoSQL database
Data Presentation	How will the data be visualized for the end user?	• Standard UI • Configurable	→ Non-customizable UI → Use a configurable dashboard
	Do you need to load at the client-side UI big amounts of data?	• No • Yes	→ - → Use batch requests
Data Privacy	Will you need to anonymize data?	• No • Yes	→ - → Use data filter + aggregator
	Will you need to encrypt data?	• No • Yes	→ - → Use public key encryption

TABLE 8.1: Excerpt of reusable architectural decision model for data management in software ecosystems

In the second step of the case study, we collected QAs that are considered according to the domain experts key QAs for the design of the smart city ecosystem. For instance, *Scalability* is important in this context as distributed systems need to be accessed and big amounts of data need to be managed at real-time. In addition, *Privacy* and *Security* are key concerns whenever sensitive personalized data is being requested. Other QAs like *Performance*, *Reliability*, and *Maintainability* strongly influence the architects' ADDs for the smart cities software ecosystem as well. The final list of considered QAs consists of 11 QAs: Availability, Data Completeness, Data Quality, Extensibility, Maintainability, Performance, Privacy, Reliability, Security, Scalability, Usability.

In architectural decision making, we often need to deal with competing requirements and therefore, these QA interdependencies also need to be considered when making trade-offs [EG04]; for instance, security comes usually at costs of usability. Using the CoCoADvISE meta-model we expressed such relationships between QAs that occurred in the decisions, i.e., a QA is in *synergy with* or in *contradiction with* another QA. Once a constraint is in place, the tool automatically checks it during decision making.

After eliciting a list of QAs in the context of our case study, we investigated whether and to which extent the design solutions of the reusable decision model (see the excerpt of Table 8.1) affect these QAs. Of course, not all QAs are relevant for every design solution and not all design solutions are related to one of the aforementioned QAs. The outcome of this step was a list of design solutions along with their positive or negative impact on the QAs. A number of prominent examples of such impacts in our reusable decision model for data management are shown in Table 8.2.

Design Solution	*Impact on QA*	
	positive	negative
Configurable dashboard	Usability	-
Batch requests	Usability, Performance	-
Vertical/horizontal scaling	Scalability	-
Offline algorithms	Performance	-
Anonymization	Security, Privacy	Performance
Cache data	Performance	-
Publish-subscriber	-	Security

TABLE 8.2: Examples of impacts of design solutions on QAs

The last step of our case study was to document ADDs for the use case scenarios collected in the first step as instances of the reusable architectural decision model for data management. That is, we tested our model by designing a number of app-level architectures.

For making and documenting ADDs, the CoCoADvISE tooling provided quality-driven decision support. Also, it was used to generate semi-complete ADD documentations for the final decisions.

8.5 Discussion

An important lesson learned of our work is that a systematic approach to deal with the software architecture and architectural qualities is specifically important in the context of a large-scale industrial software ecosystem. As many decisions are taken over and over again for multiple design situations in different applications, we decided to use reusable decisions as a basis for our systematic approach. As this approach, unfortunately, had not yet been integrated with QAs, we proposed in this work – to the best of our knowledge – the first systematic approach for integrating reusable architectural decision making and QAs.

Basing decision for multiple applications – whose requirements are often unknown at design time of the ecosystem platform – only on past experiences is not enough, but knowledge reuse from both internal experiences and external sources is needed. External sources are mainly the literature and the Web, including design and architecture patterns with their quality impacts documented as pattern consequences or other discussions of established design or architecture solutions with clear statements on the impacts on QAs. In some cases, we needed to study the solutions, e.g., empirically (like measuring performance and memory consumption of different solutions), in order to estimate the QA impacts of proposed solutions.

Patterns and similar kinds of AK introduce generic knowledge. The generic knowledge usually must be tailored to the domain in which the pattern is applied. In the ecosystem context, we usually already know much about the domain. For this reason, many of our generic decisions have a tailored counterpart in the wording of the domain or sometimes even with small changes in the model where it is known that the domain uses solutions slightly different to the AK documented in the literature. This makes it significantly easier for domain experts to use our approach.

From an architecture point of view, one of the biggest challenges in a smart city software ecosystem is the realization of new applications and extensions of the platform given the wide variety of design choices and available design patterns. Each choice may have different implications with respect to QAs. Without a systematic decision process as presented in this work, software architects need strong guidance by senior software architects to make the right decisions. Here, the reusable decision process helps to solve this problem.

Another challenge for software architects is the deep domain knowledge needed to realize smart city applications. Each domain may have its own specific protocols, standards, data formats, etc. that need to be considered when designing an application. The advantage of the presented approach is that guidance can be given, which is tailored to a specific domain. Individual steps in the overall decision model can be customized for specific domains (e.g., using a particular data format in the context of a smart grid or smart building domain). Thus, software architectures are guided through domain specific design decisions and do not need to seek help and advice from domain experts.

Finally, the reusable decision model can evolve as the body of AK evolves. Within a smart city software ecosystem, many different actors from different domains contribute new data, new algorithms, and new applications. Lessons learned on how to solve a given architectural problem or the most efficient way to solve an algorithmic problem can be added to the model. Thus, the model and the knowledge are driven by a community instead of a single architecture specification that is updated from time to time.

In addition, our approach has only been tested in the context of one large-scale ecosystem in the smart cities domain. However, we believe it can be generalized to many different contexts. While it is likely to be of minor importance in comparison to non-ecosystem contexts, we do not see any reason why the approach might not be applicable in other development scenarios where a systematic decision approach is needed and QAs are to be considered. In general, using our approach requires a certain system complexity. For very simplistic domains or small-scale systems, too much upfront investment might be required. The extra effort required for our approach is probably only justified from a certain system size and with a certain level of reuse. The break-even point for this would need to be determined in future research. Apart from this effort aspects, we consider our approach to be applicable also in other domains and for other system sizes.

The extra effort for our approach is mainly the upfront investment of (a) building a design space with quality annotations for the target domain and (b) tailoring it to the domain of the ecosystem. Both efforts are considerable, especially if detailed data on the effect on QAs are missing in the literature. But as the design space needs to be built only once and can then be reused, it was acceptable in our case study's context. Compared to other decision making approaches, using our reusable decision making approach for making decisions is minimal, as all reusable knowledge is automatically filled in. The generated decisions must just be updated with information that is specific to the current design situation. However, still a limited amount of overhead for documenting decisions remains, which is not accepted by all developers and is all sometimes forgotten under the daily pressure in typical development projects.

The various trade-offs of QAs that need to be made during the decision making process still remains a complex and challenging task. The subjectivity of quality concerns that can be interpreted differently for different stakeholders and in different contexts, the importance of some qualities that are evaluated above others, and so on, increase the complexity of decision making driven by QAs. More research on automation regarding QA-based decisions is needed, but this would require detailed data on how specific QAs are influenced by different reusable decision options. So far the documented knowledge in the literature is mainly anecdotal and informal.

8.6 Conclusions

In this work, we proposed to integrate QAs with reusable ADDs to support architectural decision making. For this, we introduced a reusable decision meta-model for modeling among others the impact of design solutions on QAs, as well as a tool prototype for supporting architectural decision making driven by QAs. We afterwards applied our proposal in a large scale software ecosystem, in particular, in the smart cities domain. That is, in the context of a case study, we modeled a reusable architectural decision model for decisions related to data management issues in smart cities software ecosystems along with the related QAs. Finally, domain experts used this model with the support of the CoCoADvISE tool to make and document ADDs for the Aspern Smart City Project. Our experiences show that reusable architectural decisions can help to systematize the decision making process in ecosystem contexts and that enriching them with QA support enables us to model many relevant QAs that must be considered during the decisions already at the time when the reusable decision model is created. That is, this way we can systematically ensure that App developers in the ecosystem consider all relevant QAs from the ecosystem architecture point of view. Still many challenges regarding quality-driven decision support based on reusable AK need to be addressed, as discussed in Section 8.5.

9 Resolving Architectural Design Decision Uncertainty using Fuzzy Logic

9.1 Introduction

Software architects are faced with recurring design problems in their everyday life. To design the right solutions they need to consider relevant patterns and pattern languages and understand how a design pattern will fit into the overall architecture. The patterns, usually written as informal texts, are subject to interpretation. In addition, in this decision making process, the software engineers must interpret the knowledge of domain experts and the requirements of the system, balance competing requirements – often vague and imprecise – and adapt the solutions to specific technologies and systems. All these factors introduce many sources of uncertainty for pattern-based design decision making. Existing approaches do not consider this inherent uncertainty of the pattern-based decision making process, which has been until now largely ad hoc and informal, without explicit, automated support.

To address this problem, we suggest a fuzzy logic based method for making pattern decisions under uncertainty. Fuzzy logic helps us to deal with the imprecision and ambiguity of the design pattern selection problem and to infer best-fitting solutions given the requirements. For this, we integrate software patterns and fuzzy logic by creating fuzzy models leveraging experts' knowledge, and provide a fuzzy inference system to make pattern decisions under uncertainty, considering patterns' forces and consequences. Along with the general fuzzy models we derive specialized fuzzy models for specific technologies and system contexts. These fuzzy models are described using a domain-specific language (DSL) and get stored in a repository. Thus, they provide reusable assets for pattern-based decision making.

We demonstrate our method for a subset of the design space for service-based platform integration we have already introduced in Section 5.6. This subset consists of four patterns and three related solutions for client asynchronous invocation design, in which we also created specialized fuzzy models for five technologies. To evaluate our method we infer the best-fitting patterns for given requirements in four case study scenarios from the Warehouse case study (see Section 5.8). We

also develop appropriate tooling to run and demonstrate our experiments. Finally, we measure the complexity and implementation effort of our DSL.

9.2 Background

9.2.1 Fuzzy Logic

Making decisions that contain uncertainty, such as the uncertainty in natural language, is not an easy task. To address this problem Zadeh introduced in 1965 Fuzzy Logic [Zad65]. Key concepts of Fuzzy Logic are fuzzy sets and their membership functions. Unlike classical sets fuzzy sets contain objects that satisfy imprecise properties of membership [Ros04]. In binary logic the membership function takes only two values: 0 and 1. Zadeh extended this binary membership to express various "degrees of membership" spanned in the interval $[0, 1]$. A membership function of a fuzzy set is a curve that defines how each point in the input space (the universe of discourse) is mapped to a membership value between 0 and 1. We use the representation $\mu_{\tilde{A}(x)}$ to express the degree of membership of element x in a fuzzy set $\tilde{A}$. Thus we can write:

$$\mu_{\tilde{A}(x)} = \text{degree to which} \quad x \in \tilde{A}, \quad \mu_{\tilde{A}(x)} \in [0, 1]$$

Fuzzy Logic allows the numerical encoding of the vague but rather simple linguistic values that humans use in their communication. These linguistic values can be interpreted using fuzzy sets which get mapped to overlapping membership functions. For example, the property *performance* could be described as *high*, *medium* or *low* which get mapped to overlapping membership functions, as shown in Figure 9.1. The membership functions can be triangular, trapetzoidal, gaussian, etc.

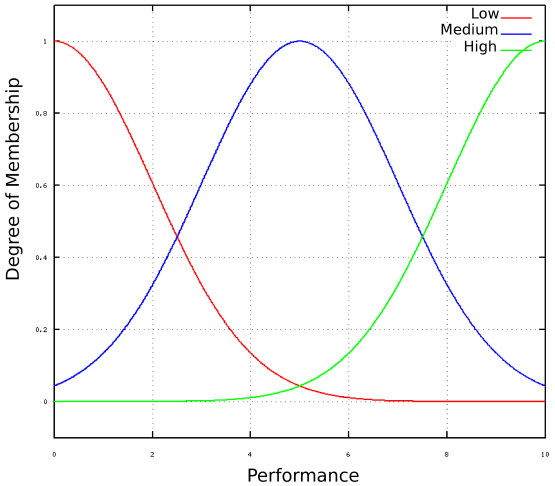


FIGURE 9.1: Gaussian membership functions for three linguistic values of property performance

9.2.2 Fuzzy Inference System

Fuzzy relations can be assembled from linguistic knowledge, expressed as IF–THEN rules. Fuzzy inference is the process of formulating the mapping from a given input to an output using fuzzy logic. Mamdani method for fuzzy inference [MA99] is the most suitable for capturing expert knowledge, as its rules allow us to describe the expertise in more intuitive and human-like manner. In this case, a fuzzy rule-based system contains simple canonical rules of the following form: "IF condition A, THEN conclusion B". The condition A can contain multiple antecedents in conjunctive or disjunctive form or a combination of both. The conclusion B is of the type "y IS $\tilde{C}$", where $\tilde{C}$ is a fuzzy value of the output variable y. The fuzzy inference process comprises the following steps: (1) fuzzify the input, (2) evaluate the fuzzy rules, (3) aggregate the outputs to reach the final decision, and (4) defuzzify the output to obtain a crisp value. In our work we integrate software patterns and fuzzy logic and apply a Mamdani-type inference method to make pattern decisions under uncertainty.

9.3 Approach Details

9.3.1 Overview

Figure 9.2 presents an overview of our approach, namely the participating tools and roles. We distinguish between two stakeholder roles: *software architect (expert)* and *software architect (user)*. That is, different levels of experience are expected for architects who create the fuzzy logic models and users of our approach.

The software architects (experts) use the *Fuzzy Decision Model Editor* (a textual DSL editor) to capture architectural knowledge. A decision model contains alternative design solutions along with their properties and quality attributes and a set of expert IF–THEN fuzzy rules that guide the design decisions. From these decision models we derive specialized fuzzy decision models in which domain and technology specific knowledge can be included. Both kinds of decision models get stored in a *Fuzzy Decision Model Repository* for reuse by a *Fuzzy Inference System*. The software architects (users) use the *Requirements Editor* to give the desired requirements in crisp values using a grading system (e.g., 1–10) for fuzzy input variables like *performance* and *reliability* and binary values (i.e., 0, 1) for variables that accommodate only two values (e.g., Yes, No) like *supports acknowledgment*. The *Fuzzy Inference System* returns the appropriate design alternatives and their ranking for the given requirements by evaluating and combining the fuzzy rules already defined in the fuzzy models. The list of the best-fitting design solutions is supposed to be used as a decision aid for the software architect who makes the final decision. After that, the input requirements and the inferred design solutions can be synthesized to produce *ADD Documentations*.

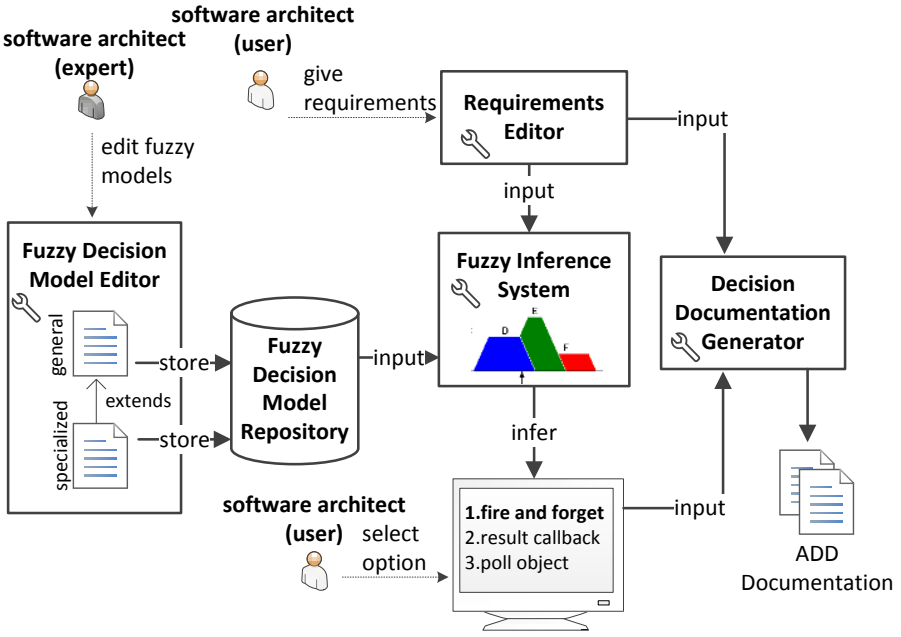


FIGURE 9.2: Fuzzy logic approach for supporting pattern-based design decisions

To create our textual DSL we used the Eclipse toolkit Xtext[1]. Along with our Eclipse tooling we used the open source Fuzzy Logic library jFuzzyLogic[2] which implements the Fuzzy Control Language (FCL) specification [Int00]. It also provides a DSL for specifying fuzzy models in FCL and a fuzzy inference system. FCL is a rather general language that implements fuzzy control. We chose, however, to focus on the domain of pattern selection and developed a simple expressive DSL to describe our fuzzy models intended to be easier to understand and less complex in this context.

9.3.2 Domain Specific Language for Fuzzy Logic Models

In the following sections, we explain how to define fuzzy pattern-based decision models using the DSL by giving some usage examples. The DSL allows to model alternative patterns for a design problem along with their forces and fuzzy rules. In particular the fuzzy rules express the relationship between forces and patterns and are of the form "if condition then pattern" where the condition comprises conjunctions and/or disjunctions of the form "force_name is force_value" or "force_name not force_value". The specification of the DSL for Fuzzy Logic Models is given in detail in Appendix C.

9.3.2.1 Fuzzy Pattern-Based Decision Models

The pattern documentations help us find the related patterns and define the forces and consequences of the patterns that are mapped to the fuzzy inputs. Patterns that refer to a specific

[1] http://www.eclipse.org/Xtext/
[2] http://jfuzzylogic.sourceforge.net

domain, problem or technology can be organized in groups. In the following example we list the alternative design patterns for asynchronous invocations:

```
group asynchronous_invocation
 Pattern fire_and_forget
 Pattern sync_with_server
 Pattern poll_object
 Pattern result_callback
 ...
```

Listing 1: Patterns for asynchronous invocations modeled in the DSL

Before writing the rules for the pattern selection we need to define the patterns' forces and consequences and their possible values – the fuzzy sets. Along with the fuzzy sets, we need to select their membership functions. For the quality attributes (e.g., *reliability*) we may choose triangular, gaussian, etc. membership functions. For properties that do not contain fuzziness (e.g., *supports acknowledgement* can be either *yes* or *no*) we use singleton membership functions. The patterns and pattern variants are the values of the fuzzy output variable *pattern* with a singleton membership function, as they do not contain any fuzziness as well. We define quality attributes with their linguistic values and membership functions as follows:

```
forces
  reliability gauss { low medium high }
  performance gauss { low medium high }
  ...
end
schemas
  gauss { gauss[1 2] gauss[5 2] gauss[10 2] }
  ...
end
```

Listing 2: Modeling of forces and consequences in the DSL

The precise shape of the membership curves is not so important, as their approximate placement on the universe of discourse, the number of curves used, and their overlapping character are the most important ideas [Ros04]. The overlapping of the curves helps us express the vague borders between the linguistic attribute values. But we should keep in mind that the type of the membership functions, the grade of their overlapping and the number of the variable values depend on the engineers' preferences, experience and intuition and affect the precision of the results. Therefore, the membership functions should be tuned manually (e.g., empirically by trial and error). Finally, the experts' experience gets translated into fuzzy rules as in the following example:

```
if performance is high and reliability is medium then poll_object
```

Listing 3: Example of a fuzzy rule

Weights can also be assigned to the rules according to their importance.

9.3.2.2 Technology-Specific Decision Models

As mentioned before, in order to include technology- and system-specific knowledge, pattern implementations extend the generic patterns and thus inherit their membership functions and rules:

```
group apache_cxf
import "generic.fuzzypattern"
 Pattern oneway ofType fire_and_forget
 Pattern transport_level_polling ofType poll_object
 ...
```

LISTING 4: Extend patterns in the fuzzy models

The generic rules will be combined with the specialized rules. Therefore, the generic pattern knowledge gets transferred to and combined with the technology-specific knowledge.

9.3.2.3 Fuzzy Inference System

A Mamdani-based fuzzy inference system [MA99] is used to infer best-fitting solutions for given requirements in crisp values (e.g., in a scale 1–10). In particular, the fuzzy inference system fuzzifies the input requirements, evaluates the fuzzy rules and aggregates the outputs, which produces an ordered list of possible design solutions with different weights. The outputs of the fuzzy inference system can be manipulated by changing the fuzzy models, namely the forces and consequences and their membership functions, the rules, or the weight of the rules. In addition, to adapt pattern-based knowledge to concrete system and technology contexts we derive specialized decision models by inheriting general fuzzy models. Pattern-based knowledge gets specialized by adding, prioritizing, or deleting choices, forces, consequences, and rules.

9.4 Design Space for Motivating Example

We use excerpts of the reusable decision model for service-based platform integration (see Appendix A) as the basis for constructing the generic fuzzy models for the alternative design patterns and design decisions. In particular, we read through the options and the decision drivers in order to define the fuzzy sets and the fuzzy rules of the fuzzy models. In the following sections, we will demonstrate the application of our approach for asynchronous invocation patterns (see Communication Style Design Decisions of Table A.3).

9.4.1 Communication Style Design Decisions

We consider the decision $D13$ of the architectural decision model for service-based platform integration (see Table A.3) to demonstrate our approach. The decision is presented in detail in Table 9.1. Most of the alternative design patterns come from the remoting pattern language in [Völ+05]. The remoting patterns describe the inner workings of distributed middleware systems (such as Web services, CORBA, Java RMI, and .NET Remoting) and explain how to use them to create distributed systems. For our demonstration we focus on the asynchronous remote invocation patterns. Unlike blocking synchronous invocations, asynchronous invocations allow client applications to resume their work while awaiting for a response to a remote invocation, thus improving scalability and performance. In our example we examine the following patterns: FIRE AND FORGET, SYNC WITH SERVER, POLL OBJECT, and RESULT CALLBACK. Ordinary synchronous invocations are an alternative to all client asynchronous patterns. A sixth alternative to handle asynchrony is to use a messaging infrastructure. Obviously, all six alternatives are mutually exclusive. Each of the solutions has different properties (acknowledgment required, result required, etc.) and different consequences for important QAs (reliability, performance, etc.) which should be considered and balanced during the decision making process. We created general fuzzy models which reflect these properties and QAs and analyzed five different technology implementations to capture application-specific knowledge in technology-specific fuzzy models.

Decision Context	Communication style between integrating services and integrating platform.
Options	• FIRE AND FORGET for one-way communication • SYNC WITH SERVER for acknowledgment only. • For result either RESULT CALLBACK or POLL OBJECT; Alternatively MESSAGING. • Synchronous invocations with/without result.
Decision Drivers	The FIRE AND FORGET pattern provides non-blocking communication with unreliable transmission. That means that the client is not notified of errors in transmission or execution of the remote object and the remote object does not deliver any execution results to the client. SYNC WITH SERVER ensures successful transmission of requests and makes the remote invocations more reliable than FIRE AND FORGET. It introduces, however, additional latency, as the client must block until the notification of successful reception is received. Thus, there is a trade-off between higher reliability and worse performance in comparison with FIRE AND FORGET. Also the server application can not inform the client for application errors as the execution of the remote invocation happens asynchronously. This pattern offers more reliable communication compared to FIRE AND FORGET, as the result is an implicit acknowledgment, but it can not immediately inform the client about an incoming result. RESULT CALLBACK is preferred over POLL OBJECT when an immediate reaction on the incoming result is needed. The same or different callback objects can be used for different invocations of the same type. MESSAGING offers high reliability, as sending a message does not require both systems to be up and running at the same time. Instead, message queues in the messaging system can temporarily store the messages when systems are down. [...]

TABLE 9.1: Exemplary reusable ADD (D13)

9.4.2 DSL-Based Specification of Design Space

The aforementioned design patterns and solutions provide the values of the output variable *Pattern* in a fuzzy decision model in our DSL. The central step to create a fuzzy decision model from the pattern-based decision model is to define the pattern forces and consequences (our fuzzy sets) and their membership functions. In our architectural decision model we have documented the following decision drivers that are used as decision criteria for the generic DSL models: *performance*, *reliability*, *acknowledgment* and *log application error*. We experimented with different numbers for fuzzy sets and membership functions in order to tune them and report our results. For writing the rules we consulted the pattern documentation as well as our experience. Listing 5 is an excerpt of the corresponding fuzzy model which captures generic decision knowledge on the asynchronous invocation patterns. Of course, what we present here reflects our interpretation of the design pattern documentation and our experience and can be modified to include more or less forces, different membership functions, different sets of rules, etc.

```
group asynchronous_invocation
Pattern fire_and_forget
Pattern poll_object
...

forces
  performance gauss { low medium high }
  acknowledgement yesno { no yes }
  ...
end

schemas
  gauss { gauss[0 2] gauss[5 2] gauss[10 2] }
  yesno { singleton[0] singleton[1] }
end

rules
  rule1 --> if performance is atmost high and reliability is low and acknowledgement is no and
     log_application_error is no then fire_and_forget
  rule2 --> if performance is atmost high and reliability is atmost medium and acknowledgement is yes and
     log_application_error is yes then poll_object
  rule3 --> if performance is atmost high and reliability is atmost medium and acknowledgement is yes and
     log_application_error is no then poll_object with 0.1
  ...
end
```

Listing 5: Excerpt of application-generic fuzzy model

This representation can be enough to capture application-generic explicit knowledge, but software engineers need more tangible technology-specific knowledge to apply these patterns to a problem at hand. To support this transition from application-generic to application-specific knowledge, which is called *Utilization* in the literature [FB09], we studied five existing frameworks and solutions

that allow us to embed the asynchronous invocation patterns in the application domain. Thus, we investigated all the pattern implementations as well as the specific decision criteria for the following frameworks: Apache CXF[3], Apache Axis 2[4], Metro 1.2[5], .NET[6] and CORBA[7]. The technology-specific DSL models import the generic DSL model, and the pattern implementations extend the generic design patterns. We present here an excerpt of a DSL model that captures knowledge about CORBA:

```
group corba
import "generic.fuzzypattern"

Pattern AMI_polling ofType poll_object : "AMI polling model"
Pattern AMI_callback ofType result_callback : "AMI callback model"
Pattern oneway_sync_none ofType fire_and_forget : "Reliable one-way SYNC_NONE"
Pattern oneway_sync_target ofType fire_and_forget : "Reliable one-way SYNC_WITH_TARGET"
Pattern oneway_sync_server ofType sync_with_server : "Reliable one-way SYNC_WITH_SERVER"

forces
  complexity gauss { low medium high }
  ...
end

rules
  rule1 --> if complexity is low then AMI_polling
  rule2 --> if complexity is atmost medium then AMI_callback
  ...
end
```

Listing 6: Excerpt of technology-specific fuzzy model

9.5 Evaluation in Case Study Scenarios

In this section, we elaborate some scenarios from the warehouse case study presented in Section 5.8, in order to infer the appropriate solutions given specific requirements. In particular, we consider the following scenarios:

1. **Book dock appointment** Shippers, carriers, and consignees are able to book dock door appointments remotely. The dock scheduling allows the YMS to manage the traffic of the trucks in and out of the yard sufficiently. The clients get a notification when their requests are received by the YMS.

[3] http://cxf.apache.org/
[4] http://axis.apache.org/
[5] http://metro.java.net/
[6] http://www.microsoft.com/net
[7] http://metro.java.net/

2. **Log trailer position** The trailers in the yard periodically send their position to the YMS using remote communication devices to allow the personnel have complete, real-time visibility of trailer locations and contents at all times. These position reports are captured in a logging system to keep track of all trailer locations and moves. Logging is also used to create reports in the future (for optimization of yard management). Loss of the log messages is acceptable.

3. **Register storage units for new products** If a new product is to be stored in the warehouse, it has to be registered first. Thus, when a new storage request arrives, the WMS will search for a suitable bin location. If there is no location available, the storage request will be rejected. If a suitable location can be found, it will be reserved and the transport to the reserved location using conveyors and lift modules is triggered.

4. **Start video stream** The security staff is able to control the incoming and outgoing personnel and vehicles using live video streaming. Also, they can monitor the flow of the goods for efficiency/safety reasons. When they start a new video stream the location of the cameras is updated and the video is streamed to their monitors. Timeouts are used to handle failures, e.g., when the camera stops streaming.

In Table 9.2 we present the appropriate solutions that our fuzzy logic based system inferred for all case studies and for both the generic and the technology-specific models (we show only the first six solutions in the ranking). The pattern implementations are given with their code names; for instance, *transport_level_polling* and *api_level_polling* (Apache Axis 2) refer to the implementations of the POLL OBJECT pattern using transport-level asynchrony or a non-blocking API at the client-side respectively. The patterns that were ranked first appear in bold letters. For the case studies the fuzzy logic system inferred more than one solutions for all technologies. For example, in the case that the FIRE AND FORGET pattern is the best-fitting solution, we are allowed, of course, to use the patterns POLL OBJECT, RESULT CALLBACK, etc. instead.

The fuzzy models as well as the requirements provided to the fuzzy inference system are just one possible solution guided by our interpretation of pattern documentation and requirements. In particular, for the membership functions we used Gaussian functions with mean 1, 5, 10 and standard deviation 2 for three linguistic values respectively. The input requirements can be seen in Table 9.2. We preferred to use Gaussian membership functions because they have the advantage of being non-zero all the time in comparison to trapezoidal or triangular functions. Unless we reduce much of the overlapping of the curves, the results did not change significantly. In that case we eliminate the number of selected patterns as we reduce uncertainty by making the borders between the linguistic values less vague. More linguistic values can lead to more accurate results, but as the number of linguistic values increases it becomes more difficult for the software engineer to perceive them. Different interpretations of the requirements may lead to different results but do not affect them dramatically.

	1. Book dock appointment	**2. Log trailer position**	**3. Storage of new products**	**4. Start video stream**
	• performance=4/10 • reliability=7/10 • acknowledgment=yes • log_app_error=no	• performance=8/10 • reliability=2/10 • acknowledgment=no • log_app_error=no	• performance=9/10 • reliability=5/10 • acknowledgment=yes • log_app_error=yes	• performance=8/10 • reliability=6/10 • acknowledgment=yes • log_app_error=yes
Generic	1. **sync_with_server** 2. sync_void_op 3. messaging 4. result_callback 5. poll_object 6. sync_with_result	1. **fire_and_forget** 2. result_callback 3. poll_object 4. messaging 5. sync_with_server 6. sync_void_op	1. **result_callback** 2. **poll_object** 3. messaging 4. sync_with_result	1. **result_callback** 2. **poll_object** 3. messaging 4. sync_with_result
Apache CXF	1. **sync_with_server** 2. sync_void 3. messaging_other 4. messaging_activemq 5. polling_ws 6. polling_approach	1. **oneway** 2. **fire_and_forget_ws** 3. polling_ws 4. polling_approach 5. messaging_other 6. messaging_activemq	1. **polling_ws** 2. **polling_approach** 3. **callback_ws** 4. **callback_approach** 5. messaging_other 6. messaging_activemq	1. **polling_ws** 2. **polling_approach** 3. **callback_ws** 4. **callback_approach** 5. messaging_other 6. messaging_activemq
Apache Axis 2	1. **sync_with_server** 2. sync_void 3. messaging_other 4. messaging_activemq 5. transport_level_polling 6. transport_level_callback	1. **oneway_nonblocking** 2. messaging_other 3. messaging_activemq 4. transport_level_polling 5. transport_level_callback 6. api_level_polling	1. **transport_level_polling** 2. **transport_level_callback** 3. **api_level_polling** 4. **api_level_callback** 5. messaging_other 6. messaging_activemq	1. **transport_level_polling** 2. **transport_level_callback** 3. **api_level_polling** 4. **api_level_callback** 5. messaging_other 6. messaging_activemq
Metro 1.2	1. **sync_with_server** 2. sync_void 3. messaging 4. polling_ws 5. async_polling 6. callback_ws	1. **oneway_ws** 2. **oneway_dispatch** 3. messaging 4. polling_ws 5. async_polling 6. callback_ws	1. **polling_ws** 2. **async_polling** 3. **callback_ws** 4. **async_callback** 5. messaging 6. sync_result	1. **polling_ws** 2. **async_polling** 3. **callback_ws** 4. **async_callback** 5. messaging 6. sync_result
.NET	1. **sync_with_server** 2. sync_void 3. msmq 4. polling_status 5. callback_state_object 6. callback_delegate	1. **fire_forget** 2. polling_status 3. msmq 4. callback_state_object 5. callback_delegate 6. sync_with_server	1. **polling_status** 2. **callback_state_object** 3. **callback_delegate** 4. msmq 5. sync_result	1. **polling_status** 2. **callback_state_object** 3. **callback_delegate** 4. msmq 5. sync_result
CORBA	1. **oneway_sync_server** 2. sync_void 3. messaging 4. AMI_polling 5. AMI_callback 6. sync_result	1. **oneway_sync_target** 2. **oneway_sync_none** 3. oneway_sync_server 4. messaging 5. AMI_polling 6. AMI_callback	1. **AMI_polling** 2. **AMI_callback** 3. messaging 4. sync_result	1. **AMI_polling** 2. **AMI_callback** 3. messaging 4. sync_result

TABLE 9.2: Design pattern selection

We have tuned the rules first through studying the literature and analysis using our own experience. In our example we use the same input requirements for all technologies, thus some technology-specific rules do not get evaluated. By making our requirements more detailed, we eliminate the returned best-fitting solutions. For example, if we provide *supported=yes* for Apache CXF the SYNC WITH SERVER pattern will not be returned and if we choose *transportlevel=no* for Apache Axis 2 the pattern implementations *transport_level_polling* and *transport_level_callback* will be eliminated.

Evaluation of Modeling Effort

To evaluate the required effort and complexity of our approach, we compare the implementation in our DSL to the implementation in FCL, the general fuzzy control language to which we translate our models and which can be leveraged by the fuzzy inference system. The most frequently used metric for quantifying required effort and complexity of software is still Lines of Code (LOC) [FP98], where the lines of the software's source code except blank lines and comments are counted. LOC gives a rough estimate of effort and complexity, but might be inappropriate when analyzing models [Wu+10]. A number of authors have suggested metrics such as Size of Model (SOM) as improvements over LOC metrics for model-based solutions [Lan07; Wu+10]. In particular, in the SOM metric we compare the number of modelling elements (nodes) in the models created from the code of both DSLs. For textual DSLs, LOC is useful as well, as it gives a rough impression of the effort needed to study and write the DSL code. Hence, we report LOC of DSL code as well. Finally, we report Model Element Density (MED), which can be defined as $MED = SOM/LOC$. This metric illustrates how many model elements (nodes) are expressed per source code line of the DSL.

In Table 9.3 we show the LOC, SOM and MED measurements for the generic and the five technology-specific fuzzy decision models, as well as the improvement that our DSL offers in comparison to the FCL. The development effort needed to create the fuzzy models using our DSL instead of the FCL decreases significantly. In the case of the generic model, our DSL requires 48.1% less LOC than the models expressed in the FCL. In the case of the technology-specific models, improvements ranging from 81.5%–87.3% have been achieved. The LOC measurements are consistent with the SOM measurements, where we achieved improvements of 33.3% for the generic model and 88.0%–91.6% for the technology-specific models, respectively. The lower improvements for the generic models can be explained by the fact that in the technology-specific models more often high-level language constructs of our DSL need to be used, which require multiple statements and model elements in the FCL. On average, improvements of 78.3% for LOC and 80.6% for SOM have been achieved.

	FCL			FuzzyPatternModel			Improvement (%)		
	LOC	SOM	MED	LOC	SOM	MED	LOC	SOM	MED
Generic	108	288	2.7	56	192	3.4	48.1	33.3	29
Apache CXF	293	1108	3.8	42	100	2.4	85.7	91.0	-37
Apache Axis 2	371	1389	3.7	47	116	2.5	87.3	91.6	-34
Metro 1.2	316	1158	3.7	45	101	2.2	85.8	91.3	-39
.NET	222	774	3.5	41	91	2.2	81.5	88.2	-36
CORBA	178	575	3.2	33	69	2.1	81.5	88.0	-35
Average	248	882	3.4	44	112	2.5	78.3	80.6	-25

TABLE 9.3: Evaluation of modeling effort results

An interesting metric is MED – in conjunction with the LOC and SOM metrics. It shows only for the generic model a plus of 29% in MED for our DSL, whereas the technology-specific models have a lower MED of −34%–−39%. On average, our DSL has a lower MED of −25% than FCL, but this is still in the range of similar MEDs for both DSLs. This can be interpreted as follows: Even though our DSL shows significant improvements regarding development effort and complexity with regard to the LOC and SOM metrics, our DSL is on average even a little more verbose than the FCL language. In other words, this shows that the improvements regarding LOC and SOM have not been solely achieved by introducing "more dense" language constructs into the DSL, but rather by introducing abstractions that better fit the design goals of the models to be expressed (i.e., to support pattern-based design decision making).

9.6 Limitations

One of the most important limitations of our method is that the fuzzy sets, the membership functions and the fuzzy rules are subject to the interpretation of the pattern documentations and acquired experience by the software architects. Therefore, many different solutions exist. The interpretation of the requirements provided to the fuzzy inference system by the requirements engineers is also subjective. Formalizing pattern decisions is challenging as a pattern describes not only a single solution template but a whole solution space for a recurring design problem. We showed in Section 9.5 how our approach is capable of dealing with many technology-specific variations of a pattern, however, the specification of pattern variants could still be simplified and more interactive support for the human making the decisions could be integrated.

9.7 Related Work on Fuzzy Logic in Software Design and Engineering

Fuzzy logic has been extensively used in software engineering for software development effort prediction [Ahm+05], estimating software testing costs and risks [EL07], software cost estimation [Mit+10], and software maintainability assessment [MB09]. A decision making support that considers uncertainty in decisions for design patterns has not yet been studied in the literature. However, some approaches propose similar decision support in the area of software design.

Aksit and Marcelloni use fuzzy logic based techniques to defer the elimination of design alternatives, in order to enhance object-oriented methods [AM01]. Moaven et al. propose a multi-criteria decision support system for the selection of architectural styles using fuzzy Choquet integral [Moa+08]. Babu et al. propose the selection of architectural styles using Analytic Network Process [Bab+10]. Both approaches require that quality attributes like flexibility and maintainability for each of the

architectural styles (pipes & filters, layered, etc.) have been evaluated using a grading system through an online survey. None of these approaches has been extended to cover the selection of alternative design patterns or cope with the typical decision complexity found in a pattern language.

9.8 Conclusions

In this chapter, we introduced a fuzzy logic based method for selecting design patterns for semi-automatic pattern-based decision making support under uncertainty. For this, we presented the steps of designing Mamdani-type fuzzy models and developed a DSL and Eclipse tooling for editing application-generic and technology-specific fuzzy models, providing requirements and inferring best-fitting solutions. These models get leveraged using a fuzzy inference system to provide an ordered list of the appropriate design patterns and pattern implementations given specific requirements. Our approach is supposed to be used as aid for software architects for making and documenting ADDs that give solutions to repeatable design problems. The whole design space, including application-generic as well as application-specific decisions is modeled once, however, the fuzzy logic decision models can be retrieved many times from the fuzzy model repository to derive appropriate solutions for desired requirements. It does not give support for making creative new design, but it rather provides useful support for design decisions that have to be made repeatedly in a specific context. The results show that our DSL leads to significant improvements with regard to the complexity and development effort, and seems to provide abstractions well fitting to the goal of supporting pattern-based design decision making.

Part IV

Consistency of Reusable Decisions with Architecture Designs and Variability

10 Maintaining the Consistency between Reusable Decisions and Designs

10.1 Introduction

ADDs play a crucial role not only during architectural design but also during development, evolution, reuse, and integration of software architectures [JB05]. The evolution of software systems requires that both ADDs and architectural design views are documented and maintained. In practice, the ADDs frequently are neither maintained nor synchronized over time with the corresponding design views. Until now, the establishment and preservation of consistency between decisions and designs have not been addressed or supported systematically. Usually, the potential inconsistencies need to be detected and resolved manually. So far there is no formal mapping or automated translation between ADDs and design views. Thus, the task of harmonizing decisions and designs remains ad hoc and tedious. making matters worse, the actual documentation of ADDs is also time consuming [Zim+08], especially for kinds of ADDs that need to be made repeatedly throughout the software design process.

The approach presented in this chapter aims at addressing the aforementioned challenges. In particular, our proposal introduces an AK transformation language. This domain-specific language supports the specification of primitive and complex actions whose enactment leads to automatic updates of design models (i.e., C&C diagrams) based on the corresponding documented ADDs derived from reusable decision models. To ensure consistency between ADDs and C&C views, constraints are automatically generated from the execution of transformation actions. That is, the constraints ensure, for instance, that manual changes in C&C views do not violate the ADDs. These constraints apply on the C&C models and their invalidation is resolved by software architects either (1) by executing automatically suggested model fixes on the C&C models or (2) by reconsidering ADDs and subsequently updating the corresponding C&C diagrams, in order to align designs to decisions. We exploit template-based generation rules and model-driven techniques for automatically instantiating and enacting the actions, as well as generating corresponding constraints. The linking of reusable ADDs to reusable actions and constraints in template form offers high reusability and automation.

In order to demonstrate our approach, we have integrated our tool ADvISE[1] – see Chapter 4 – and VbMF[2] – a tool for describing architectural view models and performing model-driven code generation. Using this prototypical implementation we evaluated our proposal in the context of an industrial case study from the warehouse automation area in terms of (a) reusability, (b) modeling effort, (c) efficiency, and (d) scalability.

10.2 Reusable AK Transformations

The main focus of our study is the harmonization of architectural design decisions and architectural models, that is, the maintenance of their consistency. This involves the tasks of making and documenting design decisions and creating and manipulating architectural models. Prior to the AK transformation language we have partially addressed the problem of bridging ADDs and designs [Lyt+12c]. In this proposal, we introduced a formal mapping model between different ADD types, on the one hand, and elements and properties of C&C models, on the other hand. Based on this formal mapping model, we could derive preliminary component models and OCL-like constraints for consistency checking. Yet, this mapping model had to be manually created and modified for each ADD separately, making this approach inefficient for large numbers of ADDs and/or complex design models. In the current approach, we take advantage of reusable ADDs to significantly enhance the productivity in creating and maintaining the formal mappings between the decisions and the designs. Thus, our proposal is an improvement to our previous work [Lyt+12c] and primarily focuses on reusable AK.

10.2.1 Introduction to VbMF

The View-based Modeling Framework (VbMF) [Tra+07] implements a model-driven, architectural view model. That is, it leverages the notion of view models for describing various aspects of the software systems at different abstraction levels and model-driven development techniques for generating code and configurations out of the view models [Tra+07]. The Core model shown in Figure 10.1 is the basis for defining various view models as well as integrating the view models using name-based matching techniques [Tra+11]. Among other views, VbMF provides a high-level component view model (see Figure 10.1) – similar to a typical C&C model such as UML component model – for representing essential architectural design elements such as components, ports, connectors, annotations, and properties that are independent from the underlying platforms and technologies. Technology- and platform-specific information will be described separately in the low-level view models that refine and enrich the high-level counterparts.

[1] http://swa.univie.ac.at/Architectural_Design_Decision_Support_Framework_(ADvISE)
[2] http://swa.univie.ac.at/View-based_Modeling_Framework

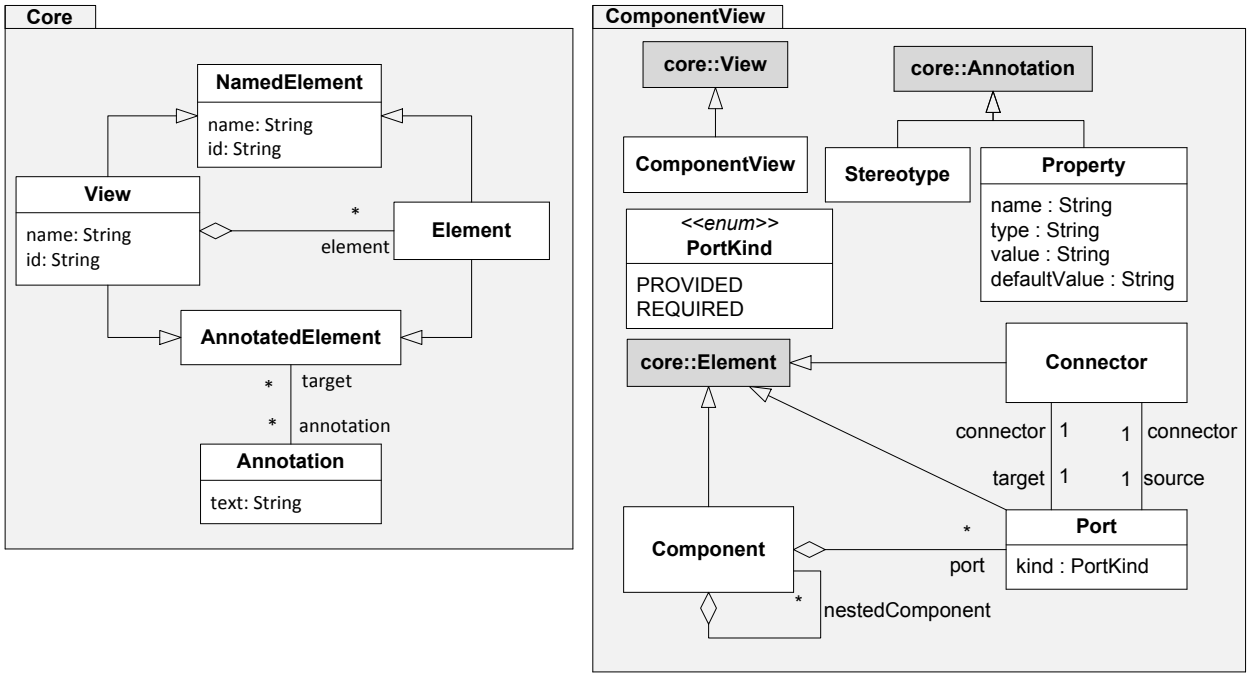


FIGURE 10.1: VbMF's core and component model

In our approach, we use the high-level component view model of VbMF (or in short form, the VbMF C&C view) for describing the architectural design of a software system. The advantage of using VbMF is that we can leverage the existing view model integration and transformation mechanisms of VbMF which are based on Eclipse technologies and therefore can be integrated well with ADvISE.

10.2.2 Approach Overview

We depict in Figure 10.2 an overview of our approach in terms of the involved stakeholders, tool suites, tools and artifacts along with their relationships. Tools are indicated with rounded rectangles while manually edited and automatically generated artifacts are denoted by rectangles having light-gray and dark-gray background, respectively.

Reusable ADD models based on Questions, Options, and Criteria (QOC) (i.e., the artifact *Reusable ADDs (QOC)*) are created using the ADvISE tool for recurring design issues. Based on these ADD models, ADvISE will automatically generate questionnaires that can be used for making actual architectural decisions (see *Actual ADDs*). Questionnaires containing ADDs instantiated from the same reusable ADD model can be generated and answered (but also modified) multiple times in similar design situations. For manipulating *C&C Diagrams*, VbMF provides a graphical *C&C View Model Editor*. In order to assist software architects and developers in *greenfield* scenarios (i.e., where a completely new C&C model needs to be derived from decisions that have been made) our approach supports generating initial instances of the *C&C Diagrams* automatically

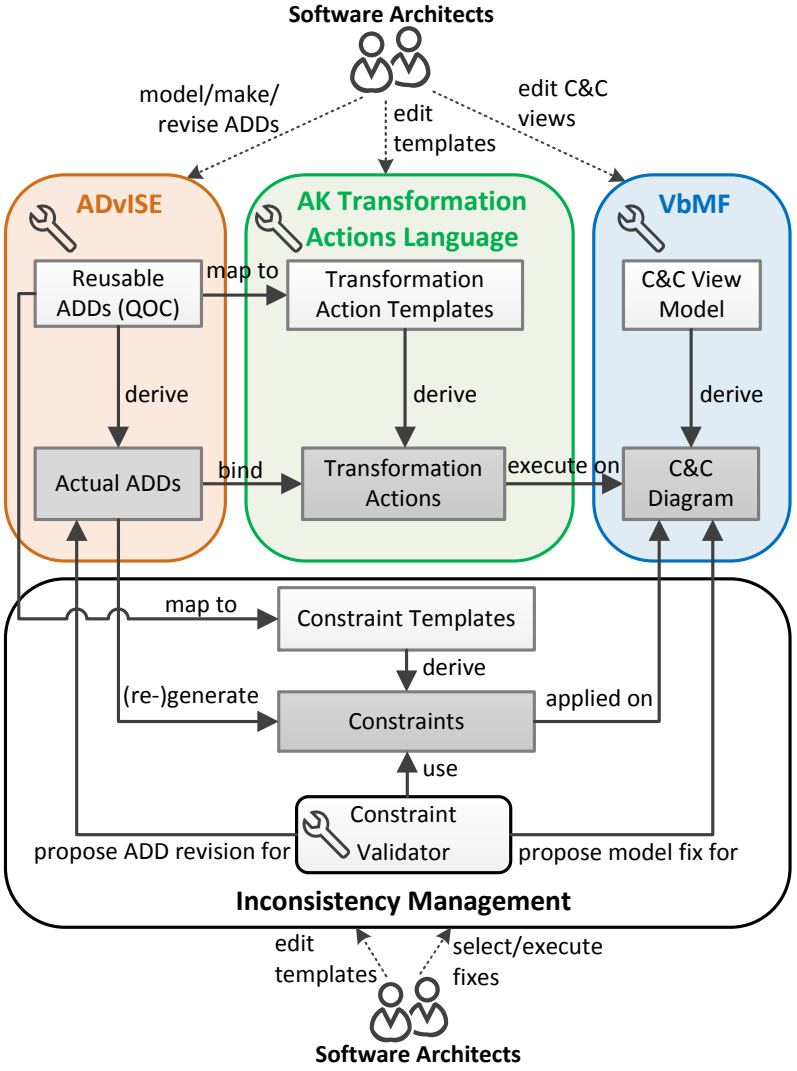


FIGURE 10.2: Approach overview

from ADDs through the execution of *Transformation Actions*. *Transformation Actions* can be also applied for updating existing view models. In a typical development scenario, design models can be changed independently of the documented decisions, that is, the *C&C Diagrams* can be manually manipulated. In this case, in order to ensure that changes in the C&C models do not invalidate existing ADDs, our approach allows the generation of OCL-like *Constraints* using predefined mappings between the reusable ADDs and constraint templates.

The *Constraint Validator* uses these constraints to check the consistency between actual ADDs and the corresponding C&C diagram. Their invalidation triggers model fixes for the C&C diagrams or recommendations for changing existing ADDs. The selection of the fixes is left to the software architects. The first kind of fixes – model fixes for the C&C diagrams – can be executed automatically, while the second kind of fixes – recommendations for revising ADDs – have to be performed manually by the architects.

To achieve the generation of transformation actions and constraints in a reusable fashion, the *Reusable ADDs* are formally mapped to *Transformation Language Templates*, which can be edited with the *AK Transformation Language Editor*. This way, for *Actual ADDs* we can instantiate the corresponding *Transformation Actions* and subsequently the *Constraints*. Using model-driven techniques, the transformation actions are automatically enacted on the corresponding VbMF C&C diagram. The AK transformation language and its enactment engine play an important role in our approach for enabling the automated and reusable transformation of ADDs into design models. In the subsequent parts of this section, we present the language and its illustrative usage in realistic development circumstances.

10.2.3 Approach Steps

We illustrate in Figure 10.3 the steps of our proposal that shall be performed by software architects, as well as the tools that will be used in each step and phase. In the first phase of the preparation of the reusable ADDs and their mappings to C&C views, the software architect uses the *ADD Model Editor* to create the reusable ADD models (i.e., define questions, options, etc. that will provide architectural decision guidance through corresponding questionnaires) based on the ADD meta-model. Then the software architect creates the transformation action templates that will transform actual ADDs into C&C view elements and consistency checking rules between decisions and designs respectively. We note that the aforementioned steps (1–2, Figure 10.3(a)) need to be done only once to define an architectural design space for particular application domains or contexts and are supposed to be performed by experienced software architects, as mentioned before. Given such a design space, the following steps (1–4, Figure 10.3(b)) will be carried out by software architects at design time who will make and document appropriate ADDs and create the corresponding design models.

The software architect follows ADvISE's guidance to make and document ADDs which will trigger the instantiation of the corresponding transformation actions and constraints. The transformation actions will then be enacted using our tools to generate initial C&C views. Finally, the software architect will be able to view and edit the C&C view(s) with the VbMF editor and validate the aforementioned constraints applied on the C&C view(s). If inconsistencies are detected, architects may choose to perform any of the recommended fixes in order to synchronize decisions with designs. These steps are supposed to be performed repeatedly in one or different projects and are based on the configuration that has been done by the experienced software architects in the Steps 1 and 2 of Figure 10.3(a).

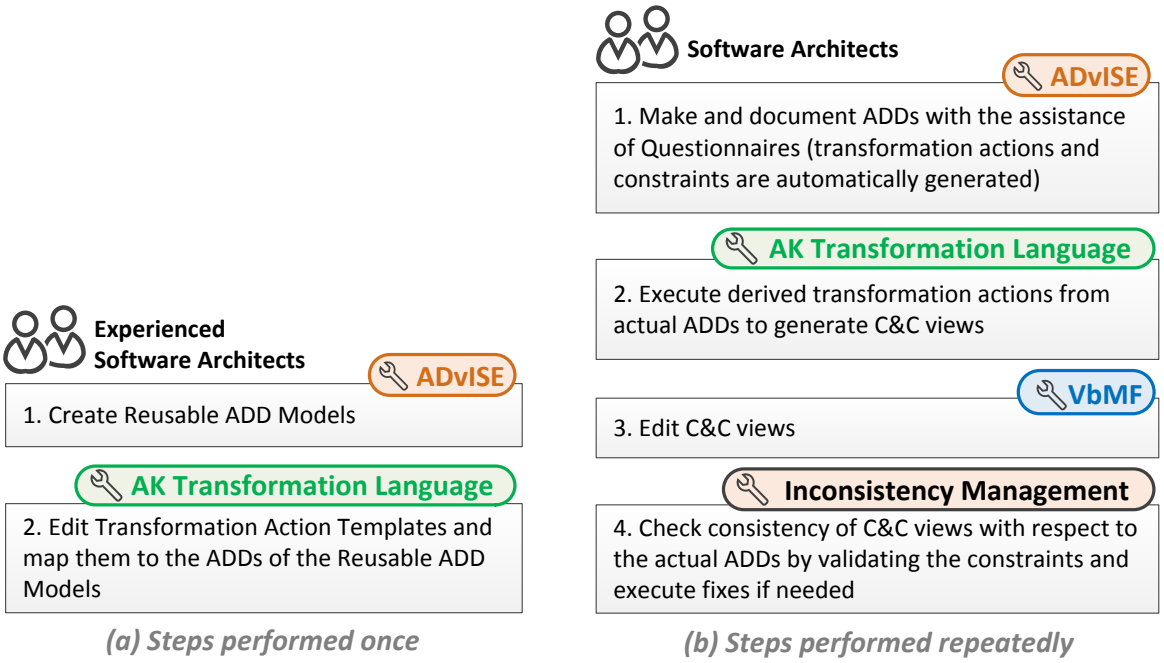


FIGURE 10.3: Approach steps performed by software architects

10.2.4 Architectural Knowledge (AK) Transformation Language

Essentially, the goal of the AK transformation language is to support the harmonization of ADDs and architectural design models via transformation actions that, when being enacted, shall create or update corresponding design models (e.g., C&C models) with respect to the intentions of the software architects reflected in the design decision models. One simple example is that making an architectural design decision on using a proxy between a client and server will lead to the creation of a new "*proxy component*" along with its properties, as well as appropriate connectors with the existing "*client*" and "*server*" components. In case none of them exists (e.g., in a "*greenfield*" scenario), these components must be created and wired accordingly.

We use the meta-model of Figure 10.4 to illustrate the main elements of the AK transformation language. The fundamental concept of our DSL is `Action` that represents the potential effects on an architectural design model such as adding (`Add`), deleting (`Delete`), and updating (`Update`) individual or a set of architectural elements such as components, connectors, ports, and their properties (e.g., `AddComponent`, `DeleteConnector`, `UpdatePort`, etc.). The enactment of an `Action` will lead to changes in the corresponding architectural design model such as the creation of a new component, the deletion of an existing connector, or the modification of a port, and so on. Apart from that, the user of the AK transformation language can define a set of components as sub-components of another component (`Group`) and refine a high level component onto one or more low-level components (`Refine`). A `Compound` is, firstly, used to represent a composite construct containing multiple actions. Secondly, it can inherit the definitions of existing `Compounds`, and therefore reduce redundancy and duplicated efforts. The semantics of a `Compound` ensures atomic

(i.e., all-or-nothing) sequential execution of its inherited compounds and constituting actions. In addition, the AK transformation language allows the execution of transformation actions under conditions (`Condition`). Finally, a `ForLoop` and a `WhileLoop` can be used in order to define one or more actions that are performed over a predefined set of design elements or under certain boolean constraints respectively.

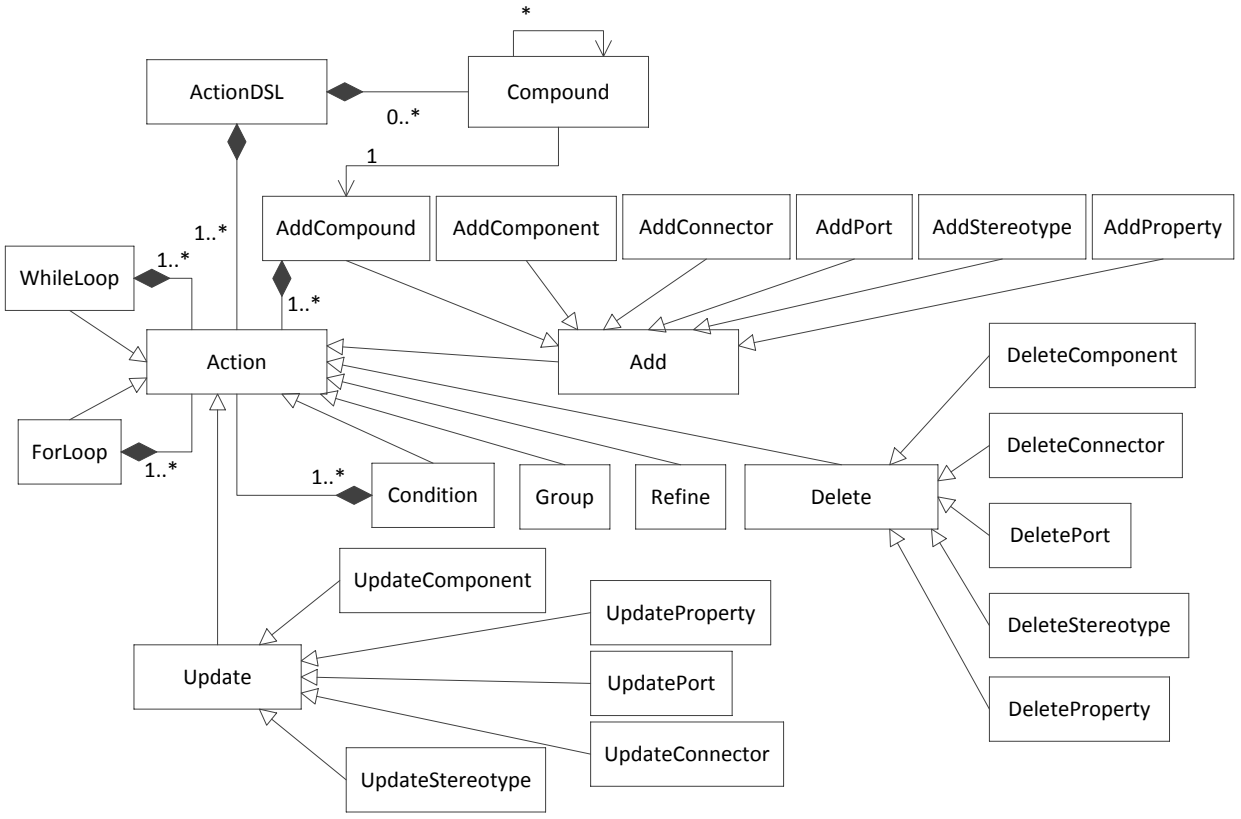


FIGURE 10.4: AK transformation language meta-model

As we mentioned above, the AK transformation language aims at aiding software architects in focusing on transforming AK into design models instead of being overwhelmed by model transformation concepts and techniques. Therefore, the language is designed to mainly emphasize on architecture-specific notions. The actions and constructs of the AK transformation language aim at expressing tentative changes to the elements of the corresponding VbMF C&C view and can be used either independently or along with ADvISE. In the latter case, the software architects, in the first and second steps of creating the design space (see Figure 10.3(a)), need to relate the options and answers of a certain ADD model with one or many transformation actions in the design decision templates. This enables the automation of creating and/or updating of the architectural C&C models when actual ADDs are made (in the Step 1–4 shown in Figure 10.3(b)), i.e., the templates are bound to concrete values. That is, once the generated questionnaires from the ADD model are answered resulting in concrete design decisions, the related actions will also be instantiated and bound to the concrete elements of the corresponding architectural models.

In the scope of our work, we developed the AK transformation language using the Eclipse Xtext framework[3]. The biggest and pragmatic advantage of using Xtext framework is its integration with Eclipse's modeling and development environment, which is widely used and supported in practice. Additionally, Xtext can generate Eclipse-based textual editors that deliver several powerful features such as syntax highlighting, content assistance and auto-completion, validation and quick fixes, automated external cross-references resolutions, and so on. Moreover, Xtext allows for partially dealing with some contextual aspects such as typing (by using Xtext type system framework[4]), uniqueness of identifiers (by using the notion of namespaces and scoping), and visibility (by scoping). The reader can refer to Appendix D for a complete specification of the AK transformation language using the Xtext grammar.

Usage Examples

Let us use the reusable ADD to introduce a *remote proxy* for calling a remote service from an application's component. This task will require a new component being created and annotated as "Remote Proxy" to denote an invocation of the remote service from the application's component. We illustrate a set of transformation actions in Listing 7, in which the parameters to be replaced are indicated with `${}`. These actions are only valid when they are bound to concrete C&C view elements. Therefore, the use of this set of actions requires that the following information is provided: the name of the C&C view model (`cv`), the name of the remote service (`A`), and the name of the component which calls this remote service (`B`).

```
add component "${A}Proxy"
add stereotype <<"Remote Proxy">> to ${cv}.${A}Proxy
add port "${A}Proxy_p1" kind=REQUIRED to ${cv}.${A}Proxy
add port "${A}Proxy_p2" kind=PROVIDED to ${cv}.${A}Proxy
add port "${A}_p" kind=PROVIDED to ${cv}.${A}
add port "${B}_p" kind=REQUIRED to ${cv}.${B}
add connector "${A}Proxy_${A}" from ${cv}.${A}Proxy.${A}Proxy_p1 to ${cv}.${A}.${A}_p
add connector "${B}_${A}Proxy" from ${cv}.${B}.${B}_p to ${cv}.${A}Proxy.${A}Proxy_p2
```

Listing 7: Example of parameterized transformation actions for creating a proxy in template form

When the software architects make concrete choices in the design decisions, the corresponding architectural models, which either exist or are newly created, are determined. As a result, the parameters of the aforementioned set of transformation actions can be bound to concrete values, for instance, `cv` $\sim$ `model`, `A` $\sim$ `Service`, and `B` $\sim$ `CompA`. The binding (denoted above as $\sim$) will be done automatically by our tools, resulting in the executable transformation actions shown in Listing 8.

[3] http://www.eclipse.org/Xtext
[4] https://code.google.com/a/eclipselabs.org/p/xtext-typesystem

```
add component "ServiceProxy"
add stereotype <<"Remote Proxy">> to model.ServiceProxy
add port "ServiceProxy_p1" kind=REQUIRED to model.ServiceProxy
add port "ServiceProxy_p2" kind=PROVIDED to model.ServiceProxy
add port "Service_p" kind=PROVIDED to model.Service
add port "CompA_p" kind=REQUIRED to model.CompA
add connector "ServiceProxy_Service" from model.ServiceProxy.ServiceProxy_p1 to model.Service.Service_p
add connector "CompA_ServiceProxy" from model.CompA.CompA_p to model.ServiceProxy.ServiceProxy_p2
```

Listing 8: Example of transformation actions for creating a proxy

The transformation actions presented above add new elements in the C&C view. Nonetheless, updating and removing existing components, connectors, etc. in a C&C view may be necessary when certain ADDs are revised or removed. Listing 9 illustrates three "`Delete`" actions that reverse the effects of the transformation actions shown in Listing 8 and an "`Update`" action that changes the name of the port `Service_p` of the component `Service`.

```
delete component model.ServiceProxy
delete port model.Service.Service_p
delete port model.CompA.CompA_p
update port model.Service.Service_p name="IService"
```

Listing 9: Examples of delete and update actions

The use of templates enables the reuse of transformation actions whenever ADDs from the reusable ADD model are made. The enactment of transformation actions will update the corresponding C&C diagram. This is achieved by performing model-to-model transformations from single transformation actions into model actions that modify the underlying C&C view models. Compound actions, conditionals, and loops are translated into their containing single actions that are afterwards enacted on the C&C view. Thus, the AK transformation language provides adequate abstractions for hiding unnecessary details of model transformation techniques, and therefore, helps software architects to better focus on architecture-specific concepts. In our example, the execution of the actions of Listing 8 will lead to the C&C view of Figure 10.5.

10.2.5 Recurring Pattern Primitives as Reusable AK Transformations

In the course of design decision making, software architects often deal with several recurring architectural elements and structures such as adapters and layers. The idea of proposing primitive abstractions as fundamental elements for describing such recurring design patterns and architectural styles has been investigated by various studies. For example, Zdun and Avgeriou described architectural patterns through a number of recurring architectural primitives in the component-and-connector view using UML profiles [ZA05]. Mehta and Medvidović developed a framework for defining abstract primitives shared by all architectural styles for composing their elements [MM03].

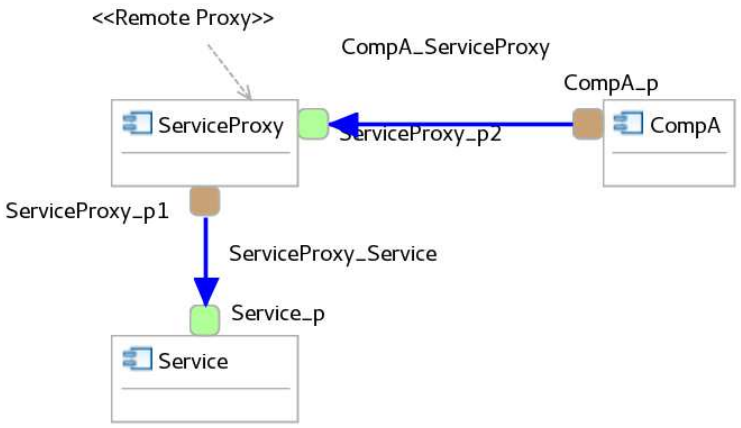


FIGURE 10.5: Example of C&C view

In this section, we describe how our approach can be used to define and use recurring architectural primitives for modeling certain patterns or styles as compounds. In particular, the expressiveness of our AK transformation language and the support for compositions and extensions through the composite structures mentioned above enable us to define *recurring architectural primitives* in a reusable and extensible way. In our approach, we specify such recurring primitives using parameterized sets of actions that are based on compounds and can be inherited and extended further. Each compound represents one primitive abstraction that can be used to realize a number of patterns that use this particular primitive as part of their solution (see [ZA05]). The compounds are used via their name and appropriate parameters. In a compound specification, we use variable access in the form `${p-name}` to refer to the parameter *p-name*. When concrete compounds are "instantiated" via the command type `add compound`, the compound parameters are replaced with the provided parameter values, which leads to the variable binding of their primitive actions.

In Listing 10, we illustrate the *indirection* compound. Indirection is applicable in case one or more related "proxy" components receive a message on behalf of one or more "target" components, forward the message to these "targets", and receive results from the "targets" also through the "proxy" components [ZA05]. Proxies and adapters are examples of indirection. The parameters `cv`, `A`, and `B` refer to the target component view, the target component, and the client respectively. The variable `n` will be bound to the name of the compound "instance" (e.g., Proxy, Adapter).

```
compound indirection (cv A B) {
  add component "${A}${n}"
  add port "${A}${n}_I1" kind=REQUIRED to ${cv}.${A}${n}
  add port "${A}${n}_I2" kind=PROVIDED to ${cv}.${A}${n}
  add port "${A}_I" kind=PROVIDED to ${cv}.${A}
  add port "${B}_I" kind=REQUIRED to ${cv}.${B}
  add connector "${A}${n}_I1_${A}_I" from ${cv}.${A}${n}.${A}${n}_I1 to ${cv}.${A}.${A}_I
  add connector "${B}_I_${A}${n}_I2" from ${cv}.${B}.${B}_I to ${cv}.${A}${n}.${A}${n}_I2
  add stereotype <<"${n}">> to ${cv}.${A}${n}
}
```

LISTING 10: Indirection compound action specification

A usage example of the indirection compound is presented in Listing 11.

```
add compound indirection "Proxy" (model Service Facade)
```

Listing 11: Example usage of the compound "indirection"

This compound action will create a proxy for invoking the component "Service" from the component "Facade". The command `add compound` contains a reference to a compound definition. When a compound action is executed, the reference to the compound definition is resolved and the actual variables are replaced. In our example, the variables `n`, `cv`, `A`, and `B` will get the values "Proxy", the C&C view model name "model", and the components "Service" and "Facade", respectively. The transformation of the C&C view includes the creation of a component, two connectors, the corresponding ports, and a stereotype. The execution of the compound transformation action triggers the execution of its containing actions. In our example, the enactment of Listing 11 will trigger the execution of the primitive actions in Listing 12.

```
add component "ServiceProxy"
add port "ServiceProxy_I1" kind=PROVIDED to model.ServiceProxy
add port "ServiceProxy_I2" kind=REQUIRED to model.ServiceProxy
...
```

Listing 12: Binding of primitive actions of indirection compound action

A compound can extend other compounds, and therefore, inherit the corresponding sets of actions contained in these compounds. For instance, in Listing 13, we specify a new compound `integrationAdapter` which extends the existing compound `adapter`.

```
compound integrationAdapter extends adapter {...}
```

Listing 13: Example of compound extension

The advantage of the compounds is that they specify a group of transformation actions that can be reused with a simple parametrized transformation action "`add compound ...`". In this way, we can increase the reusability of the AK transformations. Therefore, apart from the reuse of the low-level model actions that apply on the C&C views, the reusability of the AK transformations can be further increased by the introduction of compound actions which group other transformation actions. For the needs of a case study that we discuss in detail in the next section, we have modeled the following six compounds and have reused them in various design situations covering 21 design patterns in total:

Indirection: Indirection happens when a "proxy" component receives a message on behalf of a "target" component and forwards the message to that "target". Afterwards the result is sent back through the "proxy" component again.

Shield: A component can act as a "shield" for a set of components that form a subsystem. In this case, a client can access the members of the subsystem only through this "shield".

Grouping: A group member is part of an abstract or virtual entity. That is, there is no component in the software architecture for representing the group as a whole, but it is made only of its parts.

Callback: A callback denotes an invocation to a component B that is stored as an invocation reference in a component A. Between two components A and B, a set of callbacks can be defined, also usually implemented as methods.

Transformer: A transformer performs transformation as well as enrichment, splitting, aggregation, etc. of data that is sent from component A to component B.

Router: A router routes requests of a component to a set of other components according to specified criteria.

In Table 10.1 we present the compound specifications along with the patterns that share these common abstractions. For these compound specifications the primitive abstractions modeled in [ZA05] were used as reference with slight modifications. While Zdun and Avgeriou use OCL to document architectural primitives [ZA05], we express these abstractions using compounds. For a detailed description of the patterns for service-based platform integration that are included in Table 10.1 you can refer to Appendix B.

Compound	Compound Specification	Design Patterns

Indirection	`compound indirection (cv A B) {`
	`  add component "${A}${n}"`
	`  add port "${A}${n}_I1" kind=REQUIRED to ${cv}.${A}${n}`
	`  add port "${A}${n}_I2" kind=PROVIDED to ${cv}.${A}${n}`
	`  add port "${A}_I" kind=PROVIDED to ${cv}.${A}`
	`  add port "${B}_I" kind=REQUIRED to ${cv}.${B}`
	`  add connector "${A}${n}_I1_${A}_I" from ${cv}.${A}${n}.${A}${n}_I1 to`
	`  ${cv}.${A}.${A}_I`
	`  add connector "${B}_I_${A}${n}_I2" from ${cv}.${B}.${B}_I to ${cv}.${`
	`  A}${n}.${A}${n}_I2`
	`  add stereotype <<"${n}">> to ${cv}.${A}${n}`
	`}`
		PROXY, REMOTE PROXY, ADAPTER, REMOTE ADAPTER

Shield	`compound shield (cv T L) {`
	`  add component "${T}"`
	`  add stereotype <<"${n}">> to ${cv}.${T}`
	`  add port "${T}_S" kind=REQUIRED to ${cv}.${T}`
	`  for (c : ${L})`
	`  add port "${c}_S" kind=PROVIDED to ${cv}.${c}`
	`  add connector "${T}_S_${c}_S" from ${cv}.${T}.${T}_S to ${cv}.${c}.`
	`  ${c}_S`
	`  end`
	`}`
		FACADE, REMOTE FACADE, GATEWAY, SERVICE INTERFACE

Continued on next page

Table 10.1 – continued from previous page

Compound	Compound Specification	Design Patterns

Grouping	`compound grouping (cv T L) {`
	`  add component "${T}"`
	`  add stereotype <<"${n}">> to ${cv}.${T}`
	`  add port "${T}_G" kind=PROVIDED to ${cv}.${T}`
	`  for (c : ${L})`
	`  group ${cv}.${c} container ${cv}.${T}`
	`  end`
	`}`
		SERVICE ABSTRACTION LAYER

Callback	`compound callback (cv A B) {`
	`  add port "${A}_E" kind=PROVIDED to ${cv}.${A}`
	`  add stereotype <<"EventPort">> to ${cv}.${A}.${A}_E`
	`  add port "${A}_C" kind=REQUIRED to ${cv}.${A}`
	`  add stereotype <<"CallbackPort">> to ${cv}.${A}.${A}_C`
	`  add port "${B}_E" kind=REQUIRED to ${cv}.${B}`
	`  add stereotype <<"EventPort">> to ${cv}.${B}.${B}_E`
	`  add port "${B}_C" kind=PROVIDED to ${cv}.${B}`
	`  add stereotype <<"CallbackPort">> to ${cv}.${B}.${B}_C`
	`  add connector "${B}_E_${A}_E" from ${cv}.${B}.${B}_E to ${cv}.${A}.${A}_E`
	`  add connector "${A}_C_${B}_C" from ${cv}.${A}.${A}_C to ${cv}.${B}.${B}_C`
	`  add stereotype <<"${n}">> to ${cv}.${A}_E_${B}_E`
	`  add stereotype <<"${n}">> to ${cv}.${B}_C_${A}_C`
	`}`
		RESULT CALLBACK, REQUEST-REPLY, REQUEST-ACKNOWLEDGE CALLBACK

Transformer	`compound transformer (cv A B) {`
	`  add component "${A}${n}"`
	`  add port "${A}${n}_T1" kind=REQUIRED to ${cv}.${A}${n}`
	`  add port "${A}${n}_T2" kind=PROVIDED to ${cv}.${A}${n}`
	`  add port "${A}_T" kind=PROVIDED to ${cv}.${A}`
	`  add port "${B}_T" kind=REQUIRED to ${cv}.${B}`
	`  add connector "${A}${n}_T1_${A}_T" from ${cv}.${A}.${A}${n}_T1 to ${cv}.${A}.${A}_T`
	`  add connector "${B}_T_${A}${n}_T2" from ${cv}.${B}.${B}_T to ${cv}.${A}.${A}${n}_T2`
	`  add stereotype <<"${n}">> to ${cv}.${A}${n}`
	`}`
		DATA MAPPER, MESSAGE TRANSLATOR, SPLITTER, AGGREGATOR, CONTENT ENRICHER, CONTENT FILTER

Continued on next page

Table 10.1 – continued from previous page

Compound	Compound Specification	Design Patterns

Router	`compound router (cv T A L) {`
	`  add component "${T}"`
	`  add stereotype <<"${n}">> to ${cv}.${T}`
	`  add port "${T}_R1" kind=PROVIDED to ${cv}.${T}`
	`  add port "${A}_R" kind=REQUIRED to ${cv}.${A}`
	`  add connector "${A}_R_${T}_R1" from ${cv}.${A}.${A}_R to ${cv}.${T}.${T}_R1`
	`  add port "${T}_R2" kind=REQUIRED to ${cv}.${T}`
	`  for (c : ${L})`
	`  add port "${c}_R" kind=PROVIDED to ${cv}.${c}`
	`  add connector "${T}_R2_${c}_R" from ${cv}.${T}.${T}_R2 to ${cv}.${c}.${c}_R`
	`  end`
	`}`
		PUBLISH-SUBSCRIBER, MESSAGE ROUTER, CONTENT-BASED ROUTER

cv: component view *n*: design pattern *T*: component name *A*, *B*: components *L*: list of components

TABLE 10.1: Reusable AK transformation compounds

10.2.6 Detection and Fixing of Inconsistencies Between ADDs and C&C Views

We propose to use consistency checking to ensure the integrity of C&C views and their corresponding ADDs. For instance, if implementing an ADD implies (among other things) the creation of a component in the C&C view, the consistency is preserved as long as the underlying component exists in the view. In this context, two main sources of inconsistencies exist: (1) the C&C views are modified, so that existing ADDs are not valid anymore and (2) the ADDs are modified, so that the existing C&C view becomes outdated with regard to the ADDs. In our approach, we implement the management of this kind of inconsistencies (i.e., detection and handling) by introducing OCL-based constraints between reusable ADDs and C&C view elements. We implement these constraints using Epsilon Validation Language (EVL) [Kol+09], as EVL provides complementary support to OCL for providing user feedback, repairing inconsistencies, and introducing dependent constraints.

As presented in Figure 10.6 a number of constraints (similar to invariants in OCL) can be defined in a specific Context. A Context specifies the kind of C&C view elements on which the Constraints will be evaluated. Each Constraint has a name, an error message, and a body (expression) (i.e., ExpressionOrStatementBlock), and can additionally define two kinds of fixes: either model actions which are automatically applied on the C&C view (i.e., CnCFix) or recommendations which have to be executed by the software architect (i.e., ADDFix).

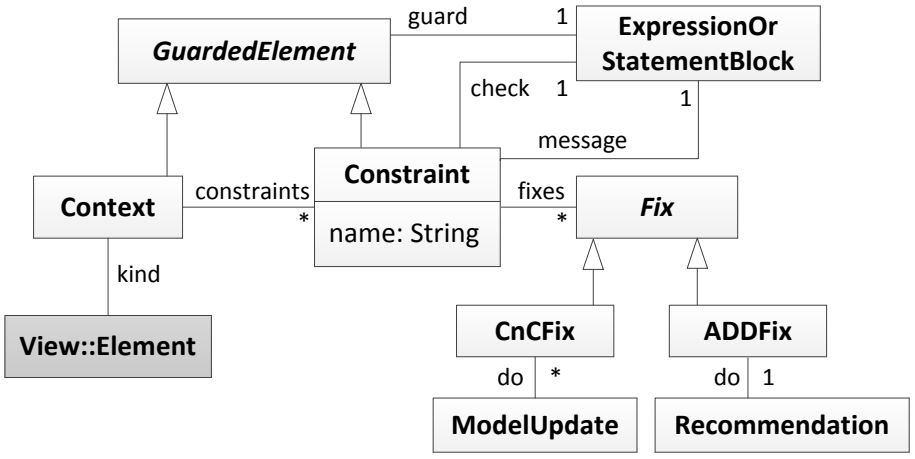


FIGURE 10.6: Abstract syntax of constraints

These constraints do not have to be edited for each ADD separately but are generated from predefined templates mapped to reusable ADDs or transformation actions. These mappings have to be defined once for each reusable ADD or transformation action and can be "instantiated" many times afterwards, whenever actual ADDs are made or modified. Also, the constraints in template form can be reused among different reusable ADDs.

Listing 20 is an example of one constraint derived from the example of Listing 8, in which a new component named ServiceProxy is introduced. The constraint ComponentServiceProxyExists is used to check if the component ServiceProxy still exists in the C&C view. If this evaluates to false it returns the error message Component ServiceProxy does not exist. For fixing this inconsistency three alternatives exist: (1) an existing component will be renamed to ServiceProxy (lines 6–12), (2) the component ServiceProxy will be added to the C&C view (lines 15–20), or (3) the related ADD will be revised (lines 23–26). The first two fixes can be applied automatically on the C&C view while the last fix (recommendation) has to be performed by the user. The selection of the fix to be executed is left to the software architect. Every time an ADD changes, the newly generated constraints will check the conformance of the C&C view with respect to the modified ADDs.

We have designed and developed respective constraint templates for each AK transformation language element and each architectural primitive defined above. Similar to the AK transformation language, the `${...}` syntax in the constraint rule templates allows to access a variable that is instantiated and bound to particular values of the related actions and models. The outcomes of the instantiation and binding of the parameterized constraint templates are concrete constraints that can be enacted using the constraint validator. The combination of transformation actions with automatically generated constraints that check that the transformation's semantics are not violated in the C&C diagram, enables us to allow developers and architects to manually change the C&C model. If a manual change violates an architectural decision that has triggered transformation actions, the corresponding constraint checking will signal an error.

```
context ComponentView {
  constraint ComponentServiceProxyExists {
    check: self.componentExists("ServiceProxy")
    message: 'Component ServiceProxy does not exist'
    fix {
      title: 'Change existing component to ServiceProxy'
      do {
        var cName: String;
        cName = UserInput.prompt("Select component of "+self.name);
        if (cName.isDefined()) {
          self.getComponent(cName).name = "ServiceProxy";
        }
      }
    }
    fix {
      title: 'Create component ServiceProxy'
      do {
        var newC: new Component;
        newC.name = "ServiceProxy";
        self.element.add(newC);
      }
    }
    fix {
      title: 'Reconsider ADD *Create a service proxy*'
      do {
        UserInput.info("Please reconsider the decision and re-generate the constraints.");
      }
    }
  }
}
operation ComponentView componentExists(name: String): Boolean {
  return Component.allInstances.exists(c:Component|c.name.equals(name));
}
operation ComponentView getComponent(name: String): Component {
  return Component.allInstances.selectOne(c:Component|c.name.equals(name));
}
```

Listing 20: Example of a constraint with fixes

In Table 10.2 we list exemplary reusable constraint templates along with error messages and suggested fixes for potential invalidations. All variables to be replaced when the templates are instantiated are indicated with "$".

Finally, we include in Table 10.3 an excerpt of the set of reusable constraints in template-based form that correspond to the actions defined in the AK transformation language. The current (OCL-like) form of the reusable constraint templates is designed for better understanding and maintaining.

Constraint	Error Message	Suggested Fixes
`Component.allInstances.exists(c:Component|c.name=$comp)`	`Component $comp does not exist`	1. Change existing component to `$comp` 2. Create component `$comp` 3. Reconsider related ADD `$add`
`Connector.allInstances.exists(c:Connector|c.name=$conn and (c.source.name=$a and c.target.name=$b))`	`Connector $conn between $a and $b does not exist`	1. Change connector between `$a` and `$b` to `$conn` 2. Create connector `$conn` between `$a` and `$b` 3. Reconsider related ADD `$add`
`Component.allInstances.exists(c:Component|c.name=$comp and c.port.exists(p:Port\|p.name=$port and p.kind=PortKind#$kind)))`	`Port $port of kind $kind for component $comp does not exist`	1. Change existing port to `$port` of kind `$kind` 2. Change kind of existing port `$port` to `$kind` 3. Create port `$port` of kind `$kind` for component `$comp` 4. Reconsider related ADD `$add`
`Element.allInstances.exists(e:Element|e.name=$elem and e.annotation.exists(p:Property\|p.name=$prop and p.type=$type and p.value=$value))`	`Element $elem does not have property $prop $type=$value`	1. Change existing property `$prop` of `$elem` to `$type=$value` 2. Change existing property of `$elem` to `$prop $type=$value` 3. Create property `$prop $type=$value` for element `$elem` 4. Reconsider related ADD `$add`
`Element.allInstances.exists(e:Element|e.name=$elem and e.annotation.exists(s:Stereotype\|s.name=$stereo))`	`Element $elem is not annotated as $stereo`	1. Change existing annotation of `$elem` to `$stereo` 2. Create annotation `$stereo` for element `$elem` 3. Reconsider related ADD `$add`
`Component.allInstances.exists(c:Component|c.name=$cont and c.nestedComponent.exists(c:Component\|c.name=$comp))`	`Component $comp is not contained in $cont`	1. Move component `$comp` into `$cont` 2. Reconsider related ADD `$add`

TABLE 10.2: Exemplary reusable constraints with fixes in template form

Transformation Action	Reusable constraint in template form
• add component • update component	`context component::ComponentView ERROR "${ADD}:Component ${component} does not exist": element.typeSelect(component::Component).exists(c\|c.name=="${component}");`
• add connector • update connector	`context component::ComponentView ERROR "${ADD}:Component ${componentA} and ${componentB} are not connected": element.typeSelect(component::Component).exists(c1\|c1.name=="${componentA}" && element.typeSelect(component::Component).exists(c2|c2.name=="${componentB}" && element.typeSelect(component::Connector).exists(conn|(c1.port.exists(p|conn.source==p) && c2.port.exists(p|conn.target==p)) || (c1.port.exists(p|conn.target==p) && c2.port.exists(p|conn.source==p)))));`
• add port • update port	`context component::ComponentView ERROR "${ADD}:Port ${port} of kind ${portKind} for component ${component} does not exist": element.typeSelect(component::Component).exists(c\|c.name=="${component}" && c.port.exists(p|p.name=="${port}" && p.kind==component::PortKind::${portKind}));`
• add property • update property	`context component::ComponentView ERROR "${ADD}:Property ${type}=${value} of ${element} does not exist": annotation.typeSelect(component::Property).exists(p\|element.exists(c\|c.annotation.exists(a\|a==p) && c.name=="${element}" && p.type=="${type}" && p.value=="${value}"));`
• add stereotype • update stereotype	`context component::ComponentView ERROR "${ADD}: ${element} is not annotated as ${stereotype}": annotation.typeSelect(component::Stereotype).exists(s\|element.exists(c\|c.annotation.exists(a\|a==s) && c.name=="${element}" && s.text=="${stereotype}"));`
• group component	`context component::ComponentView ERROR "${ADD}:Component ${container} does not contain ${component}": element.typeSelect(component::Component).exists(c\|c.name=="${container}" && c.nestedComponent.exists(c|c.name=="${component}"));`

TABLE 10.3: List of reusable constraints in template form

Dependencies between Fixes

A single modification in the C&C view (e.g., the deletion of a connector) or the revision of an existing ADD may cause many constraints to be invalidated. In the case that many changes have

been performed in both sides the software architect will be overwhelmed by a big number of error messages and possible fixes. Apart from that, some fixes may not be executable, as they depend on some preconditions. For instance, checking the existence of a port of a component requires that the component still exists in the C&C view. To reduce evaluated constraints we use guards which limit the applicability of invariants according to the result of an expression. Thus, the evaluation of some constraints is delayed until the preconditions are satisfied after the execution of other fixes.

In our running example, the constraint that checks if the port `ServiceProxy_p1` exists (created by Listing 8) will be validated only if the component `ServiceProxy` already exists in the C&C view or as soon as it is restored by executing another fix, as shown in Listing 21. By introducing dependencies between the various constraints we reduce on the one hand the number of currently invalidated constraints, and on the other hand we ensure that all proposed fixes are executable.

```
context ComponentView {
  constraint PortServiceProxyP1Exists {
    guard : ComponentServiceProxyExists
    check : self.portExists("ServiceProxy.ServiceProxy_p1")
    message : 'Port ServiceProxy.ServiceProxy_p1 does not exist'
    ...
  }
}
```

Listing 21: Example of constraint dependencies

10.3 Evaluation

We applied our approach in the context of the case study for service-based platform integration described in detail in Section 5.8. For this, we used the following tools: (1) ADvISE for modeling reusable ADDs and making concrete decisions, (2) VbMF for editing the corresponding C&C views, (3) a prototype implementation for mapping ADDs to constraints in template form and generating concrete constraints, and (4) EVL Eclipse plugin[5] for editing and validating constraints with fixes applying on the underlying C&C views.

10.3.1 Case Study

In Section 5.6 and Appendix A, we have introduced an ADD model to handle various integration aspects in the service-based platform integration domain, i.e., integration and adaptation, interface design, etc. We reuse an excerpt of this reusable decision model to demonstrate the use of AK transformation actions. Thus, we present in Table 10.4 an excerpt of the ADD model of the platform integration scenario consisting of questions and different alternative options along with

[5] http://www.eclipse.org/epsilon/doc/evl/

the corresponding transformation actions that translate the underlying decisions into elements of the C&C view. This example shows the decision making support on the type of integrating components between a platform service `PS` of one of the three platforms in our case study (i.e., WMS, YMS, and RMS) and a component of the integration layer `IC` (note: `cv` refers to the target C&C view). Every ADD option in the ADD model is associated with a set of primitive and compound actions based on pattern primitives in template form as defined in Section 10.2. The excerpt of the ADD model consists of five questions, uses seven primitive actions and two compound actions (`integrationAdapter` once and `indirection` twice), and is related to three patterns: proxy (local or remote), adapter (local or remote) and integration adapter.

The integration of the Velocity Template Language and Engine[6] with our AK transformation language allows us not only to use placeholders (${...}) but also statements (if, foreach, etc.), which begin with the # character and are parsed by the template engine, but ignored by the AK transformation language editor. When actual ADDs are made, the parameters of the transformation actions of Table 10.4 are bound to concrete values. For instance, let us assume that the software architect opts for a remote proxy as an integrating component between the YMS service *TruckMgmnt* and the integration layer's component *OperatorFacade*. The actual ADDs will be reflected in the corresponding C&C view by executing the derived transformation actions of Listing 22.

```
add compound indirection "Proxy" (example TruckMgmnt OperatorFacade)
add stereotype <<"Remote Proxy">> to example.TruckMgmntProxy
```

Listing 22: Transformation actions example from case study

In Table 10.5, we report a set of reusable ADDs that we leveraged in the warehouse case study. We also document on the number of constraints that were generated in order to check the consistency between the ADDs and the resulting C&C view. In total, we documented 24 ADDs and, using our tools, we transformed them into C&C view elements and generated 222 corresponding constraints. We note that the actual benefit of our proposed reusable constraint templates does not merely rely on the generation of concrete constraints but on the ability to be leveraged in the application context of recurring ADDs. For instance, without support from our techniques the software architects must manually make/document 24 ADDs and define 222 constraints for integrating the platform services.

10.3.2 Reusability of AK Transformation Language

Notwithstanding the initial efforts for creating the reusable AK transformations, architects will benefit from reduced total efforts in case of recurring ADDs and AK transformations in the long term. In our approach, reusability can be achieved at three different levels: (1) First of all, the AK transformation language hides the low-level model actions that are needed to transform the C&C view models. These model actions are embedded in the enactment engine of the DSL; (2)

[6] http://velocity.apache.org

ADD Options	AK Transformation Actions

Type of Integrating Component
• None
• Same Interface
• Different Interface
	`#if(${TypeOfComponent} == "None")`
	`  add port "${PS}_p1" kind=PROVIDED to ${cv}.${PS}`
	`  add port "${IC}_p1" kind=REQUIRED to ${cv}.${IC}`
	`  add connector "${IC}_${PS}" from ${cv}.${IC}.${IC}_p1 to ${cv}.${PS}.${PS}_p1`
	`  add stereotype <<"Direct call">> to ${cv}.${IC}_${PS}`
	`#elseif(${TypeOfComponent} == "Same Interface")`
	`  add compound indirection "Proxy" (${cv} ${PS} ${IC})`
	`#elseif(${TypeOfComponent} == "Different Interface")`
	`  add compound indirection "Adapter" (${cv} ${PS} ${IC})`
	`#end`

Type of Proxy • Local • Remote	`add stereotype <<"${TypeOfProxy} Proxy">> to ${cv}.${PS}Proxy`
Type of Adapter • Local • Remote	`add stereotype <<"${TypeOfAdapter} Adapter">> to ${cv}.${PS}Adapter`

Heterogeneous systems
• No
• Yes
	`#if(${HeterogeneousSystems} == "Yes")`
	`  add compound integrationAdapter "Integration Adapter" (${cv} ${PS} ${IC})`
	`#end`

Interchangeability • No • Yes	`add property "${PS}Adapter_Interchangeability" type="Interchangeability" value=` `  "${Interchangeability}" to ${cv}.${PS}Adapter`

TABLE 10.4: Excerpt an ADD model and its corresponding AK transformation actions

ADD	Times Reused	Constraints
Proxy	3	3 x 9
Adapter	1	1 x 11
Result Callback	7	7 x 12
Facade	1	1 x 8
Gateway	2	2 x 6
Publish-Subscriber	7	7 x 8
Data Mapper	2	2 x 8
Content Enricher	1	1 x 8
Total	24	222

TABLE 10.5: Reusable ADDs and generated constraints in the warehouse case study

in addition, the AK transformations are edited only once for each ADD model and are afterwards instantiated when actual ADDs are made. This kind of reuse is possible by taking advantage of the benefits of model-driven techniques and template engines; (3) finally, the use of compound

actions that can be extended and inherited increases reusability as it allows the introduction of new transformation actions with parameters which group other transformation actions. In the same way, loops also contribute to the reusability of transformation actions.

In order to show the reusability of our approach, we document the number of actions (primitive and compound), primitive actions and model actions that are needed per number of recurring ADDs and for four different ADDs of Table 10.4. Regarding the definition of the action templates of Table 10.4, the numbers of actions that have to be edited manually for the reusable decisions Direct Calls, Proxy, Adapter, and Integration Adapter, are four, two, three, and two actions, respectively. By defining compound actions, we can reduce the number of required actions in the last three cases where single actions were contained in the compound actions *add compound indirection* and *add compound integrationAdapter (extends indirection)*.

Table 10.6 shows the actions that need to be edited manually per decision – both compound and primitive actions – along with the primitive actions, as well as the model actions in which the primitive actions are translated from the enactment engine of the AK transformation language. The model actions are the actions that eventually apply on the C&C views. The number of the actions that are directly applied on the C&C model are 13, 28, 33, and 45 respectively for the four ADDs, which means that without the use of the AK transformation language the effort for editing and executing these actions would increase significantly. Clearly, primitive actions already scale much better in terms of modeling effort than manual change actions in models; reusable actions with compounds offer an additional level of support.

Reusable ADD	Actions (with compounds)	Primitive Actions	Model Actions
Direct Calls	4	4	13
Proxy	2	9	28
Adapter	3	10	33
Integration Adapter	2	14	45

TABLE 10.6: Comparison of number of actions for reusable ADDs

10.3.3 Modeling Effort for Reusable Transformation Actions

We performed a quantitative evaluation by measuring the modeling effort for editing transformation actions. In particular, we measured the increase of modeling effort if the compound actions in the AK transformation language are replaced by primitive or model actions. The results are presented in Table 10.7 for the ADDs we have documented in the warehouse case study. We notice that the modeling effort would increase up to 1100% and up to 3700% if the compound actions would be replaced by primitive or model actions respectively (i.e., for Result Callbacks). On average, we have an increase of modeling effort of 556% and 2161% for the aforementioned cases and for the setting we created for the needs of the case study.

ADD (Times Reused)	Actions (with compounds)	Primitive Actions	Model Actions	Average Increase of Modeling Effort *for Primitive Actions*	*for Model Actions*
Proxy (3)	6	27	84	350%	1300%
Adapter (1)	3	10	33	233%	1000%
Result Callback (7)	7	84	266	1100%	3700%
Facade (1)	3	8	24	167%	700%
Gateway (2)	2	12	30	500%	1400%
Publish-Subscriber (7)	7	56	189	700%	2600%
Data Mapper (2)	2	16	50	700%	2400%
Content Enricher (1)	1	8	25	700%	2400%
Total	31	221	701	556% (on average)	2161% (on average)

TABLE 10.7: Modeling effort for reusable ADDs

Table 10.8 summarizes the number of ADDs that were reused in the case study for an excerpt of the ADD model consisting of five reusable ADDs (Remote Proxy, Remote Adapter, Result Callback, Gateway, and Facade), along with the generated constraints, total number of provided fixes, and C&C view elements. We will use this excerpt for the following evaluations. Except for the actual ADDs that are made by software architects using tool support based on the ADvISE questionnaires, all other artifacts are automatically generated from the corresponding templates using model-driven techniques and a template engine.

Reusable ADD	ADDs	C&C Elements	Constraints	Fixes
Remote Proxy	3	21	24	84
Remote Adapter	1	7	8	28
Result Callback	7	42	84	280
Facade	1	10	12	43
Gateway	2	8	12	42
Total	14	88	140	477

TABLE 10.8: Exemplary reusable ADDs and generated artifacts

In the following subsections, we evaluate the efficiency and scalability for handling the inconsistencies between ADDs and C&C diagrams for different sizes of ADD models and C&C view models, and also different numbers of detected inconsistencies.

10.3.4 Efficiency and Scalability

The results documented in this section are based on the ADDs and C&C views that were captured in the context of our case study. We conducted all our measurements on a normal desktop machine, as our approach will usually run on the local machines of the software architects. The machine for testing had an Intel Quad Core i5 2.53GHz with 8GB of memory running Java VM 1.6 on Debian Linux. All measurements were performed 100 times and the resulting time was reported on average, as the deviations calculated were small. First, we measured the time needed for

validating the constraints checking the consistency of the corresponding C&C views with respect to the actual ADDs. The measurements were conducted with different versions of C&C models in different iterations of the development of the case study. In particular, C&C diagrams or parts of them were used to create the different C&C diagrams of Table 10.9. Afterwards, we validated the generated constraints with fixes for these models. The validation time increases linearly with the number of C&C view elements – at least for the C&C view sizes that we tested – and it remains very low (in a fragment of one second) even for large C&C diagrams, with more than 250 elements (models of such size are rarely used in practice).

ADDs	6	12	18	24	30	36	42	48	54	60
C&C Elements	24	48	72	96	120	144	168	192	216	240
Constraints	30	56	80	110	137	161	188	218	246	276
Validation time (ms)	54	65	73	88	102	114	141	161	185	214

TABLE 10.9: Constraint validation time

In addition, we measured the detected inconsistencies and the number of model updates that would be required for fixing these inconsistencies manually in various situations. Our aim was to estimate the manual effort that is saved by performing the fixes using our approach and accompanying tooling. To produce these inconsistencies we randomly modified (deleted, renamed, etc.) elements of the C&C view from our case study. Afterwards, we modified the related ADDs, and repeated our measurements. The results are presented in Table 10.10 and Table 10.11 respectively. In the first case, changing only 17% of the C&C view elements requires that 96 model updates have to be performed manually on the C&C view, while in the second case, changing half of the ADDs will require more than 100 model updates. Please note that these numbers are calculated as mean values based on the setting of Table 10.9 and can not be generalizable. They are documented, here, in order to give some indication about the number of inconsistencies that can be caused by changing the ADDs or the C&C views for designs of the size of our case study, as well as the effort for resolving all inconsistencies manually. From both tables we can conclude that even small or medium size changes, either on the ADDs or on the C&C views, lead to a significant amount of required model updates. Such model updates can be performed automatically in our approach.

C&C View Changes	% of all Elements	Inconsistencies	Model Updates Required
2	2%	6	13
5	6%	14	32
10	11%	28	64
15	17%	42	96
20	23%	56	128
25	28%	70	160
30	34%	84	192
44	50%	123	282

TABLE 10.10: C&C view modifications

ADD Changes	% of all ADDs	Inconsistencies	Model Updates Required
1	7%	7	14
2	14%	13	27
3	21%	20	41
4	29%	27	55
5	36%	34	69
6	43%	40	82
7	50%	47	96

TABLE 10.11: ADD modifications

10.3.5 Discussion and Limitations

Although we have implemented and demonstrated our proposal using specific tools, we claim that our approach can be generalizable to a certain extent. The transformation actions and constraint templates constitute reusable AK assets that can be customized and re-used in various reusable decisions. These templates can be applied for any existing ADD model or ADD documentation because the essential concepts and elements of these models and those in the ADvISE ADD model are almost equivalent. In most cases, the binding between the template variables and the elements of ADD models might need human intervention. That is, in order to properly associate a reusable parameterized action template containing some input parameters with a certain ADD, we need to align the parameters with the corresponding values in the ADD. This is similar to the way we pass parameters when invoking a function in traditional programming languages.

The C&C view that is created or updated by enacting the transformation actions contains all the information captured by the corresponding ADDs derived from the ADD meta-model. Nevertheless, the AK transformation language is generic and can be applied to similar C&C models or architectural views on different scenarios as well. Please note that the VbMF C&C view contains very similar elements to elements of other typical C&C views. Therefore, our approach is also applicable for most existing component models such as the UML component diagram with marginal effort for adapting the actions to accommodate new elements. Moreover, the notion of compound inheritance and extension would help to alleviate such adaptation effort.

There are many existing languages and techniques, for example, QVT[7], ATL[8], ETL[9], Xtend[10], to name but a few, that target generic and abstract concepts and elements in model transformations. They provide expressive languages and powerful transformation engines for both endogenous and exogenous model transformations [MV06]. Yet, these approaches provide no adequate abstractions and concepts for architecture-specific transformations. As a consequence, it will be inefficient to use such full-blown languages for architectural knowledge transformations as software architects

[7] http://www.omg.org/spec/QVT
[8] http://www.eclipse.org/atl
[9] http://www.eclipse.org/epsilon/doc/etl
[10] http://www.eclipse.org/xtend

must get familiar with several model transformation concepts instead of solely focusing on their particular domain of expertise. Therefore, we aim to bring the bests of both worlds by developing AK transformation language as a DSL in order to provide simple but succinct and comprehensible concepts and elements for describing and formulating the transformation of ADDs into design models in a reusable manner.

Regarding the manual steps of our approach, namely the editing of reusable ADD models and constraint templates with the mappings between them, many of the participating assets can be customized and reused in various design situations. The reusability and automation of our approach is based on reusable and recurring ADDs. Nevertheless, some non-reusable ADDs can also be integrated to the C&C views with small extra effort.

By implementing our proposal in a real-life design setting – in the context of the warehouse case study – we showed that it offers high reusability and reduces the efforts for documenting and maintaining two important design artifacts, the design decisions and C&C views. The introduced constraints with fixes also reduce the effort of maintaining the consistency between decisions and designs. Nevertheless, in the scope of our study, we have not conducted any usability studies within human designers, thus we were not able to assess the effectiveness and usefulness of our approach in the design process and in the long term. Lacking such an empirical evaluation, we are not able, for instance, to predict the time needed by software architects to learn and use the AK transformation language and from what number of reusable decisions the initial modeling effort pays off.

10.4 Conclusions

In this chapter, we presented an approach that provides reusable and extensible transformation actions and consistency checking rules for (semi-)automatically mapping the design rationale and knowledge reflected by ADDs onto architectural component models. In particular, our approach introduces the AK transformation language for specifying reusable actions that need to be enacted to automatically create or update the underlying architectural models with respect to particular ADDs. The transformation language provides basic actions for updating individual model elements, as well as expressive composite structures for describing actions applied in a set of elements such as compounds and loops. This enables us, for instance, to define recurring architectural primitives, e.g., to realize reusable specifications for architectural patterns or styles in the transformation language. In addition, our approach provides means for automatically detecting and proposing the resolution of inconsistencies to the software architect based on automatically generated constraints with fixes. These fixes can either be applied directly on the C&C view elements or provide feedback to the user for reconsidering an existing ADD.

Our evaluation in the context of an industrial case study on service-based platform integration illustrates the benefits of our approach in terms of potential modeling effort reduction and reusability. As discussed, the use of a template engine and model-driven techniques, as well as the support for inheritance and extension in the transformation language significantly enhance its reusability and extensibility. The tooling we introduced in our proposal can assist architects at inconsistency management (detection and resolution of inconsistencies) for documented architectural decisions and design views. Finally, the validation of constraints is scalable for large numbers of C&C view elements and constraints.

11 Interdependence & Integration of Variability and Architectural Design Decisions

11.1 Introduction

In software product line engineering, the design of assets for reuse and the derivation of software products entails low-level and high-level decision making. In this process, two major types of decisions must be addressed: variability decisions, i.e., decisions made as part of variability management, and ADDs, i.e., fundamental decisions to be made during the design of the architecture of the product line or the products. In practice, variability decisions often overlap with or influence ADDs. For instance, resolving a variability may enable or prevent some architectural options. This inherent interdependence has not been explicitly and systematically targeted in the literature yet, and therefore, is mainly resolved in an ad-hoc and informal manner today.

Variability management and architecture-centric development are fundamental aspects of software product line engineering [Lin+07]. Variability management aims at the explicit modeling of differences (variabilities) among the products that can be derived from a product line and, in particular, the interdependencies among individual variabilities. From a variability management perspective, the software architecture of a product line describes the design of all products in a product line in terms of reusable assets. This requires, first of all, that commonalities and variabilities among the different products of a given product line are identified. The aim of software product line engineering is to create a single architecture for a range of related products that can be tailored and customized to meet the requirements of the derivable products, for instance, imposed by different customers. This architecture is often called the *reference architecture* of the product line and may contain variabilities to represent the differences among the products [Lin+07]. Finally, each product has its own architecture derived from the reference architecture.

From an architectural perspective, variabilities may reflect different architectural options considered during the design of the product line that are independent of the products' features. Bachmann and Bass pointed out two causes of variability in the software architecture of a product line: (1) product line architectures encompass a collection of different alternatives that must be resolved

during product configuration and (2) at design time multiple alternatives may exist and need to be captured [BB01].

Currently, variability management and architectural design are mostly treated as separate activities. For their respective needs, product line and architecture communities use a variety of methods and tools for modeling, documenting, and making specific types of decisions. The product line community has mainly adopted feature models (e.g., [Kan+90]) and decision models (e.g., [Sch02]) to specify variability. The software architecture community has exploited techniques from variability modeling (such as COVAMOF [Sin+06]) and used architectural decision modeling to describe variabilities and connect them to QAs [Zdu07]. Existing variability management approaches focus on describing variabilities in a product line and managing their impact on the derived products. On the other hand, existing architectural design and decision support tools [Sha+09] cover various architectural aspects, views, and reasoning of decisions but lack adequate support for interaction with variability decisions. Our proposal intends to fill this gap and proposes the systematic integration of the two types of decisions.

Based on a motivating example, we will discuss the interdependence of variability decisions and architectural design decisions in the development of the product line and the derived products. We propose to systematically integrate these two types of decisions and suggest integrated tool support based on two existing tools: ADvISE – see Section 4.2 – and EASy-Producer[1] – a tool for variability management.

11.2 Background

In this section, we include a brief definition of product line engineering and provide a basic discussion of the dependencies between variability decisions and ADDs.

11.2.1 Product Line Engineering

To understand product line engineering it is important to notice a fundamental characteristic: the separation between development at the level of the product line as a whole and the level of an individual product; namely the *domain engineering* and the *application engineering*, respectively. Instead of domain and application engineering we will use the simplifying terms *product line level* and *product level*. At the product line level, all engineering activities that are relevant to a range of products – the product line – are performed. The logically first step is to determine what variability needs to be supported by the product line as a whole. As a basis for this, the scoping activity identifies which variability should actually be supported in a reusable manner [Sch02].

[1] http://sse.uni-hildesheim.de/en/fb4/institutes/ifi/software-systems-engineering-sse/research/projects/easy-producer/

Together with domain analysis, a precise model of the product line is derived from this. Variability decisions are captured in a variability (decision) model, which represents all variabilities (differences among the individual products) at an abstract level, as well as constraints among the individual variabilities. In later stages, the variability model is used to derive a valid product configuration for instantiation.

The reference architecture defines the realization of the product line, i.e., any decisions made in the reference architecture will be available in all products. The reference architecture may also contain variability in the sense that some ADDs are not finally taken for the product line as a whole, but several resolutions remain possible for the different variants. Thus, the fundamental potential ADDs are determined at the product line level. Besides, some ADDs may also be introduced as part of product-specific parts of the system. In the latter case, it is sometimes necessary to re-evaluate the existing variability. This may lead to introducing new variabilities to the variability model; likewise, the architecture needs to be extended.

11.2.2 Variability and Architectural Design Decisions

When creating reference architectures, a large number of decisions must be made, such as how to model a certain variability or which design pattern provides adequate support for covering particular variabilities. Other decisions are left open until product derivation time. We refer to the decisions taken as part of variability management activities as *variability decisions*. Further, we will call the decisions related to the software design of the architectures of the product line and the products *architectural design decisions (ADDs)*.

Conceptually, variability decisions and ADDs may pose distinctive as well as overlapping information. Variability decisions are any decisions that describe differences among products in a product line and are relevant to reuse (i.e., excluding product-specific aspects). Typically, variabilities are described in terms of optional (yes-no), alternative (one-out-of-many), or multiple (many-out-of-many) selections [Lin+07]. An ADD is the result of the evaluation of alternative design options in terms of architectural elements such as patterns, components, or connectors and the selection of the best-fitting solution and are considered both at product line and product level. At product line level ADDs can also be postponed to the product level. These open decisions become variabilities.

ADDs at different levels of granularity are usually taken in the early stages of design. Later, throughout the software development process, as well as the evolution of a product line, new ADDs may be considered and existing decisions might have to be revised. However, this is not a one-time process, as both engineering activities at the product line level may provide feedback to requirements engineering or variability management [CN02]. In particular, the variability supported by the product line has to be re-examined whenever new products are developed and introduce the

need for new variabilities. In general, there are two potential options for dealing with such feedback from later stages to variability modeling: (a) by using a single variability model that captures variability on all levels in a homogeneous way and (b) by using a so-called staged configuration approach in which multiple models are used, and information in one model is used to configure the subsequent level [Cza+05].

11.3 Related Work on the Interdependencies between Variability and Architectural Design Decisions

Variability and architectural design decisions have been studied often in the literature in the same context. Variability decisions mainly refer to decisions related to the differences among the products that derive from a product line. The variabilities described as optional, alternative, or multiple selections [Lin+07] are often related to architectural elements. For instance, the Feature-Oriented Domain Analysis (FODA) [Kan+90] usually mixes architecture-related decisions with domain properties and system configurations. Feature models mainly describe the solution space (i.e., focuses on modeling of commonalities and variabilities) and do not provide any guidance for selecting between alternative variants and reasoning about them. However, many approaches propose to enrich variability management with design rationale and decision support by introducing variability decision models [Sch02]. In an approach that combines both methods, Perovich et al. consider product line architectures as a set of architectural decisions, and use feature models to represent the decisions associated with the product features and transformation models to transform decisions into product architectures [Per+09]. The aforementioned approaches mainly focus on variability decisions and handle ADDs also as variability decisions without setting any boundary between the two.

Many approaches in the literature deal with modeling of reusable architectural decisions [Zim+07] or provide tool support for architectural decision making [Sha+09]. Unlike decision models for product lines that describe a set of variabilities relevant to product derivation, architectural decision models focus mainly on architecture-related options and alternatives for designing a software architecture. Some approaches suggest to integrate variability management with architectural knowledge and design rationale. For instance, Dhungana et al. suggest to capture variabilities in variability management as decisions and establish relationships between assets, such as components and decisions, explicitly [Dhu+07]. Also, Capilla and Babar suggest to integrate architectural decision models with variability models to support ADDs for product line architectures [CB08]. For this, they map design decisions to variabilities and binding times to document reasoning about decisions related to product lines. A number of approaches in software product line engineering focus on documenting ADDs and their rationale [CB08]. Unlike variability management approaches based on feature models [Kan+90] or decision models [Sch02], these approaches propose to view

ADDs in modeling and managing of product line variability models as first class citizens. For this, they distinguish between variations considered at product configuration and architectural decisions made at early stages of the design phase. As before, these approaches consider the design of product line and product architectures as an architectural decision making process and do not distinguish it from variability management. None of these approaches studies, elicits, and resolves the interdependence between variability and architectural design decisions.

11.4 Motivating example

As a motivating example for the proposed approach, we leverage a software product line for warehouse management systems. Based on the product line, custom-made software products for specific warehouses can be derived. The product line targets automated warehouses only; that is, goods in a warehouse are moved on pallets by conveyors or stapler cranes. Usually, a warehouse management system is accompanied by an Enterprise Resource Planning (ERP) system that handles all financial aspects of warehouse transactions and a base automation system that directly controls the conveyors and stapler cranes. This overall system architecture is typically layered, consisting of a Resource Planning Layer, a Warehouse Management Layer and a Basic Automation Layer (see Figure 11.1).

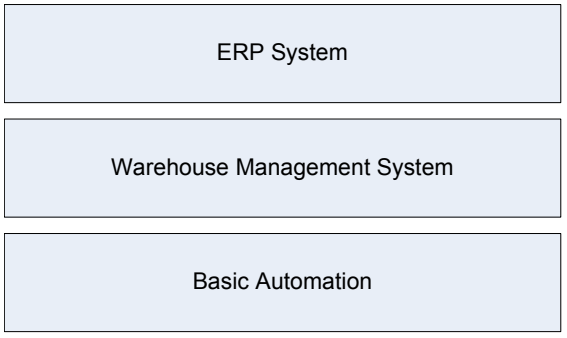


FIGURE 11.1: Architecture of a Warehouse Management System

The most important business process of a warehouse is "Order Processing" as illustrated in Figure 11.2. The process is triggered when a client orders some goods stored in the warehouse. Next, the ERP system notifies the warehouse management system about what shall be delivered ("Place Order"). The warehouse management system then maps the orders to boxes of goods that are stored in the warehouse management system ("Stock Determination"). Transport orders are sent to the base automation. The base automation will transport the boxes to a picking station ("Transport"). There, a human worker picks up the goods that are specified by the order ("Picking"). Finally, the goods are packed ("Packing") and sent to the customer ("Shipping").

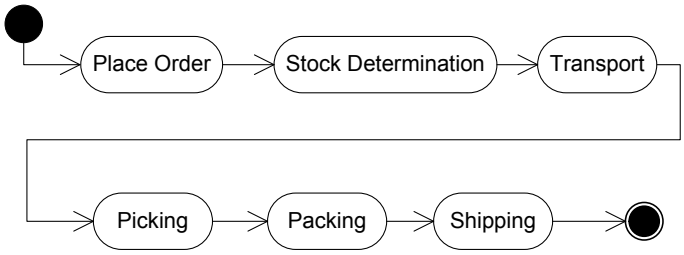


FIGURE 11.2: Goods out process of a warehouse

11.4.1 Variability Decisions

To create a product line of warehouse management systems, the variability of the domain must be managed. A selected set of variabilities (variability decisions, possible values, and binding times) is summarized in Table 11.1. The binding time is the latest point in the lifecycle when a variability decision at product level must be made. One of the variabilities we consider is the scale of the warehouse that can range from *high* (e.g., handling thousands of orders per day), *medium* (e.g., handling hundreds orders per day), and *low* (e.g., handling few dozen orders per day). Another variability concerns the strategy for handling partial pallet quantities that considers either speed or optimal reduction. The *high speed* strategy tries to only pick from a single box whilst it may possibly leave a lot of partial pallet quantities, while the *optimal reduction strategy* will remove as much partial pallet quantities as possible (thereby creating space in the warehouse), which leads to a higher amount of picks. The third variability represents different strategies for stapler cranes with one or several forks. The forth variability investigated in this example captures the user interface (UI) options including support for desktop/laptop computers or mobile devices.

ID	Variability Decision	Possible Values	Binding Time
VP1	Picking rate	High/Medium/Low	Design time
VP2	Partial pallet strategy	High speed/Optimal reduction	Runtime
VP3	Stapler crane strategy	Single fork/Multiple forks	Design time
VP4	UI device	Computer/Mobile device	Design time

TABLE 11.1: Selected variability decisions and their values in the Warehouse Product Line

11.4.2 Architectural Design Decisions

Table 11.2 presents a subset of the design space that provides the basis for making ADDs at product line level. In our motivating example, we prefer an asynchronous call-oriented to a message-oriented interaction style because it leads to less complex and more readable code (cf. **ADD1**); we prefer fix to variant interprocess communication (IPC) because fix IPC provides higher performance (cf. **ADD2**); and finally, a service-oriented to resource-oriented API style as the underlying infrastructure functionality is already provided in terms of services (cf. **ADD3**).

ID	Decision Point	Options
ADD1	Interaction Style	Asynchronous calls interaction Message-oriented interaction Synchronous calls interaction
ADD2	Interprocess Communication (IPC)	Fix Variant
ADD3	API Style	Service-oriented Object-oriented Resource-oriented

TABLE 11.2: Exemplary ADDs at product line level

Exemplary ADDs at the product level are shown in Table 11.3 along with the related variabilities. Some ADDs are influenced by the variabilities identified previously. For instance, we can not select a single interprocess communication solution for **ADD4** because of **VP1**. For *low* picking rate, the option *IPC open source* is sufficient, while for *medium* and *high* picking rates, *IPC very expensive* is necessary. This decision brings us to other subsequent ADDs. For instance, we need to decide if we will provide an abstraction interface for IPC invocations (cf. **ADD5**) and depending on decision **ADD4** we will create a wrapper component for IPC or even adapt its interface in order to support **VP1** (cf. **ADD6** and **ADD7**). Similar to the decision for an IPC solution, the deployment can not be decided until **VP1** is chosen. A single server is necessary for *low* picking rates but multiple servers with a round-robin strategy should be used with respect to *medium* picking rates. For *high* picking rates, multiple servers with load monitoring are needed. In our example, we decided to always use a client-side business delegate to identify the server, knowing that it is not necessary, and therefore, costly for single server deployment, in order to limit the variability of the architecture (cf. **ADD9**). In summary, the aforementioned ADDs are related to and influenced by the variation point **VP1** as follows:

- Low picking rate implies *IPC open source* (**ADD4**) and *Single server* (**ADD8**)
- Medium picking rate implies *IPC very expensive* (**ADD4**) and *Multiple server with round-robin* (**ADD8**)
- High picking rate implies *IPC very expensive* (**ADD4**) and *Multiple server with load monitoring* (**ADD8**)

ID	Decision Point	Options	Variability Decision
ADD4	IPC	IPC open source	VP1-low
		IPC medium price	-
		IPC very expensive	VP1-medium/high
ADD5	IPC invocations	No abstraction interface	-
		Abstraction interface (facade)	-
		Abstraction interface (gateway)	-
ADD6	IPC open source	Adapt IPC open source	-
		Create a wrapper component	-
ADD7	IPC very expensive	Create a wrapper component	-
		Other	-
ADD8	Deployment devices	Single server	VP1-low
		Multiple servers/round-robin	VP1-medium
		Multiple servers/load monitoring	VP1-high
ADD9	Server identification	Business delegate proxy	-
		Business delegate adapter	-

TABLE 11.3: Exemplary ADDs at product level

11.5 Proposed Approach for Integrating Variability with Architectural Design Decisions

Our approach is presented in the context of both variability and architectural decision making processes at product line and product level for designing reference architectures and product architectures respectively. In particular, we present the steps of variability management and architectural decision making and their relationships along with our solutions for eliciting the interdependencies among the two kinds of decisions.

11.5.1 Approach Overview

An overview of our approach is provided in Figure 11.3. First of all, the variability model at the product line level is derived from scoping and subsequent domain analysis. It leads through a manual process of derivation to a corresponding architectural decision model. As part of variability modeling, dependencies among variabilities are expressed using constraints, for instance, selecting a certain kind of warehouse restricts the range of applicable partial pallet handling strategies. To support the creation of the reference architecture, ADDs for the product line are identified in the architectural decision model. Our approach formally elicits the dependencies between the variabilities and ADDs in terms of mappings between the corresponding elements. At product line level, the resulting reference architecture will be designed to cover the whole range of variability specified by the variability decision model by selecting the appropriate architectural solutions from the architectural decision model.

At the product level, the variabilities of the product are resolved in order to obtain a valid configuration, i.e., all variability constraints are satisfied by the decisions made. Using the aforementioned formal mappings as input, we can automatically constraint the available architectural options that correspond to variability decisions made at product level. For example, if a specific variability option is chosen in the configuration, the resolution of the aforementioned predefined mappings ensure that only architectural options that are associated with that specific variability option can still be chosen in the architectural decision model. At this point, further ADDs can be made for the needs of the product architecture. This way, variability and architectural design decisions are kept consistent to each other and the product architecture conforms both to the variability and architectural constraints.

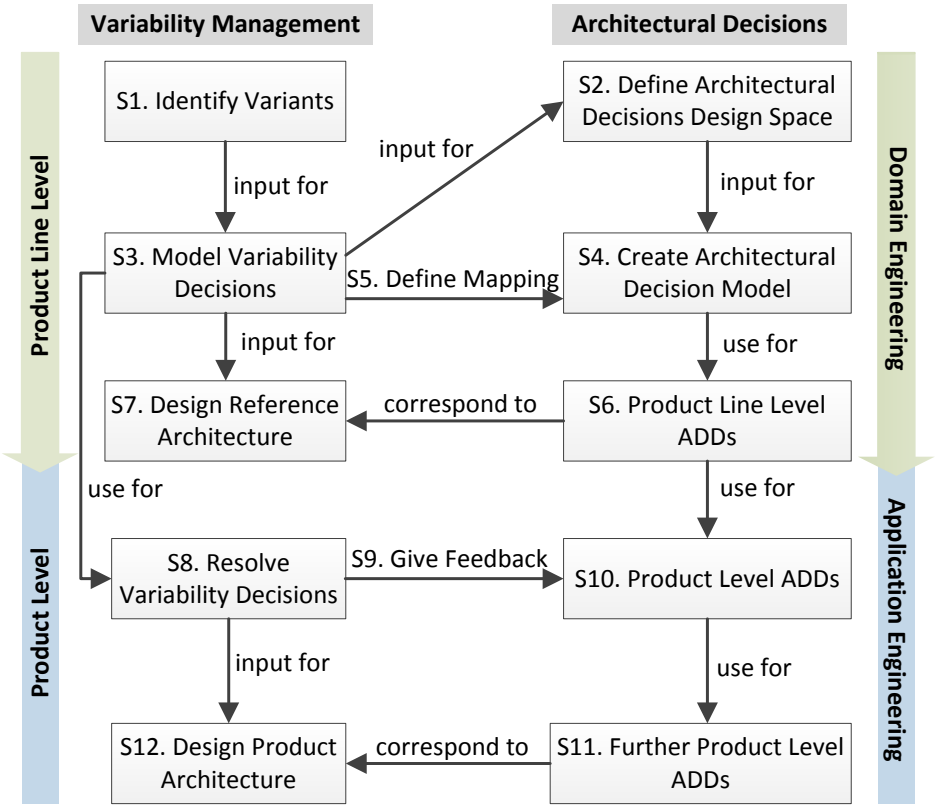


FIGURE 11.3: Approach overview

11.5.2 Product Line Level

The key idea of our approach is to first determine necessary variability decisions based on the requirements, as well as possible architectural solutions for implementing the required variability (also ADDs not related to variability). Then, the possible variability resolutions (variants) are mapped to the corresponding architectural options. This is realized in the following steps:

S1. Identify Variabilities: Based on scoping and an analysis of the requirements, we identify potential variabilities. This is often done by determining main features that are relevant to specific system instances [Sch02].

S2. Define Architectural Decisions Design Space: We consider existing architectural knowledge, such as reusable architectural models (e.g., [Zim+07]), to define the architectural decisions design space, i.e., architectural options and alternatives for the various decision points related to the design of the reference and product architectures. This information will be used as input for creating the architectural decision model.

S3. Model Variability Decisions: The individual variants that vary along an identifiable theme can be described as variability decisions. The advantage of having them as variability decisions – rather than using other variability modeling techniques such as feature modeling – is mostly that we make the inherent dimension of variability explicit[2].

S4. Create Architectural Decision Model: The analysis made in **S2** helps us define the architectural decision model that will be used as guidance for making ADDs at product line and product level.

S5. Define Mapping: The product line architects identify which variability decisions correspond to architecturally relevant requirements. They determine potential ADDs that correspond to the individual variants, and make this interdependencies explicit by introducing a mapping between the two models. For instance, a variability decision may exclude or enforce a related ADD.

S6. Product Line Level Architectural Design Decisions: The ADDs that will cover the desired variability are derived manually. The aim is to create a strategy that covers the whole range of variants described by a variability decision, considering the architectural alternatives and options provided by the architectural decision model. The ADDs at product line level are reflected in the reference architecture and can be reconsidered at product line evolution. Also, changing the product line architecture may lead to new architectural alternatives for the design of the products.

S7. Design Reference Architecture: The ADDs that are made to cover the whole range of variants implied in the variability decision model will be realized in the reference architecture.

11.5.3 Product Level

The major goal at the product level is to derive configurations based on the reference architecture to create particular products. The following steps are leveraged to accomplish a product architecture:

S8. Resolve Variability Decisions: Let us assume that, at the product line level, the kind of variability decisions and ADDs described in the previous section have been determined. We can

[2]Note that we will use a decision modeling approach [SJ04] as the tool that we will discuss later (EASy-Producer) is based on this approach. However, decision modeling and feature modeling are rather similar today and can even be equivalent for some cases [ES+12].

distinguish three situations for handling variability and architectural design decisions: (a) the variability identified at the product line level fits to the product level, thus we resolve the product line variability, (b) the variability determined at the product line level has not yet supported all product-relevant functionality, however, the additional functionality is only relevant to a single product, or (c) the variability identified in the previous step is insufficient and the needed variability is important for a range of products. The last case requires product line evolution [SE07]. Each situation will trigger the next steps for resolving the variability and architectural design decisions.

S9. Give Feedback, S10. Product Level Architectural Design Decisions: The case **S8(a)** is rather straightforward. In this case, fitting variability decisions have already been developed at the product line level. The decisions are taken and related to corresponding ADDs. Thus, selecting the variants constraints the ADDs through the mappings achieved in **S5**. If the variability decisions are sufficiently fine-grained, the ADDs can be automatically resolved. Otherwise, the ADDs are constrained and the architect performs a trade-off decision among the remaining cases. In both cases the binding time of the ADDs can be the same or later than the variability decisions binding time.

S11. Further Product Level Architectural Design Decisions: The case **S8(b)** is related to additional product-specific functionality that needs to be designed. Therefore, the variability decisions do not provide further orientation as this is outside the scope of the functionality supported by reusable assets (hence product-specific). This case is not fundamentally different from architecting a single-system. The only distinguishing point is that the existing ADDs have to be considered. This can be resolved automatically as constraints among decision points and architectural options are available in the architectural decision model. The case **S8(c)** denotes that, at the product level, the capabilities provided by the reusable assets (and hence the variabilities) are insufficient. There are two possible approaches for handling this circumstance. In the ideal case, we can go back to the product line level and evolve the product line infrastructure to cover the special case. This would entail augmenting the variability model providing (if needed) additional ADDs and establishing the relations between them. As a result, the steps from **S3** to **S7** are repeated and the variability decision model, architectural decision model, and their interdependencies are reconsidered. Afterwards, the rest can be achieved similarly to the first situation. Nevertheless, sometimes, especially if there is an urgent need for shipping the product, a different decision can be made: changes are made at the product level that might raise inconsistencies at the product line level. In this case, the product line level should be evolved or adapted at a later point in time.

S12. Design Product Architecture: The resulting product architecture based on the reference architecture will be created based on both variability decisions and product-related ADDs made in the previous stages.

11.6 Motivating Example Revisited

In this section, we present the implementation of the motivating example discussed in Section 11.4. For this, we have developed an integration of the tools EASy-Producer – for variability management – and ADvISE. The EASy-Producer tool aims at providing modeling and realization support for software product lines and software ecosystems. It provides capabilities that are standard to all product line engineering tools, like variability modeling and support for the configuration process by determining consistency and consequences of a partial configuration (e.g., some value may be derived based on constraints and other given values).

Decision Integration Tool Support

As discussed in Section 11.5.1, the first step in the tool support (cf. **S3**) is to represent the variability outlined in Table 11.1 (cf. **S1**) in the form of an EASy-Producer decision model. In our example (see Figure 11.4), four types of variability decisions are defined ("`PickingRateType`", "`PartialPalletStrategyType`", "`StaplerCraneStrategyType`", and "`UIDeviceType`") along with their resolutions (Lines 5–8). These types are used to define the variability decisions "`VP1`" to "`VP4`" of Table 11.1. The variability decisions "`VP1`", "`VP3`", and "`VP4`" will be bound at design time (Lines 15–16) and "`VP2`" at runtime (Lines 17–19).

```
1  project PL_WMS {
2
3      version v0;
4
5      enum PickingRateType {high, medium, low};
6      enum PartialPalletStrategyType {highSpeed, optimalReduction};
7      enum StaplerCraneStrategyType {single, multiple};
8      enum UIDeviceType {computer, mobile};
9
10     PickingRateType            VP1;
11     PartialPalletStrategyType  VP2;
12     StaplerCraneStrategyType   VP3;
13     UIDeviceType               VP4;
14
15     enum BindingTime {designTime, compileTime, initTime, runTime};
16     attribute BindingTime bindingTime = BindingTime.designTime to PL_WMS;
17     assign (bindingTime = BindingTime.runTime) to {
18        PartialPalletStrategyType  VP2;
19     }
20 }
```

FIGURE 11.4: Variability model of the WMS example modeled in EASy-Producer

Afterwards, we model the ADDs summarized in Table 11.2 and 11.3 (cf. **S2**) using ADvISE (cf. **S4**). Using ADvISE, for instance, we can model the options "`IPC open source`", "`IPC medium price`", and "`IPC very expensive`" as alternative options to the question "`IPC type of software`".

The next step of our approach (cf. **S5**) requires that we define the mapping between the variability and architectural design decisions. For this purpose, we establish a set of mappings from the variability decision model elements (i.e., variants) onto the corresponding architectural decision model elements (i.e., architectural options) in XML format. In particular, for each variability decision, relations of specific type to an architectural option (e.g., *excludes, enforces*, etc.) can be specified. In Listing 23, the variability decision "`PickingRateType.medium`" is mapped to the architectural option "`IPC open source`" of the ADD **ADD4** with type of relation "`excludes`". It means that selecting the *medium* picking rate will result in the rejection of *IPC open source*.

After designing the reference architecture to cover the whole range of variability (cf. **S6-S7**), it is now time to apply it to the problem of creating the product architecture. In step **S8**, we resolve the variability decisions by assigning values to the variability decisions. This is shown in Figure 11.5, where the lines 6–10 contain the decisions made for each variation point at design time. For instance, in our running example, the value "`PickingRate.medium`" was selected for "`VP1`". This case actually corresponds only to step **S8(a)**, thus the configuration is straightforward. If we have the case of **S8(b)** or **S8(c)**, the situation becomes more complex. An example of **S8(b)** would be that our customer requires an integration to a specialized ERP system, while the product line typically only supports SAP-integration. Then, a typical approach would be to introduce an extension point and a connector to this specialized ERP system. This would not necessarily be visible in the variability model or it would be simply modeled as an option to activate the extension point. An example for **S8(c)** would be that a new customer would like to have a Partial Pallet Strategy, which is not yet supported, like max two (i.e., at most two pallets may be opened for one type of article). In order to support this in the future we would extend the "`PartialPalletStrategyType`" to include the "`maxTwo`" option as well.

The configuration given in Figure 11.5 defines the product from a variability perspective. Based on the connections to the ADDs a number of ADDs can be automatically derived and further ones can be constrained. In our example, the mapping we defined in Listing 23 will enable us to reflect variability decisions on the architectural decision model at the product level design (cf. **S9-S10**). At architectural decision making (see Figure 11.6), the selection of the variant "`PickingRate-medium`"

```
<mappings>
  <aModel>ADModelIndenica</aModel>
  <vModel>PL_WMS</vModel>
  <vp name="VP1">
    <relation type="excludes">
      <vd id="PickingRateType.medium"/>
      <add id="ADD4.IPC type of software.IPC open source"/>
    </relation>
  </vp>
  ...
</mappings>
```

LISTING 23: Example of mapping between variability and architectural design decisions

will cause the option "`IPC open source`" in the related ADvISE questionnaire to be deactivated. At this point, further ADDs for the product architectural design, related to variability or not, can be made (cf. **S11-S12**).

```
1  project WMS_product {
2
3      version v0;
4      import PL_WMS;
5
6      VP1 = PickingRateType.medium;
7      /* partialPalletStrategy = PartialPalletStrategyType.high;
8                // This is not assigned, as this is only done at runtime */
9      VP3 = StaplerCraneStrategyType.single;
10     VP4 = UIDeviceType.computer;
11 }
```

FIGURE 11.5: Variability decisions in the WMS example

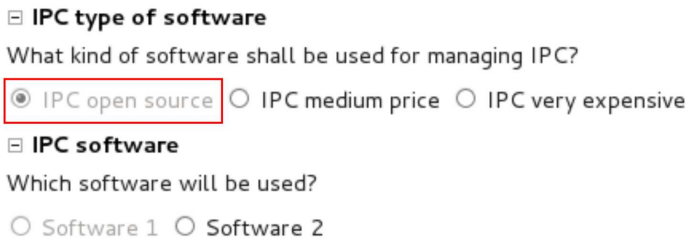


FIGURE 11.6: Architectural option deactivated due to variability decision

In our motivating example, we observed that in many cases, interdependencies between variability and architectural design decisions exist but are mainly kept implicit. Very often, these decisions are made by different stakeholders and with different tools and their overlaps and inconsistencies are resolved in an ad hoc and manual manner. We showed that formally eliciting the interdependencies requires additional efforts at the beginning but it leads to better automated support in integrating and harmonizing variability management and architectural decision making in the long run. By capturing and formalizing the links between variabilities and architectural options at the product line level, the architectural decision making process can be harmonized with the variability decision making process. As a result, ADDs can be changed or adapted whenever the variability is resolved. The advantage of our approach is that variability and architectural design decisions remain consistent at product derivation. Also, the introduction of mappings between the two kinds of decisions can significantly enrich the documentation of the design rationale. For instance, the rejection or selection of an architectural option can be justified by following the dependencies with the corresponding variability decisions.

In our approach, we assume that the variability decisions guide the ADDs for the product line and product design. In practice, an architectural decision may also influence a variability decision. For instance, a decision to use a low-cost software solution because of cost constraints may cause some variabilities to be invalid. However, that would mean that the variability needs to be reconsidered and possibly redesigned (i.e., repeat steps **S3** to **S7** in Figure 11.3).

We discussed the interaction of variability and architectural design decisions with the focus on product line design and product derivation but have not investigated the evolution and maintenance of product lines and products. As ADDs contain also interdependencies, reconsidering a variability decision may cause inconsistencies to existing ADDs. It is challenging to be able to handle this situation and also predict the impact variability decisions will have on the product architecture.

11.7 Conclusions

In this chapter, we discussed the interdependence of variability and architectural design decisions during product line and product design. Although variability decisions constraint and influence ADDs in practice, this inherent interdependence has not been studied or addressed systematically in the literature yet. We propose to make the interdependence of variability management and architectural decision making explicit and to manage variability and architectural design decisions in an integrated manner. To ensure that variability and architectural design decisions remain consistent to each other at product level, we introduce dependency mappings between them at the product line level, for instance, a specific variability decision may exclude or enforce a related architectural decision. These mappings are leveraged at the product derivation and give feedback – mainly introduce constraints – to the ADDs. In the context of a motivating example, we documented variability and architectural design decisions and their interdependencies and demonstrated our approach using EASy-Producer, ADvISE, and their integration.

Part V

Conclusions

12 Conclusions and Future Work

In this chapter, we revisit the research questions we have presented in Section 3.1 in order to summarize the main contributions of this dissertation. In addition, we discuss the limitations of our research work and introduce open challenges and our future work.

12.1 Research Questions Revisited

In Chapter 3, we introduced the main research problems this thesis is addressing and formulated five research questions that were subsequently elaborated in Chapters 4 to 11. In this section, we revisit the research questions in order to summarize our main contributions. In addition, Table 12.1 gives an overview of our contributions, the research questions they are related to, as well as the chapters where they have been presented.

Research Question 1: *Does the use of reusable AK in the form of reusable architectural decision models have a positive effect on the effectiveness and efficiency of architects during decision making and documentation?*

In order to explore this research question, we first introduced a reusable decision meta-model sharing common concepts with existing works on reusable AK. This decision model has been used as a basis for building the Eclipse-based tool ADvISE and the Web-based tool CoCoADvISE for supporting architectural decision making and documentation based on reusable ADDs. The reusable decision model and the two tools were afterwards validated in various empirical studies.

First of all, we employed multiple qualitative methods to explore the practice of architectural decision making in the domain of service-based platform integration. One important outcome of this multi-method study was a refined pattern language and a reusable architectural decision model on service-based platform integration. Apart from that, we proposed a model comprising four levels of decision making for platform integration, which was a result of our observations during an industrial case study on warehouse automation with practitioners. In the context of the European research project INDENICA, we modeled our reusable architectural decision model on service-based platform integration using ADvISE and used it to make several recurring ADDs.

Further, we performed two separate controlled experiments with software architecture students on architectural decision making and documentation based on reusable decision models. The experiments were designed to determine the supportive effect of reusable decision models for novice software architects, namely, their impact on a software architect's efficiency and effectiveness while making and documenting decisions. Our experiments provided strong evidence that reusable decision models can potentially increase both the effectiveness and the efficiency of their users while making and documenting ADDs.

Research Question 2: *To what extent do existing AK management methods and tools deal with the relationships between ADDs and QAs and how can reusable ADDs be integrated with QAs in order to provide quality-driven decision making support based on reusable AK?*

We have conducted a literature survey to study the current support of QAs in AK management methods and tools. This led us to some interesting findings and to the identification of future trends and challenges. In particular, we found evidence for three major QA-related topics in the primary studies on ADDs: (1) more than half of the primary studies provide support for QA trade-offs, (2) QA interdependencies and uncertainty are rather seldom supported, and (3) the maturity of the approaches with regard to evaluation of the QA support in AK management approaches is rather low (i.e., few empirical studies, limited design spaces and number of QAs exist). Therefore, to improve the state-of-the-art of the AK management methods and tools, more research is needed in the direction of automation support for and explicit focus on QAs and on more empirical evaluations.

Based on our findings for limited support of QAs in ADD tools and methods, we proposed to integrate QAs with reusable ADDs in order to support quality-driven architectural decision making. In this direction, our main contributions of our work were a reusable decision meta-model for modeling among others the impact of design solutions on QAs, as well as a tool prototype for supporting architectural decision making driven by QAs. The application of our approach in a large scale software ecosystem in the smart cities domain, namely, in Siemens' Aspern Smart City Project, showed that reusable architectural decisions enriched with QA support allow software architects to systematically consider all relevant QAs during architectural decision making.

Research Question 3: *How can the inherent uncertainty in architectural decision making be resolved – for both application-generic and application-specific knowledge?*

In order to address uncertainty in architectural decision making, we introduced a semi-automatic decision support method based on fuzzy logic. Our main contributions here are a domain-specific language for creating application-generic and technology-specific fuzzy models based on (pattern-based) reusable ADDs and corresponding Eclipse tooling for editing such models, providing requirements, and inferring appropriate design patterns and pattern implementations given these

requirements. Our approach provides aid for software architects for making and documenting ADDs that are expected to be repeated for similar design problems in specific contexts. The application of our approach in a motivating example and its evaluation showed that our DSL decreases complexity and development effort and seems to provide abstractions that fit well to the goal of supporting pattern-based decision making.

Research Question 4: *How can reusable ADDs be mapped to design views and how can consistency between reusable ADDs and design models be ensured over time?*

We have introduced an AK transformation language for specifying reusable actions, reflected by reusable ADDs, that need to be enacted on the corresponding architectural view models. Except for basic actions for updating individual model elements, the transformation language provides composite structures for describing sets of actions, e.g., to realize reusable specifications for architectural patterns or styles. Thus, we can translate ADDs into design views by mapping reusable ADDs to transformation action templates and derive and execute the corresponding transformation actions when concrete ADDs are made. In addition, our approach provides means for automatically generating constraints whenever transformation actions are executed for consistency checking between ADDs and architectural views. The validation of constraints leads to detecting and proposing the resolution of potential inconsistencies.

We have implemented our proposal in the case of C&C views and evaluated it in the context of an industrial case study on service-based platform integration. The main advantages of the AK transformation language are its reusability and extensibility. Our approach can significantly reduce potential modeling effort and provide automated support for software architects during the inconsistency management of documented architectural decisions and design views.

Research Question 5: *How can architectural and variability decisions be systematically integrated in the context of product line engineering?*

We have proposed to make the – until now implicit – interdependencies of variability management and architectural decision making explicit, in order to manage variability and architectural design decisions in an integrated way. For this, we have introduced dependency mappings between variability and architectural design decisions at product line level which are leveraged at product derivation to ensure that both kinds of decisions remain consistent to each other at product level. In addition, we provided tool support for our approach by integrating ADvISE with EASy-Producer – a tool for variability management.

By applying our proposal in a motivating example from the warehouse automation area, we observed that many kinds of interdependencies between variability and architectural design decisions exist but are mainly kept implicit and resolved in an ad-hoc manner. We showed that formally expressing these interdependencies, although it may initially require additional efforts, it can lead

to better automated support in integrating both kinds of decisions. The main advantage of our approach is that variability and architectural design decisions remain consistent at product derivation.

Research Question	**Contributions**	**Chapter**
RQ1 Does the use of reusable AK in the form of reusable architectural decision models have a positive effect on the effectiveness and efficiency of architects during decision making and documentation?	• Tool support (ADvISE and CoCoADvISE) for architectural decision making and documentation based on reusable decision models. • Pattern language and reusable architectural decision model for service-based platform integration. • Model comprising four levels of architectural decision making in the domain of platform integration. • Empirical evidence that reusable decision models can increase both effectiveness and efficiency of architects.	Chapter 4, 5, 6
RQ2 To what extent do existing AK management methods and tools deal with the relationships between ADDs and QAs and how can reusable ADDs be integrated with QAs in order to provide quality-driven decision making support based on reusable AK?	• Literature survey on the use of QAs in existing ADD tools and methods. • Reusable architectural decision meta-model including QAs and tool prototype for quality-driven decision making.	Chapter 7, 8
RQ3 How can the inherent uncertainty in architectural decision making be resolved – for both application-generic and application-specific knowledge?	• Approach for decision making support for reusable ADDs based on Fuzzy Logic; DSL for creating application-generic and technology-specific fuzzy models based on reusable ADDs. Inference of best-fitting solutions based on specific requirements. • Corresponding tool support.	Chapter 9
RQ4 How can reusable ADDs be mapped to design views and how can consistency between reusable ADDs and design models be ensured over time?	• AK transformation language for defining reusable actions (and action composites) reflecting reusable ADDs (mapping of decisions to designs), in order to translate ADDs into architectural view models. • Constraints for inconsistency management (detection and resolution) between ADDs and architectural views. • High reusability (by taking advantage of the model-driven paradigm and template languages) and extensibility of approach. • Tool prototype integrated with ADvISE and VbMF.	Chapter 10
RQ5 How can architectural and variability decisions be systematically integrated in the context of product line engineering?	• Dependency mappings between variability and architectural design decisions at product line level to ensure that both decisions remain consistent to each other at product level. • Integration of ADvISE with EASy-Producer (tool for variability management).	Chapter 11

Table 12.1: Overview of main contributions

12.2 Limitations

The discussion of the limitations of our research project has been included in the corresponding thesis chapters. We summarize here the points that constitute main limitations of the current thesis and indicate topics and directions for future work.

- We concentrate in our work on reusable AK, that is, reusable ADDs that provide solutions to recurring problems. Thus, creative architectural decision making is not considered in our approaches.
- The reusable decision models we have consolidated in the course of this thesis contain inevitably subjectivity. First of all, the same researchers have been involved in the consolidation and the reusable models. Secondly, these models have not been adopted in general by the software architecture community and have not been tested in many real-life design situations by practitioners working in different domains, companies, etc. However, the reusable architectural decision models we have introduced and evaluated in the context of this thesis are mainly based on reusable AK from the software architecture community.
- In addition, the controlled experiments to test the effectiveness and efficiency of software architects at architectural decision making based on reusable AK have been conducted with software architecture students with very little experience in the industry. In order to be able to consider our experiment results generalizable to stakeholders including industry experts, similar experiments need to be conducted with practitioners. This is part of our future work though.
- Our proposal to extend reusable decision models in order to support quality-driven architectural decision support does not consider some issues related to QA support like stakeholders' trade-offs, prioritization of QAs, and so on. Similar to our previous limitation, the extension of our reusable decision model and the corresponding tooling still need to be empirically validated by more practitioners and in other domains rather than the one we used for our demonstration (i.e., software ecosystems for smart cities). The same holds for our fuzzy logic based approach we introduced in order to resolve uncertainty during architectural decision making.
- Furthermore, translating reusable ADDs into designs has been demonstrated only for C&C views and need to be generalized for other architectural views. However, our approach can be easily extended to include other design views as well. Still our work lacks empirical evidence and feedback by practitioners with regard to the benefits of our approach for inconsistency management between decisions and designs in the long run.

- In general, integrating our proposals and accompanying tool support in software design and development processes should also guide our further steps. This has been rarely discussed in the current research work and should be a topic for future research.

12.3 Open Research Questions and Future Work

We claim that the work presented in this dissertation contributes significantly to the improvement of the current practice of architectural decision making and documentation based on reusable AK. In particular, it makes a step towards understanding, supporting, and automating software processes where ADDs are involved. During the research we conducted in the context of this thesis new research questions emerged that still remain open after this project. We conclude, thus, by summarizing these open challenges that outline our future and ongoing work:

- In the current thesis, a few reusable decision models were elaborated and in some cases evaluated in a few empirical studies. Essentially, part of our upcoming work is to model and use further reusable decision models for other aspects than, e.g., service-based platform integration and data management in the context of the smart cities software ecosystems. The goal of such reusable architectural decision models is to be used, updated, and configured among various projects, in similar domains, and even across development group, department, and company boundaries. These reusable decision models should be seen as a contribution to the software architecture community.

- As part of our ongoing future work, we will repeat our experiments with different reusable decision models and different types of participants (i.e., other than students). In particular, we strive for experimenting with practitioners to see if our assumptions and results are still valid in industrial contexts. At the same time, we plan to further investigate the educational aspect of reusable decision models. To that end we want to find out if and to what extent reusable decision models are conducive for novice software architects to learning how to systematically use patterns.

- Regarding our proposal to integrate QAs in reusable decision models (see Chapter 8), our QA-based decision model has been derived based on the software architectures and design principles of the smart city ICT platforms at Siemens. At this point, the model and the tool have been evaluated by core software architects of the individual projects. As the next step, the decision model(s) and the Web-based tool will be tested by an entire community of Siemens software architects. Also, as indicated by our literature survey, more efforts are needed in the direction of making QAs explicit in AK management tools, automating their support, and conducting more empirical studies in this domain.

- Our fuzzy logic based approach (see Chapter 9) leads to new challenges for the future. We are interested in studying how to modify the fuzzy decision models and fuzzy inference system in order to integrate inter-decision dependencies, which are very common in pattern-based decision making. Another open challenge is the automatic tuning of the expert system (i.e., the fuzzy rules and membership functions) by giving feedback from existing design decisions. Finally, we would also like to study the possibility of inferring generic knowledge from technology-specific knowledge.

- In our research agenda, we aim to generalize the harmonization of other architectural design views with architectural decisions using similar reusable AK transformations. Part of our future work is, thus, to consider other aspects of architectural designs apart from the elements of C&C models. Also, managing inconsistencies between decisions and designs in a systematic way (e.g., by also considering inconsistencies caused by consistency resolutions or transformations introducing inconsistencies) is an area that we consider very challenging. Of course, empirical studies are important here as well, in order to measure how much effort is needed by software architects that use tools for mapping ADDs to design views and to find out whether (and to what extent) architects can profit from our approach in the long run.

- Such interdependencies as the ones we defined between variability and architecture design decisions (see Chapter 11) need to be studied in various case study scenarios, be classified, and formalized. The systematic integration of architectural and variability decisions is important not only when creating reference architectures and derive products but also during the evolution and maintenance of product lines and products. We consider the latter to be an open challenge and plan to investigate different forms of integrating both kinds of decisions as well as the impact of changing these decisions during product line and product evolution.

Appendices

A Qualitative Multi-method Study Sources

Table A.1 summarizes the pattern collections considered during the systematic literature review performed to identify candidate patterns for a pattern language for service-based platform integration (see Section 5.4). In particular, it reports on:

- The authors and the titles of the pattern source
- The publication source
- The category the pattern collection falls into (i.e., single pattern, pattern compound, pattern story (sequence), pattern catalog, pattern language [Bus+07b])
- The year of publication (i.e., of the version selected)
- The number of patterns selected for inclusion into our pattern language and the number of patterns referenced in our pattern language (in brackets)
- The total number of patterns documented in the pattern source

Table A.2 contains the interview guide for the expert interviews reported in Section 5.5, consisting of 29 open-ended and closed-ended questions.

Table A.3 contains a list of reusable ADDs (only decision points and alternative options are documented) from the reusable architectural decision model for service-based platform integration, divided in four categories: Application Integration (AI), Interface Design (ID), Communication Style (CS), and Communication Flow (CF).

Authors	Title	Pub. Year	Source	Category[k]	Num. of patterns selected[l]	documented
Gamma et al. [Gam+94]	Design Patterns – Elements of Reusable Object-Oriented Software	1994	Addison-Wesley Book	PL	3 (1)[a]	23
Buschmann et al. [Bus+00]	Pattern-Oriented Software Architecture – A System of Patterns	2000	Wiley Book	PL	1 (1)[b]	16
Avgeriou and Zdun [AZ05]	Architectural Patterns Revisited – A Pattern Language	2005	EuroPLoP	PL	1 (1)[b]	24
Buschmann et al. [Bus+07a]	Pattern-Oriented Software Architecture – A Pattern Language for Distributed Computing	2007	Wiley Book	PL	3 (1)[c]	114
Fowler [Fow03]	Patterns of Enterprise Application Architecture	2003	Addison-Wesley Book	PL	4 (0)[d]	50
Hohpe and Woolf [HW04]	Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions	2004	Addison-Wesley Book	PL	13 (3)[e]	65
Völter et al. [Völ+05]	Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware	2005	Wiley Book	PL	5 (1)[f]	31
Daigneau [Dai12]	Service Design Patterns: fundamental design solutions for SOAP/WSDL and restful Web Services	2012	Addison-Wesley Book	PL	1 (2)[g]	28
Hentrich and Zdun [HZ12]	Process-Driven SOA: Patterns for Aligning Business and IT	2012	CRC Press Book	PL	1 (0)[h]	33
Schmidt et al. [Sch+00c]	Pattern-Oriented Software Architecture – Patterns for Concurrent and Networked Objects	2000	Wiley Book	PL	2 (0)[i]	17
Vogel [Vog01]	Service Abstraction Layer	2001	EuroPLoP	SP	1 (0)[j]	1

[a] ADAPTER^{IA}, FACADE^{ID}, PROXY^{IA}, OBJECT ADAPTER^{IA}

[b] PIPES AND FILTERS^{CF}, PUBLISH&SUBSCRIBE^{CS}

[c] REMOTE PROXY^{IA}, PUBLISH&SUBSCRIBE^{CS}, ASYNCHRONOUS COMPLETION TOKEN^{CS}, OBJECT ADAPTER^{IA}

[d] DATA TRANSFER OBJECT^{ID}, REMOTE FACADE^{ID}, GATEWAY^{ID}, DATA MAPPER^{CF}

[e] REMOTE PROCEDURE CALL^{CS}, REQUEST-REPLY^{CS}, CORRELATION IDENTIFIER^{CF}, MESSAGING^{CS}, MESSAGE^{CS}, MESSAGE CHANNEL^{CS}, POINT-TO-POINT CHANNEL^{CS}, PUBLISH-SUBSCRIBE CHANNEL^{CS}, MESSAGE ROUTER^{CF}, CONTENT-BASED ROUTER^{CF}, SPLITTER^{CF}, AGGREGATOR^{CF}, CONTENT FILTER^{CF}, CONTENT ENRICHER^{CF}, MESSAGE SEQUENCE^{CF}, MESSAGE TRANSLATOR^{CF}

[f] PROTOCOL PLUG-IN^{IA}, POLL OBJECT^{CS}, RESULT CALLBACK^{CS}, FIRE AND FORGET^{CS}, SYNC WITH SERVER^{CS}, MARSHALLER^{CF}

[g] REQUEST/ACKNOWLEDGE^{CS}, REQUEST/ACKNOWLEDGE/POLL^{CS}, REQUEST/ACKNOWLEDGE/CALLBACK^{CS}

[h] INTEGRATION ADAPTER^{IA}

[i] COMPONENT CONFIGURATOR^{IA}, EXTENSION INTERFACE^{ID}

[j] SERVICE ABSTRACTION LAYER^{ID}

[k] (SP) single pattern, (PL) pattern language

[l] (IA) Integration and Adaptation, (ID) Interface Design, (CS) Communication Style, (CF) Communication Flow

TABLE A.1: Pattern collections studied in the systematic literature review

A. Questions about service adaptation and service integration

1. Can the services from the source platform be directly used in the Virtual Service Platform (VSP)?
2. Can the same service interfaces, as provided by the remote source platform, be used for contracting the VSP services? Do the service interfaces need to be adapted in order to call the services from the VSP?
3. Is there a need for supporting additional functions such as access control, logging, or monitoring through the VSP?
4. Do you want the VSP to support the management of interface changes (e.g., by starting, stopping, suspending components)?
5. Is it tolerable to stop the system for maintenance? Is it tolerable to lose messages when the service adapters are updated?
6. Do you need to manage the service adapters in a controlled and centralized way?
7. Does the target platform need to support multiple remoting and transport protocols at the same time?

B. Questions about service interface design

8. Have the platform services been exposed as services using standard interfaces or standard technologies?
9. Is a domain-specific application interface needed?
10. Should any services be combined/integrated into a composite service by the VSP?
11. Are there platform clients requiring different kinds of transport channels (e.g., JMS, SOAP/HTTP, REST) for accessing the platform services? Is there a need for offering a common interface over these different channels?
12. Does the target platform provide a common interface for platform administration?
13. Must versioning and extending service interfaces be supported?
14. What is the expected format of the exchanged messages? Is there any common, canonical message format provided?

C. Questions about communication style

15. Does the platform support asynchronous communication (e.g., messaging, queuing)?
16. Can the platform service be invoked synchronously and/or asynchronously? Do we need to translate from asynchronous to synchronous calls and vice versa? Is there a batch mode? Are correlation identifiers supported by the platform?
17. Does the platform service expect a result or an acknowledgement from the platform client?
18. How are the messages emitted by the platform service delivered (e.g., using remote callbacks, local callbacks, or polling)? Do you want to use an event-driven or an imperative programming model in the VSP?
19. Must the messages be sent by the platform service in a reliable manner?
20. How critical is the message delivery performance from the perspective of the platform service?

D. Questions about communication flow

21. What kinds of data are/will be exchanged between the integrated platforms?
22. Which data characteristics are critical for routing the data to the correct receivers?
23. What kinds of data are sent by the source platform? What kinds of data are expected by the VSP?
24. Is it necessary to have the data, which is returned by the platform services, processed further before being delivered by the VSP?

E. Questions about platform technical details

25. Which programming language do you use to develop the platform services (e.g., Java, C#, Python, etc.)?
26. Which service-based framework do you use to develop the platform services (e.g., Axis, CXF, Metro, WCF/.NET, etc.)?
27. Which standards/technologies do you use to describe the service interfaces (e.g., WSDL)?
28. Which standard/technologies do you use to describe messages (e.g., XML, JSON)?
29. What is the transport protocol specified in the services (e.g., HTTP, JMS, SMTP)?

TABLE A.2: Interview guide

Decision Point	**Options**
$D1$ – Type of component for integrating the platform service into the integration platform	• None (direct calls) • PROXY or PROXY variant • ADAPTER or ADAPTER variant
$D2$ – Connection between service and integration platform (AI)	• Local PROXY or ADAPTER • Remote variant of PROXY or ADAPTER
$D3$ – Need for extra functionalities (such as monitoring, logging, etc.) (AI)	• Integration component forwards requests and replies • Integration component triggers extra functionalities and then forwards requests or replies
$D4$ – Connection of heterogeneous systems (in terms of synchronization and queuing behaviour) (AI)	• Plain PROXY or ADAPTER • INTEGRATION ADAPTER or variant of PROXY
$D5$ – Protocol(s) for accessing the platform (AI)	• One/Multiple fixed protocols • Support multiple protocols (PROTOCOL PLUGINS)
$D6$ – Runtime maintainability of the component (i.e., suspend, start, and stop the component) (AI)	• Use component with simple forwarding behavior • Use COMPONENT CONFIGURATOR; usually used together with input queuing as in MESSAGING; solution often employed in INTEGRATION ADAPTER
$D7$ – Accommodate diverse protocol requirements of integration services (ID)	• Canonical protocol; Several protocols • Multi-protocol support covers marshalling formats and/or transport protocols • Multiple protocols (extend the integration platform by a SERVICE ABSTRACTION LAYER and using PROTOCOL PLUG-INS)
$D8$ – Transfer multiple items of data to the remote platform (ID)	• Results from either optimizing remote invocations (batching) or the use of FACADE interfaces • REMOTE PROCEDURE CALLS: use DATA TRANSFER OBJECTS • MESSAGING: use pairs of AGGREGATORS and SPLITTERS, respectively • Use setters and getters
$D9$ – Decouple the client applications from the interface structure or the temporal consistency of the underlying platform services (ID)	• Functional (domain- or client-specific APIs) or temporal bindings (interface versioning) • Aggregating or splitting of service interfaces; or both • Temporal decoupling: explicitly bind client applications to an interface version • Intermediate interface scoped to a client/group of clients • Use FACADES to re-combine platform service interfaces. Temporal decoupling by EXTENSION INTERFACES.

Continued on next page

Table A.3 – continued from previous page

Decision Point	Options
$D10$ – Integrate the client applications and the platform services so that they can interact, while decoupling either end from the synchronization requirements of the other. Allow the integration platform to perform the necessary interface and data adaptations. (CS)	• Asynchronous forms of direct and explicit invocations (i.e., asynchronous REMOTE PROCEDURE CALLS) • Implicit and indirect invocations (i.e., MESSAGING)
$D11$ – Type of the invocations to the remote platform (CS)	• REMOTE PROCEDURE INVOCATIONS to realize synchronous invocations • Asynchronous invocation patterns to realize asynchronous communication • MESSAGING to realize asynchronous communication
$D12$ – One/multiple receivers for the invocation responses (CS)	• POINT-TO-POINT for unicasting • PUBLISH-SUBSCRIBER for broadcasting
$D13$ – Receive acknowledgement/result from the requests (CS)	• FIRE AND FORGET for one-way communication • SYNC WITH SERVER for acknowledgement only • For result either RESULT CALLBACK or POLL OBJECT; Alternatively MESSAGING • Synchronous invocations with/without result
$D14$ – Guaranteed/non-guaranteed delivery of the requests of the integrating platform (CS)	• Asynchronous communication variants • MESSAGING solution to keep requests persistent in queues
$D15$ – Translate from asynchronous to synchronous calls and vice versa (CF)	• Use CORRELATION IDENTIFIERS to keep the identities of the sent requests
$D16$ – Data transformations for the integrating platform/remote application (CF)	• SPLITTER • AGGREGATOR • CONTENT FILTER • CONTENT ENRICHER • DATA MAPPER (data exchanged in terms of in-memory objects) • MESSAGE TRANSLATOR to translate the messages from one format to another (data exchanged in terms of messages)
$D17$ – Filter data for delivering to different receivers (CF)	• MESSAGE ROUTER to filter incoming MESSAGES and republish them to an outgoing MESSAGE CHANNEL. To manage multiple reconfigurable MESSAGE ROUTERS use COMPONENT CONFIGURATOR. • CONTENT-BASED ROUTER (rules based on the content)

TABLE A.3: Overview of reusable architectural decision model for service-based platform integration

B Pattern Language for Service-based Platform Integration & Adaptation

B.1 Introduction

In the context of platform integration, the heterogeneity of service platforms with respect to their functional and non-functional properties often leads to several alternative ways of successfully tailoring, adapting, and integrating platforms. In other words, software architects and developers are usually confronted with numerous design decisions at different levels of abstraction and at different levels of granularity when designing a platform integration solution. For this, we introduce a pattern language targeting high-level and low-level design decisions for integrating and adapting platforms using services.

So far, a substantial amount of patterns have been described covering many aspects of service-based integration and adaptation. However, those patterns have been presented with a different focus, such as general software design [Gam+94], software architecture [Bus+00; HZ09], distributed system design [Bus+07a], enterprise application architecture [Fow03], messaging [HW04], remoting middleware [Völ+05], service-oriented systems [HZ09], service design [Dai12], and process-driven SOA [HZ12]. These existing patterns and design decisions are organized, in this appendix, in a comprehensive pattern language for the service-based integration and adaptation of platforms. This pattern language primarily addresses software architects who face the challenge to design, to realize, and to deploy a service-based integration architecture for software platforms – often owned by third-party vendors. As a secondary audience, developers of client applications, which are built from the integrated platforms, can consult this pattern language to evaluate the impact of the underlying integration architecture (and the design decisions embodied therein) on the observed non-functional properties (e.g., QoS properties such as execution timings, distributed exception state, etc.) observed for their applications.

B.2 The Challenge of Integrating Service-based Software Platforms

In a *service-based* integration platform, software platforms expose their interfaces in terms of services which provide the programming models for developing platform-based applications. Platform customization and configuration usually involve adaptations of service interfaces (e.g., interface aggregation) and/or service implementations (e.g., service specialization and substitution) as well as forms of service composition (e.g., batched service execution, service chaining, or process-driven service orchestration). Platforms are then integrated by applications via their exported *platform services*.

When looking at multiple service-based platforms, the exported services and their interfaces are heterogeneous in terms of the middleware technologies, the transport protocols, the programming languages, and the programming models (e.g., remote procedure calls, document-centric services) used. In addition, platform service interfaces change over time and platform services are substituted by others (e.g., service specialization, service substitution [Ruo+08]). Applications using platforms must cope with the heterogeneity of the platforms they integrate, as well as with interface changes. At the same time, (groups of) applications exhibit different requirements on the same set of platform services and may change in these requirements over time. For instance, applications might require functionally tailored interfaces (e.g., operation subsets, aggregated operations and aggregated operation data) and different interface capabilities for separate application groups based on their QoS requirements or on different authorization levels.

If this platform heterogeneity and the characteristic integration requirements were fully anticipated (for example, by analyzing the platform domains using a domain engineering approach), software platforms would be developed in terms of software product lines [Gha+12] and platform integration would turn into an issue of developing *multi software product lines* [RS10]. We are, however, interested in integration scenarios involving software platforms which were not necessarily designed as product lines and, most importantly, which were not designed to be integrated with one another (e.g., by using product line engineering techniques such as code generation or component weaving).

A strategy to deal with previously *unanticipated* platform integration is *service-based platform integration*. In such an integration strategy, applications should strive for programming towards stable, service-based interfaces that hide technology, protocol, language, and programming model dependencies as much as possible. Some platforms already offer a suitable service-based platform integration interface to their platform-driven applications, but in most cases such interfaces are not available (consider, e.g., legacy platforms). Service-based platform integration addresses situations like the one depicted on the left-hand side of Figure B.1: The client applications developed on top of a software platform have direct dependencies to the services offered by the platform. Service-based

platform integration changes this situation then to the one depicted on the right-hand side of the figure, with an intermediate abstraction layer hiding the details of the platform underneath. Thus, an application is developed on top of a service-based integration platform which integrates services from one or more platforms. With such an integration platform in place, client applications with changing requirements on the platform services, on the one hand, and platforms exposing changing platform services, on the other hand, can be effectively shielded from each other.

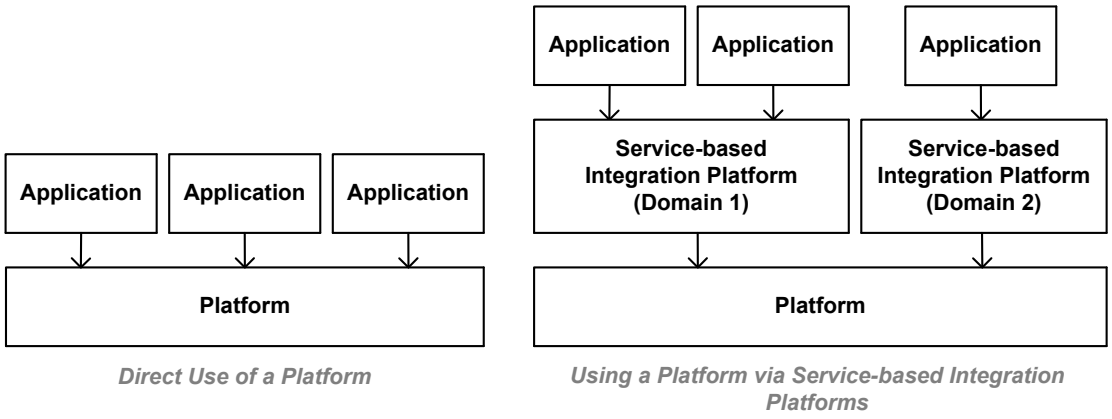


FIGURE B.1: Service-based platform integration

An intermediate platform has an additional benefit, as illustrated in Figure B.1: Applications from different domains can be programmed against different service-based integration platforms, each offering a domain-specific view on the platform abstractions. For instance, consider developing applications for Android. The integration platform could provide different views on the Android platform, with each view exposing selected Android API chunks for, e.g., business applications, Web applications, 2D and 3D games, and so on. The integration platform could provide a stable view over different versions of the Android SDK. Such a design also offers the advantage of rendering the underlying platforms exchangeable. For example, an abstraction layer can be provided in a way that similar script code can run both on Android and iOS. In this example, this is possible provided that a scripting engine running on both platforms is used.

In addition to the abstraction of platforms and their interfaces, we also consider the problem of integrating multiple platforms and their platform services into one and the same application. This problem is schematically illustrated in Figure B.2. Consider an online shopping portal that relies on an enterprise resource planning (ERP) platform such as SAP R/3 for receiving and processing purchase orders and a warehouse system for managing large inventories of goods. In a service-based integration platform, we could select only those services of the two platforms that are relevant to the online shopping portal and present them in an integrated fashion. We could offer them through the different communication channels needed for the online shopping portal, but not offered by the legacy platforms. Internally, the service-based integration platform would perform all necessary tasks of integrating, adapting, and routing messages for the two backend platforms.

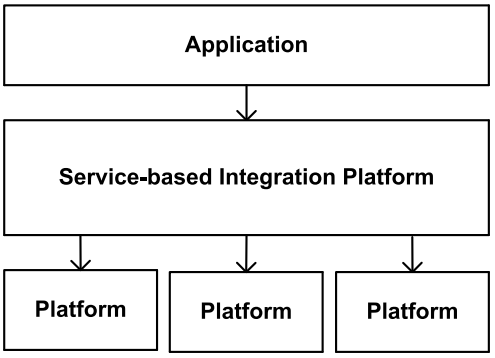


FIGURE B.2: Service-based platform integration for multiple platforms

B.3 Pattern Language Overview

In this section, we describe the pattern language for service-based platform integration and adaptation. As mentioned before, this pattern language documents interconnected design decisions by drawing from existing pattern material [Gam+94; Bus+00; HZ09; Bus+07a; Fow03; HW04; Völ+05; HZ09; Dai12; HZ12]. We introduce an overview of the main categories of design decisions documented in our pattern language in Figure B.3. The direction of the arrows implies follow-on decisions. Arrows in both directions imply that the decisions can be made in parallel. In our pattern language, we consider the following architectural decision categories: *Integration and Adaptation*, *Interface Design*, *Communication Style*, and *Communication Flow*.

The *Integration and Adaptation* category collects design decisions regarding the integration of platform services into a service-based integration platform and their interface and protocol adaptation.

The *Interface Design* category mainly covers design decisions regarding the design of the exported interface(s) of the service-based integration platform. Decisions in the categories *Integration and Adaptation* can be performed in parallel to decisions in the *Interface Design* category. These categories mainly concern developing components and interfaces for connecting applications, platforms, and the service-based integration platform.

The *Communication Style* category comprises design decisions that must be taken for each distributed component connection. These decisions relate to options for connecting two components (e.g., blocking and non-blocking component interaction styles). These decisions reside at a lower level of abstraction than the decisions of the two previous categories.

The *Communication Flow* category describes additional decisions that must be considered in case the service-based integration platform introduces more complex communication flows than simple forwarding from an exported interface to imported interfaces. For instance, such decisions relate to handling the aggregation or the splitting of the messages on their way through a service-based integration platform.

The patterns in each category are documented in the form *Problem – Solution – Decision Drivers*. These pattern sketches are based on the information extracted from the aforementioned design pattern material.

In the following sections, the four pattern-language categories are discussed in more detail and visualizations of the characteristic decision flows in each category are provided. The decision flows document relevant patterns and their interconnections, as well as the follow-on pattern category or categories to be considered. The decision flows and the related pattern sketches will help software architects and developers to structure their own decision-making processes by highlighting characteristic decision steps and by presenting pattern alternatives, pattern variants and pattern compounds. With this, we aim at offering a guideline on how to lay out a concrete decision-making process rather than presenting a ready-made "recipe" for selecting patterns in service-based platform integration.

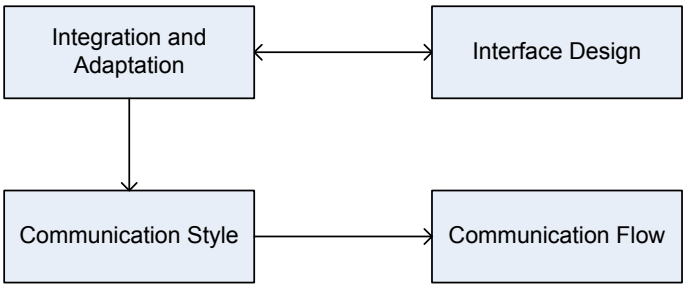


FIGURE B.3: Overview of pattern language for service-based platform integration

B.4 Integration and Adaptation

The simplest case of integrating a platform into an application is to directly invoke the platform services from the application code. However, often we would like to avoid direct invocations in order to support abstraction or stable interfaces as motivated in Section B.2. In addition, often simple direct invocations are not enough, as the integration logic should introduce extra functionality, such as logging, monitoring, indirecting, or adapting the platform access. Such situations are discussed in this section. For the case that the interfaces offered by the platform are compatible to each other and that the extra functionality needed does not change the invocation flow, the PROXY pattern [Gam+94] can be used to indirect the service invocations, to perform additional tasks on the invocation data, to select and to access the actual platform services. If extra functionality such as logging, monitoring, or access control is needed and this does not change the invocation flow, the functionality can be handled using a PROXY between the platform services and the application.

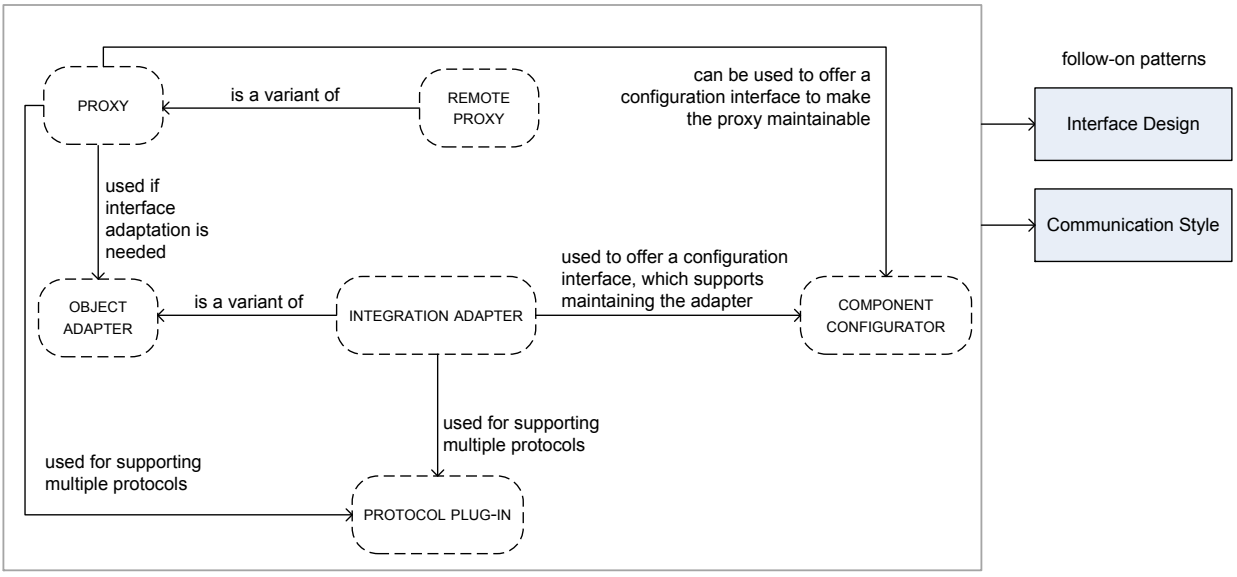


FIGURE B.4: Platform integration and adaptation patterns

Direct invocations vs proxy-based platform integration are illustrated in Figure B.5. When using direct invocations the platform services are called directly from the application, thus the application is tightly coupled with the platform interfaces. In the second schematic example, the PROXIES introduce extra functionality for monitoring the invocation flow from the application to the platform. From the viewpoint of the application, they essentially introduce a new platform abstraction, the *service-based integration platform*.

PROXY [Gam+94]

Problem There are situations in which a client does not or can not access a platform service directly, but wants still to interact with it. A surrogate or placeholder for an object to control the access to the service is needed.

Solution A PROXY acts as the intermediary between the client and the target. The PROXY has the same interface as the target. The PROXY holds a reference to the target and can forward requests to the target.

Decision Drivers A PROXY is used whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer. It introduces a level of indirection when accessing an object. It also supports creation of objects on demand. In the context of platform integration, it can be used to introduce extra functionality or control, but it does not change the interface of the invoked service or the invocation flow.

In many cases, applications and platforms are residing in different process or machine contexts. Hence, invocations must cross the process or machine boundary. In such cases, we can apply the

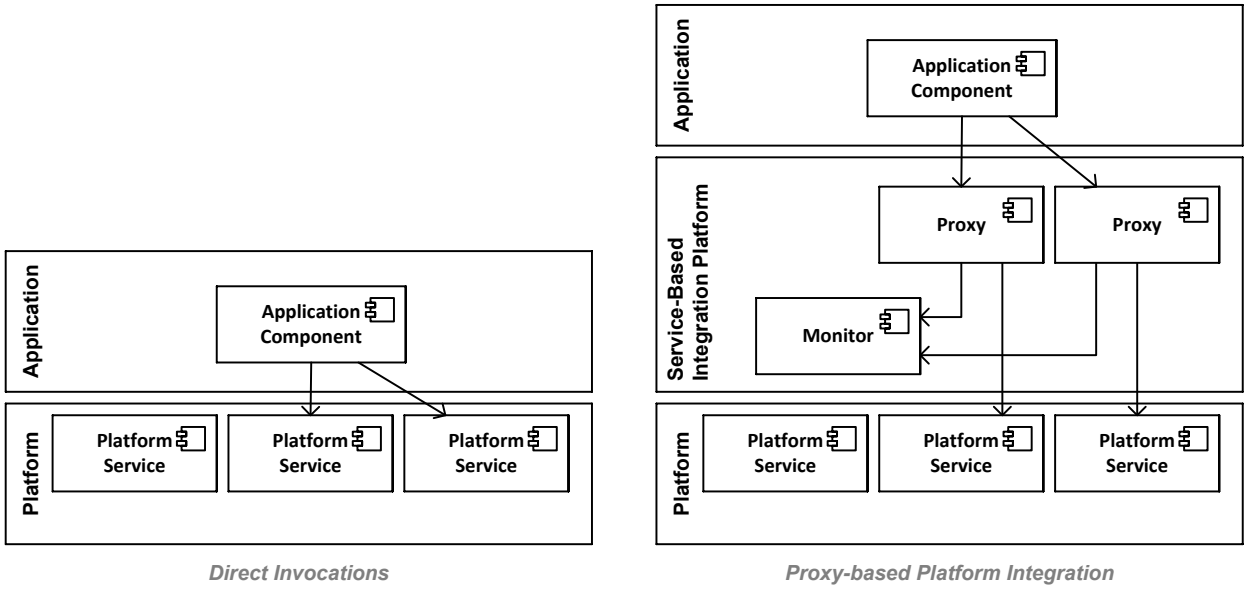


FIGURE B.5: Direct invocations vs proxy-based platform integration

remote variant of the PROXY pattern, the REMOTE PROXY [Sch+00a; Bus+07a]. In the platform integration context, the REMOTE PROXY resides in the service-based integration platform and connects application and platform. The schematic illustration on the right hand side of Figure B.5 also applies to REMOTE PROXIES, but the arrows depict remote invocations instead of local invocations.

REMOTE PROXY [Sch+00a; Bus+07a]

Problem As in the PROXY pattern, we need to access an object through a placeholder for another object to control access to it. In addition, the object and its client are residing in different process or machine contexts.

Solution A REMOTE PROXY is a PROXY that connects two objects in different process or machine contexts. Usually, communication middleware is used to cross the process or machine boundary.

Decision Drivers The REMOTE PROXY has the same decision drivers as the PROXY pattern plus the need for integration of distributed applications and platforms.

In addition to simple integration, service-based platform integration requires coping with the diversity of the interfaces that these platforms expose. Calling a remote interface directly or through a PROXY is not always possible, for instance, because the interfaces offered by a platform may not offer exactly what the calling application expects. Using the original interface might be possible, but we need to take into account that usually the applications are tightly coupled with their interfaces and implementations. Changing the interfaces of a platform is a possible solution. But,

firstly, an interface change is tedious and error-prone, and, secondly, most often it is not possible at all because many platforms that need to be integrated are provided by third parties. In addition, platforms are typically used by many applications and it is usually not possible to offer a different interface for each of them. For these reasons, an ADAPTER [Gam+94] can be inserted between the caller and the remote interface that converts the provided interface into the interface that the caller expects and vice versa. The adapter also transforms the data returned by the adaptee into the data structures expected by the caller. For distributed systems two variants of the ADAPTER pattern, the OBJECT ADAPTER [Gam+94; Bus+07a] and the INTEGRATION ADAPTER [HZ12], can be used to connect the interfaces and to perform the appropriate transformations.

OBJECT ADAPTER [Gam+94; Bus+07a]

Problem A class or component offers an interface, but the interface does not match the one that is needed by a client. We need to resolve the interface incompatibility.

Solution An OBJECT ADAPTER converts the interface of a class or component into another interface clients expect. The ADAPTER lets classes or components work together that could not otherwise because of incompatible interfaces.

Decision Drivers Interface adaptation lets us incorporate our classes or components into existing systems that might expect different interfaces. The amount of work for creating an ADAPTER depends on the similarity between the adaptee and target interfaces.

From a high-level perspective, OBJECT ADAPTERS usually have a similar structure as the PROXY example depicted in Figure B.5. The ADAPTERS would simply replace the PROXIES and introduce the additional interface adaptation behavior. Very often new versions of platforms come with new versions of interfaces. This can be hidden from the applications using the interfaces by exchanging the OBJECT ADAPTER. However, the more complex the mapping between the interfaces is, the more expensive is the mapping in terms of performance and development effort. A general problem of components like OBJECT ADAPTERS in platform integration scenarios is that invocations reaching the ADAPTER while it is being maintained (i.e., stopped and redeployed) would get lost. In many cases, this is highly undesirable. This problem is addressed by an extension of the ADAPTER pattern, the INTEGRATION ADAPTER [HZ12] pattern.

An important part of the INTEGRATION ADAPTER pattern is its use of the COMPONENT CONFIGURATOR pattern [Sch+00c] to stop, suspend, and start the adapter component during the process lifetime of the integration platform. This pattern can also be used to make other integration solutions, like the PROXY based solutions discussed before, configurable at runtime. This form of runtime adaptability complements other configuration techniques available at the deployment

time (e.g., deployment descriptors) and at the runtime of the integration platform (e.g., invocation interceptors for the middleware framework).

INTEGRATION ADAPTER [HZ12]

Problem Heterogeneous systems need to be connected and we need to shield the client from the impact of system and system interface changes. The calling and called interfaces might change over time and maintenance activities should not cause invocations or messages to get lost.

Solution The INTEGRATION ADAPTER contains two connectors: one for the client system's import interface and one for the target system's export interface. It plays the role of the translator between the heterogeneous systems and for their different interfaces, protocols, technologies and synchronization mechanisms. The adapter can be made configurable at runtime by using the COMPONENT CONFIGURATOR pattern, so that the adapter can be modified without affecting the requests to the adapter. A COMPONENT CONFIGURATOR offers a configuration interface for stopping, suspending and starting adapters. When new versions of the adapter must be deployed the adapter is stopped. When new versions of the target system are deployed or the adapter is configured at runtime the adapter is suspended. After the maintenance activities the adapter can process all requests that have arrived in the meantime.

Decision Drivers The INTEGRATION ADAPTER provides flexible integration for applications from external vendors. Generic adapters can be offered to support interconnectivity via common standards (e.g., Web services). A drawback of INTEGRATION ADAPTERS is that if many adapters from different systems exist, they need to be managed in a centralized way.

We illustrate in Figure B.6 a potential INTEGRATION ADAPTER design. The INTEGRATION ADAPTER implements a configurable component interface to realize the COMPONENT CONFIGURATOR pattern. To avoid losing messages while the adapter is being maintained, the INTEGRATION ADAPTER has an asynchronous messaging interface to the client, which queues up messages until the maintenance actions are performed (see the discussion of MESSAGING in Section B.6). The integrated platform is connected via a synchronous connector. The adapter also performs the translation from asynchronous calls to synchronous calls (see the discussion of CORRELATION IDENTIFIER in Section B.7).

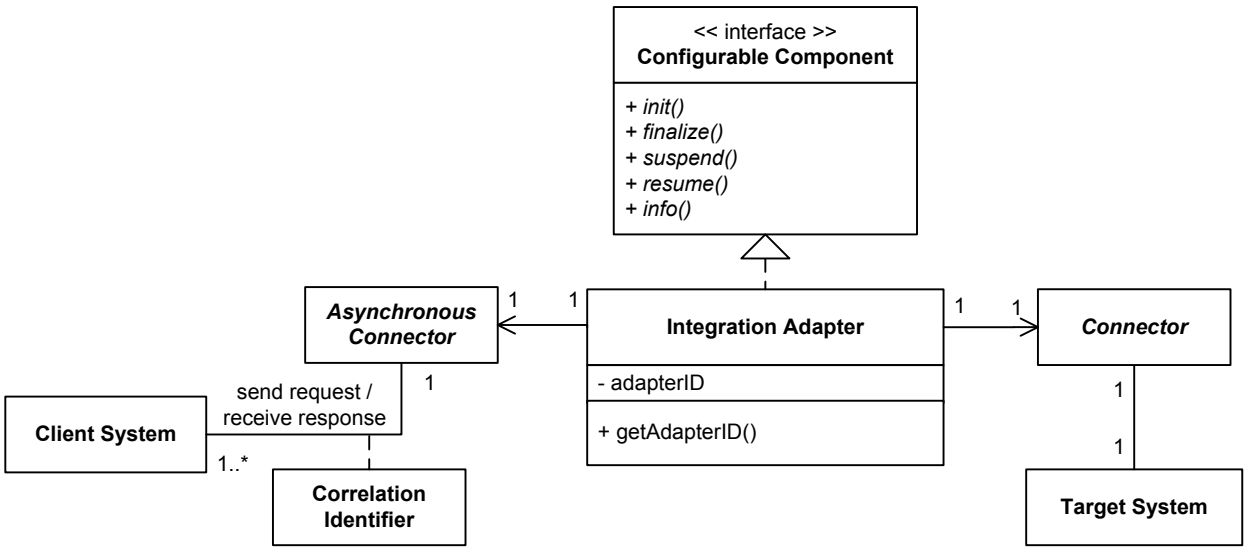


FIGURE B.6: Integration adapter: example design

COMPONENT CONFIGURATOR [Sch+00c]

Problem The application must provide a mechanism to configure components at any time of the application lifecycle. The components should be initiated, suspended, resumed, terminated, or exchanged dynamically at runtime without having any impact on the rest of the application.

Solution The component interfaces are decoupled from their implementation and used from the application to dynamically control the components. Concrete components implement these interfaces in an application-specific manner.

Decision Drivers COMPONENT CONFIGURATOR offers a common interface for the administration of components (initialize, suspend, resume, and terminate). The implementation of the components is decoupled from their configuration, thus increasing modularity and reuse. Configuration and reconfiguration of components can be performed dynamically. This pattern increases also the range of configuration alternatives. However, it has the liability of a lack of determinism, since the behavior of an application is not determined until its components are configured at runtime and a potentially lowered reliability, since the dynamically configured components can affect the execution of other components. Also, the dynamic linking adds extra levels of indirection to invocations.

When the service-based integration platform must bridge between different communication protocols, PROTOCOL PLUG-INS [Völ+05] can be used to realize translation between the different protocols.

PROTOCOL PLUG-IN [Völ+05]

Problem A distributed application must support multiple communication protocols at the same time. The protocols used for the scope of, e.g., single requests or different clients should be configurable at runtime.

Solution PROTOCOL PLUG-INS contain implementation details at the communication-protocol layer and provide common interfaces for different communication protocols. For the configuration of their parameters, the PROTOCOL PLUG-INS offer either an API or a configuration file.

Decision Drivers PROTOCOL PLUG-INS abstract from communication protocol details and allow flexible support for several communication protocols. They can also allow configuration and optimization of the communication protocols used.

B.5 Interface Design

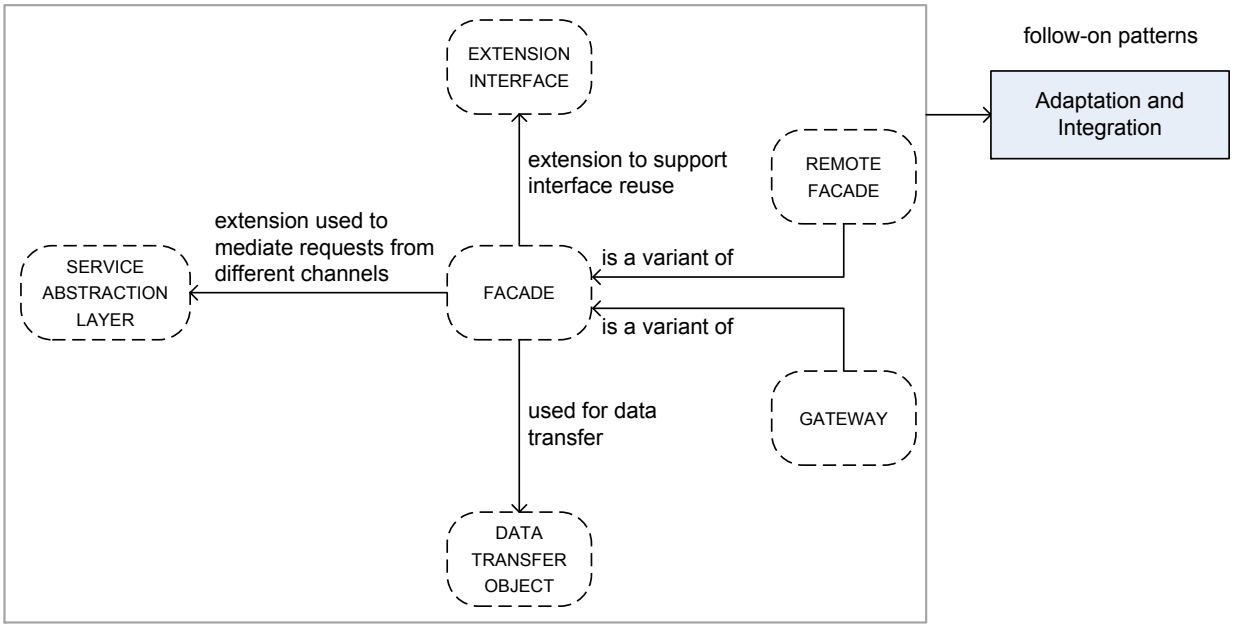


FIGURE B.7: Interface design

When developing a service-based integration platform, we need to expose interfaces to the application. Through these interfaces the applications built on top of the integration platform will be able to invoke the remote platform services. Common interfaces are also needed for monitoring, adaptation, etc. In the simplest case, we can simply expose the PROXIES and ADAPTERS, as discussed in the previous section. However, often we are faced with additional interface design

requirements, such as the unification of or the abstraction from interfaces, supporting different protocols or channels, optimizing invocation flows, avoiding redundancies in interfaces, or supporting multiple interface versions.

We might have the same requirements for one or more of the platforms to be integrated. For instance, many legacy applications do not expose an appropriate service-based interface. Sometimes it is more appropriate to first craft an appropriate service-based interface design for each of the platforms, and then to develop a service-based integration platform that offers a unified interface. When designing interfaces for platforms or integration platforms, the design of the data transfer might be an important concern. Transferring data over the network between two distributed applications can be very expensive when the number of calls increases. Therefore, we can use DATA TRANSFER OBJECTS [Fow03] which hold all the data to be sent. A DATA TRANSFER OBJECT transfers the needed information within a single call. DATA TRANSFER OBJECTS may wrap primitive data types (e.g., integers, strings) or other DATA TRANSFER OBJECTS.

DATA TRANSFER OBJECT [Fow03]

Problem When a remote interface is designed akin to a local interface, often many invocations with small sets of data are sent. This can get very expensive in terms of bandwidth use and processing of calls. Also, it leads to cluttered and unstable interfaces.

Solution DATA TRANSFER OBJECTS can be used to group the data that needs to be sent in one object. Often multiple invocations transmitting only small sets of data can be replaced by a single invocation transmitting the DATA TRANSFER OBJECT.

Decision Drivers The use of DATA TRANSFER OBJECTS makes interfaces more stable, as for many changes only the DATA TRANSFER OBJECT needs to be changed and the interface can remain the same. DATA TRANSFER OBJECTS can lead to more efficient use of bandwidth and to a reduced overhead incurred by invocation processing. DATA TRANSFER OBJECTS need to be serialized before being sent. The choice of the serialization form is critical for the performance of the transmission. Using textual instead of binary data consumes more bandwidth and can potentially lead to a significant performance penalty. Using binary serialization can introduce fragility into the communication lines. In contrast, XML serialization can be more tolerant towards changes.

From the viewpoint of the client of a platform, interface unification is often important. Platforms expose multiple interfaces, often in multiple versions. The interfaces exposed by the platforms are often not the interfaces required by the applications using the platforms. The FACADE pattern [Gam+94] describes a general way to unify interfaces. A FACADE [Gam+94] provides a coarse-grained interface on fine-grained components. In distributed systems, a REMOTE FACADE [Fow03]

can be used to specify a single point of access for a group of components which provide complex services in order to mediate client requests to the appropriate components. A REMOTE FACADE can also aggregate features of different components into new and/or higher-level services. It does not contain any domain logic and can use data from DATA TRANSFER OBJECTS. Using bulk accessors for the data ensures that invoking the remote interface remains efficient.

FACADE [Gam+94]

Problem Clients require stable interfaces over multiple versions of a component and multiple, heterogeneous components. Clients might require client-specific views on a component exposed as a stable interface.

Solution A FACADE is an object or component that provides a unified and often simplified interface for a set of software components. The FACADE or FACADES of a system are usually not bypassed by clients.

Decision Drivers FACADE can be applied when multiple interfaces should be unified. Interface unification, however, comes at a price. Often only the common denominators of interfaces can be offered (e.g., in case multiple versions of an interface need to be unified in a FACADE), or unification of overlapping functionality can be difficult to reach. Introducing an additional FACADE means to introduce an indirection that on the one hand costs performance, but on the other hand provides an additional point of control in the message flow.

REMOTE FACADE [Fow03]

Problem Interaction between objects is better understood when small objects have small methods, which leads to a fine-grained behavior. However, using fine-grained interactions when making calls between process or machine boundaries can be very expensive in terms of performance. Any object that is intended to be used as a remote object needs a coarse-grained interface to minimize the number of calls needed for a process.

Solution A REMOTE FACADE translates the coarse-grained methods onto the underlying fine-grained objects. Thus, it separates distinct responsibilities into different objects. A bulk accessor is used to replace a number of getters and setters of the underlying objects with one getter and setter.

Decision Drivers REMOTE FACADE provides access to a fine-grained object with a coarse-grained interface. Using a coarse-grained object model improves performance because of the reduced number of calls. It adds, however, additional programming effort, as the remote calls have to be translated into smaller internal calls.

A GATEWAY [Fow03] is another variant of FACADE that represents an access point to an external system used by an application. The application thus becomes independent from the specific interfaces of the external system and also from its internal structure.

GATEWAY [Fow03]

Problem Complex interfaces lead to complicated applications. When there is a need to call an external API that is difficult to understand and use, this complexity is spread through the whole system.

Solution A GATEWAY is a wrapper that translates a specialized and complicated API into a simpler API. All applications that need to call this API call instead the API offered by the GATEWAY.

Decision Drivers The introduction of a GATEWAY makes a system easier to test and any possible changes in resources flexible. When the source API changes only the GATEWAY component needs to be modified. When implementing a GATEWAY an issue that has to be considered is the handling of exceptions and return values from the source API.

When a platform needs to support consuming and providing remote objects through multiple channels, a SERVICE ABSTRACTION LAYER [Vog01] can be used.

SERVICE ABSTRACTION LAYER [Vog01]

Problem A system or platform must allow for providing and consuming remote objects through multiple channels, i.e., remoting technologies and transport protocols. This channel support should be independent from the core invocation handling for remote objects. New channels should be addable on demand.

Solution The SERVICE ABSTRACTION LAYER adds an extra layer which receives and mediates requests originating from different channels. Each channel contains a channel adapter which translates requests back and forth between the backend and frontend channel formats.

Decision Drivers SERVICE ABSTRACTION LAYER separates business from communication logic, thus clients become decoupled from the business services. Therefore, changes in the business logic do not affect the client implementations, as the clients use stable generic interfaces to interact with the remote system. The SERVICE ABSTRACTION LAYER increases, however, the level of indirection of requests. The introduction of this separate layer may reduce efficiency as all requests have to be processed at runtime.

It introduces an extra layer containing all the necessary logic to receive and delegate requests originating from the different channels. To create a SERVICE ABSTRACTION LAYER a FACADE can be used to offer an interface for creating and for sending service requests. We show in Figure B.8 an example of interface design by implementing a FACADE which uses data from different DATA TRANSFER OBJECTS. The FACADE aggregates functionality from two application components and exposes an interface for integration with the remote platform. In this example, an ADAPTER inserts additional interface adaptation between the FACADE and the remote platform services. By providing a SERVICE ABSTRACTION LAYER, as illustrated in Figure B.9, we support multiple remoting technologies through three different channels: a JMS, a SOAP, and a REST Interface. A FACADE unifies the different channels and exposes a common interface for the remote platform.

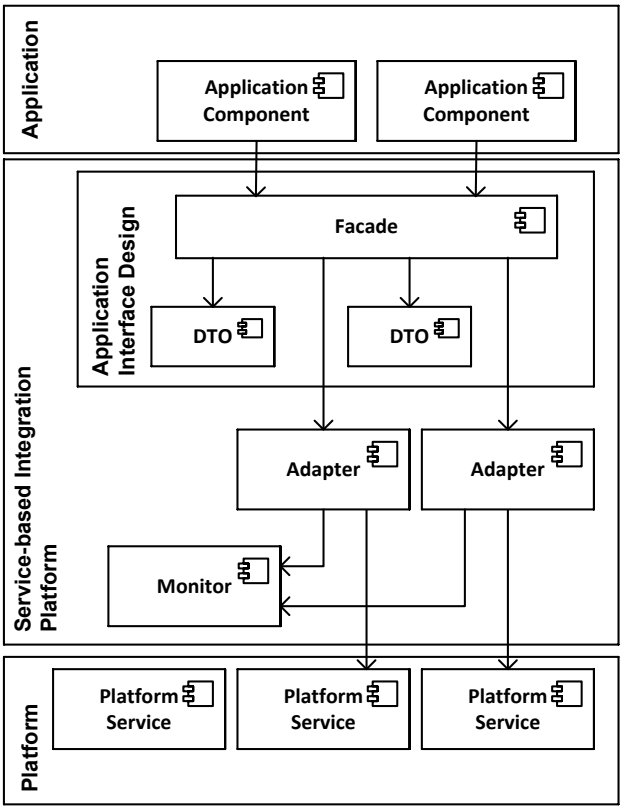


FIGURE B.8: Interface design with facade and integration with adapter

Another issue related to the design of interfaces is that the interfaces provided by platform applications are subject to adaptations and/or extensions due to changing requirements. To support different client-specific interfaces, related functionality can be grouped in separate EXTENSION INTERFACES [Sch+00b] and the common functionality can be included in a root interface. We illustrate in Figure B.10 an example of an EXTENSION INTERFACE design. Clients access component functionality via interfaces. A component may provide multiple extension interfaces which implement the root interface functionality. Clients create new components and specify the initial extension interface using a factory associated with each component type.

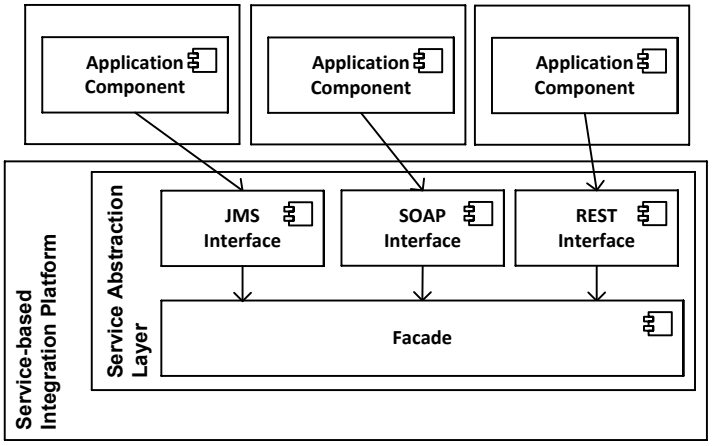


FIGURE B.9: Interface design with service abstraction layer

EXTENSION INTERFACE [Sch+00b]

Problem The interface provided and exposed by a component is subject to adaptations and/or extensions due to changing requirements of client components. Similarly, an anonymous number of clients requires alternative, client-specific interfaces for component interfaces. Being limited to a single and monolithic component interface means that changes propagate into existing client components in an uncontrolled manner.

Solution Related functionality is grouped and exported via separate EXTENSION INTERFACES. This grouping results from domain-specific (e.g., functional views) and/or temporal bindings (e.g., interface versioning). Common and/or administrative functionality (e.g., for selecting a particular view or version) is exposed by a root interface, to be included by each single EXTENSION INTERFACE.

Decision Drivers The use of EXTENSION INTERFACES decreases the coupling between the clients and components. The clients depend only on the interface roles they actually use, which ensures that they do not break when signatures of services change or new services are added to the components. To extend the functionality of a component only new EXTENSION INTERFACES need to be added. EXTENSION INTERFACES can also be aggregated to offer a new functionality of a component that aggregates other components. EXTENSION INTERFACES, however, may cause additional indirection and runtime overhead, as they are introduced between the components and the clients. It can also lead to increased complexity of client programming, as the clients must decide which EXTENSION INTERFACES are suitable for their use case.

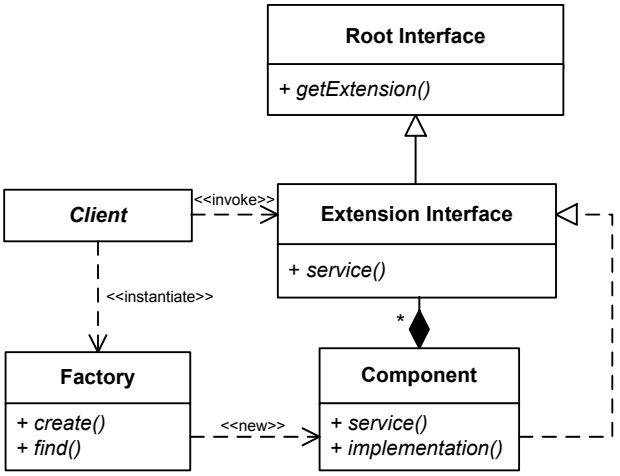


FIGURE B.10: An exemplary extension interface design

B.6 Communication Style

For each connection between two components in the platform integration solution, follow-on decisions about the communication style must be made. For instance, once the design decisions for integration and adaptation, as well as interface design, have been taken at the component or service level, decisions on the communication style between the components must be tackled. In this section, we focus on the different options for connecting distributed components. That is, in the platform integration design space, these design decisions are especially relevant for the connections between applications and the service-based integration platform, connections between the service-based integration platform and the platforms, distributed connections between the platforms, and

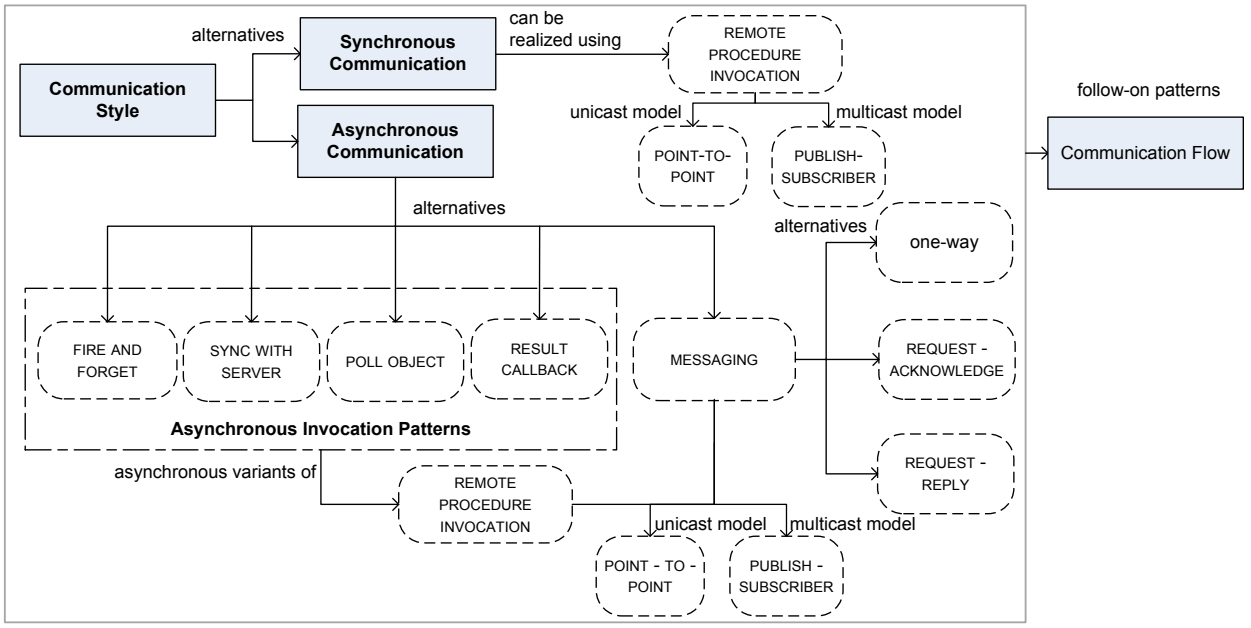


FIGURE B.11: Communication style

connections among distributed components within the service-based integration platform.

A basic option is to use synchronous invocations for the connection between two distributed components. Often synchronous invocations are realized following the REMOTE PROCEDURE INVOCATION pattern [HW04]. The remote application may respond either by sending a result value or a void result, unless an execution problem occurs and an exception is sent back. In a platform integration solution, this synchronous invocations option will rarely be used because synchronous invocations can lead to slow and unreliable systems, as the communication of the calling application must block until it receives the result.

Thus, we mainly focus on the asynchronous communication style and study the various options for implementing it. Applications that communicate with each other using asynchronous communication do not need to block their execution, but they can continue with other tasks while they are waiting for the results of their invocations. The asynchronous invocation patterns offer many alternatives for invoking a remote service asynchronously. They describe asynchronous variants of the REMOTE PROCEDURE INVOCATION pattern.

REMOTE PROCEDURE INVOCATION [HW04]

Problem Applications in different programming languages that run on different platforms need to share data and processes.

Solution Using REMOTE PROCEDURE INVOCATIONS means that each application offers a remote interface to interact with the other applications. Thus, one application can get or change data from another application by calling its remote interface.

Decision Drivers Applying the pattern results in tightly coupled applications. It is difficult to deal with application downtimes, such as system crashes or downtimes for maintenance, as incoming invocations will get lost during the downtimes. Hence, REMOTE PROCEDURE INVOCATION based systems might be more unreliable than, e.g., MESSAGING based systems. Synchronous REMOTE PROCEDURE INVOCATIONS may lead to slow and blocking applications.

In particular, when a result or application error needs to be delivered either a POLL OBJECT [Völ+05] or a RESULT CALLBACK [Völ+05] can be used. A FIRE AND FORGET interaction [Völ+05] does not return any result or acknowledgment to the application that invokes a remote object, but only offers best effort semantics. When a notification that the request arrived to the remote application is necessary, then SYNC WITH SERVER [Völ+05] can be used instead of FIRE AND FORGET.

FIRE AND FORGET [Völ+05]

Problem A client application wants to notify a remote object for an event. Neither a result is expected, nor does the delivery have to be guaranteed. A one-way exchange of a single MESSAGE is sufficient.

Solution A FIRE AND FORGET operation is performed by the communication middleware without acknowledging the processing or delivery status to the client. The thread of control is yielded back to the client immediately.

Decision Drivers The FIRE AND FORGET pattern provides non-blocking communication with unreliable transmission. That means that the client is not notified of errors in transmission or execution of the remote object. The remote object does not deliver any execution results to the client.

While FIRE AND FORGET offers one-way communication, SYNC WITH SERVER follows the communication style REQUEST-ACKNOWLEDGMENT [HW04]. In contrast to POLL OBJECT, RESULT CALLBACK requires an event-based programming style to consume the result. It has the benefit over POLL OBJECT to support immediate reaction upon the arrival of a result.

SYNC WITH SERVER [Völ+05]

Problem A client application needs to ensure higher reliability of asynchronous invocations than FIRE AND FORGET, but does not require the transmission of a result.

Solution The client sends the invocation as in FIRE AND FORGET but waits for a reply from the server about the successful transmission of the invocation. The communication middleware blocks only until the notification of the successful reception of the invocation arrives and then continues the execution.

Decision Drivers SYNC WITH SERVER ensures successful transmission of requests and makes the remote invocations more reliable than FIRE AND FORGET. It introduces, however, additional latency, as the client must block until the notification of successful reception is received. Thus, there is a trade-off between higher reliability and worse performance in comparison with FIRE AND FORGET. Also the server application can not inform the client for application errors as the execution of the remote invocation happens asynchronously.

RESULT CALLBACK and POLL OBJECT offer the REQUEST-REPLY [HW04] communication style. POLL OBJECT can be used in an imperative programming model.

POLL OBJECT [Völ+05]

Problem Remote invocations of a client must be processed asynchronously but the client needs the result to continue its computations.

Solution A POLL OBJECT receives the results from a remote invocation on behalf of the client. The client periodically queries the POLL OBJECT for the results. The client can continue with other tasks and when the results are available to the POLL OBJECT the client can fetch them the next time it queries the POLL OBJECT.

Decision Drivers The server side stays oblivious to the client-side POLL OBJECTS. The pattern offers more reliable communication compared to FIRE AND FORGET, as the result is an implicit acknowledgment, but it can not immediately inform the client about an incoming result.

RESULT CALLBACK [Völ+05]

Problem The client needs to be informed about the results of its asynchronously invoked operations once the results become available to the communication middleware.

Solution A callback-based interface for remote invocations is provided on the client which passes a callback object to the communication middleware upon a remote invocation. After the invocation, the client can continue with other tasks. When the call completes and the results become available a callback is invoked on the client to process the result.

Decision Drivers The pattern has the same basic decision drivers as POLL OBJECT. RESULT CALLBACK is preferred over POLL OBJECT when an immediate reaction on the incoming result is needed. While POLL OBJECT works well with the imperative programming model of today's OO application, RESULT CALLBACK requires an event-based programming model to handle the callbacks. The same or different callback objects can be used for different invocations of the same type.

In asynchronous remote invocations, ASYNCHRONOUS COMPLETION TOKENS [Sch+00a] are used to associate the callback with the original invocation. The pattern fulfills the same role as the CORRELATION IDENTIFIER pattern [HW04] discussed below. To ensure reliability of communication and increase decoupling of the integrating platforms, MESSAGING [HW04] provides the most convenient solution. The integrating applications exchange MESSAGES [HW04] via a MESSAGE CHANNEL [HW04] which can be either a POINT-TO-POINT CHANNEL [HW04] or a PUBLISH-SUBSCRIBE CHANNEL [HW04]. The difference between them is that in the first case we have only one receiver of the requests while in the second case multiple receivers-subscribers of the messages exist.

MESSAGING [HW04]

Problem Applications that are developed independently, are built in different languages, and run on different platforms need to share data and processes in a responsive way.

Solution MESSAGES are used to transfer packets of data frequently, immediately, reliably, and asynchronously using customized formats.

Decision Drivers MESSAGING offers high reliability, as sending a message does not require both systems to be up and running at the same time. Instead, message queues in the messaging system can temporarily store the messages when systems are down. Hence, systems become more decoupled from each other than in REMOTE PROCEDURE INVOCATION. However, complexity increases, as many decisions about the messaging system, such as the message formats, the message routing, the message transmission, and the connection of the applications to the messaging system, etc. have to be made.

The communication using MESSAGES can be either one-way or two-way. In a one-way communication, the sender directs a message towards a receiver using a one-way channel, without waiting for any notification or result of its request. A two-way communication requires a two-way channel to allow delivery of responses (void, result values, or exceptions). A REQUEST-REPLY communication can be implemented in different ways combining different asynchronous communication styles. For example, the client can first receive an acknowledgment of its request and then poll for the results (REQUEST-ACKNOWLEDGE-POLL [Dai12]) or get notified about the delivery of its request and receive the request results with a callback service (REQUEST-ACKNOWLEDGE-CALLBACK [Dai12]).

REQUEST-REPLY [HW04]

Problem Two applications communicate through an exchange of MESSAGES. Each MESSAGE realizes a one-way conversation. The sending application requires a reply from the receiver of the initial MESSAGE.

Solution To realize a two-way conversation, pairs of request and reply MESSAGES are exchanged. Depending on the intended coupling between the sender and receiver, the reply MESSAGE is sent either via the request's back channel or via its own communication channel.

Decision Drivers The pattern provides a two-way message transmission on a two-way channel. The requestor is always notified about the successful or unsuccessful completion and/or of the result of its request. The two processes (request and reply) are decoupled. If the connection between requestor and receiver fails before the reply is sent then the requestor must re-send its request unless messages are persistent.

The PUBLISH-SUBSCRIBE CHANNEL is the version of the PUBLISH-SUBSCRIBER [Bus+07a] pattern that applies for messaging. Apart from messaging, the POINT-TO-POINT and PUBLISH-SUBSCRIBER styles can be also used in synchronous or asynchronous remote invocations for unicasting and multicasting respectively. Hence, PUBLISH-SUBSCRIBER is an alternative to POINT-TO-POINT connections, mainly discussed so far, that can be applied for all communication styles discussed in this section. As in synchronous REMOTE PROCEDURE INVOCATIONS or in the asynchronous POLL OBJECT or RESULT CALLBACK patterns, messages are also often used to deliver messages in REQUEST-REPLY style or in REQUEST-ACKNOWLEDGE style (as in the SYNC WITH SERVER pattern).

PUBLISH-SUBSCRIBER [Bus+07a]

Problem Data changes in one place, but many components depend on this data and have to be updated. That means, multiple application components need to be notified about changes in one component.

Solution One dedicated component takes the role of the publisher and the other components are its subscribers who subscribe in order to get notified for state changes of the publisher. An object can be a subscriber to many publishers and can also play the role of both the subscriber and publisher.

Decision Drivers Publishers are loosely coupled to subscribers. PUBLISH-SUBSCRIBER allows listening for events without disturbing the communication flow. Thus, it can also be used for debugging and logging purposes. However, using PUBLISH-SUBSCRIBER may introduce security issues, as any subscriber is able to look at the events generated by the publisher. Although PUBLISH-SUBSCRIBER offers high scalability, it does not guarantee the delivery of events to the subscribers.

REQUEST-ACKNOWLEDGE [Dai12]

Problem A client would like to notify a system about the fact that a request has arrived or about an interesting event after a request has arrived. Requests do not need to be processed right away, and the responses to the requests do not need to be delivered.

Solution When a service receives a request it forwards the request to a background process and then returns an acknowledgment containing a unique request identifier.

Decision Drivers REQUEST-ACKNOWLEDGE communication alleviates the problem of unavailable resources or request spikes, as unlike ONE-WAY communication it lets the client know that the requests have been received and will be processed. However, the client does not get informed about application errors that may happen during the process execution.

B.7 Communication Flow

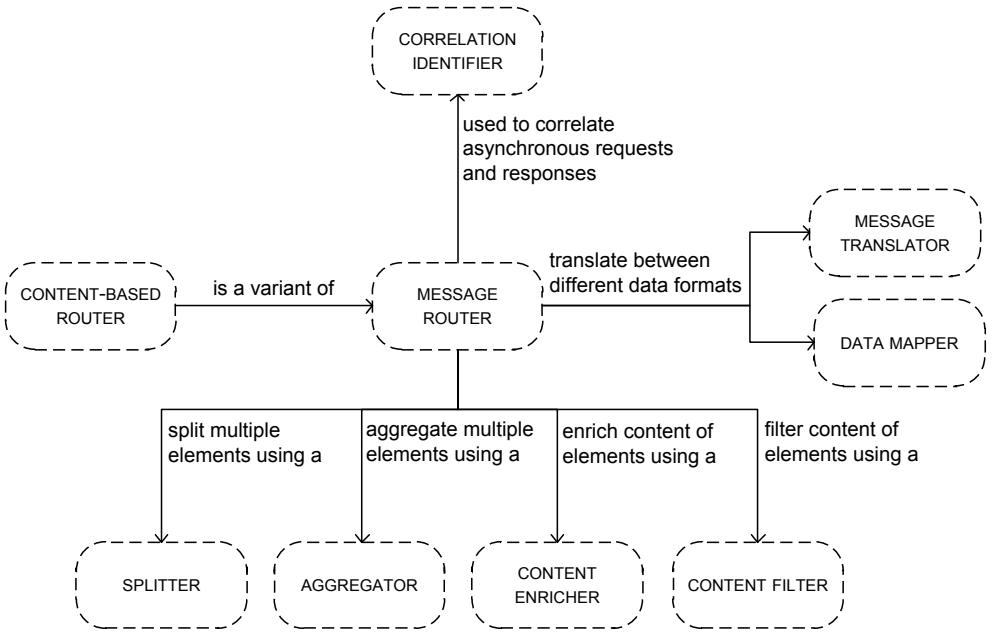


FIGURE B.12: Communication flow

Transferring distributed service invocation data from the client applications to the integrated platform services, mediated by the service-based integration platform, requires from the software designer to make design decisions related to the data transformations in the service-based integration platform. These decisions touch a variety of concerns, e.g., the routing of the invocations and their invocation data to the intended receivers, as well as all data transformations at different levels (e.g., data representation, marshalling, data transport). The communication flow perspective considers the flow of requests and replies through the integration platform as a series of data transformations, performed by infrastructure components. The relevant data items are in-memory objects (e.g., DATA TRANSFER OBJECTS) and MESSAGES. While many patterns described in this section have originally be described in the context of messaging, in variants they can also be applied in combination with the other (asynchronous) invocation patterns.

If sophisticated message or invocation routing is required, a MESSAGE ROUTER [HW04] offers an appropriate solution. The MESSAGE ROUTER listens at the incoming, or frontend, message channels and redirects the messages intercepted towards the necessary processing chains and towards the actual backend receivers, i.e., the platform services. With such a central routing component, there is a single point of responsibility for administering the routing rules and to configure the processing chains needed for preparing the messages for the individual platform services. The MESSAGE ROUTER can be made configurable following the COMPONENT CONFIGURATOR pattern (see also Section B.4).

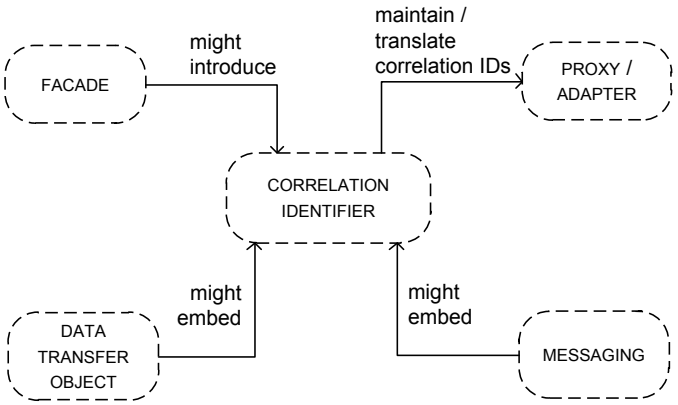


FIGURE B.13: Relationships between correlation identifier and other patterns

MESSAGE ROUTER [HW04]

Problem A message channel can be used to exchange messages of varying structure and content, thus requiring different processing steps. To decouple the processing steps, without introducing dedicated message channels, the MESSAGES have to be filtered depending on a set of conditions.

Solution A MESSAGE ROUTER component inserts a special filter that consumes a MESSAGE from an incoming MESSAGE CHANNEL and republishes it to a different outgoing MESSAGE CHANNEL, after having evaluated certain filter criteria. MESSAGE ROUTERS distribute MESSAGES either to a fixed destination or redirect the messages depending on the message content (see CONTENT-BASED ROUTER).

Decision Drivers A MESSAGE ROUTER centralizes message filters and the actual filter functionality (routing rules) in a single component which becomes the single point of maintenance and failure. Message routing however adds to the processing overhead of the integration platform.

In a service-based integration platform, routing is often performed using a CONTENT-BASED ROUTER [HW04]. As a variant of the MESSAGE ROUTER, this router accesses the message content, i.e., envelope and body elements, to evaluate the standing routing rules against the data extracted from the messages. This way, the routing conditions can be set and transmitted by the MESSAGES themselves (e.g., in their envelopes or by their type annotation), rather than by providing the routing-critical data through an external source.

CONTENT-BASED ROUTER [HW04]

Problem An integration solution deploys a MESSAGE ROUTER to have MESSAGES processed adequately. However, the message routing is not to be decided by external factors or by fixed routes, but rather by the messages' content.

Solution A CONTENT-BASED ROUTER has the capacity to examine the message content and distributes the message to a different channel based on its content (e.g., routing data in the message envelope, its structure, or message values).

Decision Drivers As a kind of MESSAGE ROUTER, the use of a CONTENT-BASED ROUTER allows for centrally managing message routing without the need to modify either the client applications or platform services. The routing functions, however, may change frequently causing extra maintenance effort. The recipients also have no control over the routing process.

Content-based routing is not applicable only for the exchange of MESSAGES representing service invocation data (e.g., implicit invocations on domain objects), but it can also be used to differentiate between invocation messages and messages carrying invocation-unrelated or opaque types of data. Imagine application scenarios which involve setting up audio/video streaming data between client applications and platform services (i.e., here, streaming services). Such data requires alternative processing steps when being mediated by the integration platform, for example, as part of an optimization which bypasses routing and processing steps applicable to handshake and invocation messages only.

Besides acting as a matchmaker between messages and the available data transformation tasks, a MESSAGE ROUTER also allows for composing processing chains to be applied on selected messages. Message processing and filter components can be organized in a PIPES AND FILTERS [Bus+00; HZ09] style. Finally, the processing chains can be constructed in a way so that the delivery to the responsible platform service is performed by republishing the transformed message to a backend, or outgoing channel.

The data sent across the network will not always be used by the data receiver, i.e., the platform services, as it is; whatever the dominating communication styles or the communication flow approach is (MESSAGING vs explicit component invocations). For example, for exposing FACADE interfaces using DATA TRANSFER OBJECTS the backend invocations must be decomposed into a series of invocations upon one or more platforms and their input and output parameter types. The MESSAGING equivalent to FACADES and DATA TRANSFER OBJECTS are compound messages, with each of the part messages addressing a distinct platform service.

A SPLITTER [HW04] disassembles the compound messages into their constituents which are expected by the target platform(s). Sometimes multiple elements need to be collected and reassembled to be delivered to their final destination and to be accepted by the platform services as message endpoints. On the back channel, e.g., for asynchronous REQUEST-REPLY interactions, there is the need for re-assembling the resulting data elements into a composite reply message. This bears the risk of duplicates or an out-of-order reassembly. The SPLITTER can, for instance, split the messages in the integration platform that are sent to the different platform services.

SPLITTER [HW04]

Problem Messages passing through an integration solution consist of multiple elements, each of which must be processed separately. The incoming message appears as a composite message.

Solution A SPLITTER component is incorporated into the integration platform to break up the composite message into a series of individual elements or element subsets. Each element subset is then published as a distinct message. Common elements of the initial message are maintained in the resulting messages (e.g., identification and sequencing tokens) in order to allow for re-integrating reply messages later on.

Decision Drivers The primary driver for adopting a SPLITTER is that target platform services, which are grouped by a dedicated FACADE, must receive the respective data subsets for which they are responsible when answering invocations dispatched onto the facade interface. A SPLITTER can not only break a message into its repetitive data chunks, but also a large message into individual messages to simplify the further processing. On the negative side, there is data overhead in the resulting messages (e.g., CORRELATION IDENTIFIERS, timestamps). Also, extra processing effort is needed for aggregating reply messages afterwards.

Conversely, an AGGREGATOR [HW04] merges individual messages or element subsets thereof into compound messages to be delivered to the platform services. The AGGREGATOR detects the related elements as well as their right order according to their CORRELATION IDENTIFIERS. On the reply channel, an AGGREGATOR might require a SPLITTER. The AGGREGATOR can, for instance, aggregate messages in the integration platform that are sent to the different platform services.

Apart from this whole-part mismatch between senders and receivers at the level of messages, the data contained in the messages might simply be too excessive or incomplete to be (efficiently) processable by the receivers. There are many possible reasons for this problem. For example, the domain model of the target system might only correspond to a subset of the source domain model or certain auxiliary invocation data contained in a message might not be relevant, for instance, the data might only be required for add-on services or constitute metadata relevant only for the underlying middleware technologies. Sometimes, security requirements demand the removal of

message parts (e.g., identity tokens). In such cases, a CONTENT FILTER [HW04] is included in the processing chain of a message to extract and drop excessive data. CONTENT FILTER can be applied in the integrated platform to filter the messages that pass through it.

AGGREGATOR [HW04]

Problem We need to combine the data of individual, but related messages so that the aggregated data can be processed as a whole by the target system.

Solution An AGGREGATOR component observes the message stream, collects and stores individual messages based on filter criteria and identification tokens (CORRELATION IDENTIFIERS) until it has received a complete set of related messages. After having assembled a single message out of these parts, based on selected aggregation strategies, the resulting message is published for delivery to the target system.

Decision Drivers In order to be able to aggregate incoming messages to an AGGREGATOR the messages need to have a correlation that indicates which messages belong together. The aggregation decisions described by the aggregation algorithm bear the risk of introducing extra development effort and additional processing complexity (depending on the aggregation strategy). At the same time, message aggregation can realize an important optimization strategy in service-based platform integration: the batch processing of service invocations. For batching, multiple content-wise unrelated messages are packed into a single composite message to be delivered to a platform service.

CONTENT FILTER [HW04]

Problem When sending messages from one system to another, it is common that situations occur in which the target system is not interested in all data included in the messages.

Solution A CONTENT FILTER component is provided to remove unneeded, obsolete, or protected data items from a message.

Decision Drivers Data filtering for messages is a critical adaptation mechanism to be supported by an integration platform. It can also simplify the structure of a message (e.g., convert a tree into a flat structure), or remove redundancy or ambiguity in a data structure. CONTENT FILTERS can act as pre-processors before handling messages using MESSAGE TRANSLATORS.

Requirements for additional data can result from domain model mismatches, different underlying middleware, or security requirements. In such cases, a CONTENT ENRICHER [HW04] augments the message with the missing information by accessing external data sources or the message context. CONTENT ENRICHER can be used in the message processing of the integration platform.

CONTENT ENRICHER [HW04]

Problem When sending messages from one system to another, it is common that situations occur, in which the target system requires more data than included in the original message.

Solution A CONTENT ENRICHER is a specialized message transformer which accesses data sources external to the message processing system to add the missing data.

Decision Drivers A CONTENT ENRICHER may need to consult external resources to find the required data, based on references contained in a message. This may not only affect the message processing throughput negatively, but also the synchronization decoupling in the communication flow. External resources might require synchronous communication, which requires special treatment for the enricher component (e.g., introducing a special message consumer internal to the integration platform). If the external source is an integrated platform service, the CONTENT ENRICHER acts as a variant of AGGREGATOR.

A frequent source of mismatch between client applications and platform services are incompatibilities between the data formats supported. Such format mismatches involve differences in data models, data types, data representation, and data transport techniques. When using explicit component invocations and in-memory object representations of the invocation data, a DATA MAPPER [Fow03] can be used to deal with the unaligned or non-canonical data formats between integrating platforms.

DATA MAPPER [Fow03]

Problem Two or more components exchange data in terms of in-memory objects. However, the receiving interfaces require an incoming data format which is incompatible with the object structures exchanged. A format mismatch is the consequence, covering inconsistencies at the level of two incompatible data models, and in-memory representation styles.

Solution DATA MAPPERS incorporate an auxiliary transformation layer in the component architecture which hosts a group of mapper components. These DATA MAPPERS provide model and representation transformations for data objects. Certain model and representation strategies are captured as dedicated modules and can be applied to data objects exchanged between different pairings of client applications and platform services.

Decision Drivers The processing of service data in terms of in-memory objects requires collocation of the two components for which the data is mapped. PROXIES can be used when process or machine boundaries need to be crossed. When crossing process and machine boundaries, MESSAGE TRANSLATORS take the role of format filters.

A DATA MAPPER transforms, e.g., the data from one object type to another. For dealing with marshalling and transport protocol mismatches, the DATA MAPPER can use the services offered by MARSHALLER [Völ+05] and PROTOCOL PLUG-IN [Völ+05] components as offered by the underlying middleware framework.

MESSAGE TRANSLATOR [HW04] can be incorporated into the processing chains of MESSAGES for transposing them from one data format into another. In the processing chains, the MESSAGE TRANSLATORS usually come last as they operate on the already filtered messages (see Figure B.14). The MESSAGE TRANSLATOR can reside, for instance, in the integration platform and translate between the client application message formats and the message formats of the platform services.

MESSAGE TRANSLATOR [HW04]

Problem Two or more interacting applications operate on different message formats and there is a message format mismatch.

Solution A dedicated filter component is used by the MESSAGE ROUTER to transform an incoming message format into an outgoing message format. A single MESSAGE TRANSLATOR can cover any, or even all, levels of transformation (model, type, representation, and transport).

Decision Drivers A key driver for providing a MESSAGE TRANSLATOR component in the integration platform is to avoid enforcing a uniform message format throughout all (existing and future) client applications and platform services, if possible at all. A MESSAGE TRANSLATOR preserves the independence of the integrated clients and services in terms of message formats. The adoption of different standard formats or the modification of proprietary ones remain a localized event without propagating to the other participants. However, depending on the degree of heterogeneity within the distributed systems in terms of message formats to be mated, there is the risk of high complexity (i.e., combinatorial explosion of message format transformations). This is particularly valid for each MESSAGE TRANSLATOR realizing all transformation steps, potentially in redundancy to other translators. This risk can be reduced by limiting a single translator's responsibility to a single transformation step (e.g., marshalling) and combining them to message processing chains for a given integration scenario (e.g., a pair of client application and platform service).

A particular source of complexity in the communication flow design of a service-based integration platform is the repeated dis- and reassembly of data items and bridging between process synchronization styles. Both the content and the synchronization decoupling require the identification of decoupled parts. Important examples are message parts of disassembled compound messages (see SPLITTER pattern) or non-blocking backend replies to blocking frontend requests. Also, the

permanent interleaving of related messages in the integration platform requires a message tracking mechanism.

Adopting CORRELATION IDENTIFIERS [HW04] is an adequate design decision to address such tracking requirements. For asynchronous communication styles, where exactly a corresponding pair among multiple communication parties has to be (implicitly or explicitly) identified, these identifiers are also referred to as ASYNCHRONOUS COMPLETION TOKENS [Bus+07a].

As for designing the frontend interfaces, for instance, CORRELATION IDENTIFIERS can be employed and stored in the FACADE to track the resulting backend invocations at a per-request level. One option is to maintain the identifier in the service descriptions, such that every communication with the service needs to refer to a specific CORRELATION IDENTIFIER. Alternatively, a FACADE could also store the CORRELATION IDENTIFIERS in the DATA TRANSFER OBJECTS, if available (see Section B.5).

In a MESSAGING infrastructure, the CORRELATION IDENTIFIER is extensively used to realize conversational interactions, i.e., for exchanging and processing messages such as in REQUEST-REPLY interactions and MESSAGE SEQUENCE interactions [HW04], to name but a few.

CORRELATION IDENTIFIER [HW04]

Problem Using asynchronous remote invocations or messages, the requesting component does not block, even if a reply is expected as part of an asynchronous REQUEST-REPLY conversation. However, upon incoming an asynchronous reply, the receiving components must align the incoming reply to the corresponding request.

Solution To correlate two messages, such as a request and a reply processed at different times, both messages embed a unique identity token. In the request message, the CORRELATION IDENTIFIER is referred to as the request ID. The reply message then includes a token, the CORRELATION IDENTIFIER, which matches or refers to the initial request ID.

Design Drivers For general MESSAGING and asynchronous REMOTE PROCEDURE INVOCATION scenarios, it is sufficient to generate and maintain unique IDs for the messages. In service-based platform integration, it is also required to preserve a reference to the client application having submitted the original request, along with the unique identifier for the message as such. This is needed to resume serving a service invocation for a particular client across the asynchronous frontend connector of an INTEGRATION ADAPTER. Assuming that the client applications should not and can not be altered to emit an additional identifier token, to be used as the CORRELATION IDENTIFIER or as a part of it, the integration platform has to maintain a mapping table which aligns the requesting clients and the CORRELATION IDENTIFIERS issued.

In some particular cases, we might need to integrate two or more software platforms that do not support compatible CORRELATION IDENTIFIER mechanisms. The reason can be that either one of the platforms does not support CORRELATION IDENTIFIERS or both support CORRELATION IDENTIFIERS but their CORRELATION IDENTIFIERS are not simply interchangeable. In such cases, components, such as the PROXIES or ADAPTERS in this pattern language, are often introduced for mediating the communication and data exchange between these platforms, i.e., translate and temporarily store the CORRELATION IDENTIFIERS. This can be realized, e.g., by letting the mediators maintain an additional table to map the CORRELATION IDENTIFIERS from one communication partner to the CORRELATION IDENTIFIERS of the other communication partner, and vice versa.

The design decisions become embodied in the way the service-based integration platform lays out the communication flow in terms of component interactions as depicted in Figure B.14. Depending on the decisions taken on the communication styles (see Section B.6), there are various possibilities for laying out the data transformation infrastructure in the service-based integration platform. For example, the integration platform can be built using basic MESSAGING principles. Alternatively, an explicit invocation style between transformer components can be applied. Both variants are sketched out as exemplary setups in Figure B.14.

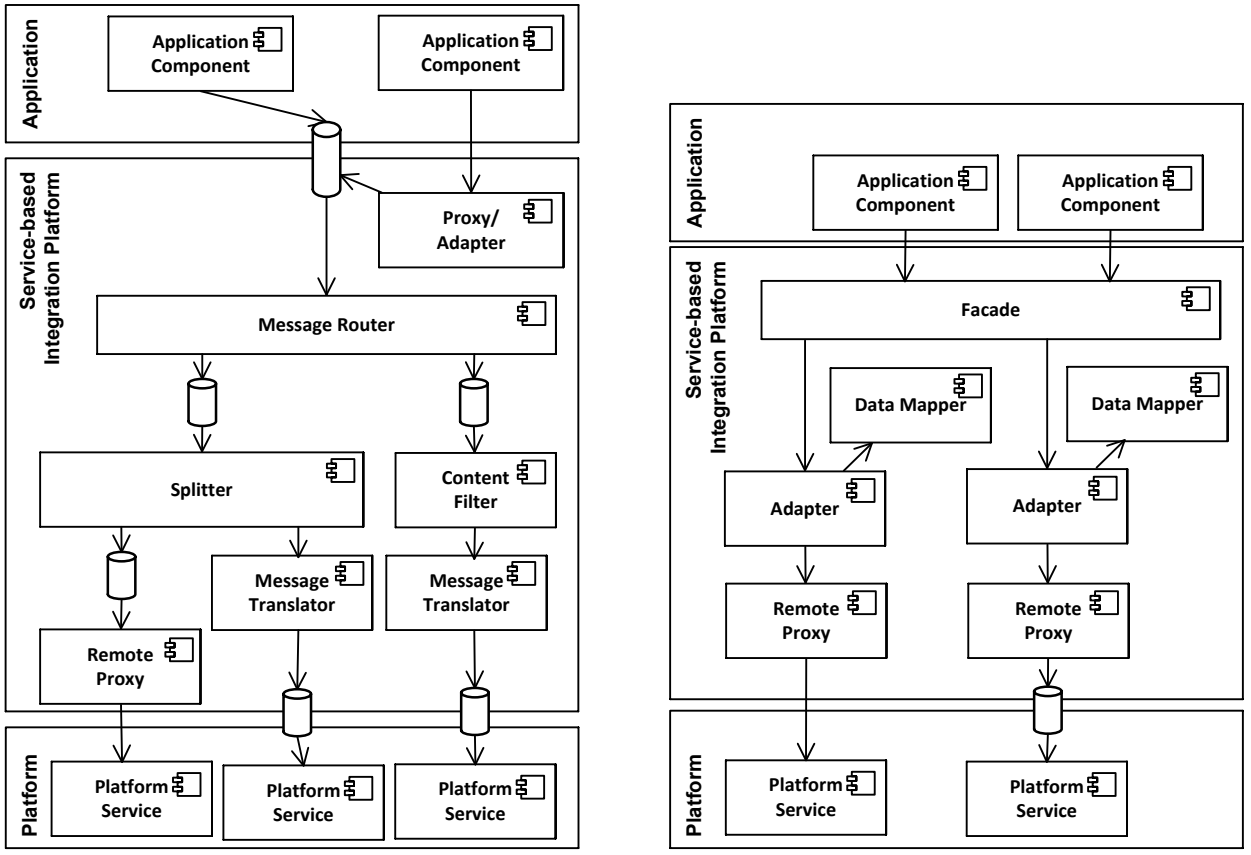


FIGURE B.14: Organizing communication flows in a service-based integration platform

The initial drivers for opting for either approach are the communication styles supported by the components to integrate (i.e., the client applications and the platform services), as well as the decoupling strategies to be implemented by the integration platform. For example, while a straightforward OBJECT ADAPTER can be easily constructed using explicit invocations, an INTEGRATION ADAPTER with an asynchronous frontend connector which attaches to the client applications can leverage an underlying MESSAGING infrastructure. Both approaches allow for minimizing, or ideally turning obsolete the need for modifying either the client applications and/or platform services to assist in the data transformations required. Client applications or platform services not enabled for MESSAGING can be integrated using bridging PROXY/ADAPTER components which act as the sending or receiving message endpoints to a frontend and backend channel, respectively. This way, client applications and the platform services do not have to be manipulated even for overcoming such a mismatch in communication style.

C DSL for Fuzzy Logic Models

The main elements defined in a fuzzy decision model expressed in our DSL are the alternative Patterns for a design problem, as well as their forces and rules. A list of alternative patterns can belong to a group and can import other fuzzy decision models expressed in our DSL, thus inheriting their forces and rules. This way, we can create technology-specific fuzzy models which are based on generic fuzzy models and their Patterns are pattern implementations (ofType) of the Patterns included in the general model. The membership functions of the fuzzy sets can be defined using some standard function types (gauss, trape, etc.) independently (schemas) and reused by multiple pattern forces. The fuzzy inference system (e.g., the fuzzy inference method) can also be configured by the user. The rules have the following syntax: if condition then Pattern where the condition can contain conjunctions and disjunctions or combinations of them of the form force_name is force_value or force_name not force_value. Also, weights can be assigned to the rules. The formal specification of the DSL given below is defined as an Xtext language grammar.

```
grammar at.ac.univie.cs.swa.dsl.FuzzyPatternModel with org.eclipse.xtext.common.Terminals

Model:
  ("group" name=ID)
  (model=Patterns);
Patterns:
  (imports+=GenericGroup)?
  (patterns+=Pattern)+
  (forces=ForceList)
  (schemas=SchemaList)?
  (rules=RuleList)
  (config=Configurations)?;
ForceList:
  "forces"
    (forceList+=Force)+
  "end";
RuleList:
  "rules"
    (ruleList+=Rule)+
  "end";
SchemaList:
  "schemas"
    (schemaList+=Schema)+
  "end";
```

```
GenericGroup:
  "import" importURI=STRING;
Pattern:
  "Pattern" name=ID ("ofType" parentPattern=[Pattern])? (":" description=STRING)?;
Force:
  name=ID (schema=[Schema])? "{" (values+=ForceValue)* "}";
ForceValue:
  name=ID (":" membership=MembershipFunction)?;
enum MembershipType:
  gauss1 | gauss2 | pi | trian | trape | singleton;
MembershipFunction:
  membership=MembershipType "["(values+=Number)*"]";
Schema:
  name=ID "{" functions+=MembershipFunction* "}";
Rule:
  name=ID "-->" "if" (parOpen+=OpenParenthesis)* (firstPredicate=Predicate) (next+=NextPredicate)* "then"
     action=[Pattern] ("with" weight=Number)?;
NextPredicate:
  connector=Connector (parOpen+=OpenParenthesis)* predicate=Predicate (parClose+=CloseParenthesis)*;
Predicate:
  variable=[Force] (positive="is" | negative="not") property=ID;
Connector:
  "and" | "or";
OpenParenthesis:
  "(";
CloseParenthesis:
  ")";
Number:
    INT | "-"INT | "-"INT"."INT | INT"."INT;
Configurations:
  {Configurations}
  "config" "{"
  ("activation=" activation=("MIN" | "PROD"))?
  ("operatorAND=" operatorAND=("MIN" | "PROD" | "BDIF"))?
  ("accumulation=" accumulation=("MAX" | "BSUM" | "NSUM"))?
  "}";
```

Listing 24: DSL specification for fuzzy logic models

D AK Transformation Language Specification

Listing 25 presents a formal definition of the AK transformation language in terms of the EBNF-like notation provided by Xtext. We note that square brackets denote the cross-references between different models. For instance, the rule "`AddConnector`" defined in Line 37–38 refers to the components' ports at the two ends of the newly added connector. Using AK transformation language simple actions that add, delete, or update components, connectors, and other elements or properties of C&C views can be enacted (e.g., `AddComponent`, `UpdateConnector`, etc.). Apart from that, AK transformation language supports the following actions: `Condition`, `ForLoop`, `WhileLoop`, `Compound`, `Group`, and `Refine`. A `ForLoop` (see Line 21–22) can be used in order to define one or more actions that are performed over a predefined set of design elements. The `WhileLoop` will do the same as `ForLoop` but under certain boolean constraints. A `Group` (see Line 91–92) can be used to define the grouping of a finite set of components as sub-components of another component, while `Refine` (see Line 94–95) indicates a mapping of an abstract and high-level component onto one or more low-level components. A `Compound` (see Line 25–26) is used to represent a composite construct containing multiple actions. It can inherit the definitions of existing `Compounds` via the keyword "`extends`", and therefore, reduce redundancy and duplicated efforts.

```
1  grammar at.ac.univie.cs.swa.dsl.action.ActionDSL with org.eclipse.xtext.common.Terminals

3  ActionDSL:
4    {ActionDSL}
5    "module" name=FQN
6    (compounds+=Compound)*
7    (actions+=Action)+;

9  Action:
10   Add | Delete | Update | Group | Refine | Condition | WhileLoop | ForLoop;
11 Condition:
12   "if" "(" condition=BooleanExpression ")"
13   (thenActions+=Action)+
14   (=>"else" (elseActions+=Action)+)?;
15 WhileLoop:
16   {WhileLoop}
17   "while" expression=BooleanExpression "do" (actions+=Action)+ "end";
18 ForLoop:
19   {ForLoop}
20   "for" "("element=ID ":" params=LIST")" (actions+=Action)+ "end";
```

```
Compound:
  "compound" name=ID ("extends" (parent+=[Compound|FQN])+)? spec=Spec;
Spec:
  "("(args+=ID)+")" "{" (actions+=Action)* "}";

Add:
  AddComponent | AddConnector | AddPort | AddProperty | AddStereotype | AddCompound;
AddComponent:
  "add component" name=STRING;
AddConnector:
  "add connector" name=STRING "from" source=[component::Port|FQN] "to" target=[component::Port|FQN];
AddPort:
  "add port" name=STRING "kind=" kind=PortKind "to" component=[component::Component|FQN];
enum PortKind:
  provided="PROVIDED" | required="REQUIRED";
AddStereotype:
  "add stereotype" "<<" text=STRING ">>" "to" target=[core::Element|FQN];
AddProperty:
  "add property" name=STRING "type=" type=STRING "value=" value=STRING "to" target=[core::Element|FQN];
AddCompound:
  "add compound" compound=[Compound|FQN] name=STRING "("(args+=ID)+")";

Delete:
  DeleteComponent | DeleteConnector | DeletePort | DeleteProperty | DeleteStereotype;
DeleteComponent:
  "delete component" component=[component::Component|FQN];
DeleteConnector:
  "delete connector" conn=[component::Connector|FQN];
DeletePort:
  "delete port" port=[component::Port|FQN];
DeleteProperty:
  "delete property" property=[component::Property|FQN];
DeleteStereotype:
  "delete stereotype" stereotype=[component::Stereotype|FQN];

Update:
  UpdateComponent | UpdateConnector | UpdatePort | UpdateProperty | UpdateStereotype;
UpdateComponent:
  "update component" component=[component::Component|FQN] "name=" newName=STRING;
UpdateConnector:
  "update connector" conn=[component::Connector|FQN] "name=" newName=STRING;
UpdatePort:
  "update port" port=[component::Port|FQN] ("name=" newName=STRING)? ("kind=" newKind=PortKind)?;
UpdateProperty:
  "update property" prop=[component::Property|FQN] ("name=" newName=STRING)?  ("type=" newType=STRING  "value=
     " newValue=STRING)?;
UpdateStereotype:
  "update stereotype" stereotype=[component::Stereotype|FQN] "text=" newText=STRING;

Group:
  "group" component=[component::Component|FQN] "container" container=[component::Component|FQN];
Refine:
  "refine" component=[component::Component|FQN] "in" (refinedComp+=STRING)+;
```

Listing 25: AK transformation language specification

Bibliography

[Abr+06] S. Abrams, B. Bloom, P. Keyser, D. Kimelman, E. Nelson, W. Neuberger, T. Roth, I. Simmonds, S. Tang, and J. Vlissides. "Architectural Thinking and Modeling with the Architects' Workbench." In: *IBM Systems Journal* 45.3 [July 2006], pp. 481–500.

[AF14] David Ameller and Xavier Franch. "Assisting software architects in architectural decision-making using Quark." In: *CLEI electronic journal* 17.3 [2014], pp. 1–20.

[Ahm+05] Moataz Ahmed, Moshood Omolade Saliu, and Jarallah AlGhamdi. "Adaptive fuzzy logic-based framework for software development effort prediction." In: *Information & Software Technology* 47.1 [2005], pp. 31–48. DOI: 10.1016/j.infsof.2004.05.004.

[Ale+77] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. New York: Oxford University Press, 1977.

[AM01] Mehmet Aksit and Francesco Marcelloni. "Deferring Elimination of Design Alternatives in Object-Oriented Methods." In: *Concurrency and Computation: Practice and Experience* 13.14 [2001], pp. 1247–1279. DOI: 10.1002/cpe.611.

[Amo+14] Simone da Silva Amorim, Eduardo Santana de Almeida, and John D. McGregor. "Scalability of Ecosystem Architectures." In: *2014 IEEE/IFIP Conference on Software Architecture, WICSA*. IEEE Computer Society, 2014, pp. 49–52. DOI: 10.1109/WICSA.2014.36.

[AN+05] Tariq Al-Naeem, Ian Gorton, Muhammed Ali Babar, Fethi Rabhi, and Boualem Benatallah. "A Quality-driven Systematic Approach for Architecting Distributed Software Applications." In: *27th International Conference on Software Engineering*. ICSE'05. New York, NY, USA: ACM, 2005, pp. 244–253. DOI: 10.1145/1062455.1062508.

[AZ05] Paris Avgeriou and Uwe Zdun. "Architectural Patterns Revisited – A Pattern Language." In: *Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPlop 2005)*. Irsee, Germany, July 2005, pp. 431–470.

[Bab+10] K. Delhi Babu, P. Govinda Rajulu, A. Ramamohana Reddy, and A. N. Aruna Kumari. "Selection of Architecture Styles using Analytic Network Process for the Optimization of Software Architecture." In: *International Journal of Computer Science and Information Security, IJCSIS* 8.1 [2010].

[Bac+05] F. Bachmann, L. Bass, M. Klein, and C. Shelton. "Designing Software Architectures to Achieve Quality Attribute Requirements." In: *Software, IEEE Proceedings* 152.4 [Aug. 2005], pp. 153–165. DOI: 10.1049/ip-sen:20045037.

[Bas+02] Leonard J. Bass, Mark Klein, and Felix Bachmann. "Quality Attribute Design Primitives and the Attribute Driven Design Method." In: *Revised Papers from the 4th International Workshop on Software Product-Family Engineering*. PFE'01. London, UK: Springer-Verlag, 2002, pp. 169–186. DOI: 10.1007/3-540-47833-7_17.

[Bas+03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[BB01] Felix Bachmann and Len Bass. "Managing Variability in Software Architectures." In: *Symposium on Software Reusability: Putting Software Reuse in Context (SSR)*. ACM, 2001, pp. 126–132. DOI: 10.1145/375212.375274.

[BC08] Muhammad Ali Babar and Rafael Capilla. "Capturing and Using Quality Attributes Knowledge in Software Architecture Evaluation Process." In: *First International Workshop on Managing Requirements Knowledge*. IEEE Computer Society, 2008, pp. 53–62. DOI: 10.1109/MARK.2008.3.

[BG07] Muhammad Ali Babar and Ian Gorton. "A Tool for Managing Software Architecture Knowledge." In: *Proceedings of the Second Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent*. SHARK-ADI'07. Washington, DC, USA: IEEE Computer Society, 2007. DOI: 10.1109/SHARK-ADI.2007.1.

[BM11] Artur Boronat and José Meseguer. "Automated Model Synchronization: A Case Study on UML with Maude." In: *Electronic Communications of the EASST* 41 [2011].

[Boe+09] R.C. de Boer, P. Lago, A. Telea, and H. Van Vliet. "Ontology-driven visualization of architectural design decisions." In: *Software Architecture, 2009 European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 51–60. DOI: 10.1109/WICSA.2009.5290791.

[Bos04] Jan Bosch. "Software Architecture: The Next Step." In: *Software Architecture, First European Workshop (EWSA)*. Vol. 3047. Lecture Notes in Computer Science. Springer, 2004, pp. 194–199. DOI: 10.1007/978-3-540-24769-2_14.

[Bos09] Jan Bosch. "From Software Product Lines to Software Ecosystems." In: *Proceedings of the 13th International Software Product Line Conference*. Vol. 446. SPLC'09. San Francisco, California, 2009, pp. 111–119.

[Bre+07] P. Brereton, B. Kitchenham, D. Budgen, M. Turner, and M. Khalil. "Lessons from applying the systematic literature review process within the software engineering domain." In: *Journal of Systems and Software* 80.4 [2007], pp. 571–583. DOI: 10.1016/j.jss.2006.07.009.

[Bu+09] Wanfeng Bu, Antony Tang, and Jun Han. "An Analysis of Decision-centric Architectural Design Approaches." In: *Proceedings of the 2009 ICSE Workshop on Sharing and Reusing Architectural Knowledge*. SHARK'09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 33–40. DOI: 10.1109/SHARK.2009.5069113.

[Bud+11] David Budgen, Andy Burn, and Barbara Kitchenham. "Reporting computing projects through structured abstracts: a quasi-experiment." In: *Empirical Software Engineering* 16.2 [2011], pp. 244–277. DOI: 10.1007/s10664-010-9139-3.

[Bus+00] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture – A System of Patterns*. Wiley, 2000.

[Bus+07a] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture – A Pattern Language for Distributed Computing*. Vol. 4. Wiley Series in Software Design Patterns. New York: John Wiley & Sons Ltd., 2007. ISBN: 0470059028.

[Bus+07b] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture – On Patterns and Pattern Languages*. Wiley Series on Software Design Patterns. Chichester, England: John Wiley & Sons Ltd., Apr. 2007.

[Cap+07] Rafael Capilla, Francisco Nava, and Juan C. Duenas. "Modeling and Documenting the Evolution of Architectural Design Decisions." In: *Proceedings of the Second Workshop on SHAring and Reusing Architectural Knowledge Architecture, Rationale, and Design Intent*. SHARK-ADI'07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 9–15. DOI: 10.1109/SHARK-ADI.2007.9.

[Cap+11] Rafael Capilla, Olaf Zimmermann, Uwe Zdun, Paris Avgeriou, and Jochen Küster. "An Enhanced Architectural Knowledge Metamodel Linking Architectural Design Decisions to other Artifacts in the Software Engineering Lifecycle." In: *Proceedings 5th European Conference Software Architecture*. Vol. 6903. LNCS. Springer, 2011, pp. 303–318. DOI: 10.1007/978-3-642-23798-0_33.

[CB08] Rafael Capilla and Muhammad Ali Babar. "On the Role of Architectural Design Decisions in Software Product Line Engineering." In: *2nd European Conference on Software Architecture (ECSA)*. Vol. 5292. Springer, 2008, pp. 241–255. DOI: 10.1007/978-3-540-88030-1_18.

[CH94] H. Cooper and L. V. Hedges. *The handbook of research synthesis*. New York, NY: Russel Sage Foundation, 1994.

[Che+09] Marsha Chechik, Winnie Lai, Shiva Nejati, Jordi Cabot, Zinovy Diskin, Steve Easterbrook, Mehrdad Sabetzadeh, and Rick Salay. "Relationship-based change propagation: A case study." In: *Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering (MISE)*. MISE'09. IEEE, 2009, pp. 7–12. DOI: 10.1109/MISE.2009.5069890.

[Cho+06] Heeseok Choi, Keunhyuk Yeom, Youhee Choi, and Mikyeong Moon. "An Approach to Quality Achievement at the Architectural Level: AQUA." In: *8th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems*. Vol. 4037. FMOODS'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 20–32. DOI: 10.1007/11768869_4.

[Cho+09] Youhee Choi, Heeseok Choi, and MoonKyun Oh. "An architectural design decision-centric approach to architectural evolution." In: *11th International Conference on Advanced Communication Technology (ICACT), Gangwon-Do, South Korea*. Vol. 1. IEEE Press, 2009, pp. 417–422.

[Cle+02] Paul Clements, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers, and Reed Little. *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002.

[Cle+10] Viktor Clerc, Edwin de Vries, and Patricia Lago. "Using wikis to support architectural knowledge management in global software development." In: *Proceedings of the 2010 ICSE Workshop on Sharing and Reusing Architectural Knowledge*. ACM. 2010, pp. 37–43. DOI: 10.1145/1833335.1833341.

[CN02] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, 2002.

[Cop96] J. Coplien. *Software Patterns*. SIGS Mgt. Briefings. SIGS Books & Multimedia, 1996.

[CP09] Lawrence Chung and Julio Cesar Prado Leite. "On Non-Functional Requirements in Software Engineering." In: *Conceptual Modeling: Foundations and Applications*. Ed. by Alexander T. Borgida, Vinay K. Chaudhri, Paolo Giorgini, and Eric S. Yu. Vol. 5600. Berlin, Heidelberg: Springer-Verlag, 2009. Chap. On Non-Functional Requirements in Software Engineering, pp. 363–379. DOI: 10.1007/978-3-642-02463-4_19.

[CP14] Meiru Che and Dewayne E. Perry. "Architectural Design Decisions in Open Software Development: A Transition to Software Ecosystems." In: *23rd Australian Software Engineering Conference, ASWEC*. 2014, pp. 58–61. DOI: 10.1109/ASWEC.2014.37.

[Cre08] J. W. Creswell. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. 3rd ed. Sage Publications Ltd., 2008.

[CS07] Murray Cantor and John D. Sanders. *Operational IT governance*. Tech. rep. IBM developerWorks, 2007.

[Cui+08] Xiaofeng Cui, Yanchun Sun, and Hong Mei. "Towards Automated Solution Synthesis and Rationale Capture in Decision-Centric Architecture Design." In: *Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA)*. WICSA'08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 221–230. DOI: 10.1109/WICSA.2008.14.

[Cza+05] K. Czarnecki, S. Helsen, and U. Eisenecker. "Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models." In: *Software Process Improvement and Practice* 10.2 [2005], pp. 143–169. DOI: 10.1002/spip.225.

[Dai12] Robert Daigneau. *Service Design Patterns: fundamental design solutions for SOAP/WSDL and restful Web Services*. Addison-Wesley, 2012.

[Der+12] Diego Dermeval, Joao Pimentel, Carla Silva, Jaelson Castro, Emanuel Santos, Gabriela Guedes, Marcia Lucena, and Anthony Finkelstein. "STREAM-ADD - Supporting the Documentation of Architectural Design Decisions in an Architecture Derivation Process." In: *Proceedings of the 2012 IEEE 36th Annual Computer Software and Applications Conference*. COMPSAC'12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 602–611. DOI: 10.1109/COMPSAC.2012.81.

[Dhu+07] Deepak Dhungana, Paul Grünbacher, and Rick Rabiser. "DecisionKing: A Flexible and Extensible Tool for Integrated Variability Modeling." In: *1st International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*. Vol. 2007-01. Lero Technical Report. 2007, pp. 119–128.

[DW11] Hoa Khanh Dam and Michael Winikoff. "An agent-oriented approach to change propagation in software maintenance." In: *Autonomous Agents and Multi-Agent Systems* 23.3 [Nov. 2011], pp. 384–452. DOI: 10.1007/s10458-010-9163-0.

[Eas+08] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. "Selecting Empirical Methods for Software Engineering Research." In: *Guide to Advanced Empirical Software Engineering*. Springer London, 2008, pp. 285–311.

[Eck10] Jutta Eckstein. *Agile Software Development with Distributed Teams: Staying Agile in a Global World*. New York: Dorset House, 2010. ISBN: 978-0-932633-71-2.

[EG04] Alexander Egyed and Paul Grünbacher. "Identifying Requirements Conflicts and Cooperation: How Quality Attributes and Automated Traceability Can Help." In: *IEEE Software* 21.6 [2004], pp. 50–58. DOI: 10.1109/MS.2004.40.

[Egy07] Alexander Egyed. "Fixing Inconsistencies in UML Design Models." In: *29th International Conference on Software Engineering (ICSE)*. IEEE, 2007, pp. 292–301. DOI: 10.1109/ICSE.2007.38.

[EL07] Avner Engel and Mark Last. "Modeling software testing costs and risks using fuzzy logic paradigm." In: *Journal of Systems and Software* 80.6 [2007], pp. 817–835. DOI: 10.1016/j.jss.2006.09.013.

[ES+12] Sascha El-Sharkawy, Stephan Dederichs, and Klaus Schmid. "From Feature Models to Decision Models and Back Again: An Analysis Based on Formal Transformations." In: *16th International Software Product Line Conference (SPLC)*. ACM, 2012, pp. 126–135. DOI: 10.1145/2362536.2362555.

[Fal+10] Davide Falessi, Muhammad Babar, Giovanni Cantone, and Philippe Kruchten. "Applying empirical software engineering to software architecture: challenges and lessons learned." In: *Empirical Software Engineering* 15.3 [2010], pp. 250–276. DOI: 10.1007/s10664-009-9121-0.

[Fal+11] Davide Falessi, Giovanni Cantone, Rick Kazman, and Philippe Kruchten. "Decision-making Techniques for Software Architecture Design: A Comparative Survey." In: *ACM Computing Survey* 43.4 [Oct. 2011], 33:1–33:28. DOI: 10.1145/1978802.1978812.

[Far+08] R. Farenhorst, R. Izaks, P. Lago, and H. Van Vliet. "A Just-In-Time Architectural Knowledge Sharing Portal." In: *Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA)*. Feb. 2008, pp. 125–134. DOI: 10.1109/WICSA.2008.20.

[FB09] Rik Farenhorst and Remco C. Boer. "Knowledge Management in Software Architecture: State of the Art." In: *Software Architecture Knowledge Management: Theory and Practice*. Springer Berlin Heidelberg, 2009. Chap. 2, pp. 21–38. DOI: 10.1007/978-3-642-02374-3_2.

[Fow03] Martin Fowler. *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003, p. 533.

[FP98] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach (2nd Edition)*. PWS, 1998.

[Gam+94] Erich Gamma, Richard Helm, Ralph Johnson, and John M Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[Gau+15] Patrick Gaubatz, Ioanna Lytra, and Uwe Zdun. "Automatic Enforcement of Constraints in Real-time Collaborative Architectural Decision Making." In: *Journal of Systems and Software* [2015]. accepted for publication.

[GB14] Matthias Galster and Muhammad Ali Babar. "Empirical Study of Architectural Knowledge Management Practices." In: *IEEE/IFIP Conference on Software Architecture (WICSA)*. 2014, pp. 239–242. DOI: 10.1109/WICSA.2014.28.

[Gha+12] Yaser Ghanam, Frank Maurer, and Pekka Abrahamsson. "Making the leap to a software platform strategy: Issues and challenges." In: *Information and Software Technology* 54.9 [2012], pp. 968–984. DOI: 10.1016/j.infsof.2012.03.005.

[Grü+03] Paul Grünbacher, Alexander Egyed, and Nenad Medvidović. "Reconciling Software Requirements and Architectures with Intermediate Models." In: *Software Systems Modeling* 3.3 [2003], pp. 235–253. DOI: 10.1007/s10270-003-0038-6.

[GS67] B. Glaser and A. Strauss. *The Discovery of Grounded Theory*. Aldin, 1967.

[HA05] Siw Elisabeth Hove and Bente Anda. "Experiences from Conducting Semi-structured Interviews in Empirical Software Engineering Research." In: *Proceedings 11th IEEE International Software Metrics Symposium*. IEEE, 2005, pp. 23–32. DOI: 10.1109/METRICS.2005.24.

[HA07] Neil B. Harrison and Paris Avgeriou. "Leveraging Architecture Patterns to Satisfy Quality Attributes." In: *First European Conference on Software Architecture (ECSA)*. Vol. 4758. Springer, 2007, pp. 263–270. DOI: 10.1007/978-3-540-75132-8_21.

[HA09] Uwe van Heesch and Paris Avgeriou. "A Pattern-based Approach Against Architectural Knowledge Vaporization." In: *Proceedings 14th Annual European Conference on Pattern Languages of Programs*. Vol. 566. CEUR Workshop Proceedings. CEUR-WS.org, 2009.

[Har+07] Neil B. Harrison, Paris Avgeriou, and Uwe Zdun. "Using Patterns to Capture Architectural Decisions." In: *IEEE Software* 24.4 [2007], pp. 38–45. DOI: 10.1109/MS.2007.124.

[Har+10] Colin Harrison, B. Eckman, R. Hamilton, Perry Hartswick, Jayant Kalagnanam, Jurij Paraszczak, and P. Williams. "Foundations for Smarter Cities." In: *IBM Journal of Research and Development* 54.4 [2010], pp. 1–16. DOI: 10.1147/JRD.2010.2048257.

[Hee+12a] Uwe van Heesch, Paris Avgeriou, and Rich Hilliard. "Forces on Architecture Decisions – A Viewpoint." In: *Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, WICSA/ECSA*. IEEE, 2012, pp. 101–110. DOI: 10.1109/WICSA-ECSA.212.18.

[Hee+12b] Uwe van Heesch, Paris Avgeriou, Uwe Zdun, and Neil Harrison. "The supportive effect of patterns in architecture decision recovery – A controlled experiment." In: *Science of Computer Programming* 77.5 [2012], pp. 551–576. DOI: 10.1016/j.scico.2011.11.008.

[Hee+13] U. van Heesch, P. Avgeriou, and A. Tang. "Does decision documentation help junior designers rationalize their decisions? A comparative multiple-case study." In: *Journal of Systems and Software* 86.6 [2013], pp. 1545–1565. DOI: 10.1016/j.jss.2013.01.057.

[Her+07] Sebastian Herold, Holger Klus, Yannick Welsch, Constanze Deiters, Andreas Rausch, Ralf Reussner, Klaus Krogmann, Heiko Koziolek, Raffaela Mirandola, Benjamin Hummel, Michael Meisinger, and Christian Pfaller. "CoCoME - The Common Component Modeling Example." In: *The Common Component Modeling Example: Comparing Software Component Models [result from the Dagstuhl research seminar for CoCoME, August 1-3, 2007]*. 2007, pp. 16–53. DOI: 10.1007/978-3-540-85289-6_3.

[HG99] James D. Herbsleb and Rebecca E. Grinter. "Architectures, Coordination, and Distance: Conway's Law and Beyond." In: *IEEE Software* 16.5 [Sept. 1999], pp. 63–70. DOI: 10.1109/52.795103.

[Hoo+11] Johan F. Hoorn, Rik Farenhorst, Patricia Lago, and Hans van Vliet. "The Lonesome Architect." In: *Journal of Systems and Software* 84.9 [Sept. 2011], pp. 1424–1435. ISSN: 0164-1212. DOI: 10.1016/j.jss.2010.11.909.

[HW04] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. 2nd. Addison-Wesley, 2004.

[HZ09] Carsten Hentrich and Uwe Zdun. "A Pattern Language for Process Execution and Integration Design in Service-Oriented Architectures." In: *Transactions on Pattern Languages of Programming I*. Ed. by James Noble and Ralph Johnson. Vol. 5770. Lecture Notes in Computer Science. Springer Berlin/Heidelberg, 2009, pp. 136–191. DOI: 10.1007/978-3-642-10832-7_6.

[HZ12] Carsten Hentrich and Uwe Zdun. *Process-Driven SOA: Patterns for Aligning Business and IT*. Infosys Press. CRC Press, 2012.

[Inz+13] Christian Inzinger, Waldemar Hummer, Ioanna Lytra, Philipp Leitner, Huy Tran, Uwe Zdun, and Schahram Dustdar. "Decisions, Models, and Monitoring – A Lifecycle Model for the Evolution of Service-Based Systems." In: *Proceedings of the 17th IEEE International Enterprise Distributed Object Computing Conference*. EDOC'13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 185–194. DOI: 10.1109/EDOC.2013.29.

[ISO11] ISO/IEC/IEEE. "Systems and software engineering – Architecture description." In: *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)* [Jan. 2011], pp. 1–46.

[Jac+99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley Professional, 1999.

[Jan+07] Anton Jansen, Jan Van Der Ven, Paris Avgeriou, and Dieter K. Hammer. "Tool Support for Architectural Decisions." In: *Proceedings of the 6th working IEEE/IFIP Conference on Software Architecture*. IEEE Comp. Soc., 2007. DOI: 10.1109/WICSA.2007.47.

[Jan13] Slinger Jansen. "How Quality Attributes of Software Platform Architectures Influence Software Ecosystems." In: *Proceedings of the 2013 International Workshop on Ecosystem Architectures*. WEA 2013. New York, NY, USA: ACM, 2013, pp. 6–10. DOI: 10.1145/2501585.2501587.

[JB05] Anton Jansen and Jan Bosch. "Software Architecture as a Set of Architectural Design Decisions." In: *The 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*. IEEE Comp. Soc., 2005, pp. 109–120. DOI: 10.1007/978-3-540-88030-1_37.

[JS10] Chris Jensen and Walt Scacchi. "Governance in Open Source Software Development Projects: A Comparative Multi-level Analysis." In: *Open Source Software: New Horizons*. Vol. 319. IFIP Advances in Information and Communication Technology. Springer Berlin Heidelberg, 2010, pp. 130–142. DOI: 10.1007/978-3-642-13244-5_11.

[Kan+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. Carnegie-Mellon University Software Engineering Institute, Nov. 1990.

[Kaz+01] R. Kazman, J. Asundi, and M. Klein. "Quantifying the Costs and Benefits of Architectural Decisions." In: *23rd International Conference on Software Engineering (ICSE)*. 2001, pp. 297–306.

[KC07] Barbara Kitchenham and Stuart Charters. *Guidelines for performing Systematic Literature Reviews in Software Engineering*. Tech. rep. EBSE 2007-001. Keele University and Durham University Joint Report, 2007.

[KF08] Hermann Kaindl and Jürgen Falb. "Can We Transform Requirements into Architecture?" In: *3rd International Conference on Software Engineering Advances (ICSEA), Sliema, Malta*. IEEE Computer Society, 2008, pp. 91–96. DOI: 10.1109/ICSEA.2008.19.

[Kit+02] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. "Preliminary Guidelines for Empirical Research in Software Engineering." In: *IEEE Transactions on Software Engineering* 28.8 [Aug. 2002], pp. 721–734.

[Kit+09] B.A. Kitchenham, O.P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman. "Systematic literature reviews in software engineering – A systematic literature review." In: *Information and Software Technology* 51.1 [2009], pp. 7–15. DOI: 10.1016/j.infsof.2008.09.009.

[Kit+95] B. Kitchenham, L. Pickard, and Shari Lawrence Pfleeger. "Case Studies for Method and Tool Evaluation." In: *Software, IEEE* 12.4 [1995], pp. 52–62. DOI: 10.1109/52.391832.

[Kof+09] Alexander Kofman, Avi Yaeli, Tim Klinger, and Peri Tarr. "Roles, Rights, and Responsibilities: Better Governance Through Decision Rights Automation." In: *Proceedings of the 2009 ICSE Workshop on Software Development Governance*. SDG'09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 9–14. ISBN: 978-1-4244-3736-8. DOI: 10.1109/SDG.2009.5071330.

[Kol+09] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. "On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages." In: *Rigorous Methods for Software Construction and Analysis*. Ed. by Jean-Raymond Abrial and Uwe Glässer. Vol. 5115. Berlin, Heidelberg: Springer-Verlag, 2009. Chap. On the evolution of OCL for capturing structural constraints in modelling languages, pp. 204–218. DOI: 10.1007/978-3-642-11447-2_13.

[Kon+13] Marco Konersmann, Zoya Durdik, Michael Goedicke, and Ralf H. Reussner. "Towards Architecture-centric Evolution of Long-living Systems (the ADVERT Approach)." In: *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*. QoSA'13. New York, NY, USA: ACM, 2013, pp. 163–168. DOI: 10.1145/2465478.2465496.

[Kru+06] Philippe Kruchten, Patricia Lago, and Hans van Vliet. "Building Up and Reasoning About Architectural Knowledge." In: *Proceedings of the Second International Conference on Quality of Software Architectures*. Vol. 4214. QoSA'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 43–58. DOI: 10.1007/11921998_8.

[KS06] Alexander Königs and Andy Schürr. "Tool Integration with Triple Graph Grammars - A Survey." In: *Electronic Notes in Theoretical Computer Science* 148.1 [2006], pp. 113–150. DOI: 10.1016/j.entcs.2005.12.015.

[KZ10] Patrick Könemann and Olaf Zimmermann. "Linking Design Decisions to Design Models in Model-Based Software Development." In: *4th European Conference in Software Architecture (ECSA), Copenhagen, Denmark*. Vol. 6285. Springer, 2010, pp. 246–262. DOI: 10.1007/978-3-642-15114-9_19.

[Küs13] Martin Küster. "Architecture-Centric Modeling of Design Decisions for Validation and Traceability." In: *Proceedings of the 7th European Conference on Software Architecture*. Vol. 7957. ECSA'13. Berlin, Heidelberg: Springer, 2013, pp. 184–191. DOI: 10.1007/978-3-642-39031-9_16.

[Lam03] Axel van Lamsweerde. "From System Goals to Software Architecture." In: *School on Formal Methods*. Ed. by M. Bernardo and P. Inverardi. Vol. LNCS 2804. Springer, 2003, pp. 25–43. DOI: 10.1007/978-3-540-39800-4_2.

[Lan07] Christian Lange. "Model Size Matters." In: *Models in Software Engineering*. Ed. by Thomas Kühne. Vol. 4364. Lecture Notes in Computer Science. Springer Berlin/Heidelberg, 2007, pp. 211–216. ISBN: 978-3-540-69488-5. DOI: 10.1007/978-3-540-69489-2_26.

[Lia+09] Peng Liang, Anton Jansen, and Paris Avgeriou. *Knowledge Architect: A Tool Suite for Managing Software Architecture Knowledge*. Tech. rep. University of Groningen, 2009.

[Lin+07] F. van der Linden, K. Schmid, and E. Rommes. *Software Product Lines in Action - The Best Industrial Practice in Product Line Engineering*. Springer, 2007.

[LK07] Larix Lee and Philippe Kruchten. "Capturing Software Architectural Design Decisions." In: *2007 Canadian Conference on Electrical and Computer Engineering*. IEEE Computer Society, 2007, pp. 686–689. DOI: 10.1109/CCECE.2007.176.

[Lou+08] Neil Loughran, Pablo Sánchez, Alessandro Garcia, and Lidia Fuentes. "Language Support for Managing Variability in Architectural Models." In: *Software Composition*. Vol. 4954. 2008, pp. 36–51. DOI: 10.1007/978-3-540-78789-1_3.

[Lyt+a] Ioanna Lytra, Carlos Carrillo, Rafael Capilla, and Uwe Zdun. "A Survey on Quality Attributes Use in Architectural Design Decisions." submitted to: *ACM Computing Surveys* [submitted in December 2014].

[Lyt+b] Ioanna Lytra, Patrick Gaubatz, and Uwe Zdun. "Two Controlled Experiments on Model-based Architectural Decision Making." submitted to: *Information and Software Technology* [revised in January 2015].

[Lyt+12a] Ioanna Lytra, Stefan Sobernig, Huy Tran, and Uwe Zdun. "A Pattern Language for Service-based Platform Integration and Adaptation." In: *Proceedings of the 17th European Conference on Pattern Languages of Programs*. EuroPLoP'12. Irsee, Germany: ACM, 2012, 4:1–4:27. DOI: 10.1145/2602928.2603080.

[Lyt+12b] Ioanna Lytra, Stefan Sobernig, and Uwe Zdun. "Architectural Decision Making for Service-Based Platform Integration: A Qualitative Multi-Method Study." In: *Proceedings of the 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*. WICSA-ECSA'12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 111–120. DOI: 10.1109/WICSA-ECSA.212.19.

[Lyt+12c] Ioanna Lytra, Huy Tran, and Uwe Zdun. "Constraint-Based Consistency Checking Between Design Decisions and Component Models for Supporting Software Architecture Evolution." In: *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*. CSMR'12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 287–296. DOI: 10.1109/CSMR.2012.36.

[Lyt+13] Ioanna Lytra, Huy Tran, and Uwe Zdun. "Supporting Consistency Between Architectural Design Decisions and Component Models Through Reusable Architectural Knowledge Transformations." In: *Proceedings of the 7th European Conference on Software Architecture*. Vol. 7957. ECSA'13. Montpellier, France: Springer-Verlag, 2013, pp. 224–239. DOI: 10.1007/978-3-642-39031-9_20.

[Lyt+14a] Ioanna Lytra, Huy Tran, and Uwe Zdun. "Harmonizing architectural decisions with component view models using reusable architectural knowledge transformations and constraints." In: *Future Generation Computer Systems* [2014]. DOI: 10.1016/j.future.2014.11.010.

[Lyt+14b] Ioanna Lytra, Holger Eichelberger, Huy Tran, Georg Leyh, Klaus Schmid, and Uwe Zdun. "On the Interdependence and Integration of Variability and Architectural Decisions." In: *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*. VaMoS'14. Sophia Antipolis, France: ACM, 2014, 19:1–19:8. DOI: 10.1145/2556624.2556634.

[Lyt+15] Ioanna Lytra, Gerhard Engelbrecht, Daniel Schall, and Uwe Zdun. "Reusable Architectural Decision Models for Quality-driven Decision Support: A Case Study from a Smart Cities Software Ecosystem." accepted for publication at: *ICSE 3rd International Workshop on Software Engineering for Systems-of-Systems (SESoS)*. 2015.

[LZ13] Ioanna Lytra and Uwe Zdun. "Supporting Architectural Decision Making for Systems-of-systems Design Under Uncertainty." In: *Proceedings of the First International Workshop on Software Engineering for Systems-of-Systems*. SESoS'13. Montpellier, France: ACM, 2013, pp. 43–46. DOI: 10.1145/2489850.2489859.

[LZ14] Ioanna Lytra and Uwe Zdun. "Inconsistency Management Between Architectural Decisions and Designs Using Constraints and Model Fixes." In: *Proceedings of the 23rd Australian Software Engineering Conference*. ASWEC'14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 230–239. DOI: 10.1109/ASWEC.2014.33.

[MA99] E. H. Mamdani and S. Assilian. "An Experiment in Linguistic Synthesis with a Fuzzy Logic Controller." In: *International Journal of Man-Machine Studies* 51.2 [1999], pp. 135–147. DOI: 10.1006/ijhc.1973.0303.

[Mac+91] Allan MacLean, Richard Young, Victoria Bellotti, and Tom Moran. "Questions, Options, and Criteria: Elements of Design Space Analysis." In: *Human-Computer Interaction* 6 [1991], pp. 201–250.

[Mak+08] Majid Makki, Ebrahim Bagheri, and Ali A. Ghorbani. "Automating Architecture Trade-Off Decision Making through a Complex Multi-attribute Decision Process." In: *2nd European Conference on Software Architecture (ECSA)*. Vol. 5292. Lecture Notes in Computer Science. Springer, 2008, pp. 264–272. DOI: 10.1007/978-3-540-88030-1_20.

[Mal+11] Ivano Malavolta, Henry Muccini, and V. Smrithi Rekha. "Supporting Architectural Design Decisions Evolution through Model Driven Engineering." In: *Proceedings of the 3rd International Conference on Software Engineering for Resilient Systems*. Vol. 6968. SERENE'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 63–77. DOI: 10.1007/978-3-642-24124-6_6.

[Mau98] Todd J.; Pierce Heather R. Maurer. "A Comparison of Likert Scale and Traditional Measures of Self-Efficacy." In: *Journal of Applied Psychology* 83.2 [Apr. 1998], pp. 324–329.

[May+09] Christine Mayr, Uwe Zdun, and Schahram Dustdar. "Reusable Architectural Decision Model for Model and Metadata Repositories." English. In: *Formal Methods for Components and Objects*. Ed. by FrankS. de Boer, MarcelloM. Bonsangue, and Eric Madelaine. Vol. 5751. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 1–20. ISBN: 978-3-642-04166-2. DOI: 10.1007/978-3-642-04167-9_1.

[MB09] Harish Mittal and Pradeep Bhatia. "Software maintainability assessment based on fuzzy logic technique." In: *SIGSOFT Software Engineering Notes* 34.3 [May 2009], pp. 1–5. DOI: 10.1145/1527202.1527210.

[MCH11] Mehdi Mirakhorli and Jane Cleland-Huang. "Using Tactic Traceability Information Models to Reduce the Risk of Architectural Degradation during System Maintenance." In: *27th IEEE International Conference on Software Maintenance (ICSM), Williamsburg, VA, USA*. IEEE Computer Society, 2011, pp. 123–132. DOI: 10.1109/ICSM.2011.6080779.

[Men+06] Tom Mens, Ragnhild Van Der Straeten, and Maja D'Hondt. "Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis." In: *9th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*. 2006, pp. 200–214. DOI: 10.1007/11880240_15.

[Mit+10] Anish Mittal, Kamal Parkash, and Harish Mittal. "Software cost estimation using fuzzy logic." In: *IGSOFT Software Engineering Notes* 35.1 [Jan. 2010], pp. 1–7. DOI: 10.1145/1668862.1668866.

[MM03] Nikunj R. Mehta and Nenad Medvidović. "Composing Architectural Styles From Architectural Primitives." In: *Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-11)*. New York, NY, USA: ACM, 2003, pp. 347–350. DOI: 10.1145/940071.940118.

[Moa+08] Shahrouz Moaven, Jafar Habibi, Hamed Ahmadi, and Ali Kamandi. "A Decision Support System for Software Architecture-Style Selection." In: *Proceedings of the 2008 Sixth International Conference on Software Engineering Research, Management and Applications*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 213–220. DOI: 10.1109/SERA.2008.26.

[MR47] H. B. Mann and Whitney D. R. "On a Test of Whether One of Two Random Variables is Stochastically Larger than the Other." In: *Annals of Mathematical Statistics* 18.1 [1947], pp. 50–60.

[MS03] David G. Messerschmitt and Clemens Szyperski. *Software Ecosystem: Understanding an Indispensable Technology and Industry*. Cambridge, MA, USA: MIT Press, 2003.

[MS95] Salvatore T. March and Gerald F. Smith. "Design and Natural Science Research on Information Technology." In: *Decision Support Systems* 15.4 [Dec. 1995], pp. 251–266. ISSN: 0167-9236. DOI: 10.1016/0167-9236(94)00041-2.

[MT00] N. Medvidović and R. N. Taylor. "A classification and comparison framework for software architecture description languages." In: *Software Engineering, IEEE Transactions on* 26.1 [2000], pp. 70–93. DOI: 10.1109/32.825767.

[MV06] Tom Mens and Pieter Van Gorp. "A Taxonomy of Model Transformation." In: *Electronic Notes in Theoretical Computer Science* 152 [Mar. 2006], pp. 125–142. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2005.10.021.

[MW13] Cornelia Miesbauer and Rainer Weinreich. "Classification of Design Decisions: An Expert Survey in Practice." In: *Proceedings of the 7th European Conference on Software Architecture*. Vol. 7957. ECSA'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 130–145. DOI: 10.1007/978-3-642-39031-9_12.

[Nak+13] Agnes Nakakawa, Patrick van Bommel, and Henderik Alex Proper. "Supplementing Enterprise Architecture Approaches with Support for Executing Collaborative Tasks - a Case of TOGAF ADM." In: *International Journal of Cooperative Information Systems* 22.2 [2013]. DOI: 10.1142/S021884301350007X.

[Nen+03] Christian Nentwich, Wolfgang Emmerich, and Anthony Finkelstein. "Consistency Management with Repair Actions." In: *25th International Conference on Software Engineering (ICSE)*. 2003, pp. 455–464.

[Nor+09] Robert Nord, Paul C. Clements, David Emery, and Rich Hilliard. *A Structured Approach for Reviewing Architecture Documentation (CMU/SEI-2009-TN-030)*. Tech. rep. Pittsburgh, Pennsylvania: Software Engineering Institute, Carnegie Mellon University, 2009.

[Now+10] Marcin Nowak, Cesare Pautasso, and Olaf Zimmermann. "Architectural Decision Modeling with Reuse: Challenges and Opportunities." In: *Proceedings of the 2010 ICSE Workshop on Sharing and Reusing Architectural Knowledge*. SHARK'10. Cape Town, South Africa: ACM, 2010, pp. 13–20. DOI: 10.1145/1833335.1833338.

[NP13] Marcin Nowak and Cesare Pautasso. "Team Situational Awareness and Architectural Decision Making with the Software Architecture Warehouse." In: *Proceedings of the 7th European Conference on Software Architecture*. Vol. 7957. ECSA'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 146–161. DOI: 10.1007/978-3-642-39031-9_13.

[Ova+03] Päivi Ovaska, Matti Rossi, and Pentti Marttiin. "Architecture as a Coordination Tool in Multi-site Software Development." In: *Software Process: Improvement and Practice* 8.4 [2003]. Ed. by Daniela Damian. Special Issue: Global Software Development: Growing Opportunities, Ongoing Challenges, pp. 233–247. DOI: 10.1002/spip.186.

[Pef+07] Ken Peffers, Tuure Tuunanen, Marcus Rothenberger, and Samir Chatterjee. "A Design Science Research Methodology for Information Systems Research." In: *Journal of Management Information Systems* 24.3 [Dec. 2007], pp. 45–77. ISSN: 0742-1222. DOI: 10.2753/MIS0742-1222240302.

[Per+09] Daniel Perovich, Pedro O Rossel, and Maria Cecilia Bastarrica. "Feature Model to Product Architectures: Applying MDE to Software Product Lines." In: *Joint Working IEEE/IFIP Conference on Software Architecture European Conference on Software Architecture*. Vol. 11. IEEE, 2009, pp. 201–210. DOI: 10.1109/WICSA.2009.5290806.

[PMSA06] Jorge Enrique Pérez-Martínez and Almudena Sierra-Alonso. "From Analysis Model to Software Architecture: A PIM2PIM Mapping." In: *Model Driven Architecture - Foundations and Applications (ECMDA-FA), Biblao, Spain*. Vol. 4066. 2006, pp. 25–39. DOI: 10.1007/11787044_3.

[Pui+12] Jorge Pinna Puissant, Ragnhild Van Der Straeten, and Tom Mens. "Badger: A Regression Planner to Resolve Design Model Inconsistencies." In: *8th European Conference on Modelling Foundations and Applications (ECMFA)*. Vol. 7349. Springer, 2012. DOI: 10.1007/978-3-642-31491-9_13.

[Raa+08] Bas Raadt, Sander Schouten, and Hans Vliet. "Stakeholder Perception of Enterprise Architecture." In: *Proceedings of the 2nd European Conference on Software Architecture*. ECSA'08. Paphos, Cyprus: Springer, 2008, pp. 19–34. DOI: 10.1007/978-3-540-88030-1_4.

[Rob02] C. Robson. *Real World Research - A Resource for Social Scientists and Practitioner-Researchers*. Blackwell Publishing, 2002.

[Ros04] T.J. Ross. *Fuzzy logic with engineering applications*. John Wiley, 2004.

[RS10] Marko Rosenmüller and Norbert Siegmund. "Automating the Configuration of Multi Software Product Lines." In: *Proceedings of the Fourth International Workshop on Variability Modelling of Software-Intensive Systems*. Vol. 37. ICB-Research Report. Universität Duisburg-Essen, 2010, pp. 123–130.

[Ruo+08] Anna Ruokonen, Vilho Raisanen, Mika Siikarla, Kai Koskimies, and Tarja Systa. "Variation Needs in Service-Based Systems." In: *Proceedings of the 6th European Conference on Web Services*. IEEE Computer Society, 2008, pp. 115–124. DOI: 10.1109/ECOWS.2008.22.

[Sch+00a] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. "Pattern-Oriented Software Architecture." In: Chichester, England: John Wiley & Sons Ltd.Wiley, 2000. Chap. Extension Interface, pp. 141–174.

[Sch+00b] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. "Pattern-Oriented Software Architecture." In: Chichester, England: John Wiley & Sons Ltd.Wiley, 2000. Chap. Interceptor, pp. 109–141.

[Sch+00c] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture – Patterns for Concurrent and Networked Objects*. Vol. 2. Wiley Series in Software Design Patterns. Chichester, England: John Wiley & Sons Ltd.Wiley, 2000.

[Sch+07] Nelly Schuster, Olaf Zimmermann, and Cesare Pautasso. "ADkwik: Web 2.0 Collaboration System for Architectural Decision Engineering." In: *Nineteenth International Conference on Software Engineering and Knowledge Engineering (SEKE)*. Knowledge Systems Institute Graduate School, 2007, pp. 255–260.

[Sch02] Klaus Schmid. "A Comprehensive Product Line Scoping Approach and its Validation." In: *International Conference on Software Engineering (ICSE'24)*. ACM, 2002, pp. 593–603. DOI: 10.1145/581339.581415.

[SD06] Ragnhild Van Der Straeten and Maja D'Hondt. "Model refactorings through rule-based inconsistency resolution." In: *ACM symposium on Applied computing (SAC)*. 2006, pp. 1210–1217. DOI: 10.1145/1141277.1141564.

[SE07] Klaus Schmid and Holger Eichelberger. "A Requirements-Based Taxonomy of Software Product Line Evolution." In: *Electronic Communications of the EASST* 8 [2007].

[Sea99] Carolyn B. Seaman. "Qualitative Methods in Empirical Studies of Software Engineering." In: *IEEE Transactions on Software Engineering* 25.4 [July 1999], pp. 557–572. ISSN: 0098-5589. DOI: 10.1109/32.799955.

[SG96] Mary Shaw and David Garlan. *Software architecture - perspectives on an emerging discipline*. Prentice Hall, 1996.

[Sha+09] M. Shahin, Peng Liang, and M.R. Khayyambashi. "Architectural design decision: Existing models and tools." In: *Software Architecture, 2009 European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*. IEEE, Sept. 2009, pp. 293–296. DOI: 10.1109/WICSA.2009.5290823.

[Sha+10] Mojtaba Shahin, Peng Liang, and Mohammad Reza Khayyambashi. "Improving Understandability of Architecture Design Through Visualization of Architectural Design Decision." In: *Proceedings of the 2010 ICSE Workshop on Sharing and Reusing Architectural Knowledge*. SHARK'10. New York, NY, USA: ACM, 2010, pp. 88–95. DOI: 10.1145/1833335.1833348.

[Sha+11] Mojtaba Shahin, Peng Liang, and Zengyang Li. "Architectural Design Decision Visualization for Architecture Design: Preliminary Results of A Controlled Experiment." In: *Proceedings of the 1st Workshop on Traceability, Dependencies and Software Architecture (TDSA)*. ACM, 2011, pp. 5–12. DOI: 10.1145/2031759.2031762.

[Sil+10] Marcos Aurélio Almeida da Silva, Alix Mougenot, Xavier Blanc, and Reda Bendraou. "Towards Automated Inconsistency Handling in Design Models." In: *CAiSE*. Vol. 6051. 2010, pp. 348–362. DOI: 10.1007/978-3-642-13094-6_28.

[Sin+06] M Sinnema, J Van Der Ven, and S Deelstra. "Using Variability Modeling Principles to Capture Architectural Knowledge." In: *ACM SIGSOFT Software Engineering Notes* 31.5 [2006].

[SJ04] Klaus Schmid and Isabel John. "A Customizable Approach to Full Lifecycle Variability Management." In: *Science of Computer Programming* 53.3 [Dec. 2004], pp. 259–284. DOI: 10.1016/j.scico.2003.04.002.

[Sjo+07] Dag I. K. Sjoberg, Tore Dyba, and Magne Jorgensen. "The Future of Empirical Methods in Software Engineering Research." In: *2007 Future of Software Engineering*. FOSE'07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 358–378. ISBN: 0-7695-2829-5.

[SM11] M. Strembeck and J. Mendling. "Modeling Process-related RBAC Models with Extended UML Activity Models." In: *Information and Software Technology* 53.5 [May 2011], pp. 456–483. DOI: 10.1016/j.infsof.2010.11.015.

[Soc+06] Periklis Sochos, Matthias Riebisch, and Ilka Philippow. "The Feature-Architecture Mapping (FArM) Method for Feature-Oriented Development of Software Product Lines." In: *IEEE International Conference on the Engineering of Computer-Based Systems*. Los Alamitos, CA, USA: IEEE Computer Society, 2006, pp. 308–318. DOI: 10.1109/ECBS.2006.69.

[SP02] Kari Smolander and Tero Päivärinta. "Describing and Communicating Software Architecture in Practice: Observations on Stakeholders and Rationale." In: *Proceedings of the 14th International Conference on Advanced Information Systems Engineering*. Vol. 2348. CAiSE'02. London, UK: Springer, 2002, pp. 117–133.

[SW65] S. S. Shapiro and M. B. Wilk. "An analysis of variance test for normality (complete samples)." In: *Biometrika* 3.52 [1965].

[SZ01] George Spanoudakis and Andrea Zisman. "Inconsistency management in software engineering: Survey and open research issues." In: *in Handbook of Software Engineering and Knowledge Engineering*. World Scientific, 2001, pp. 329–380.

[TA05] J Tyree and A Akerman. "Architecture Decisions: Demystifying Architecture." In: *IEEE Software* 22.2 [2005], pp. 19–27. DOI: 10.1109/MS.2005.27.

[Tan+10] Antony Tang, Paris Avgeriou, Anton Jansen, Rafael Capilla, and Muhammad Ali Babar. "A comparative study of architecture knowledge management tools." In: *Journal of Systems and Software* 83.3 [2010], pp. 352–370. DOI: 10.1016/j.jss.2009.08.032.

[Tea+05] R Development Core Team et al. "R: A language and environment for statistical computing." In: *R foundation for Statistical Computing* [2005].

[Tib+06] Chouki Tibermacine, Régis Fleurquin, and Salah Sadou. "On-Demand Quality-oriented Assistance in Component-based Software Evolution." In: *9th International Conference on Component-Based Software Engineering*. Vol. 4063. CBSE'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 294–309. DOI: 10.1007/11783565_21.

[Tof+] Dan Tofan, Matthias Galster, Ioanna Lytra, Paris Avgeriou, Uwe Zdun, Mark-Anthony Fouche, Remco de Boer, and Fritz Solms. "Empirical Evaluation of GADGET – a Process to Increase Consensus in Group Architectural Decision Making." to be submitted.

[Tof+14] Dan Tofan, Matthias Galster, Paris Avgeriou, and Wes Schuitema. "Past and future of software architectural decisions – A systematic mapping study." In: *Information and Software Technology* 56.8 [2014], pp. 850–872. DOI: 10.1016/j.infsof.2014.03.009.

[Tra+07] Huy Tran, Uwe Zdun, and Schahram Dustdar. "View-based and Model-driven Approach for Reducing the Development Complexity in Process-Driven SOA." In: *International Conference Business Process and Services Computing (BPSC)*. Vol. 116. Lecture Notes in Informatics (LNI), 2007, pp. 105–124.

[Tra+11] Huy Tran, Uwe Zdun, and Schahram Dustdar. "Name-based view integration for enhancing the reusability in process-driven SOAs." In: *International Journal of Business Process Integration and Management* 5.3 [2011], pp. 229–239. DOI: 10.1504/IJBPIM.2011.042527.

[Tra+13] Huy Tran, Ioanna Lytra, and Uwe Zdun. "Derivation of Domain-specific Architectural Knowledge Views from Governance and Security Compliance Metadata." In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. SAC'13. Coimbra, Portugal: ACM, 2013, pp. 1728–1733. DOI: 10.1145/2480362.2480689.

[Tse12] Konstantinos Tselios. "Two empirical studies on decision-making processes in software architecture." MA thesis. the Netherlands: University of Groningen, 2012.

[TT10] Abbas Tashakkori and Charles Teddlie. *Handbook of Mixed Methods in Social & Behavioral Research*. 2nd. Sage Publications, Inc., 2010.

[Ven+06] Jan S. van der Ven, Anton Jansen, Paris Avgeriou, and Dieter K. Hammer. "Using Architectural Decisions." In: *Perspectives in Software Architecture Quality*. Ed. by C. Hofmeister, Crnkovic, R. Reussner, and S. Becker. Universität Karlsruhe, Fakultät für Informatik, 2006, pp. 1–10.

[Vig77] G. Vigderhous. "The level of measurement and 'permissible' statistical analysis in social research." In: *Pacific Sociological Review* 20.2 [1977], pp. 61–72.

[VK07] Vijay K Vaishnavi and William Kuechler. *Design Science Research Methods and Patterns: Innovating Information and Communication Technology*. Auerbach, 2007.

[Vog01] Oliver Vogel. "Service Abstraction Layer." In: *Proceedings of EuroPLoP 2001*. Irsee, Germany, 2001.

[Völ+05] Markus Völter, Michael Kircher, and Uwe Zdun. *Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware*. Software Design Patterns. Chichester, England: John Wiley & Sons Ltd., 2005.

[WG14] Rainer Weinreich and Iris Groher. "A Fresh Look at Codification Approaches for SAKM: A Systematic Literature Review." In: *Proceedings of the 8th European Conference on Software Architecture (ECSA)*. ECSA'14. Berlin, Heidelberg: Springer-Verlag, 2014, pp. 1–16. DOI: 10.1007/978-3-319-09970-5_1.

[Woh+00] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering: An Introduction*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[Woh+03] Claes Wohlin, Martin Höst, and Kennet Henningsson. "Empirical Research Methods in Software Engineering." In: *Empirical Methods and Studies in Software Engineering*. Vol. 2765. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 7–23. DOI: 10.1007/978-3-540-45143-3_2.

[Wu+10] Yali Wu, Frank Hernandez, Francisco Ortega, Peter J. Clarke, and Robert France. "Measuring the effort for creating and using domain-specific models." In: *Proceedings of the 10th Workshop on Domain-Specific Modeling*. DSM'10. New York, NY, USA: ACM, 2010, 14:1–14:6. ISBN: 978-1-4503-0549-5. DOI: 10.1145/2060329.2060360.

[Yin02] Robert K. Yin. *Case Study Research: Design and Methods, 3rd Edition (Applied Social Research Methods, Vol. 5)*. 3rd. SAGE Publications, Inc, Dec. 2002.

[ZA05] Uwe Zdun and Paris Avgeriou. "Modeling Architectural Patterns Using Architectural Primitives." In: *Proceedings of OOPSLA'05*. San Diego, California, USA: ACM, Oct. 2005, pp. 16–20. DOI: 10.1145/1094811.1094822.

[Zad65] L A Zadeh. "Fuzzy sets." In: *Information and Control* 8.3 [1965], pp. 338–353.

[Zal+11] Andrzej Zalewski, Szymon Kijas, and Dorota Sokolowska. "Capturing Architecture Evolution with Maps of Architectural Decisions 2.0." In: *Software Architecture - 5th European Conference, ECSA 2011, Essen, Germany, September 13-16, 2011. Proceedings*. Vol. 6903. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 83–96. DOI: 10.1007/978-3-642-23798-0_9.

[Zdu07] Uwe Zdun. "Systematic Pattern Selection Using Pattern Language Grammars and Design Space Analysis." In: *Software: Practice & Experience* 37.9 [2007], pp. 983–1016. DOI: 10.1002/spe.799.

[Zim+07] Olaf Zimmermann, Thomas Gschwind, Jochen Küster, Frank Leymann, and Nelly Schuster. "Reusable Architectural Decision Models for Enterprise Application Development." In: *3rd International Conference on Quality of Software Architectures (QoSA)*. Vol. 4880. Springer, 2007, pp. 15–32. DOI: 10.1007/978-3-540-77619-2_2.

[Zim+08] Olaf Zimmermann, Uwe Zdun, Thomas Gschwind, and Frank Leymann. "Combining Pattern Languages and Reusable Architectural Decision Models into a Comprehensive and Comprehensible Design Method." In: *Proceedings 7th Working IEEE/IFIP Conference Software Architecture*. IEEE, 2008, pp. 157–166. DOI: 10.1109/WICSA.2008.19.

[Zim+09] Olaf Zimmermann, Jana Koehler, Frank Leymann, Ronny Polley, and Nelly Schuster. "Managing architectural decision models with dependency relations, integrity constraints, and production rules." In: *Journal of Systems and Software* 82.8 [2009], pp. 1249–1267. DOI: 10.1016/j.jss.2009.01.039.

[Zim11] Olaf Zimmermann. "Architectural Decisions as Reusable Design Assets." In: *IEEE Software* 28.1 [2011], pp. 64–69. DOI: 10.1109/MS.2011.3.

[ZM05] Carmen Zannier and Frank Maurer. "A qualitative empirical evaluation of design decisions." In: *ACM SIGSOFT Software Engineering Notes* 30.4 [2005], pp. 1–7. DOI: 10.1145/1082983.1083124.

[ZS09] Uwe Zdun and M. Strembeck. "Reusable Architectural Decisions for DSL Design: Foundational Decisions in DSL Development." In: *Proceedings of 14th European Conference on Pattern Languages of Programs (EuroPLoP 2009)*. Irsee, Germany, 2009, pp. 1–37.

[IEE98] IEEE. "IEEE Std 1061-1998." In: *IEEE Standard for a Software Quality Metrics Methodology* [1998].

[Int00] International Standard CEI/IEC. *Programmable Controllers Part 7: Fuzzy Control Programming*. Tech. rep. 61131-7. IEC-International Electrotechnical Commission, 2000.