An Architectural Decision Modeling Framework for Service-Oriented Architecture Design

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Olaf Zimmermann

aus Salzgitter, wohnhaft in Zürich (Schweiz)

Hauptberichter:	**Prof. Dr. Frank Leymann**
Mitberichter:	**Prof. Dr. Michael Papazoglou**

Tag der mündlichen Prüfung: 17. März 2009

Institut für Architektur von Anwendungssystemen
der Universität Stuttgart
2009

Zusammenfassung

Die Nutzung von Informationstechnologie ist heutzutage für Unternehmen in nahezu allen Branchen unentbehrlich. Unternehmensanwendungen unterstützen die Ausführung von Geschäftsprozessen und automatisieren diese teilweise. Die Entwicklung und Integration qualitativ hochwertiger Unternehmensanwendungen, die als geschichtete und verteilte Softwaresysteme charakterisiert werden können, stellt eine große Herausforderung dar. In den letzen Jahren sind die Konzepte der dienstorientierten Architektur (engl. Service-Oriented Architecture, SOA) zu einem wichtigen Architekturstil für die Entwicklung und Integration von Unternehmensanwendungen herangereift. Aus Benutzersicht betont SOA die geschäftliche Ausrichtung der in Software realisierten Dienste. Aus architektonischer Sicht stellen Modularität, loose Kopplung (d.h. Plattform-, Orts-, Protokoll-, und Formatunabhängigkeit) sowie Schichtenbildung und Flussunabhängigkeit wichtige SOA-Prinzipien dar. Zentrale SOA-Muster sind Dienstnehmer-Geber-Vertrag, unternehmensweiter Dienstbus (engl. Enterprise Service Bus, ESB), Dienstkomposition und Dienstverzeichnis.

Dienstnehmer und -geber sowie ESB- und Dienstkompositions-Infrastruktur müssen zahlreiche nichtfunktionale Anforderungen (NFA) erfüllen. Viele NFA betreffen Software-Qualitätsattribute in Bereichen wie Zuverlässigkeit, Benutzerfreundlichkeit, Effizienz, Wartbarkeit, und Portierbarkeit. Andere NFA resultieren aus unternehmensweiten Architekturrichtlinien und Limitationen von Altanwendungen. Von anderen Architekturstilen bereits bekannte, aber auch neue Herausforderungen sind zu meistern, wie zum Beispiel Schnittstellenvertragsgestaltung und die gleichzeitige Versorgung vieler heterogener Dienstnehmer. Die resultierenden Entwurfsfragen sind nicht einfach zu beantworten; es werden daher Entwurfsmethoden benötigt. Die heute existierenden Entwurfsmethoden stellen viele Dienstidentifikations- und Dienstspezifikationstechniken zur Verfügung; sie decken die Dienstrealisierung jedoch nur unzureichend ab. In der Praxis hat sich gezeigt, dass zur erfolgreichen Realisierung von Diensten hoher Qualität und Entwurfseleganz wesentlich mehr gehört als die Dienste in fachlichen Anforderungen zu identifizieren, Web Services Description Language (WSDL)-Schnittstellen für diese Dienste zu spezifizieren und WSDL-nach-Code-Generatoren aufzurufen: Zahlreiche Architekturentscheidungen müssen getroffen werden.

Die Modellierung von Architekturentscheidungen ist ein aufkommendes Gebiet in der Softwarearchitekturforschung. Im Unterschied zu anderen Notationen für Softwarearchitekturen erfassen Architekturentscheidungsmodelle das Wissen, das zu bestimmten Entwürfen führt und diese begründet (engl. Rationale). Architekturentscheidungen betreffen ein Softwaresystem als Ganzes oder die Kernkompo-

nenten eines derartigen Systems. Architekturentscheidungen bestimmen die nichtfunktionalen Eigenschaften eines Softwaresystems, zum Beispiel seine Qualitätsattribute. Jede Entscheidung behandelt ein konkretes Entwurfsproblem, für das eine oder mehrere Lösungsalternativen auszuwählen sind. Beispiele sind die Wahl von Programmiersprachen und Werkzeugen sowie von Architekturmustern, Integrationstechnologien und Middlewareprodukten.

Zwei Arten von Architekturentscheidungen, die im SOA-Entwurf erforderlich sind, resultieren aus den oben aufgezählten SOA-Mustern: Eine Art von Architekturentscheidungen behandelt Dienstschnittstellenvertragsgestaltung inklusive der Frage der Granularität (z.B. Struktur der ausgetauschten Nachrichten, Gruppierung von Dienstoperationen). Eine andere Art von Architekturentscheidungen betrifft nichtfunktionale Aspekte der ESB-Integration und der Dienstkomposition wie zum Beispiel Nachrichtenaustauschmuster und Systemtransaktionsgrenzen.

Durch die Vorauswahl des Architekturstils können Architekten von einem großen Fundus an Architekturwissen profitieren. Dieses Wissen kann in zwei Bereiche eingeteilt werden: Wissen, das zu der Definition der SOA-Prinzipien und SOA-Muster geführt hat und Wissen, das in Projekten gesammelt wurde, welche die Prinzipien und Muster zuvor angewendet haben. Beide Wissensbereiche beeinflussen die Architekturentscheidungen, die in SOA-Projekten zu treffen sind.

Um Architekten durch den Architekturentscheidungsprozess zu führen, ist eine SOA-Entwurfsmethode erforderlich. Der Entwurf einer derartigen Methode ist das in dieser Arbeit gelöste Problem:

Wie kann das Fällen von Architekturentscheidungen während des SOA-Entwurfs organisiert werden, ausgehend von funktionalen und nichtfunktionalen Anforderungen und bereits gesammeltem Architekturwissen, das in Form von SOA-Prinzipien und -Mustern dokumentiert ist?

Nach dem heutigen Stand der Technik werden Architekturentscheidungen ad hoc und retrospektiv im aktuellen Projekt dokumentiert; dabei handelt es sich um eine zeitintensive Aufgabe ohne unmittelbare positive Auswirkungen. Im Gegensatz dazu untersuchen wir die Rolle, die wieder verwendbare Architekturentscheidungsmodelle während des SOA-Entwurfs spielen können: Wir behandeln wiederkehrende Architekturentscheidungen als genuines Konzept in unserer Methode und stellen ein Architekturentscheidungsmodellierungsrahmenwerk sowie ein wieder verwendbares SOA-Entscheidungsmodell vor, das Architekten durch den Entwurfprozess führt. Unsere Methode arbeitet werkzeuggestützt.

In unserem Rahmenwerk stellen wir eine Technik zur systematischen Identifikation von wiederkehrenden Architekturentscheidungen zur Verfügung. Unser SOA-Entscheidungsmodell ist nach einem Metamodell strukturiert, das Wiederverwendung und Zusammenarbeit unterstützt. Die Modellorganisation folgt den Prinzipien der modellgetriebenen Architektur und separiert länger aktuell bleibende plattformunabhängige Entscheidungen von sich häufig ändernden plattformspezifischen. Auf einer konzeptuellen Ebene werden SOA-Muster referenziert, was die initiale Befüllung und laufende Pflege des Entscheidungsmodells erleichtert. Unser Entscheidungsabhängigkeitsmanagement hilft Architekten, die Modellkonsistenz zu prüfen und irrelevante Entscheidungen gar nicht erst zu betrach-

ten. Eine verwaltete Entscheidungsliste (engl. Managed Issue List) führt durch den Entscheidungsprozess. Um Entscheidungs- und Entwurfsmodelle abzugleichen, werden Entscheidungsausgangsinformationen in Entwurfsmodelltransformationen injiziert. Ein Web-basiertes Kollaborationssystem bietet Werkzeugunterstützung für die Schritte und Konzepte im Rahmenwerk.

Einer der Anwendungsfälle für das Entscheidungsmodellierungsrahmenwerk und das wieder verwendbare SOA-Entscheidungsmodell ist die Nutzung als Entwurfsmethode; weitere Anwendungsfälle sind Ausbildung, Wissensaustausch, Review-Technik und Steuerungsinstrument (engl. Governance Instrument).

Es folgt eine Zusammenfassung der einzelnen Kapitel.

Kapitel 1: Einleitung

In diesem Kapitel führen wir das Anwendungsgenre der Unternehmensanwendungen sowie SOA-Entwurf als Kontext dieser Arbeit ein. Wir geben einen Überblick über den Stand der Technik im Bereich SOA-Entwurfsmethoden und leiten sieben Forschungsprobleme ab. Weiterhin geben wir einen Überblick über unsere Lösung, die aus einem Rahmenwerk für die Modellierung von Architekturentscheidungen, einem wieder verwendbaren Architekturentscheidungsmodell für SOA und Werkzeugunterstützung besteht. Die Kombination von Rahmenwerk und Modell ergibt die gesuchte entscheidungszentrische SOA-Entwurfsmethode.

Kapitel 2: Stand der Technik und Praxisrealität

In diesem Kapitel charakterisieren wir die Herausforderungen für die Konstruktion von Unternehmensanwendungen und stellen SOA als Architekturstil für die Entwicklung und Integration von Unternehmensanwendungen vor. Wir verwenden Architekturprinzipien und -muster, um SOA zu definieren und präsentieren eine motivierende Fallstudie. Wir zeigen den Stand der Technik in den Bereichen Methoden für Software Engineering und Entwurf, Methoden für Softwarearchitekturentwurf, Methoden für Entwicklung und Integration von Unternehmensanwendungen, Methoden für SOA-Entwurf sowie Architekturwissensmanagement auf und stellen einige in der Praxis verwendete Werkzeuge vor.

Kapitel 3: Anforderungen an SOA-Entwurfsmethoden und resultierende Forschungsprobleme

In diesem Kapitel etablieren wir zunächst einen Anforderungskatalog für Methoden, die den SOA-Entwurf unterstützen. Wir verdichten diesen in die für den Entwurf einer entscheidungszentrischen Methode besonders relevanten Anforderun-

gen und formulieren die Forschungsprobleme, die zu lösen sind, damit diese Anforderungen erfüllt werden können. Abschließend verwenden wir Anforderungskatalog und Forschungsprobleme, um die Stärken und Schwächen der den heutigen Stand der Technik repräsentierenden Methoden aus Kapitel 2 zu analysieren.

Kapitel 4: Ein Rahmenwerk zur Modellierung von Architekturentscheidungen im SOA-Entwurf

In diesem Kapitel führen wir das Konzept eines wieder verwendbaren Architekturentscheidungsmodells (engl. Reusable Architectural Decision Model, RADM) ein. Wir unterscheiden zu treffende Entscheidungen (engl. Issues) von bereits getroffenen (engl. Outcomes). Wir definieren ein Rahmenwerk für die Modellierung von SOA-Architekturentscheidungen (engl. SOA Decision Modeling, SOAD), das aus sieben Schritten besteht:

1. Entscheidungen identifizieren.
2. Einzelne Entscheidungen modellieren.
3. Modell strukturieren.
4. Entscheidungen auch zeitlich ordnen.
5. Modell auf Projektbedürfnisse zuschneiden.
6. Entscheidungen treffen unter Verwendung eines Entscheidungsmodells als Architekturentwurfsmethode.
7. Entscheidungen durchsetzen.

Wir erläutern wie sich das Rahmenwerk in den Software-Lebenszyklus integriert und stellen eine Architektur für die Werkzeugunterstützung des Rahmenwerkes vor. Außerdem wenden wir das Rahmenwerk auf den SOA-Entwurf und die motivierende Fallstudie aus Kapitel 2 an.

Kapitel 5: Umfang wieder verwendbarer Architekturentscheidungsmodelle festlegen

Entscheidungen identifizieren. Wir stellen eine neuartige Technik für die Identifikation von wieder verwendbarem Architekturentscheidungswissen in Architekturmustern als Schritt 1 im Rahmenwerk vor:

Welche Architekturentscheidungen kehren wieder im SOA-Entwurf?
Können solche Entscheidungen systematisch in Mustern identifiziert werden?

Die Technik arbeitet mit mehreren Typen von Entscheidungen:

1. *Executive-Entscheidungen* zur Projektdefinition und technischen Projektausrichtung sowie zur Anforderungsanalyse.
2. Entscheidungen zur *Selektion und Adoption von konzeptuellen Mustern*.

3. *Technologieentscheidungen* wie zum Beispiel Wahl von Containern, Protokollen, Betriebssystemen sowie Festlegung von technologiespezifischen Nutzungsprofilen (engl. Technology Profiling).
4. Entscheidungen zur *Auswahl und Konfiguration von Softwareprodukten*.

Wir stellen Identifikationsregeln für diese Typen auf und führen einen Katalog von architekturstilunabhängigen Meta-Entscheidungen als Technikelemente ein.

Kapitel 6: Befüllen wieder verwendbarer Architekturentscheidungsmodelle

Einzelne Entscheidungen modellieren. Für Schritt 2 stellen wir ein Metamodell für die Architekturentscheidungsmodellierung bereit. Es löst das folgende Problem:

Welche Informationen sind für jede zu treffende und wiederkehrende Architekturentscheidung (engl. Issue) zu modellieren?

Aufgrund des Umfangs und der inhärenten Komplexität des Entscheidungswissens hat eine Modellierung der Issues Vorteile gegenüber der Erfassung in strukturiertem oder unstrukturiertem Text. Es ist unerlässlich, ein einheitliches Format zu definieren, um das Entscheidungswissen austauschbar und vergleichbar zu machen. Dazu erweitern wir existierende Arbeiten aus dem Bereich Architekturwissensmanagement. Um die Entscheidungsmodelle wieder verwendbar zu machen, modellieren wir den wiederkehrenden Entscheidungsbedarf, das Issue, getrennt vom projektspezifischen Entscheidungsausgang, dem Outcome.

Modell strukturieren. Schritt 3 behandelt die Organisation eines in den vorherigen beiden Schritten gewonnenen Modells:

Wie können Architekturentscheidungsmodelle in einer intuitiven, anwendungsfallgetriebenen Art und Weise organisiert werden?

Architekturentscheidungsmodelle sind komplex: sie müssen nicht nur die Issues detailliert beschreiben, sondern auch die logischen Beziehungen zwischen den Issues. Eine Organisation nach Verfeinerungsebenen und Architekturschichten stellt die Benutzbarkeit derartiger Modelle sicher. Weiterhin können mit dieser Modellorganisation Entwurfsfehler aufgedeckt werden.

Entscheidungen auch zeitlich ordnen. Um Entscheidungsmodelle als Entwurfsmethode nutzbar zu machen, klären wir im Schritt 4:

Wie können zeitliche Abhängigkeiten von Entscheidungen repräsentiert werden? Wir können die Entscheidungen geordnet werden in Vorbereitung der Nutzung eines Entscheidungsmodells als Methode?

Wir erweitern unser Metamodell um Aspekte der kontextabhängigen, dynamischen Nutzung des Entscheidungswissens. Dieses ermöglicht uns, dem Architek-

ten nur eine Untermenge der Entscheidungen im Modell anzubieten basierend auf bereits getroffenen Entscheidungen. Damit ist der Architekt mit weniger Entscheidungen konfrontiert, was zu einem effizienteren Entscheidungsprozess führt.

Kapitel 7: Erstellen und Benutzen von Architekturentscheidungsmodellen in Projekten

Modell auf Projektbedürfnisse zuschneiden. Nachdem Entscheidungen identifiziert, modelliert und organisiert sind, behandelt Schritt 5:

Wie kann ein wieder verwendbares Architekturentscheidungsmodell auf Projektbedürfnisse angepasst werden?

Das in diesem Schritt eingeführte Konzept ist das Filtern von Entscheidungen.

Entscheidungen treffen. Das in Schritt 6 gelöste Problem ist:

Wie kann ein Architekturentscheidungsmodell als SOA-Entwurfsmethode eingesetzt werden?

Wir führen eine verwaltete Entscheidungsliste (engl. Managed Issue List), einen projektweiten Makroprozess und einen entscheidungsweiten Mikroprozess ein. Um den Methodeneinsatz zu demonstrieren, wenden wir die Prozesse auf die motivierende Fallstudie aus Kapitel 2 an.

Entscheidungen durchsetzen. Schritt 7 führt Modelltransformationen ein, die Architekturentscheidungen als Eingabeparameter berücksichtigen:

Wie kann durchgesetzt werden, dass die getroffenen Entscheidungen bei den weiteren Entwurfs- und Entwicklungsaktivitäten berücksichtigt werden?
Wie kann der Entscheidungsausgang in Entwurfsmodelle und Programmcode eingebracht werden?

Wir fokussieren auf die Beziehung zwischen Architekturentscheidungsmodellierung und modellgetriebener Softwareentwicklung. Zunächst identifizieren wir die benötigen Plattformmodelle und definieren Modelltransformationen innerhalb des Entscheidungsmodells. Anschließend integrieren wir Entscheidungsmodelle mittels Entscheidungsinjektion in die Entwurfsmodelltransformationskette. Das vorgestellte Konzept ergänzt existierende Ansätze zur Durchsetzung von Entscheidungen.

Kapitel 8: Ein Kollaborationswerkzeug für die Modellierung von Architekturentscheidungen

Der konzeptuelle Entwurf und die Implementierung eines Kollaborationssystems für Architekten, welches die Konzepte des Rahmenwerkes zur Modellierung von Architekturentscheidungen unterstützt, stellen den letzten Beitrag der Arbeit dar:

Welche Bausteine muss ein Werkzeug haben, das Architekten beim Untersuchen, Treffen und Durchsetzen von Architekturentscheidungen unterstützt? Wie kann die Zusammenarbeit beim Erstellen und Nutzen von Architekturentscheidungsmodellen unterstützt werden?

Kapitel 9: Praxistest der Beiträge

In diesem Kapitel diskutieren wir, wie wir Rahmenwerk, Entscheidungsmodell und Werkzeug im Hinblick auf Praxisnutzen und Benutzbarkeit validiert haben. Wir klären zunächst die Ziele und Kriterien für die Validierung und stellen unseren Ansatz vor. In einem zweiten Schritt bewerten wir unsere Lösung hinsichtlich des Anforderungkatalogs aus Kapitel 3. Auf diese Selbsteinschätzung folgen die Vorstellung von fünf industriellen Fallstudien und eine Diskussion der Rückmeldungen aus der Zielgruppe. Abschließend behandeln wir ergänzende Validierungstypen wie Selbstexperimente und Schulungen und fassen die Validierungsergebnisse zusammen.

Kapitel 10: Diskussion von Forschungsansatz und Forschungsergebnissen

Dieses Kapitel enthält eine Reflektion über den gewählten Forschungsansatz, eine Interpretation der Validierungsergebnisse hinsichtlich der Stärken und Schwächen des vorgestellten Ansatzes und einen Vergleich mit verwandten Arbeiten. Wir gehen kurz auf die Möglichkeiten zur Implementierung der vorgestellten Konzepte in marktgängigen Architektur- und anderen Entwicklungswerkzeugen ein.

Kapitel 11: Zusammenfassung und Ausblick

Die Arbeit schließt mit einer Zusammenfassung der erzielten Ergebnisse, einem Ausblick auf zukünftige Forschungsaktivitäten sowie einer Vision für eine umfassende Nutzung des Rahmenwerkes und wieder verwendbarer Entscheidungsmodellen im industriellen Umfeld.

Anhang A: Ernten von Architekturentscheidungswissen

Anhang A stellt einen Prozess für das systematische Extrahieren von Wissen über Architekturentscheidungen aus Projektartfakten vor.

Anhang B: Auszug aus dem „RADM for SOA“

Anhang B ist ein Auszug aus dem im Rahmen der Validierung der Arbeit erstellten, wieder verwendbaren Entscheidungsmodell „RADM for SOA“.

Abstract

Enterprises in numerous industries rely on Information Technology (IT) solutions today; enterprise applications support and partially automate the execution of the business processes in these enterprises. It is challenging to develop and integrate such enterprise applications, which can be characterized as logically layered and physically distributed software systems. In recent years, Service-Oriented Architecture (SOA) concepts have matured into an important architectural style for enterprise application development and integration. From a usage perspective, a key principle in SOA is the business alignment of services. From an architectural perspective, SOA principles include modularity, loose coupling (i.e., platform, location, protocol, and format transparency), as well as logical layering and flow independence. Key SOA patterns are service consumer-provider contract, Enterprise Service Bus (ESB), service composition, and service registry.

Service consumers and providers as well as ESB and service composition infrastructure have to fulfill numerous Non-Functional Requirements (NFRs). Many NFRs concern software quality attributes in areas such as reliability, usability, efficiency, maintainability, and portability. Other NFRs result from enterprise architecture guidelines and constraints of legacy systems. Old and new challenges arise, e.g., interface contract design and serving a large number of heterogeneous service consumers. There are no straightforward answers to the resulting SOA design questions; design methods are required. Existing design methods provide many service identification and specification techniques; however, they do not cover service realization sufficiently. Project experience makes evident that there is more to realizing services of quality and style than identifying abstract services in functional requirements, specifying them with technical interface contracts such as Web Services Description Language (WSDL) port types, and applying WSDL-to-code transformation wizards: Many architectural decisions are required.

Architectural decision modeling is an emerging field in software architecture research. Unlike other architecture documentation approaches, architectural decision models capture the architectural knowledge justifying certain designs (rationale). Architectural decisions concern a software system as a whole, or one or more of the core components of such a system. Architectural decisions directly or indirectly determine the non-functional characteristics of a system, e.g., its software quality attributes. Each decision describes a concrete design issue which has several potential solutions (alternatives) that are chosen from. Examples are the selection of programming language and tools, of architectural patterns, of integration technologies, and of middleware assets.

Two areas of architectural decisions required during the SOA design work result from the SOA patterns introduced above: One area of architectural decisions relates to service contract design including the specification of the service granularity (e.g., structure of the messages exchanged, operation grouping). Another area concerns non-functional aspects of ESB integration and service composition such as defining message exchange patterns and system transaction boundaries.

Having preselected the architectural style, architects can benefit from a large body of knowledge. This knowledge can be split into two parts: knowledge that resulted in the definition of the SOA principles and patterns, and knowledge gained on projects that have applied these SOA principles and patterns previously. Both knowledge parts influence the decisions to be made in an SOA project.

To guide architects through the decision making process, a SOA design method is required. The design of such a method is the problem solved by this thesis:

How to facilitate the architectural decision making in SOA design, starting from functional and non-functional requirements and already gathered architectural knowledge captured in SOA principles and patterns?

In the current state of the art, architectural decisions are captured ad hoc and retrospectively on each project, if at all; this is a labor-intensive undertaking without immediate benefits. On the contrary, we investigate the role reusable architectural decision models can play during SOA design: We treat recurring architectural decisions as first-class method elements and propose an architectural decision modeling framework and a reusable decision model for SOA which guide the architect through the SOA design. Our approach is tool supported.

In the framework, we provide a technique to systematically identify recurring decisions. Our reusable architectural decision model for SOA conforms to a metamodel supporting reuse and collaboration. The model organization follows Model Driven Architecture (MDA) principles and separates long lasting platform-independent decisions from rapidly changing platform-specific ones. The alternatives in a conceptual model level reference SOA patterns. This simplifies the initial population and ongoing maintenance of the decision model. Decision dependency management allows knowledge engineers and software architects to check model consistency and prune irrelevant decisions. Moreover, a managed issue list guides through the decision making process. To update design artifacts according to decisions made, we inject decision outcome information into design model transformations. Finally, a Web-based collaboration system provides tool support for the framework steps and concepts.

One of the use cases for architectural decision modeling framework and reusable decision model for SOA is usage as a design method; other use cases are education, knowledge exchange, review technique, and governance instrument.

A summary of each chapter follows.

Chapter 1: Introduction

In this chapter, we introduce the enterprise application genre and SOA design as the context of this thesis. We give an overview of the state of the art regarding SOA design methods and derive seven research problems from it. We outline our solution, which comprises an architectural decision modeling framework and a reusable architectural decision model for SOA, which yield the desired decision-centric SOA design method, as well as related tool support.

Chapter 2: State of the Art and State of the Practice

In this chapter, we first introduce the key characteristics of enterprise applications and SOA as an architectural style for development and integration of such applications. We use architectural principles and patterns to define SOA. To illustrate the state of the practice, we present a motivating case study. Furthermore, we describe the state of the art in software engineering and design methods, software architecture design methods, methods for enterprise application development and integration, SOA design methods, and architectural knowledge management. Finally, we give examples for SOA design tools used in practice.

Chapter 3: SOA Design Method Requirements and Research Problems

In this chapter, we first establish the requirements for methods supporting SOA design. From these requirements, we distill those which are particularly relevant for a decision-centric SOA design method and formulate the research problems to be solved in this thesis to satisfy these requirements. Finally, we use the requirements and research problems to analyze the state-of-the-art methods introduced in Chapter 2 and to demonstrate that the problems have not been properly solved yet.

Chapter 4: An Architectural Decision Modeling Framework for SOA Design

In this chapter we introduce the concept of a Reusable Architectural Decision Model (RADM) separating decisions required (issues) from decision made (outcomes). We introduce a conceptual framework for SOA Decision Modeling (SOAD). The SOAD framework steps are:

1. Identify decisions.
2. Model individual decisions.

3. Structure model.
4. Add temporal decision order.
5. Tailor model.
6. Make decisions, using a decision model as architecture design method.
7. Enforce decisions.

We explain how the framework is positioned in the software engineering process and outline context and architecture of a tool supporting the framework concepts. We apply the framework to the motivating case study from Chapter 2.

Chapter 5: Scoping Reusable Architectural Decision Models

Identify decisions. As SOAD step 1, we present a novel technique for the identification of reusable architectural decision knowledge in architectural patterns:

Which architectural decisions required (issues) recur during SOA design?
Can such decisions be identified systematically in patterns?

The technique works with several types of issues:

1. *Executive decisions* regarding project scoping and technical directions, as well as business requirements analysis.
2. Decisions regarding *selection and adoption of conceptual patterns*.
3. *Technology decisions* concerning the selection and profiling of containers, protocols, operating systems, and the like.
4. Decisions regarding *vendor asset selection and configuration*.

We provide identification rules for the issue types and present a style-independent meta issue catalog as technique elements.

Chapter 6: Populating Reusable Architectural Decision Models

Model individual decisions. A metamodel supporting the modeling of architectural decisions is provided in step 2, solving the following problem:

Which information to model for each architectural decision required (issue)?

Due to the inherent complexity of the architectural decision knowledge, it is beneficial to model the recurring decisions, rather than capture them in structured or unstructured text. It is essential to agree on a common format to make the knowledge exchangeable and comparable. We extend existing work in architectural knowledge management to make decision models reusable: The recurring part, the issue, is separated from the project-specific part, the outcome.

Structure model. Once individual decisions have been identified and documented as described in steps 1 and 2, we can take step 3:

How to organize architectural decision models in an intuitive, use case-driven way?

The model resulting from step 2 is fairly complex: It must provide detailed information about the issues, but also about their logical relations. Organizing the model by refinement levels and architectural layers makes the model comprehensible. Furthermore, design errors can be detected with this model organization.

Add temporal decision order. In support of constraint management and design method usage of decision models, we answer the following questions in step 4:

How to represent temporal dependencies between decisions?
How to order the decisions to prepare for decision making?

Based on the metamodel defined in the previous two steps, context-dependent, dynamic usage of the decision knowledge can be expressed. This allows us to present to the architect only a subset of the decisions to be made based on past decisions. Thus, the architect has to cope with fewer decisions, which leads to a more efficient decision making process.

Chapter 7: Creating and Using Architectural Decision Models on Projects

Tailor model. Having collected, modeled, and organized the required knowledge in the previous four steps we can now take step 5:

How to tailor a Reusable Architectural Decision Model (RADM) for a project?

Decision filtering is the concept we introduce to support this step.

Make decisions. The problem we investigate in step 6 is:

How to use an Architectural Decision Model (ADM) as a SOA design method?

We define a managed issue list, a project-wide macro process, and a decision-wide micro process supporting decision making. To demonstrate the method, we apply it to the motivating case study introduced in Chapter 2.

Enforce decisions. Step 7 introduces decision-aware model transformations as an additional solution to the decision enforcement problem:

How to enforce that made architectural decisions are respected during subsequent design activities and during development?
How to update design models and code according to outcome information in an architectural decision model?

We focus on the relation between architectural decision modeling and model-driven development. First we identify the involved platform models and define model transformations within our decision models. Next we integrate decision models into a design model transformation chain via decision injection. This semi-automatic support for decision enforcement complements existing manual approaches such as coaching.

Chapter 8: A Collaboration Tool for Architectural Decision Modeling

The conceptual design and implementation of a collaboration tool supporting the SOAD framework concepts is the final contribution of this thesis:

Which logical building blocks comprise a tool that supports architects when they investigate, make, and enforce architectural decisions?
How to support collaborative creation and usage of decision models?

Chapter 9: Validation of Research Results

In this chapter we present the validation of SOAD regarding its practical value and usability. We first clarify the validation objectives and scope, present our approach, and give an overview of the results. Next we assess if SOAD meets the requirements for SOA design methods from Chapter 3. After this self assessment we present five industrial case studies. We also feature supplemental evaluation techniques such as self experiments, industry workshops, teaching, and implementation of advanced concepts, and we summarize the validation results.

Chapter 10: Discussion of Research Approach and Results

In this chapter, we reflect upon our research approach and interpret the pros and cons of SOAD that became apparent during the validation. We discuss applicability criteria and compare SOAD with related work. Finally, we outline how the SOAD concepts can be realized in existing architecture design and other tools.

Chapter 11: Conclusions and Outlook

This chapter summarizes the thesis and its contributions. It discusses future work and presents a vision for an extended usage of SOAD in the industry.

Appendix A: Harvesting Architectural Decision Knowledge

Appendix A defines a four step process to syndicate architectural decision model content from architectural decisions made on industry projects.

Appendix B: Excerpt from RADM for SOA

Appendix B contains an excerpt from the Reusable Architectural Decision Model (RADM) for SOA we created during thesis validation.

Acknowledgements

Everybody who has written a book knows it is hard. Having been there, I thought I knew what was coming. It turned out to be harder. There are a number of people who stand out in their essential role in having made this work possible. I would like to thank all of them very sincerely for supporting me on my journey from practice to research.

Prof. Dr. Frank Leymann for his scientific guidance and passion in supervising this thesis, especially for availability despite busy schedules, SOA and product architect knowledge, and moral support in difficult times.

Prof. Dr. Michael Papazoglou for co-supervising this thesis as well as inspiration and advice regarding service-oriented computing methods and other related work.

Past and present members of the Business Integration Technologies Team (BIT) at IBM Zurich Research Laboratory, particularly: Jana Koehler, who provided additional scientific guidance as well as input to the formalization and reviewed selected thesis chapters. Nelly Schuster who developed Architectural Decision Knowledge Wiki and reviewed selected thesis chapters. Ronny Polley who implemented the model formalization. Jochen Küster who reviewed early versions of selected sections. Other Ph. D. students Jussi Vanhatalo, Ksenia Wahler, and Michael Wahler who went through similar learning curves at the same time. Hagen Völzer who provided BPMN consulting. Christian Hoertnagel (late arrival).

Paper co-authors for supporting my research ideas, contributing their architectural knowledge, and providing supplemental technical writing advice: Cesare Pautasso, Stefan Tai, Uwe Zdun, Jonas Grundler, Thomas Gschwind, and Jochen Küster.

All internal and external PhD students at IAAS for inspiring discussions, pointers to related work, and university logistics support. Rania Khalaf (IBM Watson Research Center) for her first-of-a-kind thesis work at IAAS and giveback, including putting up with my many questions about the PhD process at Universität Stuttgart.

Members of the patterns, software architecture, architectural knowledge management, and model-driven development communities for review feedback and many inspiring discussions at conferences, invited talks, guest lectures, and workshops: Paris Avgeriou, Remco de Boer, Grady Booch, Rafael Capilla, Davide Falessi, Rik Farenhorst, Harald Gall, Gregor Hohpe, Anton Jansen, Doug Kimelman, Patrick Könemann, Philippe Kruchten, Patricia Lago, Gerald Reif, Willem-Jan van den Heuvel, Hans van Vliet, and the participants of the Dagstuhl Seminar on Software Service Engineering.

Mark Tomlinson, Ulf Hollberg, Albert Maier, and Frank Müller for reviewing the writeup of the SOAD concepts also from a practitioner's perspective.

IBM ITA profession leaders, product and solution architects, and creators of IBM Global Services Method for teaching me how to architect solutions, for sharing lead architect responsibilities, and/or for taking over responsibilities when I left services to conduct this research. The following individuals serve as community proxies (in alphabetical order): Steve Abrams, Jim Amsden, Dan Berg, Michael Brandner, Kyle Brown, Vadim Dubrovski, Karin Dürmeyer, Peter Eeles, Celso Gonzalez, Kerard Hogg, Petra Kopp, Sven Milinski, Bertrand Portier, Christian Ringler, Philippe Spaas, Gerd Watmann, as well as the members of the IBM SWG Normative Guidance, Team Blueprint, and patterns for e-business working groups.

Members of IBM GTS leadership teams for giving me the opportunity to develop, harden, and validate the research concepts presented in this thesis in close connection with practice: Sesh Murthy, Keith (KC) Goodman, Stefan Pappe, Kavita Chavda, Dale Davis, Tony Shan, Liz Smith, and Marie Wieck.

My past and present management line at IBM Zurich Research Laboratory for supporting my work: Matthias Kaiserswerth, Krishna Nathan, Peter Buhler, Doug Dykeman, Michael Waidner, and Jana Koehler. Phil Janson and Andreas Kind and their teams for joint work, SOA and other architectural decision making domains.

IBM SWG Emerging Internet Technologies and alphaWorks teams for QEDWiki and Architectural Decision Knowledge Wiki release support: David Sink, Krishna Akella, Wing Lee, Jim Smith, Keyur Dalal, Terry Finch, Jim Chao, Brent Zupp, and Lynn Haney.

Executives and technical staff at clients I consulted to from 1999 to 2005 for supplying practical problems and providing me with numerous opportunities to learn and to grow my solution architect experience. Again there are too many to mention, so one group of individuals serves in proxy role: Michael Craes, Guido Oellermann, Guido Ranft, and Reinhold Nolte from Sparkassen Informatik (at that time).

Thomas Kasemir and Caro Funk, IBM SWG, for their hospitality during my many visits to Stuttgart.

Everybody at Gampel 2006, St. Gallen 2007, and Southside 2008.

Last but definitely not least: Family and friends simply for being there before, during, and after writing "crunch mode" – you know who you are!

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ACD	Asset Configuration Decision
ACE	Attempto Controlled English
ACID	Atomicity, Consistency, Isolation, Durability
AD	Architectural Decision
ADD	Attribute-Driven Design
ADM	Architectural Decision Model
a.k.a.	also known as
ASC	Architectural Separation of Concerns
ASD	Asset Selection Decision
ASR	Architecturally Significant Requirement
BA	Business Activity
BAPO	Business Architecture Process and Organization
BDD	Business-Driven Development
BPEL	Business Process Execution Language
BPEL4WS	Business Process Execution Language for Web Services
BPM	Business Process Model, Business Process Modeling
BPMN	Business Process Modeling Notation
BR	Business Rule
CAD	Custom Application Development
CBAM	Cost Benefit Analysis Method
CBDI-SAE	CBDI Service Architecture & Engineering
CEI	Common Event Infrastructure
CMMI	Capability Maturity Model Integration
CORBA	Common Object Request Broker Architecture
CRC	Class, Responsibility, Collaborator
CRUDS	Create, Read, Update, Delete, Search
CT	Communication Transactionality
DAG	Directed Acyclic Graph
DCE	Distributed Computing Environment
DDD	Domain-Driven Design
DDR	Design Decision Rationale
Dojo	(not an acronym)
DSS	Decision Support System
EA	Enterprise Application
EAD	Enterprise Application Development
EAI	Enterprise Application Integration
ECOWS	European Conference on Web Services

EJB	Enterprise JavaBean
EP	Entry Point
ER	Enterprise Resource
ESB	Enterprise Service Bus
FTP	File Transfer Protocol
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IAAS	Institut für Architektur von Anwendungssystemen
IAT	Invocation Activity Transactionality
IC	Integrity Constraint
ICSOC	International Conference on Service-Oriented Computing
IDE	Integrated Development Environment
IDL	Interface Description Language
IIOP	Internet Inter-ORB Protocol
ILD	Integration Layer Decisions
IR	Identification Rule
IT	Information Technology
ITP	Invocation Transactionality Pattern
J2EE	Java 2 Enterprise Edition
JAX-WS	Java XML Web Services
JEE	Java Enterprise Edition
JMS	Java Message Service
JSON	JavaScript Object Notation
KBSE	Knowledge-Based Software Engineering
LC	Legacy Constraint
MDA	Model Driven Architecture
MDD	Model-Driven Development
MOF	Meta Object Facility
MOM	Message-Oriented Middleware
MQ	Message Queuing
MSOA	Mainstream SOA Method
MVC	Model-View-Controller
NFA	Nichtfunktionale Anforderung
NFR	Non-Functional Requirement
OCL	Object Constraint Language
OOA	Object-Oriented Analysis
OOAD	Object-Oriented Analysis and Design
OOP	Object-Oriented Programming
OOPSLA	Object-Oriented Programming, Systems, Languages, and Applications
PAD	Pattern Adoption Decision
PAT	Process Activity Transactionality
PHP	PHP: Hypertext Preprocessor
PIM	Platform-Independent Model
PoEAA	Patterns of Enterprise Application Architecture
POJO	Plain Old Java Object

POSA	Patterns of Software Architecture
POX	Plain Old XML
PSD	Pattern Selection Decision
PSM	Platform-Specific Model
QOC	Question, Option, Criteria
QoS	Quality of Service
RADM	Reusable Architectural Decision Model
RDBMS	Relational Database Management System
REST	Representational State Transfer
RIHA	Review, Integrate, Harden, Align
RPC	Remote Procedure Call
RUP	Rational Unified Process
S4V	Siemens Four Views
SC	System Context
SCA	Service Component Architecture
SCL	Service Composition Layer
SCLD	Service Composition Layer Decisions
SDLC	Service Development Lifecycle
SLD	(Atomic) Service Layer Decisions
SOA	Service-Oriented Architecture
SOAD	SOA Decision Modeling, Service-Oriented Analysis and Design
SOAP	(not an acronym)
SOMA	Service Oriented Modeling and Architecture
SQL	Structured Query Language
SRD	SOA Service Realization Decisions
SSL	Secure Sockets Layer
ST	Service Provider Transactionality
SWOT	Strengths, Weaknesses, Opportunities, Threats
TDD	Test-Driven Design
TLS	Transport Layer Security
TOGAF	The Open Group Architecture Framework
TPD	Technology Profiling Decision
TSD	Technology Selection Decision
Tx	Transaction
UDDI	Universal Description, Discovery, and Integration
UMF	Unified Method Framework
UML	Unified Modeling Language
UNIX	(not an acronym)
UPIS	User, Process and Resource Integrity, Integration, Semantics
URL	Uniform Resource Locator
VP	Viewpoint
WBM	IBM WebSphere Business Modeler
WID	IBM WebSphere Integration Developer
WPS	IBM WebSphere Process Server
w.r.t.	with respect to
WS	Web Service

WSAT	Web Services Atomic Transaction
WSDL	Web Services Description Language
WS-I	Web Services Interoperability
WWW	World-Wide Web
XML	Extended Markup Language
XSD	XML Schema

1 Introduction

In this chapter, we introduce the enterprise application genre and SOA design as the context of this thesis. We give an overview of the state of the art regarding SOA design methods and derive seven research problems from it. We outline our solutions to these problems, which comprise an architectural decision modeling framework, a reusable architectural decision model for SOA, and tool support.

1.1 Context: Enterprise Applications and SOA Design

Enterprises in many industries rely on Information Technology (IT) solutions today; *enterprise applications* such as customer relationship and supply chain management systems support and partially automate the execution of *business processes* such as order management and procurement of production goods. For many functional areas, enterprise applications are available as commercially-off-the-shelf software packages. In other areas, custom development is conducted, either because no suited packages exist or because enterprises seek to gain a competitive advantage with specialized in-house solutions.

It is challenging to develop such custom enterprise applications [Fow03] and to integrate them [HW04]. As logically layered and physically distributed software systems, they have to serve multiple user channels and integrate heterogeneous backend systems. The integrity of the business processes and many underlying resources, for instance the content of databases and message queues, has to be managed. Many fields contribute to the body of knowledge required to solve these design problems. Important areas of research that have been adopted in enterprise application development and integration are relational database management systems [SKS02], transaction processing [GR93], distributed computing [TV03], component-based development [Eme03], business process management [LR00], and software engineering [Som95].

In recent years, *Service-Oriented Architecture (SOA)* concepts [KBS05] have matured into an important architectural style for the enterprise application genre; Web services [ACK+03] and other technologies are used to implement these concepts. In this thesis, we define the SOA style through its architectural *principles* and *patterns*: From a usage perspective, a key principle in SOA is the business alignment of services. From an architectural perspective, the defining principles include modularity, loose coupling (i.e., platform, location, protocol, and format transparency), as well as logical layering and flow independence. Key SOA pat-

terns are service consumer-provider contract, Enterprise Service Bus (ESB), service composition, and service registry. When these patterns are applied, service consumers do not interact with service providers directly but exchange messages via the ESB. Loose coupling is achieved with service registry lookups and ESB capabilities such as message queuing, dynamic routing, and message mediations; logical layering and flow independence are supported by the composition of services into executable workflows [WCL+05].

Service consumers and providers as well as ESB and service composition infrastructure have to fulfill numerous Non-Functional Requirements (NFRs). Many NFRs concern *software quality attributes* [BCK03] in areas such as reliability, usability, efficiency (e.g., performance, scalability), maintainability, and portability [ISO01]. Other NFRs result from *enterprise architecture* [SZ92] guidelines. The constraints of already existing enterprise applications, often called *legacy systems*, form a third source of NFRs.

To satisfy NFRs, the right *architectural decisions* must be made. Architectural decisions concern a software system as a whole, or one or more of the core components of such a system. Architectural decisions directly or indirectly determine the non-functional characteristics of a system [ZGK+07]. Unlike other notations for software architecture design, architectural decision models capture the knowledge justifying a certain design (i.e., its rationale). Each decision describes a concrete design issue for which several potential solutions exist; one or more of these alternative solutions are chosen. Examples are the selection of programming languages and tools, of architectural patterns, of integration technologies, and of middleware assets.

When conducting *SOA design* activities, software architects make architectural decisions when defining the service contracts and when designing service consumers and providers. Two areas of such architectural decisions result from the SOA patterns introduced above: One area of architectural decisions pertains to service contract design including the specification of the service granularity (e.g., structure of the messages exchanged, operation grouping). Another area concerns non-functional aspects of ESB integration and service composition such as defining message exchange patterns and system transaction boundaries [ZZG+08].

Having preselected the architectural style, architects can benefit from a large body of *architectural knowledge*. This knowledge can be split into two parts: knowledge that resulted in the definition of the SOA principles and patterns, and knowledge gained on projects that have applied these SOA principles and patterns previously. Both knowledge parts impact the decisions made in an SOA project.

To guide architects through the decision making process, an SOA design method is required [ZKL07]:

How to facilitate the architectural decision making in SOA design, starting from functional and non-functional requirements and already gathered architectural knowledge captured in SOA principles and patterns?

The design of such a method is the problem solved by this thesis.

1.2 State of the Art and the Practice in SOA Design

A rather large body of related work from different fields is relevant in this design context, for instance software engineering processes [Boe88] and design methods such as object-oriented analysis and design [Boo94], architectural patterns [BMR+96], and software architecture design methods [HKN+07]. Genre-specific methods for enterprise application development and integration [CCS07] and SOA [AGA+08, CBD+06, PV06] also are eligible. Concepts from design decision rationale and architectural knowledge management help to capture and share knowledge about principles, patterns, and their application [LL91, KLV06].

General purpose *software engineering processes* and *design methods* [Boe88, Boo94] organize the required design activities, for instance "define quality attributes" or "specify interface contract". They do not elaborate on concrete quality attributes pertaining to individual SOA design issues, pros and cons of alternatives available, and logical dependencies between them. *Architectural patterns* present proven solutions such as "broker" [BMR+96] and "macroflow" [ZD06]. Such patterns are highly educational, but do not aim at guiding software architects through the genre-specific decision making [ZZG+08].

Software architecture design methods such as the five ones presented in [HKN+07] can also be applied. Moreover, refinements of software engineering processes and design methods that are specific to *enterprise application development and integration* such as Custom Application Development (CAD) in the IBM Unified Method Framework (UMF) [CCS07] exist. Being independent of any architectural style, such assets can not provide SOA-specific design advice, for instance regarding the service contract granularity, ESB integration, and transactionality issues outlined in Section 1.1.

Existing *service modeling methods* [AGA+08, CBD+06, PV06] define the stages of SOA design, for instance service identification, specification, and realization. However, they insufficiently cover service realization and the architectural decisions required to transition from business-level service identification to the instantiation and adaptation of SOA patterns. They address only superficially how to cope with NFRs such as quality attributes and legacy system constraints.

Many of the architectural decisions made during SOA design materialize in the analysis and design artifacts produced. Others are less tangible; however, the literature argues that architectural decisions should be made explicit [KLV06]. In *architectural knowledge management*, metamodels and ontologies for decision capturing exist. Existing work focuses on capturing and representing decisions that have been made already (which we call *outcomes*). It does not advise architects how to anticipate and resolve architectural decisions required (which we call *issues*) when applying SOA principles and patterns in a particular design context on an enterprise application development and integration project.

As a consequence, making architectural decisions remains a challenge for practicing architects [ZGK+07]: Intuition often is the only, but not always a suitable decision driver; educated guesses and personal preferences dominate the decision making. A champion-apprenticeship model is the primary model for education and

knowledge transfer. This leads to low decision maker productivity. It remains hard to trace whether architectural decisions made have been implemented accurately; inconsistencies between architectural documentation and code occur often. Such a lack of rigor in architectural decision making leads to acceptance issues and quality problems with the SOA under construction. The constructed enterprise applications fail to meet stakeholder expectations and project requirements; technical project risk often is high. There is little reuse of architectural knowledge and cross-project collaboration beyond copy-paste of document fragments.

1.3 A Decision-Centric Approach to SOA Design

In this thesis, we reveal how the decision making challenges outlined in Section 1.2 can be overcome. Many of the architectural decisions required (issues) are not specific to any particular project; they recur due to the availability of SOA principles and patterns as well as corresponding technology standards. This allows us to develop a method that follows a novel paradigm: Instead of focusing on process (i.e., responsible roles, activities to be performed, and artifacts to be produced), our method centers on reuse of genre- and style-specific architectural knowledge; it anticipates many of the issues. In our decision-centric method, the architect is presented only eligible issues in the context of decisions already made.

Objectives and use cases. For this purpose, we develop a conceptual *SOA Decision Modeling (SOAD)* framework and a *Reusable Architectural Decision Model (RADM) for SOA*. SOAD is a framework for an active, tool-supported management of issues and decisions made. It has the following use cases:

- *Education*, informing inexperienced architects about the details of the issues, e.g., the NFRs and candidate solutions to consider (alternatives).
- *Knowledge exchange*, facilitating discussions about the issues (and the rationale for certain outcomes) in communities of practicing architects.
- *Design method*, presenting an ordered subset of issues to an architect confronted with a particular design task in a given project context.
- Technical quality assurance *review technique,* allowing architects to analyze and compare architectures via the decisions made.
- *Governance instrument*, customizing SOAD framework and RADM for SOA to establish architectural guidelines for an entire enterprise.

The RADM for SOA is a knowledge repository, capturing decisions required (issues). Taking an *active*, guiding role during the design, it goes beyond the capabilities of a *passive* repository of decisions made. As a side effect, the RADM for SOA captures proven designs as recommendations (often called “best practices”).

We do not propose a Decision Support System (DSS) or automated expert system with artificial intelligence, or design wizard; architectural thinking, taking project-specific requirements into account, is still required when applying SOAD. In the foreseeable future, this requires the skills and experience of humans.

1.4 Research Problems and Contributions Overview

The objective of this thesis is to realize the design method use case for SOAD outlined in Section 1.3. This requires solutions to the following research problems:

- Where and how to *identify* the architectural decisions required during SOA design (issues)?
- Which information to *model* for each identified issue?
- How to *structure* the resulting decision model in a user-friendly way?
- How to represent logical and temporal *decision dependencies*?
- How to use the decision model as an architecture *design method*?
- How to *enforce* that the decisions made are implemented correctly?
- How to support the SOAD framework in a decision modeling tool facilitating *collaboration* between decision makers and other stakeholders?

Solutions to these research problems are the contributions of this thesis; they define the SOAD framework steps. Figure 1 introduces these steps, along with the excerpts from the RADM for SOA used as examples later in this thesis:

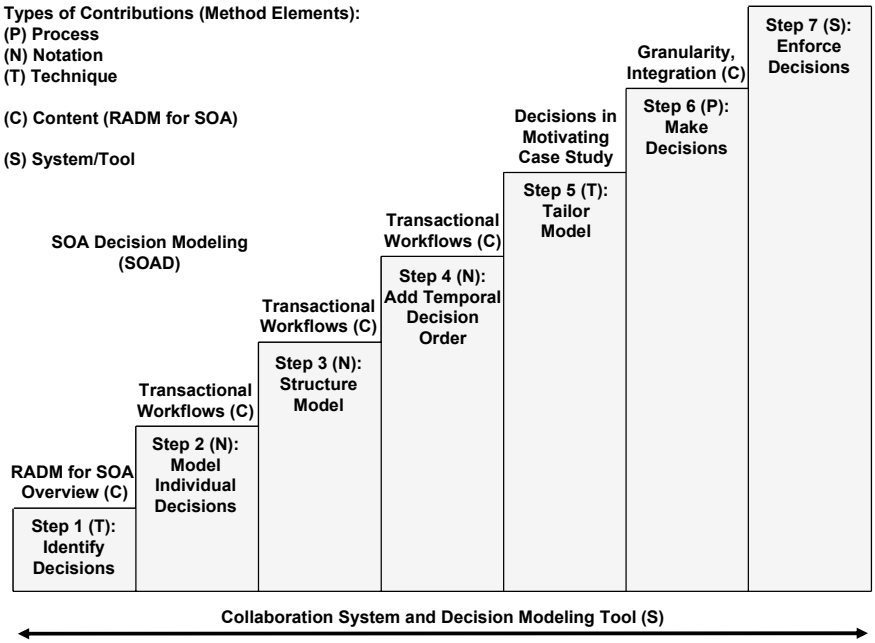


Figure 1. SOA Decision Modeling (SOAD) contributions overview

The SOAD framework is organized in seven steps which contribute process, notation, technique, or system elements. The RADM for SOA is an additional thesis contribution (method content); it supplies examples for all steps. SOAD framework and RADM for SOA yield the desired SOA design method. Our final contribution is the design of a collaboration system providing tool support for SOAD.

Step 1: Identify decisions. To support this step, we provide a novel technique for the systematic identification of reusable architectural decision knowledge. As discussed previously, SOA is an architectural style which we define through principles and patterns; many of the architectural decisions required when applying the patterns recur. The related architectural decision knowledge can be identified systematically and classified into levels of refinement:

1. *Executive decisions* dealing with project scoping and technical directions, as well as business requirements analysis.
2. Decisions about the selection and adoption of *conceptual patterns*.
3. Decisions about *technology choices* such as selection and configuration of containers, protocols, operating systems, and the like.
4. Decisions regarding *vendor asset selection and configuration*.

We provide identification rules for these levels, as well as a catalog of style-independent meta issues. To demonstrate the technique, we use 35 of the 389 issues we captured during thesis validation. The architectural decision knowledge modeled by these issues was harvested from industry projects.

Step 2: Model individual decisions. Due to the inherent complexity of architectural decision knowledge and the many dependencies between decisions, it is beneficial to model the decisions and their dependencies rather than capture them in structured or unstructured text. To support this step, we define a common metamodel. This metamodel extends existing work in architectural knowledge management to make the knowledge in metamodel instances such as our RADM for SOA reusable, exchangeable, and comparable: We separate the recurring part, the *issue*, from the project-specific part, the *outcome*, and introduce attributes that support decision lifecycle management and an alignment with software engineering processes. Decisions required when designing transactional workflows for SOA serve as our example in this step.

Step 3: Structure model. A model resulting from step 2 is fairly complex: It has to provide detailed information about the issues, but also about their dependencies in order to become comprehensive and comprehensible. To make such a model comprehensive we integrate descriptions of the patterns identified in step 1. To make it comprehensible we introduce containment and logical dependency relations, which allow knowledge engineers to organize the model content by refinement levels, architectural layers, and other structuring principles.

Step 4: Add temporal decision order. This step addresses the decision dependency management problem. We express dynamic usage of the architectural decision knowledge by adding temporal dependency relations to our metamodel. This allows us to present to the architect only a subset of the decisions to be made. This subset is calculated from past decisions and other context information. As a consequence, the architect has to cope with fewer decisions, which results in a more efficient decision making process.

Step 5: Tailor model. In this step we demonstrate how to tailor a model for a particular design context on a project. Such tailoring is required to adapt a reusable

decision model for a certain SOA project; issues are removed, updated, and added during the tailoring. We introduce decision filtering as a supporting concept.

Step 6: Make decisions. In this step we leverage the decision model as design method. We define a managed issue list. This list is used in two decision making processes we also introduce: Architects use a (project-wide) macro process and a (decision-wide) micro process to traverse the decision model and choose alternatives solving the issues. The rationale for the selection is recorded in outcomes.

Step 7: Enforce decisions. We focus on the relation between architectural decision modeling and Model-Driven Development (MDD) in this step. We specify the platform models and model transformations within a decision model and present decision injection as a new concept for integrating decision models into a design model transformation chain. This semi-automatic support for decision enforcement complements existing manual approaches such as coaching.

Finally, we propose a *collaboration system* as a tool for decision modeling, making, and sharing which we call *Architectural Decision Knowledge Wiki.* This tool supports the SOAD use cases and framework steps. It is implemented.

1.5 Industrial Use of Presented Solution

Project initiation motivated by practical problems. The base set of architectural knowledge captured by the Reusable Architectural Decision Model (RADM) for SOA originates from industry projects conducted from 2001 to 2005.

The proposed decision-centric method has its roots in these projects, as well as our key hypothesis that the architectural decisions required in SOA design (issues) recur. For instance, we observed that there was significant overlap between the issues we encountered in two projects realizing diverse business services and processes in the finance and the telecommunications industries [ZMC+04, ZDG+05].

Validation approach. Software engineering contributions in general and architecture design methods in particular must be validated in practice. Several non-trivial validation challenges must be overcome: Experiments are costly to set up as there are many influencing factors and the participants must have certain skills and experience. Industry projects face high economic pressure and other external forces that limit their ability to experiment with emerging concepts and technologies. We were able to conduct five industrial *case studies* and supplemental activities such as self experiments and teaching to validate the results of this thesis in an iterative and incremental fashion. *Action research* was applied in two of the cases.

Validation results and industrial adoption. The presented decision-centric SOA design method has been used in ten industry projects. We found evidence for an acceleration of decision identification. The case study participants also reported improvements of decision making quality.

SOAD has already begun to be adopted in practice on a broader scale. We began to train practitioners in SOAD (about 120 at the time of writing). Architectural

Decision Knowledge Wiki was released on IBM alphaWorks, an emerging technologies Web portal. It is in company-internal use within a community of software architects. Leveraging the results of this thesis, IBM Global Technology Services (GTS) has announced an *SOA Architecture Decision Accelerator*.

1.6 Thesis Structure

The remainder of this thesis is structured in the following way. Chapter 2 defines the enterprise application genre as the problem domain (context) addressed by this thesis. It also introduces SOA as an architectural style for enterprise application development and integration. The chapter illustrates the state of the practice in a motivating case study that also serves as a source of examples throughout the thesis. Chapter 2 also introduces the state of the art in software engineering processes and design methods, software architecture design methods, methods for enterprise application development and integration, SOA design methods, and architectural knowledge management. Finally, SOA design tools used in practice are presented.

The following Chapter 3 compiles a set of requirements for SOA design methods and derives the seven research problems outlined in Section 1.4 as well as detailed research questions for them. The chapter also assesses the state of the art to demonstrate that the problems have not been solved satisfyingly so far.

Chapter 4 introduces the SOAD framework and its key concepts. It also positions our work in the software engineering process and outlines the architecture of a tool for SOAD. Finally, it applies SOAD to SOA design.

The following Chapters 5 to 8 then detail the SOAD framework steps and their implementation. Chapter 5 and 6 cover decision model asset creation, beginning with identification of reusable architectural decision knowledge for SOA design (Chapter 5) and progressing to modeling individual decisions, model structuring, and dependency management (Chapter 6). Chapter 5 also gives an overview of the RADM for SOA we created during thesis validation; excerpts from this decision model serve as examples in Chapter 6. Asset consumption is described in Chapter 7, comprising tailoring of decision models, their usage as design method, and decision enforcement in model-driven development. Finally, Chapter 8 presents design and implementation of Architectural Decision Knowledge Wiki, our collaboration system providing tool support for the SOAD steps and concepts.

Chapter 9 presents how we validated SOAD framework, RADM for SOA, and tool. The subsequent Chapter 10 discusses research approach and results, as well as strengths and weaknesses observed in the validation. The chapter also compares SOAD with related work and outlines how our concepts can be supported in commercial tools. Finally, Chapter 11 concludes with a summary of the thesis results, answers to the research questions, and an outlook to future work. It also presents a grand vision for a broader adoption of SOAD in the industry.

There are two appendices: Appendix A presents our bottom-up process for harvesting architectural decision knowledge from projects, and Appendix B contains excerpts from the RADM for SOA developed during thesis validation.

2 State of the Art and State of the Practice

In this chapter, we first characterize the enterprise application genre and introduce Service-Oriented Architecture (SOA) principles and patterns (Section 2.1). Next, we present a case study motivating the state of the practice in SOA design (Section 2.2). Furthermore, we describe the state of the art regarding SOA design methods (Section 2.3), which we will later analyze (Chapter 3) and compare with our solution (Chapter 10). Finally, we present SOA design tools used in practice (Section 2.4).

2.1 Introduction to Problem Domain

In this section, we introduce the enterprise application genre, related development and integration challenges, and SOA as an architectural style for this genre.

2.1.1 The Enterprise Application Genre

Companies in industries such as finance, telecommunications, automotive, as well as retail and distribution rely on Information Technology (IT) systems today. For instance, customer relationship management systems reach out to customers over Web-based self service channels to improve customer satisfaction and retention. In order management scenarios, the IT systems partially automate certain business functions such as inventory checking so that processing times and cost can be reduced. Supply chain management systems integrate business partners into the company-internal processes, which makes the procurement of production goods and other materials more efficient and more flexible.

Such IT systems support the *business processes* in a company, which comprise multiple *business activities* [LR00]. In such a setting, business information is represented in data structures, industry domain-specific algorithms operate on these data structures, and user interfaces display input and output of the algorithms to humans. From an information management perspective, complex and sometimes ambiguously defined entities such as customer profiles, invoices, and bills of material have to be represented in software. The algorithms range from simple data transfer logic to sophisticated calculations and computations as well as long-running process control flows that codify a company's intellectual property. The human-computer interactions deal with a variety of users such as customers and

staff who are served over multiple channels. Business processes execution can be triggered by human users, but also systems such as sensors. First and foremost, such IT systems are distributed systems [TV03]:

Definition 2.1 (Enterprise Application, System Context, User Channel). *An* enterprise application *is a distributed, software-intensive system that automates parts or all of selected business processes and business activities in an enterprise and supports human users during strategic planning and operations. An enterprise application has a* system context*, which we define as the set of its uses relations [BCK03] with primary and secondary actors [RJB99]. A* user channel *is the technical realization of a uses relation between a primary actor and an enterprise application. The actors can be human users or other systems.*

Figure 2 gives an example. An insurance company exposes its customer care, contract, and risk management applications to three types of external and internal human users, its customers, independent agents, and internal back office staff:

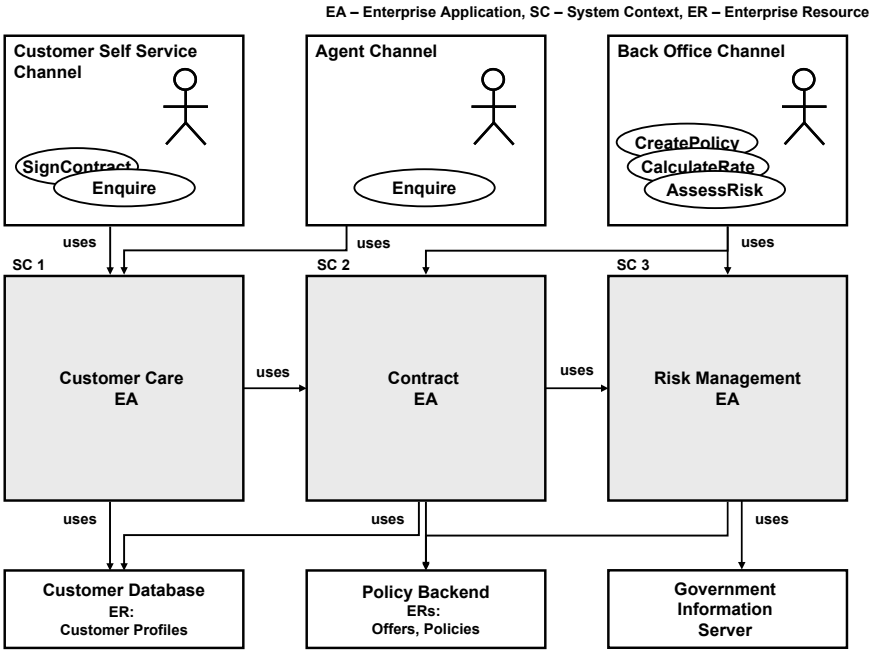


Figure 2. System context diagram for a sample enterprise application landscape

There are three user channels, a customer self service, an agent, and a back office channel. Each of these channels supports one or more business activities initiated by users: enquire, assess risk, calculate rate, sign contract, and create policy. Let us assume that these activities jointly realize a customer enquiry process.

The applications work with a customer database, a policy backend, and a government information server, which appear in the system contexts of the applications. Additional uses relations indicate how the enterprise applications interact

with each other. For instance, the customer care application communicates with the contract application (e.g., when processing a customer or agent enquiry).

To fulfill their responsibilities, the enterprise applications use enterprise resources, which may reside inside or outside their system context boundaries:

Definition 2.2 (Enterprise Resource, Backend System, Backend Channel). *An* enterprise resource *is a persistent data entity which provides business-relevant information to one or more of the enterprise applications in an enterprise. It is stored by one of the enterprise applications using it, which may be a* backend system*. A backend system is an enterprise application whose primary actors are enterprise applications, not human users. A* backend channel *is the technical realization of a uses relation between an enterprise application and a backend system.*

In the example, the three enterprise applications store their enterprise resources (customer profiles, offers, and policies) in the customer database and in the policy backend as shown in Figure 2, which creates uses relations between these systems. The policy backend is only accessed from the contract and the risk management application; it is not exposed to any human user directly. Backend systems that have been developed some time ago are often called *legacy systems*.

The government information server in the system context of the risk management application is an example of an external system that is not operated by the insurance company. It does not host any enterprise resources in the example, but may provide additional statistical data required to perform risk assessments.

2.1.2 Characteristics of Enterprise Applications

Several properties make enterprise applications such as the customer care, contract, and risk management applications in the example difficult to develop and integrate. As distributed systems, enterprise applications must deal with challenges such as addressing, remote communication, workload and failover management, and concurrency [Fow03]. *Software quality attributes* in the areas defined in ISO/IEC specification 9126-2001 [ISO01] must be respected (see Chapter 1).[1] We now refine these general quality attributes in a genre-specific way. User and channel diversity, process and resource integrity, integration needs, and semantic ambiguities are four related characteristics.

User and channel diversity. Many enterprise applications serve multiple human users with rather diverse wants and needs, skills, and IT experience (quality attribute: usability). In automation scenarios, non-human actors such as sensors also must be served. Multiple lines of business and external legal entities (e.g., customers and suppliers) collaborate during business process execution. Hence, multiple user channels must be provided. These channels differ in the way they allow users to interact with an enterprise application (e.g., interactive vs. batch processing).

[1] For instance, an aspect of scalability is that a system is able to cope with a growing number of concurrent users and requests without an unacceptable negative impact on other quality attributes such as performance and usability.

The security requirements, e.g., regarding access control and data privacy, also vary by channel: Channels crossing company boundaries have demanding authentication, authorization, and confidentiality requirements. The number and the nature of the user channels change often over time.

Process and resource integrity. The integrity of business processes must be preserved until they terminate, which may take days, months, or even years (quality attribute: accuracy). In the example (Figure 2), multiple users (i.e., customers and agents) might login to the customer care application simultaneously and interact with it in a conversational fashion (i.e., users send multiple related requests within a single login session). The user request processing has to be coordinated: conversational state must be managed throughout the process lifetime [ZDG+05]. Furthermore, the integrity of the involved enterprise resources (e.g., customer profiles) must be ensured during these conversations throughout the process lifetime: Phantom reads, loss of updates, deadlocks, and other concurrency problems must be avoided [Fow03]. Relational Database Management Systems (RDBMS) [SKS02] and system transactions are commonly used to store enterprise resources persistently and to prevent the concurrency problems from happening. Less advanced data management technologies such as flat, custom formatted files are still used in practice as well; not all of these technologies support transactional invocation. Moreover, in long running processes, system transactions alone are not sufficient to manage the integrity of the enterprise resources; business transactions [Fow03] such as compensation [LR00] are complementary approaches.

Integration needs. Modern enterprises are geographically distributed; virtual enterprises exist. In the insurance industry example, some of the users are external parties; the applications must be physically distributed and provide remote interfaces. Secondly, enterprise applications are strategic assets for a company, which means that their lifecycle often spawns several years (or even decades). Technology evolves over the lifetime of the applications. Thirdly, most modern enterprise applications are composites; already existing software packages, custom developed applications, and external systems must be integrated [HW04]. Finally, enterprise applications often use off-the-shelf middleware assets to manage processes and enterprise resources. The various enterprise applications and the used middleware assets often follow different architectural principles and run on multiple technology platforms (e.g., operating systems, programming languages). Enterprise applications have to cope with such heterogeneity and the resulting integration needs (quality attribute: interoperability).

Semantics. The vagueness and change dynamics of the business information captured by enterprise resources is another challenge (quality attribute: functionality). This challenge is also known as conceptual dissonance [Fow03]. For example, semantics of real-world concepts such as “customer” must be modeled. In our example, a master data management solution might refer to a customer as a “party” and use its own data model to represent parties. It is rather difficult to define such entities precisely so that they are machine readable, as they convey the human understanding of a particular business. Humans are able to interpret data flexibly

(e.g., work with synonyms and homonyms) and to identify and handle exceptional cases in a nondeterministic way. This is more difficult for machines if many ambiguities and contradictions in the IT representation of enterprise resources exist.[2]

When constructing enterprise applications, these challenges must be overcome.

2.1.3 Enterprise Application Development and Integration

To construct enterprise applications, software engineering concepts are applied:

Definition 2.3 (Enterprise Application Development, Enterprise Application Integration). Enterprise Application Development (EAD) *comprises all software engineering activities required to construct an enterprise application, i.e., analysis, design, development, testing, integration, and operations [Som95]*. Enterprise Application Integration (EAI) *provisions messaging and other distributed computing technologies to let enterprise applications with a uses relation exchange information about enterprise resources and invoke each other's functions [HW04]*.

In this thesis, we focus on analysis, design, and integration. During analysis, functional requirements, often articulated as use cases [BMR99], process models [LR00], or user stories [Bec00], describe *what* the features of an enterprise application are; Non-Functional Requirements (NFRs) define quality attributes and constraints regarding *how* an enterprise application delivers this functionality. NFRs usually are captured in free form or structured text, although more rigid approaches have been proposed.

During design, software architectures are viewed from multiple *viewpoints*. An example of such an approach is the 4+1 views model of software architecture [Kru95]. It defines five viewpoints – the logical, the process, the development, the physical, and the scenario viewpoint.[3] The rationale for this is to manage complexity and divide labor without compromising overall integrity and consistency. For instance, the focus on a *logical view* is different from a focus on the *physical view*. The logical view defines *components* (with certain functional responsibilities) and their *connectors*. The physical view focuses on IT infrastructure such as operating system *processes* and hardware *nodes*; the NFRs drive its design. Different skills are required to create logical and physical designs; architectural diagrams for these views range from informal rich pictures to Unified Modeling Language (UML) [RJB99] class diagrams profiled for architecture design to proprietary representations of *deployment units*, nodes, locations, and network topologies [CCS07].

With the support of a viewpoint schema, we can position the EAD and EAI design activities in the *software lifecycle* [Som95] (Figure 3). The top row of Figure 3 shows that the software lifecycle defines a *software engineering process* including *design phases* on EAD and EAI projects. We use terms from the IBM Unified

[2] Fuzzy logic, neuronal networks, advanced database technologies, and the semantic Web movement aim at improving the situation. Even with such support, the challenge remains.

[3] This work was fed into a standard for documenting architecture descriptions, IEEE 1471 (equivalent to ISO / IEC, 42010) [IEEE07]. Other viewpoint schemas exist [CCS07].

Method Framework (UMF) to decompose the design phase into *solution outline*, *macro design*, and *micro design activities* (shown in the middle part of the figure).[4] During these design phases, architects produce the *architecture documentation* (bottom of the figure). Not all design activities qualify as architecture design activities; hence, we position architecture design as a sub-phase of design.

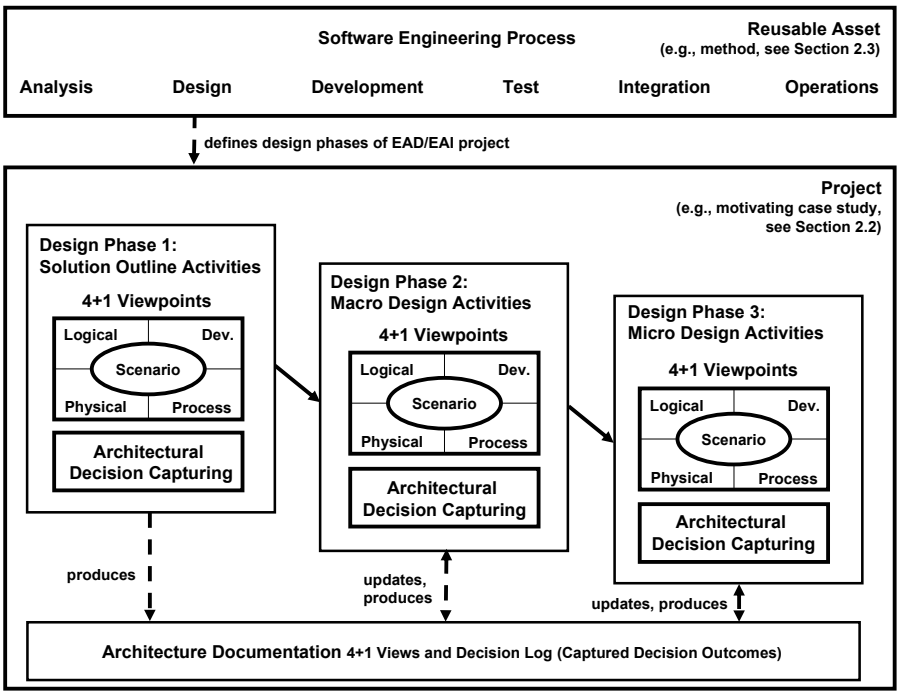


Figure 3. EAD/EAI design activities in software engineering process

The architecture documentation is organized into *views*, which follow a particular *viewpoint scheme*. In this case, we chose the 4+1 views defined by [Kru95]. In each of the three design phases, the views are elaborated progressively. Figure 2 is an example of a diagram that is part of architecture documentation; showing the system contexts of three applications, it provides a scenario view.

To rationalize a design, the *architectural decisions* [KLV06, TA05] should be captured; we defined the term in Chapter 1. The resulting *decision log* becomes an additional architectural view [DC05].

To refine the genre-specific challenges from Section 2.1.2 is part of the analysis activities. The selection of *architectural patterns* [BMR+96] is an example of a design activity. We investigate such patterns next.

[4] Alternatively, we could have used the taxonomy from [Kru03] or [PV06].

2.1.4 Principles and Patterns in SOA Design

When designing an architecture, architects can start from scratch or base their design work on already existing assets. The four user channel (U), process and resource integrity (P), integration (I), and semantics (S) challenges outlined in Section 2.1.2 can also be observed in other software-intensive systems; hence many related *architectural principles* [OG07] and patterns have already been documented, e.g., in object-oriented programming [Mey00, GHJ+95], in distributed computing [BHS07, VKZ04], and in genre-specific literature [Fow03, HW04].

In recent years, the existing principles and patterns have been combined and extended to form an important *architectural style* [BCK03, SG96] for EAD and EAI: *Service-Oriented Architecture (SOA)* [KBS05]. As the term architectural style is used ambiguously in the literature, let us clarify its meaning for this thesis:

Definition 2.4 (Architectural Style). *An* architectural style *consists of a set of architectural principles and patterns that are aligned with each other to make designs recognizable and design activities repeatable: The principles express architectural design intent; the patterns adhere to the principles and are commonly occurring (proven) in practice. They can be combined into workable solutions.*

Architects apply an architectural style to benefit from already gained architectural knowledge and to ensure their solutions to complex design problems are workable. One example of an architectural style is *pipes and filters*, which is applied in UNIX to aggregate shell commands from predefined ones and in the World Wide Web to create composite applications such as Yahoo! Pipes [Yah].

Most existing architectural patterns take a logical viewpoint to define components and connectors that comprise the pattern; patterns for other viewpoints can also be found. The architectural patterns appearing in the definition of an architectural style may be assembled from more primitive ones [ZHD07] or from design patterns [GHJ+95]; if that is the case, we call them *composite patterns*.

As we positioned SOA as an architectural style for EAD and EAI, we can define SOA with application genre-specific principles and patterns. To do so, we take multiple perspectives, (a) a *business analysis* perspective, (b) an *architecture design* perspective, and (c) a *development* perspective. The perspectives correspond to phases in the software engineering process from Figure 3:[5]

Definition 2.5 (Service-Oriented Architecture). *(a) From a* business analysis perspective*, an SOA provides a set of* services *that an organizational unit of an enterprise exposes to its customers, business partners, and company-internal organizational units.* Business alignment *of enterprise applications and underlying IT infrastructure is the principle motivating the introduction of this pattern.*

(b) From an architecture design perspective*, SOA introduces a* service consumer *(a.k.a. requestor), a* service provider*, and a* service contract *(see Definition 2.6). This pattern promotes the principles of* modularity *and* platform transparency. *A composite architectural pattern,* Enterprise Service Bus (ESB)*, governs*

[5] Many other definitions of SOA exist, none of which is precise and detailed enough to base a decision-centric SOA design method on it. Our definition evolved from [Ars04].

the service consumer-provider interactions and physical distribution in support of principles such as protocol transparency *and* format transparency *(Definition 2.7). The* service composition *pattern organizes the processing logic, adhering to the principles of* logical layering *and* flow independence *(Definition 2.8). The* service registry pattern *defines how service providers are looked up; the related principles are* location transparency *and* service virtualization *(Definition 2.9).*

(c) As far as the development perspective *is concerned, SOA provides a* standardization *of an implementation and deployment model, which may be realized by technology standards such as Web services [ACK+03], Service Component Architecture (SCA) [OSOA], and Java Web services [SunWS].*

Table 1 summarizes these perspectives, principles, and patterns defining SOA, and relates them back to the EAD and EAI challenges from Section 2.1.2:

Table 1. SOA principles and patterns

Perspective	Principle (Challenge)	Pattern or Technology
(a) Business analysis	Business alignment (U, S)	Service
(b) Architecture design (logical and physical viewpoint)	Modularity (U, P, I, S)	Service consumer-provider contract
	Platform transparency (I)	
	Protocol transparency (U, I)	Enterprise Service Bus (ESB)
	Format transparency (I)	
	Logical layering (U, P, I)	Service composition
	Flow independence (P, I)	
	Location transparency (I)	Service registry
	Service virtualization (S, I)	
(c) Development	Standardization (I)	Web services specifications
		Service Component Architecture (SCA)
		Java Web services

Figure 4 describes the architectural patterns from Definition 2.5 in a UML class diagram.

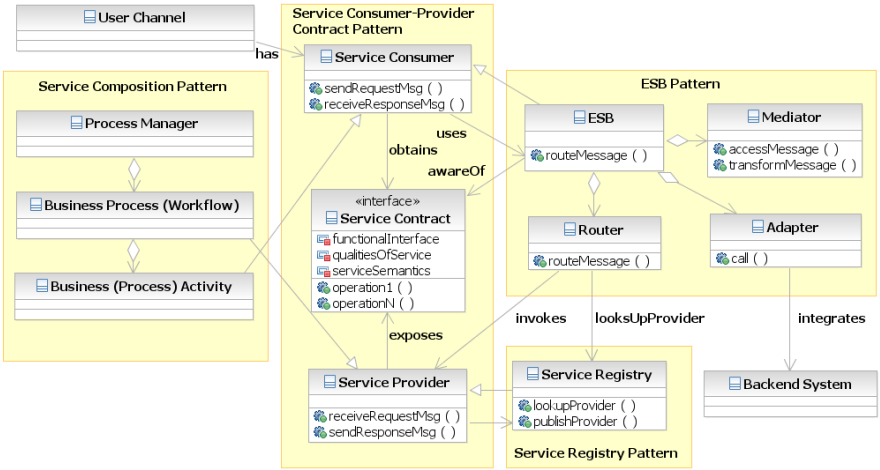


Figure 4. SOA patterns in UML (logical viewpoint)

The essence of the style is the decoupling of service consumer and service provider via the service contract, ESB messaging, and the service registry. We now introduce all four patterns from Definition 2.5 and Figure 4 in detail.

Definition 2.6 (Service Consumer-Provider Contract). *A* service contract *defines a service invocation interface. The contract has a functional and a behavioral part. The functional contract is machine-readable and specifies one or more* operations *which comprise* request *and, optionally,* response messages*. A* service provider *realizes the operations defined in the contract; a* service consumer *invokes them. We jointly refer to a service provider-contract pair as* service*. A service consumer sends a request message to invoke an operation defined in the functional contract; optionally, the service provider returns a result as a response message. The behavioral part of the service contract defines the non-functional characteristics of the message exchange and the operation invocation semantics.*

The motivating principles for this pattern are *modularity* [Mey00] and *platform transparency*. As a foundation of all four patterns, the modularity principle indirectly addresses all four challenges from Section 2.1.2; platform transparency addresses the integration challenge. The service contract separates interface and implementation; it is the only knowledge shared by service consumer and service provider. Security policies are examples of non-functional aspects expressed in the contract (e.g., should a request message be encrypted?). Operation invocation semantics include pre- and postconditions. The service lifecycle (e.g., provisioning and decommissioning of providers) is not exposed in the behavioral part of the contract, i.e., no distributed call stacks or heaps exist, which would require a remote memory management. This gives services an "always-on" appearance from a consumer's perspective. As a consequence, the request and response messages can only be exchanged as documents which do not include any memory references.

Once such contract has been agreed upon, the implementation details of service providers are hidden from the consumers; they can change without effect on service consumers. Consumer and provider do not have to be implemented in the same programming language; they can run on multiple hardware and operating system platforms. Assuming that service invocations do not have any unspecified side effects such as uncoordinated manipulations of enterprise resources, service consumers can share and reuse service providers freely.

The principles and similar patterns are known from object-oriented programming [Mey00] and distributed (client/server) computing [TV03]. Many technology platforms can be used to implement them. If Web services technologies are used, the Web Services Description Language (WSDL) [W3C03] defines functional service contracts. WS-Policy [W3C07] can be used to specify non-functional characteristics. Web Service Semantics (WSDL-S) [AFM+05] annotates WSDL contracts with semantic descriptions; several other notations have been proposed.[6] At runtime, service consumers and providers exchange SOAP [W3C01] messages to transfer request and response documents which are format-

[6] In practice, the service invocation semantics are often specified informally, e.g., in text.

ted according to the WSDL contract. All these technologies are based on XML languages [W3C00].

The next pattern focuses on the integration challenge from Section 2.1.2. It details the message exchange capabilities introduced in Definition 2.6:

Definition 2.7 (Enterprise Service Bus). *The* ESB *pattern is the SOA-specific refinement of the general broker pattern [BMR+96]: An ESB provides a remote communication infrastructure that allows service consumers and service providers to exchange request and response messages using one or more message exchange patterns, communication protocols, and message exchange formats [KBH+04].*

In general, brokers provide many-to-many connectivity between technically diverse and physically distributed communication parties; they decouple the parties from each other. The primary responsibility of an ESB is to *route* request and response messages. Introducing a central ESB and a service registry (Definition 2.9) creates a hub-and-spoke architecture known from EAI middleware; a direct communication variant of the broker pattern also exists, in which the communication partners know about each other and address each other directly [BMR+96]. ESBs support message exchange patterns such as synchronous request-reply invocations, asynchronous one way messaging, and publish-subscribe [HW04].

Unlike traditional message brokers, an ESB is aware of the type and structure of the messages exchanged: The ‚S' in the pattern name refers to a machine-readable service contract as introduced in Definition 2.6. The ‚E' in the pattern name indicates that the ESB must provide architectural qualities that make it possible to overcome the integration challenges outlined in Section 2.1.2. For instance, high volumes of messages have to be processed, possibly exchanging large amounts of data over local or wide area networks when transferring enterprise resource information (data). Channels serving human users must respond instantly; sub-second response times are often required. If devices such as sensors and actuators are integrated into the SOA to monitor and control the physical environment, communication may have to happen in real time.

The World Wide Web (WWW) as a distributed communication infrastructure partially implements the ESB pattern, providing universal connectivity over a single protocol, HTTP [W3C04]. Advanced ESBs also provide *protocol transparency*. Multiple communication protocols are supported, including HTTP, but also asynchronous message queuing, e.g., via Java Message Service (JMS) providers [SunJMS], and, with the help of *adapters*, proprietary protocols used by legacy systems [KBH+04]. Other advanced ESB capabilities are content- and workload-based routing and *mediations*. Mediations transform request and response messages if service consumer and provider use different formats. This provides *format transparency* to service consumers and providers. ESBs also provide access to the message payload (i.e., request and response message data) for security and systems management purposes, e.g., authentication, authorization, monitoring, and billing. The service contract is interpreted to process the payload.

The next pattern addresses the process and resource integrity challenges from Section 2.1.2. This is required when a large number of services are integrated:

Definition 2.8 (Service Composition). *If two or more service providers are assembled into an additional service provider we speak of* service composition. *This additional service provider invokes the assembled service providers via their service invocation interfaces. Service composition may form a dedicated architectural layer in an SOA, which we call* service composition layer.

If an enterprise application employs SOA principles such as modularity, many different service providers with rather diverse responsibilities may exist, e.g., technical logging services, atomic business functions such as customer lookups and address validations, and entire business processes such customer enquiries and claim processing in the insurance industry example. The characteristics of these service providers differ. To avoid a tight coupling between service consumers and providers with different responsibilities and quality attributes as well as undesired dependencies between the services, the permitted invocations must be defined. For instance, a process service may be permitted to invoke a business function service, but not to invoke a technical service directly. Similarly, a technical logging service should be unaware of its consumers and not call a process service itself. Otherwise, a change to the interface of the process service, which is required to respond to a change in the business requirements, requires the technical utility service to be changed as well (if the change is not backward compatible). This violates the modularity principle and degrades the maintainability and portability of the application.

As a solution, the SOA should be organized into three or more *logical layers*. Selecting the layers pattern [BMR+96] is an architectural decision driven by the desire for structure and flexibility: Architectural elements in a particular logical layer fulfill a certain architectural responsibility jointly and cohesively. They only interact with each other and with architectural elements in adjacent lower layers; interfaces isolate the layers from each other. As a result, layer implementations can seamlessly switch from one technology to another, without causing a need to change the architectural elements contained within one of the adjacent layers.

Traditionally three logical layers are used in EAD [Fow03]: The *presentation layer* contains all rich or thin client logic displaying user interfaces to human users. In an SOA, many service consumers reside in the presentation layer. The *domain layer* contains business logic such as control flow, but also calculations and modifications of enterprise resources. It is typically activated in response to stimuli from the presentation layer or from other systems (e.g., when realizing business event and timeout management). The *data source layer* lets enterprise resources and other data persist. It also provides interfaces allowing the domain layer to access the data when executing its logic.

The service composition pattern refines the above logical layering scheme: The domain layer is divided into two sub-layers, a *service composition layer* and an *atomic service layer*. Service providers either reside in the atomic service layer or, as composed services, in the service composition layer. The implementations of atomic services in a programming language may also reside in the atomic service

layer or be placed in an additional *component layer*. The ESB and the service registry patterns are co-located in a separate *integration layer* [Ars04].[7]

Business activities that are assigned to end users[8] are placed in the service composition layer. The service composition layer keeps track of the conversational state. Business activities invoke services in the atomic service layer. The atomic service layer contains calculations and manipulations of enterprise resources, which are not permitted to invoke services in the service composition layer. A business *process manager* [HW04] (e.g., a *workflow management system* [LR00]) is the central middleware component in the service composition layer. It is aware of the business activities that have to be performed and the appropriate order, which defines an executable business process control flow (a.k.a. *workflow*). Each process manager can host more than one process; it is responsible for creating and terminating process instances, and relating incoming requests to such process instances (correlation). Such process instances may run for a long time: The process manager can ensure that the logical order of the process execution is adhered to and that the integrity of enterprise resources is preserved throughout the process lifetime and across user channels (coordination). This includes handling logical and technical processing errors (e.g., invalid request data, network connectivity problems). The process manager can also ensure that process instances complete in a timely manner if that is a business requirement.

Having divided the business logic this way, *flow independence* can be achieved; just like presentation and domain layer are unaware of the way a relational database stores the enterprise resource data and optimizes access to it (providing data independence [SKS02]), the basic computations in the atomic service layer are unaware of the way they are composed into business processes.

If workflow patterns and technologies are used to realize the service composition layer, the formal foundations for its execution semantics can be Petri nets, Pi-calculus, or graph theory [LR00]. One technology option is the Web Services Business Process Execution Language (WS-BPEL or, in short, BPEL), which evolved from several proprietary languages and has been standardized [OAS07].

Our fourth and final pattern addresses the integration and semantics challenges from Section 2.1.2, extending the ESB pattern:

Definition 2.9 (Service Registry). *A service registry provides information about services that can be invoked via the ESB. It makes service contracts and service provider access information available to the ESB and to service consumers. Organizational information such as service ownership, service level agreements, and billing information can optionally be stored in the service registry as well.*[9]

To ensure flexibility during deployment and service invocation, service consumers should not use fixed service provider addresses; ideally, they should even be unaware of the actual service provider and let the ESB decide where to route a

[7] At present, no single SOA layering scheme has been agreed upon; many proposals exist.

[8] End users are primary or secondary actors (in UML terminology [RJB99]) in the system context with business relevance. Actors can be human users or other systems.

[9] If used at design time, the service registry is also referred to as service repository. We use the terms interchangeably in this thesis.

service request to (e.g., for load balancing purposes). To provide such *location transparency* is the objective of the service registry pattern.

A service registry provides a design time interface to architects and developers which allows these users to publish and lookup service contracts and providers. At runtime, a service registry may also act as a service provider, so that ESB and service consumers in other applications have access to the information about service contracts and service providers that is stored in the registry.

Selecting service providers at runtime is an advanced usage scenario for a service registry; such dynamic lookup requires semantic annotations that can be used to automate the provider lookup based on the Quality of Service (QoS) expected by a consumer. Service consumers and providers are no longer aware of each other (*service virtualization*). Many open research and industry adoption challenges exist, such as trust, negotiation, and monitoring of dynamically negotiated service level agreements.

This pattern is the SOA pendant of naming and directory services known from the Common Request Broker Architecture (CORBA) [OMG04], Java Enterprise Edition (JEE) [SunJEE], Distributed Computing Environment (DCE) [OG97], and other remoting middleware. The Universal Description, Discovery, and Integration (UDDI) [OAS04] specifications realize the service registry pattern in a Web services context; vendor-specific UDDI extensions and alternative realizations exist. An example is the IBM WebSphere Service Repository and Registry (WSRR) [IBM]. A detailed analysis of the novelties of the pattern and its implementation alternatives is out of scope of this thesis.

With application genre characterized and SOA principles and patterns defined, we can define what we mean by SOA design in this thesis:

Definition 2.10 (SOA design). SOA design *comprises all architecture design activities on EAD and EAI projects employing SOA principles and patterns.*

2.2 State of the Practice: Motivating Case Study

To demonstrate the state of the practice in EAD and EAI and to motivate the design issues that occur in SOA design, we introduce a scenario from the insurance industry in this section. The scenario concerns a fictitious company, which we already used in [ZTP03]. Business scenario, requirements, and technical design considerations in the case originate from real SOA projects conducted in several industries, e.g., [ZMC+04, ZDG+05]. Due to space constraints, we simplify the case. However, we present it in such a way that it is still representative for the state of the practice and helps to motivate our research problems.

2.2.1 An Insurance Industry Scenario: Customer Enquiry Processing

Let us assume that PremierQuotes Inc., a fictitious insurance company, acquired DirtCheap Insurance, another fictitious insurance company, and formed the PremierQuotes Group to fulfill the growth expectations of its stakeholders [ZTP03].

Shortly after the takeover, a strategic initiative to improve the customer enquiry processing is established by the executive management. The objectives of the initiative are to *improve customer service*, measured by the conversion rate (i.e., ratio between accepted offers and enquiries processed), and *increase profit* by not making an offer if there is a high risk of fraudulent claims.

To contribute to the strategic initiative, the Chief Information Officer (CIO) launches an EAD and EAI project, with the goal to develop a new process-centric customer enquiry system which reuses logic from existing customer care, contract, and risk management applications operated by the two merged companies. We introduced these enterprise applications in Figure 2 on page 10. Let us further assume that the policy backend and the risk management application are COBOL applications running on the IBM System z platform [IBM]. The contract application is a Java Enterprise Edition (JEE) application [SunJEE]. Customer care is a Web application consisting of PHP scripts. An external data source, currently provided by a government information server available on the Internet, has to be integrated, providing the crime statistics (fraud history) for a certain geographical area in a proprietary file format.

2.2.2 Business Process Model

To understand the business needs and solicit functional requirements from a scenario viewpoint, a *Business Process Model (BPM)* can be created. Such analysis-phase BPMs are typically created by business domain experts, not software architects or workflow technology specialists.[10]

Figure 5 gives an example, using the Business Process Modeling Notation (BPMN). The users of the new system are prospective customers, independent agents, and the PremierQuotes Group back office staff. Each horizontal swim lane in the diagram represents one user channel shown in Figure 2 (page 10). The *business event* triggering the process execution is a customer enquiring about an offer for a certain type of insurance, e.g., health care. The processing can either start because a prospective customer enquires about insurance over the self service channel or because an agent enquires on behalf of the customer. The following business activities are conducted by the involved parties: *request offer*, *assess risk*, *calculate rate*, *receive offer*, *sign contract*, and *create policy*. The enterprise resources are displayed as documents accessed or manipulated by the business activities (e.g., customer profile in assess risk activity).

[10] Use case modeling [RJB99] is another requirements analysis technique; the agile community favors user stories [Bec00] over BPM and use cases.

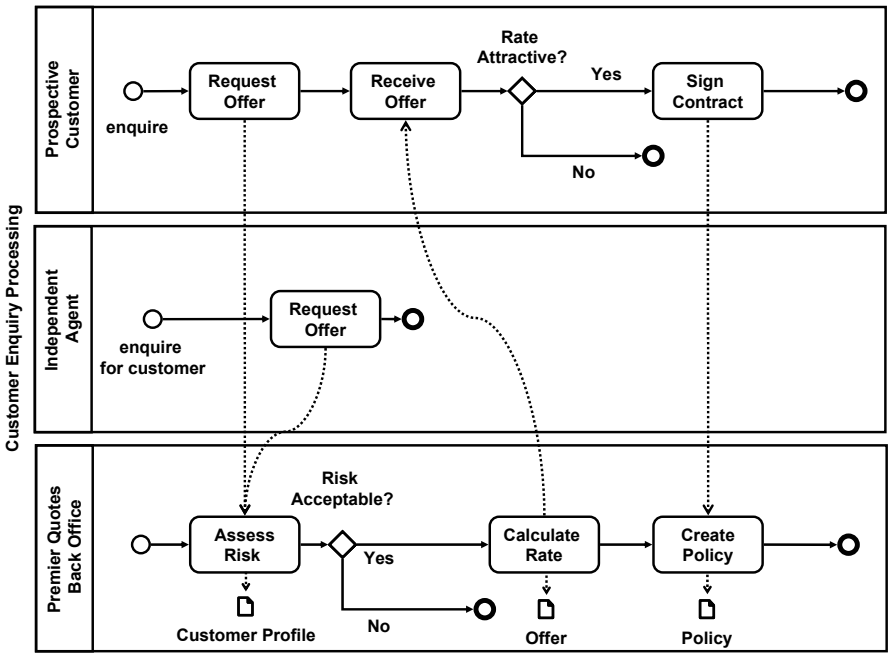


Figure 5. Analysis-phase BPM for customer enquiry processing

Note that the processing is not fully specified: It is not clear what happens in the customer swim lane if no offer is made and what happens in the back office if the customer does not sign the contract until a certain date. Such information could be added to the diagram. However, even if such information is added, the diagram does not provide enough information to design an SOA or implement any Web services. Important technical information is missing:

- User information such as their location and supporting IT infrastructure (e.g., hardware, user channel middleware, and existing applications).
- Data structures transferred from one business activity to another, e.g., from request offer to assess risk, and data definitions for enterprise resources, e.g., customer profile, offer, and policy.
- Request correlation and process instance management required to overcome the process and resource integrity challenges (see Section 2.1.2).
- Technical error handling needs (e.g., network and server timeouts).
- Quality of Service (QoS) requirements and other non-functional concerns, e.g., regarding transactionality, security, and reliability.
- Availability and interfaces of involved enterprise applications (e.g., legacy systems), network topology layout, and used integration middleware.

Such technical aspects are not expressed as the BPM is created during business requirements analysis. The model can be exported to BPEL to become directly executable in a BPEL process manager that implements the service composition pat-

tern (Definition 2.8). However, such a direct execution often is not sufficient for production workflows as the source model does not cover the above design concerns sufficiently.[11]

2.2.3 Business Rules, NFRs, and Legacy Constraints

More information about the requirements than that provided by the analysis-phase BPM is required so that the customer enquiry processing system can be designed. We now summarize the architecturally significant *business rules, non-functional requirements*, and *legacy constraints*. These requirements concretize the generic user channel, process and resource integrity, integration, and semantics challenges from Section 2.1.2. They are solicited during the analysis phase of the project.

Business Rules (BRs) capture functional requirements that can not be expressed easily in control-flow oriented BPMs or stimulus-response-based use case models:

BR 1. Only one offer should exist per customer at any given time.

BR 2. To improve customer service and conversion rate, PremierQuotes Group must respond to a prospective customer within three working days.

BR 3. A prospective customer can see the status of the enquiry processing via a Web application, but not the detailed justification for the calculated rate, or any information belonging to the profiles of other existing and prospective customers.

BR 4. The back office must always be able to obtain up-to-date information about the processing (enquiry status). All customer-facing activities must be monitored.

BR 5. If the customer does not accept an offer within two weeks (sign contract activity), the enquiry processing is terminated and archived in the customer database (so that the archived information can be used later for risk calculation purposes).

Non-Functional Requirements (NFRs) state *how* a system performs its functions, rather than *what* functions it provides:[12]

NFR 1. The system must be able to handle up to 50 concurrent users. This NFR has been calculated based on the number of active agents, the existing customers and the business growth strategy. The average customer self service and agent session length is ten minutes; one customer enquiry process instance is triggered per session. Back office users are logged in permanently during regular business hours and trigger up to one business activity per second.

NFR 2. New communication protocols and interfaces should be built on mature, open industry standards if these standards are supported by at least two vendors.

---

[11] This is not a language limitation, but a role and phase issue: Defining technical properties is not a responsibility of business domain experts analyzing the functional requirements.

[12] The NFRs may differ per user channel and per used backend system. They should be specified per business activity or use case, not globally (a common mistake in practice).

NFR 3. Other business processes must be able to work with a customer profile while a customer enquiry process is using it. Update conflicts must be prevented.

NFR 4. Sub-second response time is required in the customer self service and the agent channels (for all processing steps defined in the analysis-phase BPM). The system must be available during working hours, after hours, and on weekends.

NFR 5. The request volumes are expected to grow if the business strategy succeeds. The new customer enquiry application should be portable, as an IT infrastructure migration project is currently underway. Additional functionality is likely to be required in future releases; for instance, the board members of PremierQuotes Group have already expressed a desire to run business simulations.

Legacy Constraints (LCs) are a special type of NFRs. LCs are architecturally relevant characteristics of other systems appearing in the system context of the enterprise applications that can not be changed within the scope of the current project. The backend and external systems in the case introduce the following LCs:

LC 1. The only interface to the customer database is a Structured Query Language (SQL) [SKS02] interface provided by a Relational Database Management System (RDBMS). To format customer profiles, the customer database interface uses identifiers and data types that are different from those understood by the contract application.

LC 2. The policy backend offers a synchronous Remote Procedure Call (RPC) [HW04] interface. The risk management application provides an asynchronous Message-Oriented Middleware (MOM) interface [HW04]; both of these interfaces are non-transactional from a consumer's perspective.

LC 3. The government information server, which must be integrated to be able to perform the risk assessment, does not provide an online interface. A proprietary file transfer format is defined. A batch job can be run to obtain the fraud history data from a Website accessible via FTP. When a request file is uploaded, it takes up to 24 hours until result data or an error report becomes available for download.

Our exemplary compilation of BRs, NFRs, and LCs is not complete; on industry projects, several hundred of such requirements typically have to be dealt with.

2.2.4 Candidate Architectures

To illustrate the complexity of the application genre and the need for architectural decision making, we present two conceptual architectures for the case now: A traditional client-server architecture and one based on SOA principles and patterns.

Client-server architecture. Figure 6 illustrates a *three-tier client-server architecture* [ACK+03, OHE99] for the customer enquiry system, not yet using any SOA pattern.

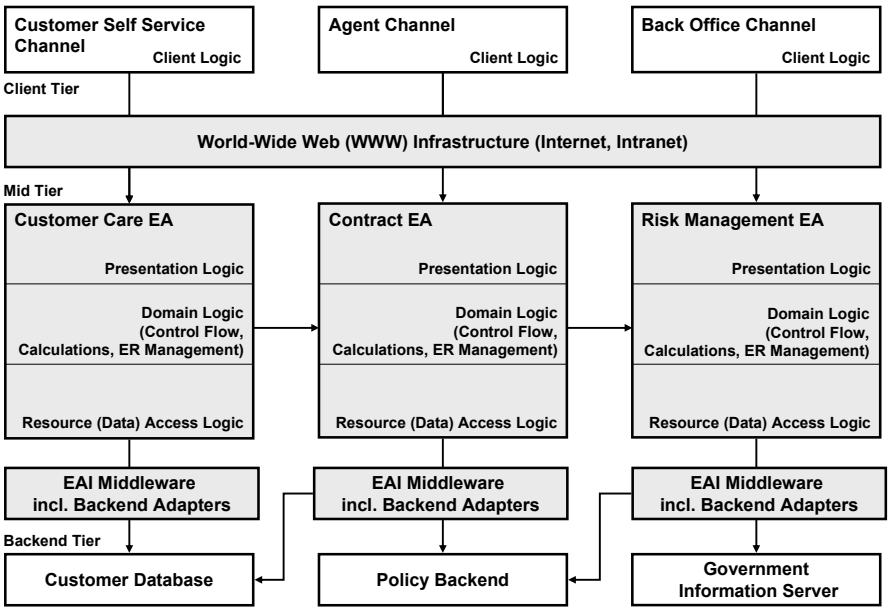


Figure 6. Customer enquiry architecture 1 (three-tier client-server)

The three physical tiers are the *client tier*, the *mid tier* hosting presentation, domain, and resource (data) access logic, and the *backend tier*. World-Wide Web (WWW) infrastructure connects the client tier with the mid tier (over the Internet for the customer self service channel and the agent channel, over an intranet for the back office channel). Traditional Enterprise Application Integration (EAI) middleware is used to connect the mid tier with the backend tier.

The client tier contains all application components directly serving the users appearing in the system context diagram (Figure 2 on page 10) and the analysis-phase BPM (Figure 5 on page 23). Examples are Web browsers and rich client applications running on Personal Computers (PCs) used by customers, agents, and back office staff. This tier is out of scope of our integration-centric case study.

The mid tier comprises the three applications shown in the system context diagram. These applications are logically layered[13] into *presentation*, *domain*, and *resource (data) access logic* layers. Typical responsibilities of the mid tier are input validation, processing control, session state management, calculations, and manipulations of enterprise resources in response to the EAD and EAI challenges discussed in Section 2.1.2. We detail the architecture of this tier in Figure 7.

The backend tier stores enterprise resources persistently and coordinates concurrent access to the enterprise resources (i.e., customer profiles, offers, and policies). This tier hosts database servers, but also other systems which in themselves may be physically tiered, but are located external to the company or in another or-

[13] Tiers provide separation of concerns in the physical viewpoint, layers in the logical one.

ganizational domain. The policy backend and the government information server are examples. This tier is out of scope of our case study as well.

Figure 7 decomposes the mid tier of the contract application according to Fowler's and Brown's layering scheme [Fow03] and shows the logical components required to realize the contract application:

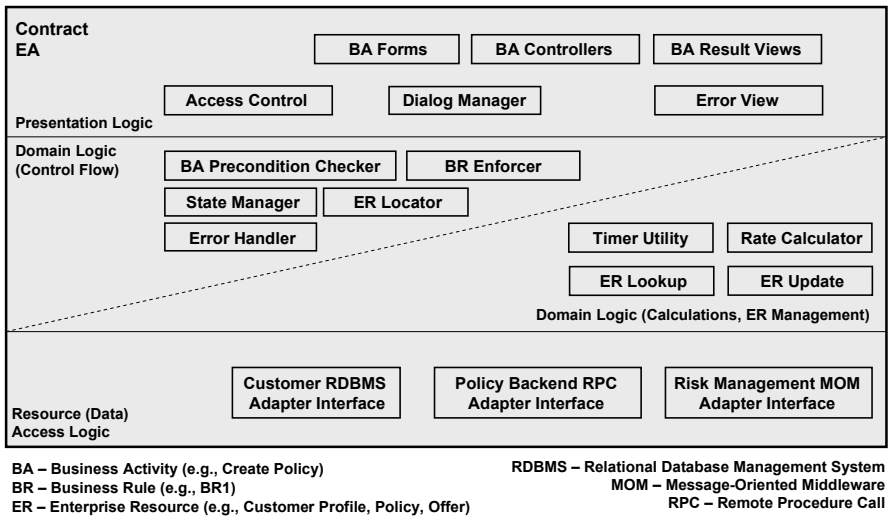


Figure 7. Logical decomposition of mid-tier layers

The presentation logic validates user input, controls which domain logic to invoke for each incoming request, and prepares the result views. The domain logic verifies whether all prerequisites for request processing are met, coordinates the requests, and controls the processing flow (upper part); it also performs calculations and creates, reads, updates, and deletes enterprise resources such as customer profiles, offers, and policies (lower part). The resource (data) access logic connects the mid tier with other enterprise applications and the backend tier (here: customer database, policy backend, and risk management application via adapters). We do not introduce all logical components in detail here, but refer the reader to the literature [Fow03, ZDG+05]. The mid tiers of the other two applications (customer care, risk management) have similar logically layered architectures.

All business activities specified in the analysis-phase BPM and related use cases can be supported with such architecture. One of its strengths is that the logical layering principle is followed; the layers in the three applications can be developed independently of each other. Another advantage is that design time reuse can be achieved through code libraries [Mey00]. For instance, the policy ER lookup component and the timer utility may also be needed in the customer care application.[14]

However, it is rather difficult to maintain overall consistency of the processing state (as required to satisfy BR 1, 2, 4, and 5) because the contract application is

[14] In practice, such code is often duplicated due to organizational matters.

only one of three applications; it can influence the processing in the customer care and in the risk management applications only via information exchanges over the EAI middleware or via the customer RDBMS. Business activity monitoring, time-out management, and business transaction management as required by the business rules and NFRs from Section 2.2.3 can be implemented this way; however, this is a labor-intense and error-prone undertaking. The required custom code often is hard to maintain.

Another drawback is the peer-to-peer approach to integration (i.e., no *hub-and-spoke* broker [HW04] is used). Each application has its own resource (data) access logic and adapter; a change in a backend interface (e.g., addition of a parameter) causes changes in all adapters. The format transparency principle is not followed.

SOA. Figure 8 outlines an alternative architecture, now employing SOA principles and patterns to organize the tiers: The system context and application boundaries from Figure 6 no longer exist; the mid tier domain logic is refactored into atomic services (i.e., providers with contracts). The ESB and service composition patterns are applied. In this architecture, SOA layers from Definitions 2.8 are used: integration, presentation, service composition, and atomic services layer.

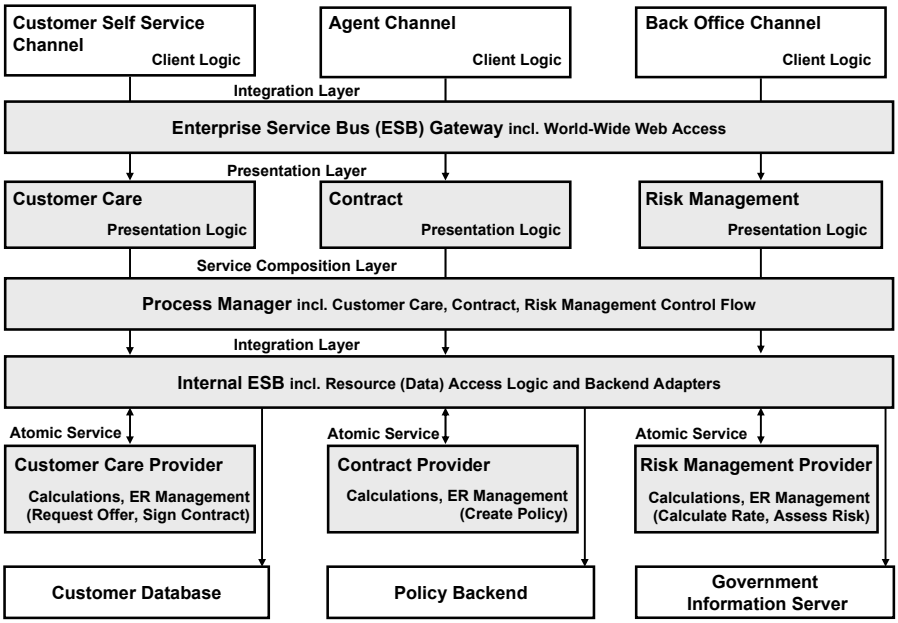


Figure 8. Customer enquiry architecture 2 (SOA)

The atomic services comprise the calculations and enterprise resource management part of the domain logic layer from the traditional tree-tiered architecture. There is one service provider per refactored application; each service operation implements the calculations and enterprise resource management required by one

business activity from the analysis-phase BPM.[15] Each provider exposes a service contract, which is used by service consumers in higher layers to prepare request messages and process response messages before and after service invocation.

The internal ESB (see Definition 2.7) is responsible for providing resource (data) access logic and adapters (which was formerly implemented in the three applications and traditional EAI) and for request routing. It is aware of the service contracts and can monitor service invocations in response to BR 4.

The process manager consolidates the control flow part of the domain logic formerly spread over the domain layers of the three applications. The seven business activities from the BPM (Figure 5 on page 23) appear in the executable process model (this is not shown in Figure 8). The process manager interfaces with the presentation logic which serves the customer, agent, and back office channels. It is responsible for preserving correct processing states, which helps to enforce the BR 1, 2, 4, and 5. Modern process managers (e.g., BPEL engines) provide rather sophisticated timeout and compensation management capabilities, which can be leveraged to satisfy business rules such as BR 2 (respond within three days).

The ESB gateway provides the users of the customer enquiry SOA access to the presentation layer, which invokes services in the lower layers, starting with the business activities hosted by the process manager.

Like in the previous architecture, design time reuse is possible. Business and resource (data) access logic can also be shared at runtime: Being exposed on the internal ESB, the contract service can be used by the contract consumer, but also by the customer care consumer. This makes the architecture more flexible, but causes extra runtime dependencies between consumers and providers.

Note that we do not claim either one of the two architectures to be superior to the other; they merely serve as examples of EAD and EAI challenges, of the usage of the SOA patterns, and of related design issues. We present such issues next.

2.2.5 Design Issues in the Case

When receiving functional specifications like the one in Figure 5 and producing architecture design models such as Figure 6, Figure 7, and Figure 8, software architects have to *select an appropriate architectural style*. As already discussed, SOA is a state-of-the-art option; a more conservative alternative is to develop three separate three-tier applications. Many follow-up design issues arise before the design can be implemented with Web services and/or other technologies.

Strategic design issues. Assuming that SOA is the preferred option, *a particular SOA reference model must be selected*, which includes agreeing on terminology and identifying relevant pattern languages, and setting technology and product procurement directions. The business strategy[16] and technical principles, e.g., to

[15] This is a simplification which is acceptable at this design refinement stage.

[16] E.g., mergers and acquisitions planned, or divestitures and outsourcing?

prefer or ban open source assets and to prefer certain software vendors and server infrastructures, must be considered. NFR 2 is an example of a related requirement.

Conceptual design issues. Next, *conceptual patterns must be selected and adopted*, decomposing the ones that define SOA as an architectural style. All components appearing in Figure 4 on page 16 have to be refined, e.g., the router in the ESB pattern. Functional requirements, business rules, NFRs, and legacy constraints such as those from Section 2.2.3 influence the design work.

In the case, we identified customer care, contract, and risk management services. It is required to *design service providers* for these services. The granularity of the service contracts in terms of number of service operations and structure of request and response messages (see Definition 2.6) must be decided. Once such service contracts are in place, it becomes possible to *design service consumers*.

The *detailed design and configuration of the internal ESB and the ESB gateway* triggers another set of concerns: According to our ESB definition, message exchange patterns and formats, as well as mediation, routing, and adapter patterns have to be selected (or banned). In this pattern selection and adoption process, format transformations, security settings, service management (e.g., monitoring), and communications transactionality must be defined precisely.

The *service composition design* also must be refined if this SOA pattern is selected. As already outlined, the choice of a central process manager implementing workflow concepts as opposed to distributed state management in individual applications is an important architectural concern. Other key architecture design issues regarding service composition are where to draw the line between the service composition layer and the atomic service layer, how to interface with the presentation layer (in terms of request correlation and coordination), and how to integrate legacy workflows, e.g., those residing in software packages. System transaction boundaries and higher level error handling strategies such as business transactions and compensation handlers have to be defined as well.

In the case, we did not introduce a *service registry* (Definition 2.9) in the architecture. If we had done so, several design issues would deal with the adoption and implementation of this pattern, as well as the related operational aspects. Design time versus runtime registry lookup is an example of a related design issue.

Platform-related design issues. *Implementation technologies for the conceptual patterns must be selected and profiled*, for instance WS-* [WCL+05] or another integration technology. Once technologies have been chosen, *implementation platforms must be selected and configured*. Many of the SOA patterns are implemented in commercial or open source middleware assets. It must be decided whether middleware assets should be procured and how the chosen ones should be installed and configured. Performance, scalability, interoperability, and portability are key quality attribute types when selecting and configuring implementation platforms.

In summary, PremierQuotes Group has two architecture alternatives when supporting the customer enquiry process with enterprise applications: SOA (Figure 8) or three-tiered client-server applications integrated via traditional EAI middleware

(Figure 6). Making this decision is only the start of the architecture design; detailed design work follows. Numerous design issues are encountered, which qualify as architectural decisions as per our introduction of the term in Chapter 1. The design issues differ substantially depending on the architectural style and patterns chosen. Numerous, often conflicting forces influence the decision making: Quality attributes in categories such as reliability, usability, efficiency (performance, scalability), maintainability and portability, as well as the four user channel, process and resource integrity, integration, and semantics challenges drive the selection of architectural style, the adoption of conceptual patterns, and the design of their platform-specific refinements. Many dependencies exist between the design issues encountered on the project, but also from and to those on other projects.

Various methods and other assets help the architect cope with this complexity and to overcome these challenges.

2.3 State of the Art Regarding Methods for SOA Design

As the introduction to the problem domain and the motivating case study demonstrated, SOA design is a broad and interdisciplinary topic; a wide range of related work is eligible. In this section, we focus on *software engineering methods* and *design methods* and particularly relevant supporting assets. Many such assets exist, which vary in scope, maturity, and practical adoption widely. In the interest of space, we only introduce selected representatives in this section, e.g., those contributing important concepts and those popular in practice.

The section is structured according to categories of design methods: General purpose software engineering methods and design methods (Section 2.3.1), software architecture design methods that narrow the focus to the architecturally relevant design activities (Section 2.3.2), enterprise application genre-specific methods (Section 2.3.3), and SOA style-specific methods (Section 2.3.4). As a supplemental field that is orthogonal to all method categories, we also cover architectural knowledge management (Section 2.3.5).

We use these methods several times throughout the thesis: To define our focus area, we conduct a fit/gap analysis for them in Chapter 3. Furthermore, we reference elements from these methods when developing our solution in Chapters 4 to 8. We complete the related work coverage with a comparison between the existing methods and our solution in Chapter 10.

2.3.1 Software Engineering Methods and Design Methods

Software engineering covers the entire spectrum of the software lifecycle, from analysis to design, development, test, integration, and operations [Som95]. The literature provides a rich body of supporting concepts, including software engineering methods; design methods then focus on the design phase. Such methods are *reusable assets* [OMG05] that define:

- *Processes* and *notations* that advise when to produce which *artifacts*. Process and notation are mandatory method elements [OMG08].
- *Techniques* to create artifacts and sample or reference artifact *content*. Comprehensive and mature methods provide such method elements.

Figure 9 shows this method anatomy on its left side. It also emphasizes that as a reusable asset a method must be *adopted* for project usage. The project incarnation of the method is shown on right side:

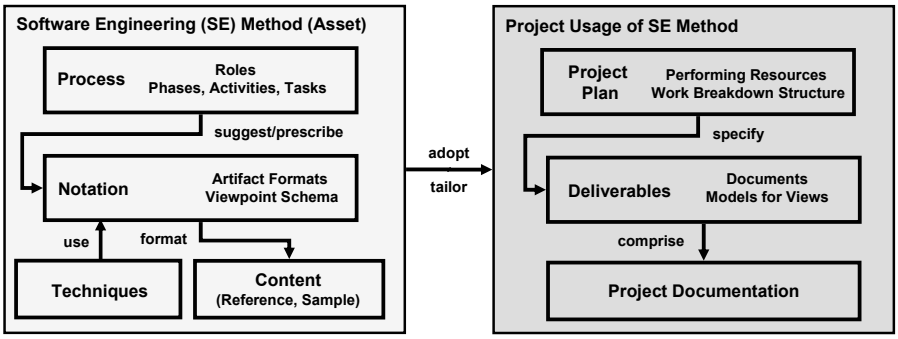


Figure 9. Method anatomy and project adoption

Methods define performing *roles* and a breakdown of their process into *phases*, *activities*, and *tasks* [OMG08]. Boehm's spiral model is an example of such process [Boe88]. Some processes are rather lightweight and *agile* [Bec00]. Others prescribe process and notation in great detail. An example of such comprehensive process is the *Rational Unified Process (RUP)* [Kru03].

Roles distinguish technical professions and areas of responsibility and specialization, e.g., lead and subsystem architects, developers, and technology platform specialists; processes define roles such as "application architect", "integration architect", "developer", and "tester". Comprehensive methods define the skills required to create artifacts and recommend a related education curriculum. The involved roles change during a project; e.g., analysis is performed by a business domain expert; design, development, test, and integration by software engineers specializing on certain subject matters.

Artifacts (a.k.a. work products [OMG08]) are the output of a task within a process. If the method supports an iterative and incremental process, the same artifacts are worked upon multiple times, going through several refinement steps [LL07]. The method prescribes or suggests a notation for the artifacts; using a common notation allows practitioners to exchange and reuse artifacts seamlessly. Examples are architecture documentation artifacts and code, but also test cases. Many requirements analysis and software design notations can be used to describe the artifacts, e.g., the Unified Modeling Language (UML) [RJB99]. In RUP, the role "software architect" creates the artifact "software architecture document", which is structured according to Kruchten's 4+1 viewpoints (see Section 2.1.3); UML can be used to model this architectural artifact [YRS+99]. The diagrams in

Section 2.2 are examples of analysis and design artifacts, which may be deliverables of the PremierQuotes SOA project.

Techniques and sample and/or reference content educate the method user how to create certain artifacts in a method-conformant way. They make the application of the method reproducible and repeatable.

OOAD. *Object-Oriented Analysis and Design (OOAD)* [Boo94] is an example of a mature design method. OOAD leverages UML or another object-oriented modeling language; RUP is one of many OOAD processes (although it has outgrown its OOAD heritage in recent years). Use cases capture the functional behavior of a system observable at the system context boundary. UML class diagrams specify the attributes and the behavior of the entities that are relevant in a particular domain, as well as various types of associations between them. Sequence and collaboration diagrams describe the interaction dynamics. Many OOAD techniques support class design, for example Class, Responsibility, Collaborators (CRC) cards [BC89]. The "OOPSLA school of software development" [Fow07] is particularly popular nowadays; it combines an agile OOAD process with Test-Driven Development (TDD) [Bec02] and Domain-Driven Design (DDD) [Eva03] techniques. Reusable design advice (content) is conveyed in pattern form.

Patterns. Architecture and design *patterns* capture proven solutions to commonly occurring problems. In 1995, Gamma et al. published the seminal Design Patterns book [GHJ+95]. Many different types of patterns have been documented since then, for example Patterns of Software Architecture (POSA) [BMR+96], Java design patterns [ACM03], domain analysis patterns [Fow97], and even patterns for non-IT topics. A pattern is a *proven solution* to a *problem* in a *context*. The context refers to a recurring set of situations in which the pattern applies. The problem refers to a set of goals and constraints that typically occur in this context and influence the pattern's solution, called the *forces* of the pattern [ZZG+08]. To systematically explain how to apply a number of patterns in combination, many pattern authors document patterns within larger *pattern languages*, containing rich pattern relationships and extensive examples and known uses sections.

Software engineering assets cover the entire software lifecycle. We now narrow our focus to architecturally relevant design activities and artifacts.

2.3.2 Software Architecture Design Methods

Software architecture [BCK03] is a sub-discipline of software engineering. A large body of software architecture literature focuses on software quality attributes [ISO01] as architecturally significant requirements [HKN+07]. Defining different *stakeholders* and viewing architectures from multiple *viewpoints* [Kru95, CCS07] are important complexity management strategies in software architecture.

Due to the problem solved in this thesis we focus on *architecture design methods* here. Five prominent examples of such methods are introduced and compared in [HKN+07], which we briefly review now. They emphasize an iterative architec-

ture design process, use viewpoints to organize the architectural artifacts, and provide certain techniques for important architecture design activities.

Five industrial methods. *Attribute-driven design (ADD)* [BCK03] uses software quality attributes as its base. ADD follows a recursive decomposition process, during which *architectural tactics* and patterns are chosen. Five *decomposition steps* define the ADD process: choose the architectural drivers, choose an architectural pattern that satisfies the drivers, instantiate modules and allocate functionality from use cases, define interfaces of the child modules, and verify and refine the use cases and quality attributes, making them constraints for the child attributes [HKN+07]. The artifacts to be produced are defined in [BCK03] as well. ADD can be used with traditional and with agile software engineering methods.

Siemens Four Views (S4V) [HNS00] defines four views, *conceptual, execution, module and code architecture* plus a *global analysis* activity in which organizational, technological, and product factors are identified. From these views and activities, the key architectural issues or challenges are identified; the method is aware that there are many related factors, which may conflict. From a process perspective, S4V emphasizes the need for iterations. The concept of an *issue card* and many illustrative examples of issues are introduced in [HNS00]. In the method anatomy shown in Figure 9 on page 32, issue cards can be positioned as artifacts. The design issues in the motivating case study may be represented by such cards.

In Section 2.3.1, we positioned *RUP* as a general purpose software engineering process. Combined with UML, RUP covers the full method anatomy (i.e., process, notation, techniques, and content). It specifically supports architecture design tasks; the notation of *elaboration points* emphasizes the importance of an iterative and incremental architecture design process. The *discipline* concept organizes the method by concerns. RUP emphasizes risk mitigation and defines an *issue list* artifact. The design issues in the motivating case study may appear in such list.

The *Business Architecture Process and Organization (BAPO)* method was developed by Philips Research. BAPO defines five views to organize the artifacts: *customer, application, functional, conceptual*, and *realization*. Quality attributes are used across these views to assess whether the architecture, which is developed iteratively, already provides enough information to start the implementation and to assess whether the design is free of discrepancies.

Nokia developed *Architectural Separation of Concerns (ASC)*, which is also known as the *ARES System of Concepts*. ASC has the notion of *design inputs* and *Architecturally Significant Requirements (ASRs)*. ASRs can be grouped. ASC also introduces the notion of architecturally relevant design decisions and links them to the ASRs. Separation of concerns is applied during the project phases (design, build, upgrade, load, and runtime). ASC supports the concept of an issue *backlog*.

Architectural patterns and other architectural knowledge. Complementary to architecture design methods, architectural patterns provide reusable designs (method content). Examples are general architectural patterns [BMR+96], patterns related to message-based integration [HW04], and remoting patterns [VKZ04]. As part of his Software Architecture Handbook project, Booch compiled and categorized more than 2000 patterns commonly used in various application genres [Boo].

We now narrow our focus to the enterprise application genre.

2.3.3 Enterprise Application Development and Integration Methods

All general purpose software engineering and architecture design methods presented so far can be applied to EAD and EAI. Genre-specific method extensions and additional methods also exist. They have the same general characteristics as the more general methods, but tend to put more emphasis on Business Process Modeling (BPM) and model-driven development.

OOAD extensions. OOAD processes are commonly applied on EAD and EAI projects. For instance, UMF [CCS07] defines a method extension for "Custom Application Development (CAD)" which embraces OOAD (but also structured analysis [You89]). Version 4 defines more than 100 artifacts such as "system context diagram", "process definition", "use case model", "nonfunctional requirements", "component model", "operational model", and "architectural decisions". The method is tailored at project startup to identify an adequate subset of artifacts.

RUP provides a *business modeling discipline*, which uses UML activity diagrams as BPM notation both during analysis and during design. Domain-specific method extensions can be integrated [Joh05]. *Business-Driven Design (BDD)* [Mit05] starts from BPM artifacts and progresses to the design and implementation level, leveraging UML along the way.

Model-Driven Development (MDD). In MDD, *models* are formal abstractions of systems [Wah08]. Analysis, design, and other types of models are distinguished. From a method perspective, the models are project deliverables which correspond to artifacts. *Model Driven Architecture (MDA)* defined by the Object Management Group introduces the Meta Object Facility (MOF) as a metametamodel and distinguishes between *platform-independent* and *platform-specific* models to make designs portable and facilitate incremental design refinement. *Model transformations* between different types of models are defined [OMG03].

EAD/EAI patterns and other genre-specific knowledge. Enterprise application architecture [Fow03] and integration [HW04] patterns provide method content. Workflow patterns exist [VT]; however, fine-grained language primitives (e.g., split, merge) do not qualify as common solutions to recurring design problems.

Enterprise architecture frameworks [SZ92] and *maturity models* [AH05, SEI] define their own methods and method content which can be used for *governance* purposes. Industry reference models such as SCOR [Sup] and IBM Industry Models [IBM] provide method content; many such reference models come with their own, domain-specific design processes and artifact creation techniques.

We now narrow focus a third time, this time to methods for design and implementation of enterprise applications employing a certain architectural style, SOA.

2.3.4 SOA Design and Service Modeling Methods

All methods presented so far can be applied to SOA design, particularly to design individual services [ZKG04, ZSW+05]. We focus on style-specific methods next.

CBDI-SAE. *CBDI Service Architecture & Engineering (CBDI-SAE)* is defined through a series of reports [CBD+06] and, more recently, through a knowledge repository available online. It provides a reference model for SOA defining four concepts, *glossary*, *SOA principles*, *service lifecycle*, and *SOA metamodel*. Twelve principles are defined (both abstract and technical), resembling those from Definition 2.5. The metamodel provides eight views on SOA (from business modeling to specification and implementation to runtime). A service-oriented process framework is defined, comprising ten steps covering the entire service lifecycle. CBDI-SAE also gives some guidance how to define high quality services.

SDLC. *Web Services Development Lifecycle (SDLC)* [Pap08] is a multi-step process for service-oriented design and development. It defines the *service lifecycle* to spawn from planning to analysis and design to construction and test to provisioning to deployment to execution to monitoring. Within SDLC, top-down, bottom-up, and meet-in-the-middle approaches to development can be taken. Six *milestones* establish high-level architectural principles and design goals (as method content): Reusing existing functionality, minimizing costs of disruption, employing an incremental mode of integration, providing increased flexibility, providing scalability, and complying with standards. *Qualities* such as low coupling and high cohesion (in multiple dimensions) and the distinction between coarse and fine granularity (i.e., scope of functionality) provide further method content. Concrete design advice is given how to deal with legacy systems and how to realize service monitoring. Web services concepts are presented in detail (corresponding to the development perspective in our Definition 2.5). To recommend techniques for each of its steps, SDLC embraces a rich set of existing work in software engineering [Kru03], business process modeling [Sup], and service modeling [Joh05].

SOMA. *Service-Oriented Modeling and Architecture (SOMA)* is the service modeling method from IBM [AGA+08, Ars04]. *Service identification*, *specification*, and *realization* are three of the process activities in SOMA. SOMA covers top-down service identification in business process models and other business analysis artifacts, suggesting techniques for goal service modeling, domain decomposition, and process decomposition (adhering to the SOA principle of business alignment). To specify services, SOMA defines a *service model* artifact, which works with the elements of the service contract introduced in Definition 2.6 and the organizational information to be stored in a service registry according to Definition 2.9. A *service litmus test* assists with the decision whether an identified service should be realized. Making architectural decisions is seen as a key activity during service realization. A conceptual model for SOA exists.

Other methods. Several other proposals exist, which resemble the presented ones. Many of them only cover a subset of the method anatomy, i.e., they focus on the

process aspect and put less emphasis on notation, techniques, and content. The importance of the business alignment principle is often stressed.

Erl proposes a full service design method called Mainstream SOA Method (MSOAM). He defines a process and related guidance in [Erl05]. SOA principles are defined in [Erl08], SOA patterns in [Erl09]. The concepts in the process are abstractions of those found in SOA technology specifications. For example, BPEL knows a concept *partner link*; hence, Erl's method defines a step "formalize partner service conversations". A main use case of the method is education.

Erradi et al. define their SOA Framework (SOAF) in overview form [EAK06]. The main focus of SOAF is to define five process steps along with inputs, activities, and deliverables; some high-level advice regarding service identification and aspects such as granularity is given. Shishkov et al. describe a process called "software derived from business components" in [SVQ06] and [SVT07]. Principles such as layering and loose coupling are promoted; 13 process steps are defined. Norm analysis gives guidance regarding interaction design. Chang et al. present a service-oriented analysis and design approach to developing adaptable services and a comprehensive approach to service adaptation [CLK07, CK07]. There is an SOA metamodel; five process phases are defined. The main focus is on a single quality attribute, adaptability. Types of variability are defined.

SOA patterns. SOA patterns have emerged over recent years, e.g., *ESB patterns* [KBH+04] to connect distributed system components (see Definition 2.7), and *process-driven SOA* [ZD06] patterns to realize long-running sequences of business activities as service compositions and workflows (Definition 2.8).

All methods presented to far promote a process- and artifact-centric approach. Architectural knowledge appears in artifacts serving as input or output of activities, or in the form of techniques and method content. Another stream of existing work focuses on the explicit management of such knowledge. We cover this field next as it is relevant for our decision-centric method creation paradigm.

2.3.5 Architectural Knowledge Management

Having been neglected both in academia and industry for a long time, the importance of capturing architectural decisions as defined in Chapter 1 (page 2) is now widely acknowledged: The *architectural knowledge management* [KLV06] field has its roots in *Design Decision Rationale (DDR)* [LL91, MYB+91]. The main focus of the field is architecture documentation, which may happen during or after the design work. Decisions made are stored in a repository; after-the-fact reuse is possible. The field distinguishes a tacit personalization strategy for architectural knowledge management from one based on explicit codification [Jan08].

IBM Unified Method Framework (UMF) [CCS07], which has been in use on IBM client engagements since 1998 (under a different name), defines an artifact "architectural decisions". The artifact description defines a text table template for decision capturing. It advises architects to capture all their decisions and the rationale behind them in a decision log. The rationale includes motivation, alterna-

tives considered, final decision with justification, assumptions, implications, and related decisions.

One of the IBM reference architectures comes with a filled out architectural decisions artifact containing decisions made during Web application design. Having worked with this artifact, Tyree and Akerman defined another rich decision capturing template, structured into 13 sections: issue, decision, status, group, assumptions, constraints, positions, argument, implications, related decisions, related artifacts, related principles, and notes [TA05].

In [KLV06], Kruchten, Lago, and van Vliet present an ontology that describes the attributes that should be captured for a decision, the types of decisions to be made, how decisions are made (i.e., their lifecycle), and decision dependencies. The ontology defines certain types such as executive, existence, and property decisions, dependencies such as constrains, forbids, enables, subsumes, conflictsWith, overrides, comprises, isAnAlternativeTo, isBoundTo, and isRelatedTo, as well as a decision lifecycle implementing a basic status management.

Jansen and Bosch view software architecture as a composition of a set of design decisions. Their model for architectural design decisions focuses on the time dimension, defining a dedicated entity representing architectural modifications occurring over the software lifecycle. Hence, their model is not only useful for architecture documentation, but also during design and operations (maintenance).

Architecture Rationale and Element Linkage (AREL) uses UML to capture design rationale [Ta07]. A UML profile is defined for that purpose.

Several other decision capturing templates and metamodels exist in industry and academia [Bre], which resemble the presented ones.

Before we analyze the presented existing assets, we complete the introduction to the problem domain with an overview of tools used on SOA projects today.

2.4 SOA Design Tools Used in Practice

In this section we give a brief overview of tools supporting the method concepts and other assets introduced in this chapter. The information originates from personal experience and contacts with the target audience (see Chapter 9 for details). It has exemplary character and does not aim to be complete.

Conceptual and technology view. The following tools are commonly used by practicing architects during SOA design on industry projects:

- *Method browsers* expose process, notation, techniques, and content defined in a method to users, following the method anatomy and relationships between the method elements introduced in Section 2.3.
- *Modeling* and *development environments* provide graphical editors supporting the creation of analysis, design, and development artifacts such as those shown in Section 2.1 and Section 2.2. They may also support code generation.

- *Office suites and traceability management tools* are used to process structured and unstructured text, e.g., issue lists and decision logs.
- *Reusable asset repositories* are used to exchange knowledge, including, but not limited to industry models, patterns, and code libraries [OMG05].
- Architects also work with *project management software* when performing tasks such as work breakdown structure creation.

Figure 10 illustrates this tooling landscape and how architects use the tools to create or review artifacts such as analysis-phase BPMs, process models of conceptual (i.e., design time) workflows, and service contracts as well as their BPEL and WSDL refinements. These artifacts implement the concepts from Definitions 2.6 to 2.9. Other users of the tools exist but are not shown in Figure 10, e.g., requirements engineers (a.k.a. business analysts and domain experts) and developers.

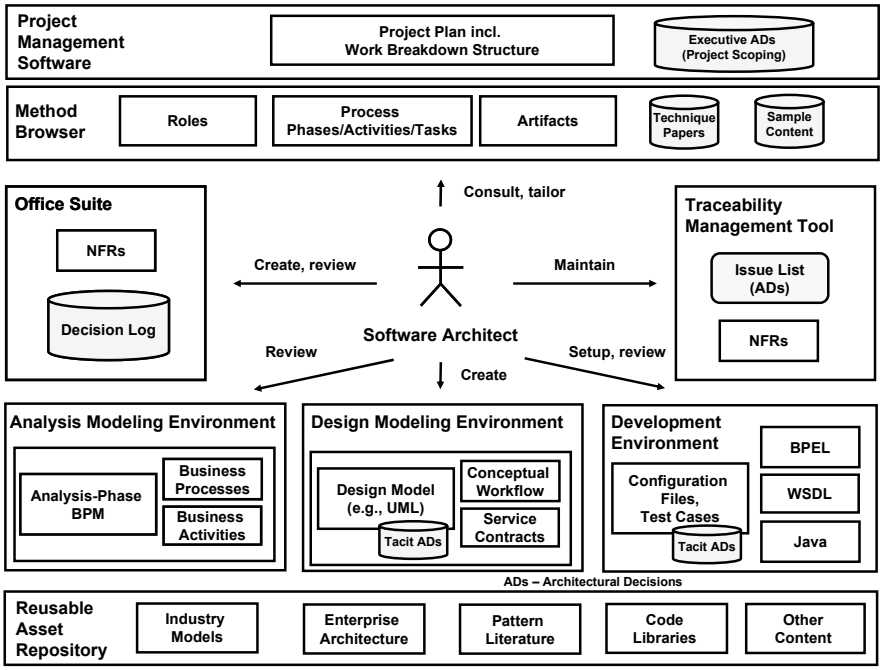


Figure 10. State of the practice regarding SOA design tools

Architectural decisions are not only captured in a central decision log, but also materialize in other tools and artifacts (this is shown as *tacit ADs* in Figure 10).

Exemplary mapping to vendor assets. Numerous commercial products and open source assets provide implementations of all tools introduced conceptually in Figure 10. In the interest of space, we only give a few examples here.

Microsoft Office Project 2007 [MS07] is an example of a project management software which can be used to create and monitor a *work breakdown structure* and capture project scoping decisions. IBM Rational Method Composer (RMC) [IBM] is a method authoring and browsing product for method engineers and practitio-

ners following a method defined in RMC. The method content can be exported from RMC so that only method engineers have to work with the product.

IBM Rational RequisitePro [IBM] can be used as a traceability management tool. As indicated by the product name, the metamodel of the tool supports requirements management by default. It can also be configured to capture architectural decisions, e.g., formatted according to the suggestion in the description of the architectural decisions artifact in UMF (which we introduced in Section 2.3.5).

IBM Rational Software Modeler [IBM] is one of many UML tools. It can be used for all OOAD activities, e.g., serve as an analysis and as a design modeling environment. It extends Eclipse which is a popular platform for Integrated Development Environments (IDEs) supporting Java and other languages [Ecl]. Many commercial and open source BPEL editors are also based on Eclipse.

IBM Rational Asset Manager [IBM] is one example of an asset repository. Custom developed repositories such as wikis are a popular choice in practice.

2.5 Summary of the Problem Domain Characteristics

Enterprise applications are distributed, software-intensive systems facing particular user channel, process and resource integrity, integration, and semantics challenges. SOA is a state-of-the-art architectural style for developing and integrating such enterprise applications. Many NFRs including software quality attributes must be met when developing service consumers and providers and integrating them with the help of the ESB, service composition, and service registry patterns. Numerous architectural decisions must be made during the SOA design activities.

Many software engineering as well as phase-, genre-, and style-specific design methods exist. These methods take a process- and artifact-centric approach; architectural decisions are captured retrospectively. The methods do not anticipate the architectural decisions required when applying the patterns defining the SOA style; there is no notion of recurring SOA decisions taking an active, guiding role.

SOA design is supported by many methods and tools. These assets treat architectural decisions as documentation artifacts, not as genuine method elements.

We provide more evidence for this statement when assessing state of the art and the practice in SOA design methods in detail in the next chapter, Chapter 3.

3 SOA Design Method Requirements and Research Problems

In this chapter, we first establish requirements for methods supporting SOA design (Section 3.1). From these requirements, we distill the ones that are particularly relevant for a decision-centric SOA design method and formulate the research problems that must be solved to satisfy them (Section 3.2). Finally, we use the requirements and research problems to analyze the methods introduced in Chapter 2 (Section 3.3) and derive an overall problem statement for this thesis (Section 3.4).

3.1 Requirements for SOA Design Methods

In this section we establish requirements for methods supporting the design and integration of enterprise applications employing SOA as their architectural style. We classify the requirements according to the categories we used to introduce existing methods and supporting assets in Chapter 2:

1. The *software engineering method* category covers the entire software engineering process independent of any application genre.
2. The *software architecture design method* category focuses on the subset of the design phase of the software engineering process that deals with architecturally relevant design issues and artifacts.
3. Requirements specific to methods targeting the *enterprise application genre* form the next category.
4. *SOA-specific requirements* follow.
5. Finally, a cross-cutting *architectural knowledge management* category comprises requirements for capturing and sharing architectural decisions.

The requirements catalog originates from three sources of input: academic and industrial *literature* [HKN+07], *interviews* with members of our target audience, practicing software architects (see Chapter 9), and personal industry *project experience* [ZMC+04, ZDG+05]. An earlier version of the requirements catalog was presented in [ZZG+08]. Obviously, such a catalog will never be complete. We will use the presented version to establish the focus area for this thesis and assess existing methods (Sections 3.2 and 3.3), to justify the design and implementation of our decision-centric SOA design method (Chapters 4 to 8), and to assess whether our method meets its design goals (Chapter 9).

3.1.1 General Requirements for Software Engineering Methods

The state of the art and the practice we introduced in Chapter 2 suggests requirements shared by all methods, general purpose ones as well as those specializing on certain project phases, application genres, and architectural styles. Table 2 lists these requirements and recapitulates their rationale:

Table 2. General requirements for software engineering methods

Requirement	Justification
R1-1: Method anatomy = process + notation + supporting techniques and content	Clarifies method scope [Boe88, Boo94, OMG08]; techniques and content ensure repeatability [Kru03]
R1-2: Provide standard description format, metamodel, or formal underpinning	Allows developing tool support, e.g., supporting collaboration and knowledge exchange [OMG08]
R1-3: Be broadly applicable and actionable, e.g., provide templates and examples	Method elements must fit in the software lifecycle and be detailed and concrete [ZZG+08]
R1-4: Provide link between requirements engineering (analysis) and design work	Makes requirements traceable in design, helps to verify soundness of design [Kru03]
R1-5: Provide link to project management methods	Supports secondary responsibility of software engineers and architects on industry projects [ZZG+08]
R1-6: Ease method content authoring (extensibility)	Projects are different; ease of authoring makes method adaptable and broadly applicable [ZZG+08]
R1-7: Be consumable and comprehensible, provide tailoring means (usability)	Content must be easy to understand and customize; user must be able to remove irrelevant parts and add missing elements rapidly [ZZG+08]

Methods must provide both a *process*, defining which role has to produce which artifact at which point in time in the project, and a *notation*, specifying the layout of the artifacts to be produced when following the process. To ensure repeatability, there should be *techniques* as well as reference or sample *content* teaching method users how to produce method-conformant artifacts (R1-1).

To ensure that artifacts produced by different method authors and project users can be exchanged seamlessly, it is necessary that the artifacts are documented according to a *standard description format or metamodel* (R1-2). A formal definition of the metamodel makes tool development feasible. To be useful on many different projects, the method content must be *broadly applicable and actionable*, e.g., reside on a sufficient level of detail and support one or more phases; to accelerate the creation of artifacts, documentation templates should be available (R1-3).

Methods targeting architects should be integrated with those used by other roles on development and integration projects, for instance *requirements engineering* (R1-4) and *project management* (R1-5). Such integration is required in practice. Finally, *extensibility* (R1-6) and *usability* (R1-7) of the method documentation are important. Extension points make a method adaptable and broadly applicable. It must be simple to contribute method elements of various sizes; a quality assurance and maintenance process should be defined. Providing rapid orientation in problem and solution space is an important usability aspect as practitioners have a limited education budget. It must be possible to locate reusable assets and tailor them according to project needs rapidly; the benefit must not be overcompensated by the effort.

3.1.2 Software Architecture Design Method Requirements

General purpose methods target multiple practitioner roles and cover the entire software lifecycle. Design, and more specifically architecture design, focuses on one role (software architect) and one phase (design) in this lifecycle. Therefore, there is an opportunity – and a necessity – to become more specific and concrete (e.g., in response to R1-3 and R1-7). Table 3 enumerates the related requirements originating from the consensus view of the creators of five industrial methods presented in [HKN+07] and from our own experience:

Table 3. Software architecture design method requirements

Requirement	Justification
R2-1: Refine general purpose methods: Provide multiple architectural viewpoints	Required to deal with complex design issues and to support division of labor [HKN+07]
R2-2: Be driven by quality attributes and stakeholder goals	Quality attributes are key to success, but often conflicting, "architectural concerns, context"[HKN+07]
R2-3: Support decomposition of complex design issues (architectural analysis)	Top-down refinement of design problem [ZZG+08], "candidate architectural solutions" [HKN+07]
R2-4: Support composition of resolved design issues (architectural synthesis)	Bottom-up assembly into overall design, prototype or full scope ("validated architecture" in [HKN+07])
R2-5: Define relationships between design issues and leverage them in method design	To maintain consistency and remove irrelevant method elements during method tailoring [ZZG+08]
R2-6: Provide a managed to do list	"Backlog" concept in [HKN+07]
R2-7: Support architecture evaluation, feedback loops, and backtracking	To ensure architecture meets requirements, e.g., with prototyping, to improve design iteratively [HKN+07]

Architecture design methods must complement general purpose methods in a phase-specific way, e.g., supporting *multiple architectural viewpoints*. IEEE standard 1471 [IEEE07] and the 4+1 model from [Kru95] are examples of such viewpoint schemes (R2-1). The design method must be driven by architecturally significant requirements, particularly *stakeholder goals* and NFRs including software *quality attributes* (R2-2). To overcome analysis-paralysis effects, the design method has to support the top-down *decomposition* of a complex problem to be solved by a software system into smaller, more manageable design issues (R2-3).[17] To create end-to-end architectures and support cross-project reuse, it must be possible to *compose* solutions to resolved design issues into complete architecture designs (R2-4). Furthermore, it must be possible to express dependency *relationships between the design issues*; such dependencies have to be taken into account during the architecture planning, design, and evaluation activities (R2-5). To facilitate team work and keep track of open issues, there should be a managed to do list. Such dynamic backlog (a.k.a. issue list) can serve as a checklist and a reminder function for the architect (R2-6). Finally, the method should advise how to evaluate architectures. It must be possible to backtrack and to provide feedback across roles whether designs are sound, e.g., technically feasible (R2-7).

[17] The nature of design issues depends on the method creation paradigm: General purpose methods typically suggest a process- and artifact-centric approach. The issue can also be a single quality attribute, a pattern, a design model element, or an architectural decision.

3.1.3 Requirements Specific to the Enterprise Application Genre

In Chapter 2, we introduced enterprise applications as a particular application genre. Enterprise application architectures have several genre-specific characteristics (as discussed in Section 2.1.2). Taking these genre-specific characteristics into account, Table 4 establishes five requirements specific to EAD and EAI:

Table 4. EAD- and EAI-specific architecture design method requirements

Requirement	Justification
R3-1: Refine architecture design methods for EAD and EAI: Support pattern-based architecture design	Mature general purpose patterns and EAD/EAI-specific ones exist [Fow03, HW04]; high-level application architecture and candidate styles known upfront [ZZG+08]
R3-2: Align with analysis methods (e.g., BPM, OOA), enterprise architecture frameworks, and maturity models	Common forms of requirements engineering (analysis) in EAD [Mit05, Boo94]; enterprise architecture assets used for IT governance and maturity management [AH05]
R3-3: Cover integration of legacy systems and software packages	Often hosting enterprise resources and/or containing other valuable data and logic (asset character) [Fow03]
R3-4: Support model-driven development concepts, use industry models	Separate platform-independent from platform-specific concerns, support portability and reuse [ZZG+08]
R3-5: Align with contemporary commercial EAD and EAI project delivery and procurement practices	Executive decisions such as in-house development vs. procurement of packages, offshoring and outsourcing influence the technical design work [HKN+07, ZMC+04]

Architectural *patterns* are a state-of-the art form of capturing established knowledge. Pattern knowledge originates from successful software architectures in which a pattern author has observed a common solution (the pattern). EAD and EAI patterns should therefore be integrated into the method (R3-1).

Well-established forms of *requirements engineering* such as Business Process Modeling (BPM) and OOAD should be supported. Similarly, *enterprise architecture frameworks*, *governance methods*, and *maturity models* should be integrated as the principles established by such assets have a significant impact on the architectural decision making (R3-2). It must also be possible to find advice how to deal with existing *legacy systems* and *software packages* (R3-3), for instance, when to leave as-is, when to adapt and integrate, and when to redesign a legacy system. The EAD and EAI challenges from Section 2.1.2 must be addressed.

Many EAD and EAI design issues require abstract conceptual thinking as well as detailed technology expertise or vendor-specific know how. A design method should therefore separate technology *platform-independent* and *platform-specific* concerns. This also makes a design portable and reusable. Modeling is one way of addressing the requirement. Support for other Model-Driven Development (MDD) concepts such as model transformations facilitates automation (R3-4).

Finally, it is important that a method is *applicable in today's business environments* (R3-5). For instance, requirements engineering and project management might be performed in-house, but a professional services firm might be contracted for design and development, possibly involving offshoring and outsourcing. The process defined in a method must respect that; a balance between standardization and flexibility must be found so that organizational changes do not cause major

method adjustment efforts. It must be possible to use the method in collaborative environments; the responsibilities of the involved roles (parties) must be defined.

3.1.4 SOA-Specific Design Method Requirements

General purpose and software architecture design methods deliberately do not make any assumptions about a particular application genre. EAD- and EAI-specific methods do so, but are independent of any architectural style. An SOA-specific design method, however, must meet additional requirements to become comprehensive and actionable. Table 5 derives such SOA-specific requirements:

Table 5. SOA-specific design method requirements

Requirement	Justification
R4-1: Refine previous three categories: Support *service* engineering process	E.g., SDLC phases [Pap08] and SOMA service identification, realization, and specification steps [AGA+08]
R4-2: Define notation for multiple service contract dimensions	Express functional and behavioral service contract: Syntax, QoS, semantics as per Definition 2.6 and [Joh05]
R4-3: Integrate SOA principles and patterns (Definitions 2.6 to 2.9)	Service consumer-provider contract, ESB, service composition, service registry patterns must be refined
R4-4: Give advice regarding granularity and other SOA-specific design issues	Many recurring design issues pertain to the patterns defining SOA as an architectural style (see Chapter 2)
R4-5: Cover service lifecycle management, e.g., ownership and versioning	Full lifecycle of a shared service transcends that of a single application or EAD/EAI project [Pap08, KBS05]

To support the service engineering process (R4-1), the entire software lifecycle introduced in Section 2.1.3 must be covered. An SOA design method must define how *services* can be constructed systematically, starting from analysis artifacts (e.g., business process models, use cases, or user stories) and covering the entire service engineering process defined in the literature [Pap08].

Regarding *service contract dimensions* (R4-2), functional and behavioral aspects such as syntax, Quality of Service (QoS) policies, and invocation semantics must be covered by the notation defined by a method (see Definition 2.6).

Taking advantage of our definition of SOA, the principles and patterns defined in Chapter 2 (Definitions 2.6 to 2.9) initiate the SOA design; general software quality attributes, EAD- and EAI-specific challenges steer the subsequent activities (R4-3). Methods must provide techniques and content in addition to process and notation to satisfy R1-1; in an SOA context, it is essential for SOA design methods to provide SOA-specific content, e.g., providing design advice regarding *granularity* and other recurring SOA design issues such as those discussed in Section 2.2.5 (R4-4). Designing for composability and reuse are related challenges.

Regarding *service lifecycle management* (R4-5), it is not sufficient to focus on early stages such as service identification and specification: A shared service deserves a more sophisticated approach to lifecycle management than an OOAD class or component if it is treated as a company asset (e.g., if it has product status). A related design issue is how to version shared services when supporting multiple service consumers. These service consumers usually evolve independently of each other; their change management plans may differ, e.g., in their release schedules.

3.1.5 Requirements for Architectural Knowledge Management

This last requirements category does not refine a previous one, but is orthogonal to these predecessors. It is required because our method creation paradigm centers on architectural decision knowledge. Some of the requirements in the previous categories already dealt with such decision knowledge implicitly (as method content).

In [FBC06], thirteen general use cases for decision capturing are identified, covering a wide range of activities such as conflict detection, validation, documentation, coordination, and communication. Due to our extended usage of architectural decisions, additional requirements for building up *architectural decision knowledge* apply, particularly if the decision making responsibilities are shared within and across teams. R1-7 (usability), R2-6 (managed issue list), R3-5 (commercial EAD and EAI project delivery practices) and R4-4 (SOA design advice) lead to a set of cross-cutting *collaboration requirements* (Table 6):

Table 6. Architectural decision knowledge capturing and sharing requirements

Requirement	Justification
R5-1: Obtain required knowledge	Obtain architectural knowledge from third parties, e.g., company-wide enterprise architecture group [SZ92] or community of practice [GR01]
R5-2: Adopt identified knowledge	Tailor obtained knowledge according to project-specific needs: delete, update, and add content, e.g., design issues and solution alternatives [SZP07]
R5-3: Delegate decisions	Delegate architecture design work to other architects and lead developers and support review activities with bidirectional feedback loops [FBC03]
R5-4: Involve community	Involve network of peers in search of additional expertise during architecture design work, e.g., platform specialists [FBC03]
R5-5: Document decisions	Enforce decision outcome via generation of artifacts, e.g., decision log and code snippets serving as architectural templates [FBC03, KLV06]
R5-6: Align with other models	Inject decision outcome into design models, development, and deployment artifacts such as source code, configuration files, and test cases (R3-4)
R5-7: Share gained knowledge	Share gained architectural knowledge with third parties such as the actors from R5-1, having cleansed the project deliverables [SZP07]

If architectural decisions are supposed to be used actively during design, broad and deep architectural decision knowledge from different sources is required. It must be possible to obtain such knowledge from completed projects. The overhead of reusing such knowledge must be low (R5-1). Furthermore, it must be possible to tailor the method content (R5-2). In support of R5-3, it is required to ensure the consistency of decisions in a global context (having delegated decisions to multiple team members, e.g., subsystem architects or platform specialists). Furthermore, it should be possible to compare decisions from different projects with each other and with industry best practices. It is often desirable to involve peers (R5-4); this requires a common understanding of problem and solution space and a common vocabulary. R5-5 and R5-6 deal with communicating decisions and propagating them to design model elements and code. The exchange of architectural knowledge across project and, possibly, company boundaries must be supported (R5-7): To cleanse the projects deliverables, project-specific and company-internal information has to be removed and text and figure elements renamed to avoid confidentiality problems and misunderstandings regarding terminology.

3.2 Research Problems and Questions

The requirements for SOA design methods that we compiled in the previous section indicate that the creation of a decision-centric architecture design method is an ambitious undertaking. A detailed investigation of all 31 requirements would exceed the scope of this thesis. Hence, we now distill those that are specific to our decision-centric method creation paradigm and have open research problems attached which must be solved to create a decision-centric SOA design method. These research problems constitute the focus area of this thesis.

Problem identification in method requirements. We identified seven research problems to be particularly relevant in the requirements context from Section 3.1. Table 7 introduces them and indicates from which requirements they originate:

Table 7. Research problems distilled from method requirements

Research Problem	Method Requirements	Rationale (from Section 3.1)
Decision identification	R1-4, R2-2, R3-2, R4-3, R5-1	To create and reuse knowledge, to ensure applicability and extensibility
Decision modeling	R1-2, R1-3, R1-6, R1-7, R2-3, R2-4, R3-4, R5-3, R5-5	To make knowledge exchangeable, to partially automate its processing
Model structuring	R1-2, R1-6, R1-7, R2-1, R3-2, R5-2, R5-4	To organize method content, to ensure logical consistency and usability
Dependency management	R2-3, R2-4, R2-5	To order decisions for design method usage, to prune irrelevant ones
Design method usage	R1-1, R2-6, R3-1, R4-1	To facilitate decision making
Decision enforcement	R3-4, R5-5, R5-6	To align decisions with other artifacts
Collaboration system	R3-5, R5-3 to R5-7	To support teamwork

The remaining requirements (i.e., those not listed in the table) continue to be relevant and serve as input to the design of our solution presented in Chapters 4 to 8. We will return to all requirements in Chapter 9 to assess our solution.

To specify the seven problems, we formulate the corresponding research questions now.

Decision identification. The first problem we distilled from the requirements deals with scoping the method and finding related architectural knowledge:

1. What are the architectural decisions required during SOA design (issues)? Do these issues recur?
2. If so, can the issues be identified systematically in patterns?
3. Can this systematic approach be transferred to other application genres and architectural styles?[18]

Decision modeling. Our second problem concerns the documentation of individual issues which are identified with the help of solutions to the first problem:

[18] Positive answers to the questions form the hypotheses to be verified or falsified during validation, e.g.: *Architectural decisions required (issues) recur; an identification technique can be defined*. We will return to these questions and hypotheses in Chapters 4 to 8 when presenting solutions to them and in Chapter 9 when presenting validation results.

4. Which information to model for each issue (and its alternatives)?
5. Which level of detail is appropriate so that the given advice is detailed enough to be actionable and generic enough to be broadly applicable and not subject to overly frequent, unmanageable changes?
6. Which aspects are not covered by existing templates and metamodels used to document architectures and to capture decisions made?

Model structuring. Due to the broad scope a decision-centric SOA design method must have, many decisions have to be modeled. The third problem investigates how to structure the resulting decision models:

7. Assuming that a large number of issues recurs, how can a decision model be organized in an intuitive, use case-driven way?
8. How to separate rarely changing conceptual knowledge from rapidly evolving technology information and platform-specific know how?
9. How to leverage existing problem solving concepts such as architectural layers and viewpoints in the decision models?

Dependency management. Architectural decisions rarely occur in isolation. Our fourth problem deals with the many relations between intertwined decisions:

10. Which logical and temporal dependencies exist between decisions? How can such dependencies be represented in decision models?
11. Can these dependencies be used to detect design errors, to organize the decision making process, and to prune irrelevant decisions?
12. If so, how to order the decisions to prepare for decision making?

Design method usage. The fifth problem investigates how to realize our primary use case for decision models, taking the current design context into account:

13. How to use an architectural decision model as an SOA design method?
14. Can a process be defined that considers only the decisions required by a particular role in a certain project phase and design context?
15. What is the relation to software engineering and design methods?

Decision enforcement. Our sixth problem concerns the connection between decision models and design models as well as other development artifacts:

16. How to enforce that made architectural decisions are respected during subsequent design activities and during development?
17. How to update design models and code according to outcome information in an architectural decision model?
18. What is the relation between decision models and Model-Driven Development (MDD)?

Collaboration system. The seventh and final problem deals with tool design:

19. Which logical building blocks comprise a tool that supports architects when they investigate, make, and enforce architectural decisions?
20. How to support collaborative creation and usage of decision models?
21. How to integrate such tool with other tools used during SOA design?

We now return to the existing methods and other assets introduced in Chapter 2 and investigate whether they fully or partially solve these problems.

3.3 Analysis of State-of-the-Art Design Methods

In this section, we assess the methods introduced in Chapter 2 with respect to the seven problems we identified in Section 3.2. The objective of this analysis is to assess whether the problems have been solved partially or fully already and to locate the concepts in existing work we can build upon. Table 8 gives an overview of the features in existing assets that are particularly relevant for this analysis:[19]

Table 8. Research problems and existing solutions (methods and other assets)

Research Problem	Software Engineering	Software Architecture	EAD/EAI Methods	SOA Methods
Decision identification	OOAD (classes), pattern literature	Architecturally significant requirements	BPM; EAD and EAI patterns	Service model [AGA+08]
Decision modeling	Pattern templates	AREL UML profile [Ta07], decision capturing metamodels	MDA MOF [OMG03], UMF artifact template	–
Model structuring	Pattern languages and catalog taxonomies [Boo]	Viewpoints [Kru95], ontology in [KLV06]	MDA model types, enterprise architecture	SOA reference models [Ars04]
Dependency management	Pattern relations	Ontology in [KLV06]	–	–
Design method usage	OOAD, patterns-based design [BHS07]	Issue cards in S4V, ASC backlog, RUP issue list [HKN+07]	BPM methods (analysis), RUP/UMF extensions	e.g., CBDI-SAE, SDLC, SOMA
Decision enforcement	Agile development, governance models	Architectural evaluation [HKN+07]	MDD, code generation	–
Collaboration system	Eclipse plugins, wikis, Jazz [Jaz]	Decision capturing tools [AGJ05, Jan08]	–	–

Decision identification. OOAD defines how candidate classes can be identified in use cases. However, this identification technique deals with domain models rather than architectural knowledge. The pattern literature presents conceptual solutions, but does not elaborate on the origin of the knowledge and the architectural decisions required when adopting the pattern. Advice how to realize the conceptual design in a technology or vendor asset is given in the form of examples, if at all.

The architecture design methods presented in [HKN+07] define a backlog (or similar concepts), but it is undefined where the entries of this backlog, the design issues, come from. The starting points for decision identification are architectur-

[19] To simplify the discussion without loosing generality, we assume that all general purpose and SOA design methods can be used in combination, e.g., ADD plus UMF plus SDLC. Within OOAD, this is the case; e.g., the UML notation is the result of the fusion work of the "three amigos". It yet has to be proven that such a fusion is possible in SOA design.

ally significant requirements. Architectural knowledge management focuses on decisions made already, not on sources of decisions yet to be made.

BPMs can be used to identify services; service models as created in SOMA also can be used to initiate the SOA design work. However, it remains unclear which architectural decisions have to be made during this work. We demonstrated this when listing the design issues in the motivating case study in Chapter 2. EAD and EAI patterns exist, but they have not been fully integrated into emerging SOA design methods yet. They have the same limitations as general purpose patterns.

In summary, no existing approach gives the software architect advice how to identify the SOA design issues, e.g., the architectural decisions required when realizing services or SOA infrastructures with ESBs and service composition engines. We can conclude that the decision identification problem is unsolved so far.

Decision modeling, model structuring, and dependency management. A similar assessment can be made for the *decision modeling* problem. In the patterns community, several pattern templates exist; pattern languages specify relations between patterns in a particular domain. Formalizations of these concepts have been proposed [Zdu07]. Many different interpretations of the term "pattern" exist; there is no consensus on a single template or metamodel. In all templates, the main focus is on the presented solution rather than the design problem solved [ZZG+08].

Architecture knowledge management advises to capture decision knowledge in structured or unstructured text; several metamodels and ontologies exist [Bre, DFL+07, JB05, TA05]. Harrison, Avgeriou, and Zdun [HAZ07] propose to minimize the capturing effort by referencing already published patterns such as "layers". The Architecture Rationale and Element Linkage (AREL) method [Ta07] uses UML to capture rationale. AREL sees UML as the only architectural notation; the decision rationale is embedded in and added to UML models with the help of profiles. This can lead to usability issues if the rationale texts are large. All these approaches focus on architecture documentation rather than design method support.

The Meta Object Facility (MOF) in Model Driven Architecture (MDA) [OMG03] provides a formal underpinning for modeling. However, not all existing implementations are faithful to the original vision of MDA; the practical adoption of the paradigm is limited at present (see discussion in Section 7.3).

The existing service modeling methods provide rich texts describing what good services are (in terms of design principles and design activities to be performed). However, they do not capture such SOA decision rationale in model form.

Partial solutions to the *model structuring* problem are available: Pattern catalogs [Boo] structure the solution domain, but not the problem domain. Viewpoint models from software and enterprise architecture [Kru95, OG07], decision ontologies [KLV06], and MDA model types [OMG03] provide suitable structuring mechanisms, but have not yet been applied to SOA issues steering design work.

Decision *dependency management* has been studied in a pattern education or architecture documentation context only [KLV06], not in architecture design.

We can conclude that while partial solutions to the decision modeling, model structuring, and dependency management problems exist, none of them is suffi-

cient to support a usage of architectural decisions as a design method. However, we can build on existing assets to create our decision-centric design method.

Design method usage. As explained in Chapter 2, general purpose software engineering as well as software architecture design methods, EAD and EAI methods, and service modeling methods can be applied during SOA design. These methods vary in their support for architectural decisions and phases and roles supported.

Software engineering methods, design methods, and patterns. OOAD meets requirements R1-1 to R1-7. For instance, the Classes, Responsibilities, and Collaborators (CRC) card technique is not limited to class design; service contracts can be conceptualized with it as well. A customization of OOAD processes, notations, and techniques to SOA design is possible with some restrictions [ZKG04].[20]

However, OOAD by default focuses on the development viewpoint rather than logical and physical architectures. It was designed before the Internet and XML became popular; there is no inherent support for SOA-specific principles and patterns such as ESB and the related Web service design issues (R4-3, R4-4). A design issue that is not addressed by the OOAD literature properly is finding the right service granularity (i.e., number of operations and their message parameter structure) when facing requirements such as shared service usage and distribution over possibly slow network connections that do not preserve a message sequence.

While patterns accurately describe the technological options to solve a given design problem, they are not designed to guide through the pattern selection and overall architecture design process. Relationships between the patterns are only defined within a single pattern language. However, many industry projects use multiple pattern languages. Systematic approaches to identify and evaluate candidate patterns from different languages are only beginning to emerge [ZAH+08].

Software architecture design methods. Software architecture design methods go a long way in supporting practitioners when designing enterprise applications. However, the methods introduced in Chapter 2 fail to provide SOA-specific guidance. To give an example, none of the existing approaches gives concrete advice which predefined patterns to select in the motivating case study and how to refine them into a design that can be implemented. While an issue backlog has been suggested in [HKN+07], it remains unclear how to populate, order, and process it.

EAD- and EAI-specific methods and method extensions. Just like the general purpose and architecture design methods, the EAD- and EAI-specific methods and method extensions introduced in Section 2.3.3 have a process- and artifact-centric anatomy. There is no emphasis on architectural decisions required during design.

Due to their breadth, depth, and popularity in architect and developer communities, we consider Fowler's patterns [Fow03] a de facto standard for enterprise application development. However, the patterns in the book mainly focus on Web

[20] It is important to take SOA-specific principles into account. For instance, remote object references should not be modeled as this would violate the principle of defining service providers with "always on" semantics. ESB messaging does not support the passing of remote object references ("programming without a call stack" [Hoh07], Definition 2.7).

applications. Hohpe and Woolf's enterprise integration patterns [HW04] describe messaging patterns accurately and in great detail; the book is widely accepted as the "lingua franca" of messaging. However, the authors take a middleware- rather than an application-centric viewpoint. Generally speaking, older enterprise application literature does not cover all elements of the SOA style. For instance, the two mentioned pattern languages do not present patterns for service composition with workflows. Top-down guidance from problem to solution is given informally and incompletely; giving such advice is not the main objective of patterns books.

SOA-specific methods and method extensions. Existing service modeling methods structure the SOA design process. For instance, CBDI-SAE, SDLC, and SOMA define the phases, activities, and/or tasks in their processes. However, they do not cover all phases of the service lifecycle on the same level of detail (R4-1).

The reference model, SOA principles, metamodel, and process framework in CBDI-SAE addresses all requirements for SOA design methods. Multiple viewpoints are taken. The method has been developed over several years; supporting techniques and content are available. However, quality attributes and architectural decisions are only touched upon. They are not a first class citizen in the method.

The advice about versioning and service lifecycle in SDLC is highly educational. There is coverage of software quality attributes and design tasks. However, all advice is given in text form. SOA concepts and Web services technologies are not separated. Most of the advice pertains to the design of individual services, not an entire SOA. For instance, logical layering is covered incompletely.

SOMA outlines several service identification techniques, e.g., goal service modeling and process decomposition. It has a service specification format (service model) and a litmus test governing service exposure; Web services usage is not mandated. SOMA sees architectural decisions as an important service realization concept. However, no detailed catalog of such service realization decisions exists.

MSOAM evolved from two text books on Web services technology [Erl04] and SOA concepts [Erl05]. It serves well to educate readers about these concepts and technologies, which is different from a requirements- and context-driven SOA design process. The SOA principles and patterns in [Erl08, Erl09] reside on an abstract, vendor-independent level; their descriptions are not detailed enough to support all of the architectural decision making activities required during SOA design. The high number of patterns in the catalog (85) calls for additional guidance.

Other articles on service-oriented analysis and design from authors in industry and academia fail to satisfy requirement R1-1 and R1-3: They are not broad, deep, and mature enough to be able to assist during SOA design on industry projects. For instance, Shiskov's discussion remains on a high level; while some advice is valuable, most method elements repeat advice already known from non-SOA literature [SVQ06, SVT07]. Chang's phases are specified informally without stating artifacts as input and output of the phases. Design techniques and information about the responsible roles are missing [CLK07, CK07].

In all methods presented in Section 2.3.4, SOA-specific design advice is given in text form in technique papers and method extensions. Design patterns can be integrated (R3-1, R4-3). However, the required architectural decisions are not stated.

EAD/EAI challenges such as those discussed in Section 2.1.2 and SOA design issues such as those from Section 2.2.5 are not covered in depth. Service interface design, communication protocol selection, and transactional management settings are examples for issues that are not addressed sufficiently (R4-4). Furthermore, technology and vendor recommendations (often called "best practices") are not integrated well; if present, they tend to oversimplify the picture.

We see another point of critique regarding existing SOA design methods: The state of the art in software engineering and architecture has not been taken into account sufficiently. Supporting multiple viewpoints (R2-1) and quality attribute-driven design (R2-2) are key aspects in this regard. Several commercial methods provide technique papers, but only few validated research results exist. None of the existing assets provides a strong connection with SOA principles and patterns or explicit support for tradeoff analysis and architecture evaluation (based on cost, quality attributes, and other criteria). It remains unclear when and how to make which architectural decision when applying the SOA principles and patterns.

We can conclude that the problem has not been solved yet; none of the existing methods follows a decision-centric paradigm for method creation and provides SOA-specific method content that satisfies all requirements from Section 3.1.

Decision enforcement. Regarding the *decision enforcement problem,* agile development has been designed to provide a close connection between architecture design and development (which are seen as one activity [Fow03]). On the other end of the spectrum, governance and maturity models such as The Open Group Architecture Framework (TOGAF) and Capability Maturity Model Integration (CMMI) can be leveraged to ensure consistency between designs and emerging implementations. In these approaches, decision enforcement remains a human activity.

Existing software architecture design methods [HKN+07] emphasize the importance of architectural evaluation, but do not see MDD as a solution. Conventional techniques such as prototyping and incremental design dominate.

While MDD transformation chains for BPM and SOA [BB06] support code generation, none of the existing implementations allows using architectural decisions as input to model transformations; it is unclear how to address NFRs when transforming and how to integrate architectural viewpoints and method roles.

SOA design and service modeling assets do not provide any style-specific extensions to the decision enforcement capabilities of the more general methods.

This problem is also unsolved: Today's enforcement techniques are informal. The existing MDD approaches do not integrate architectural decisions properly.

Collaboration system. Regarding the *collaboration system problem*, existing decision capturing and sharing tools [AGJ05, CNP+06, Jan08] have architecture documentation and knowledge exchange as their primary use cases, not active design method support. Existing design tools with collaboration support do not use architectural decisions as their central metaphor and do not implement solutions to the previous six problems; they have been created to support other roles.

In the proposed extended usage context for architectural decisions, design method support, the collaboration system (tool) problem is open as well.

3.4 Overall Problem Statement and Summary

As we have shown in Chapter 2, SOA principles and patterns make the high-level architectures of enterprise applications straightforward to design. However, when refining such high-level architectures, many architectural decisions must be made to satisfy numerous, often conflicting requirements. Hence, a method is required that assists software architects when making these decisions.

Full lifecycle methods define a process with roles, as well as input and output of the tasks in the process; they do not give advice how to design solutions (e.g., applicable NFRs and potential solutions to particular design issues). OOAD is a mature general design method which can be applied to SOA design, but does not exploit any SOA-specific principles and patterns. By default, OOAD focuses on the analyst and the developer rather than the architect, although its concepts can be applied to architecture design. Pattern languages mainly have educational character; their usage as design method has been proposed, but stands at an early stage.

Software architecture design methods focus on the design phase and the architect role; however, they do not follow a decision-centric approach although the notion of a backlog has been proposed. Backlog management remains a manual task. Furthermore, software architecture design methods do not provide any enterprise application genre- or SOA style-specific design guidance.

Several application genre-specific methods add BPM as an analysis technique, but fail to use architectural knowledge and model-driven development in a practical way. Their main focus is on the analyst and the developer, not the architect.

A variety of service modeling methods has been defined in recent years, some of which have already matured. However, the advice given in these methods is informal and not always driven by software quality attributes. These methods integrate existing work from software architecture research insufficiently; they focus on ensuring the business alignment of conceptual services (targeting the analyst) and on best practices for Web services implementations (targeting the developer).

In summary, existing methods do not cover all SOA design aspects sufficiently. The main reason for that is that they do not treat the architectural decisions required in SOA design as genuine method elements. To overcome this limitation, decision identification, decision modeling, model structuring, dependency management, decision making, and decision enforcement problems must be solved:

An SOA design method is required which:
(a) identifies architectural decisions in patterns and project-specific artifacts,
(b) guides architects through the decision making when they refine the patterns into platform-specific designs, and
(c) supports architects during the enforcement of the decisions made.

The definition of such a method is the objective of the subsequent chapters: Chapter 4 introduces the method and its supporting concepts. Chapter 5 covers part (a) in detail, targeting knowledge engineers. Chapter 6 defines a metamodel supporting and underlying parts (a), (b), and (c). Chapter 7 completes the coverage of part (a) and covers part (b) and (c) in detail, targeting software architects.

4 An Architectural Decision Modeling Framework for SOA Design

In this chapter we introduce a conceptual framework for architectural decision modeling. We call it *SOA Decision Modeling (SOAD)* framework.[21] SOAD concepts and tool support for them solve the research problems from Chapter 3. SOAD framework and a particular SOA decision model we developed with it comprise our decision-centric SOA design method. This method and related tool support satisfy the requirements from Chapter 3.

The chapter is organized in the following way: Section 4.1 introduces the two key framework concepts, decision reuse and decision modeling, and seven framework steps. Section 4.2 positions SOAD in the software engineering process and proposes a tool architecture supporting the framework. Section 4.3 applies SOAD to SOA design and the motivating case study from Chapter 2. Section 4.4 summarizes the chapter. Chapters 5 to 7 then cover SOAD steps and concepts in depth.

4.1 Key Concepts: Decision Reuse and Modeling

With SOAD, we extend the usage of architectural decisions from architecture documentation to architecture design method. As defined in Chapter 2, a method is a *reusable asset* [OMG05] that is created by a method engineer for use on multiple projects. Hence, SOAD must provide a concept for *decision reuse*. As a first step towards decision reuse, we define two *phases* of knowledge processing:

1. *Asset creation* is performed by a *knowledge engineer*, i.e., a software architect [BCK03] tasked with the creation of a reusable asset comprising architectural decision knowledge. The asset is created for (and with input from) a community [GR01], e.g., the architects in one enterprise.
2. During *asset consumption* one or more *software architects* use the architectural decision knowledge in the reusable asset on their projects.

We *model* the decisions in the reusable asset rather than capture them in text: We specify the structure of and the relations between decisions in a *metamodel*. This makes it possible to exchange the knowledge and to automate parts of the

[21] Despite its name, the framework is applicable to multiple application genres and architectural styles. It supports several use cases. We focus on its usage as a SOA design method in this thesis. Applicability to other genres and styles is discussed in Chapter 10.

knowledge processing, e.g., model instantiation, export and import of knowledge, consistency checking, and report generation.

In support of these two concepts, we introduce two forms of architectural decision models. They are treated differently during asset creation and consumption.

Definition 4.1 (Reusable Architectural Decision Model, RADM). *A* Reusable Architectural Decision Model (RADM) *is a reusable asset containing knowledge about architectural decisions required when applying an architectural style (comprising of architectural principles and patterns according to Definition 2.4) in a particular application genre. A RADM is shared by a community of architects.*

Such a RADM can capture architectural knowledge from already completed projects that employed the architectural style for which the RADM is created. SOA is such an architectural style (see Definitions 2.4 and 2.5 in Chapter 2).[22]

Project-specific architectural decision models are created from such RADMs:

Definition 4.2 (Architectural Decision Model, ADM). *The* Architectural Decision Model (ADM) *for a project contains knowledge about architectural decisions required, but also captures information about architectural decisions made.*

An ADM is *created* and *used* during the asset consumption phase, reusing one or more RADMs. Information about decisions made is added throughout the project; it can be fed back to the RADM after project closure (we call the feedback process *asset harvesting*). Figure 11 shows the ADM and RADM processing:

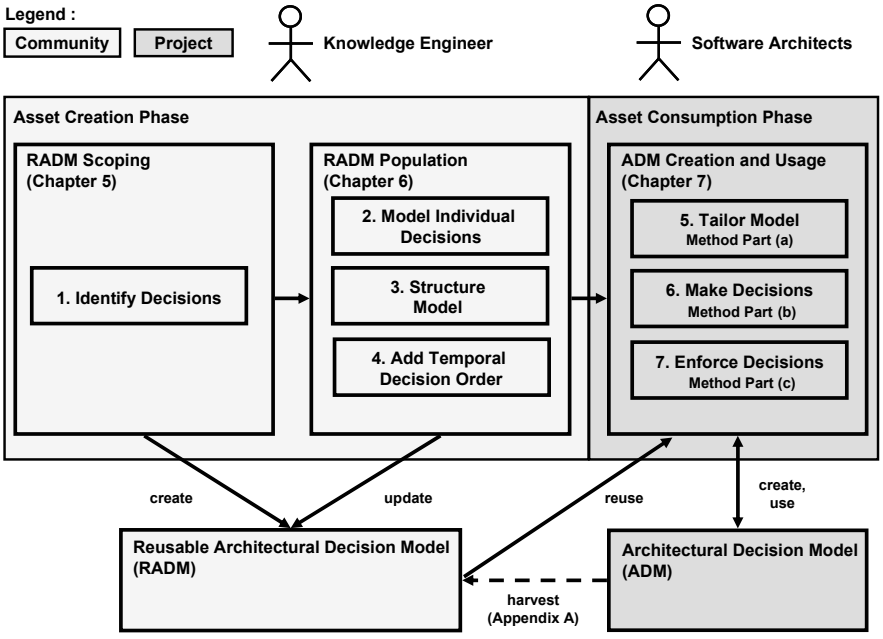


Figure 11. SOAD users and framework steps

[22] As part of the thesis validation activities we created one particular RADM, which we call *RADM for SOA*. Excerpts from this RADM for SOA serve as examples in this thesis.

Figure 11 also introduces the seven steps that are performed by the knowledge engineer during asset creation and software architects during asset consumption. The asset creation steps are organized into two sub-phases: the RADM is first *scoped* and then *populated*. Asset consumption does not have sub-phases. We now give an overview of the steps, which are specified in detail in Chapters 5 to 7.

RADM scoping (asset creation phase). To define the boundaries of a RADM during asset creation, a knowledge engineer performs the following step:

1. *Identify decisions* required when applying an architectural style in an application genre. This step starts with a review of the patterns defining the style. It returns a list of decisions to be included in the RADM.

This step can be performed top down, starting from the definition of the architectural style in use, or bottom up, studying architectural artifacts from previous projects. It is possible to combine top-down and bottom-up RADM scoping.

To make the framework flexible and to avoid an overloading of the individual steps, we separate the identification of decisions (RADM scoping) from their detailed documentation (RADM population):

RADM population (asset creation phase). A knowledge engineer performs the following steps to populate a RADM:

2. *Model individual decisions*. In this step, the decisions in the list delivered by step 1 are documented in such a way that the modeled knowledge can support the decision making on projects. The required level of detail depends on the software engineering method adopted by the project and the knowledge sharing practices in the community: A model targeting communities that employ an agile process and a knowledge personalization strategy (as introduced in Chapter 2) can be less detailed than a model targeting communities that apply traditional processes and a knowledge codification strategy (also introduced in Chapter 2).
3. *Structure model* according to logical dependencies between decisions. The model structure developed in this step has the objective to make the RADM easy to navigate and to adapt to project needs.
4. *Add temporal decision order* by modeling temporal decision dependencies. This order is leveraged during the decision making step 6.

It is worth noting that steps 1 to 4 may be executed repeatedly and in an overlapping fashion to scope and populate a RADM iteratively and incrementally.

Having completed the asset creation phase with its four steps, we can progress to asset consumption on a project:

ADM creation and usage (asset consumption phase). ADMs are created and used on projects that apply SOAD. Three steps are performed by the architects:

5. *Tailor model*, creating an ADM from one or more RADMs by taking project-specific requirements into account. An initial set of decisions required on the project is determined in this step. These may or may not appear in the tailored RADMs; hence, architectural decision knowledge

can be added, updated, or deleted in this step. Together with steps 1 to 4 (asset creation), this step realizes the decision identification part (a) of the decision-centric design method sketched in Section 3.4.

6. *Make decisions*. In this step, architects review the architectural decision knowledge in the ADM created in step 5, match this information against the project requirements, make their decisions, and update the ADM. When locating the relevant parts of the model in a given project situation, they are assisted by the model structure and the temporal order of the decisions developed in steps 3 and 4. This step realizes the decision making part (b) of the decision-centric design method sketched in Section 3.4.
7. *Enforce decisions*. In this step, architects share the rationale for the decisions made in step 6 and captured in the ADM. They update other architectural artifacts accordingly. Via decision logs, they instruct the project team which chosen alternatives to implement. Furthermore, they provide fragments of development artifacts to demonstrate how to implement certain architectural concepts. This step realizes the decision enforcement part (c) of the decision-centric design method sketched in Section 3.4.

It is worth noting that steps 5 to 7 may be executed repeatedly and in an overlapping fashion. The execution rhythm depends on the software engineering methods and design practices in use (see discussion in Section 7.2 in Chapter 7). For instance, agile processes advise practitioners to reprioritize and reorganize the design work daily [Yip]; they put little emphasis on upfront architecture design.

The architectural knowledge gained on the project and captured in the ADM can be fed back to the community-level RADM. Details of the architectural decision harvesting activities are out of scope here; they are covered by Appendix A.

Supporting concepts. The seven steps required to create and consume a RADM asset are not straightforward to perform. To assist knowledge engineers and architects using SOAD, we provide supporting concepts corresponding to these steps:

1. A pattern-centric *technique for decision identification* provides instructions how to scope a RADM in a reproducible way (step 1). Starting from the definition of an architectural style, the technique uses *identification rules* and style-independent *meta issues* to scope a RADM for this style.
2. A common *metamodel* for RADMs and ADMs ensures that individual decisions are modeled consistently and can be exchanged within the community (step 2). To facilitate reuse, we extend existing work and distinguish decisions required from decisions made in our metamodel.
3. Modeling *logical dependencies* such as decision refinement and decomposition organizes RADMs and ADMs into levels and layers (step 3).
4. A formal definition of *temporal dependencies* creates an order in a decision model that can be followed during design (step 4).
5. *Decision filtering* supports the RADM tailoring into an ADM (step 5).
6. The above concepts allow us to define a *macro* and a *micro process for decision making* (step 6). The project-wide macro process is supported by

an actively *managed issue list* comprising decisions required and decisions made. It launches the micro process for each decision required.

7. *Decision injection* into logical design models and development artifacts integrates ADMs and models for other architectural viewpoints (step 7). This makes it possible to reflect decisions in other artifacts.

In addition to the concepts, *tool support* for them is required so that the framework steps can be applied by knowledge engineers and teams of software architects easily. Table 9 gives an overview how the concepts and tool support for them solve the research problems from Chapter 3 and where in this thesis these solutions are described.

Table 9. Research problems solved by framework steps, concepts, and tool

Research Problem	SOAD Step and Coverage in Thesis	State of the Art (from Section 3.3)	SOAD Concept (Chapters 5 to 8)
Decision identification	Step 1: Identify decisions (Chapter 5)	To be pulled from literature (patterns), as well as analysis and design artifacts	Identification rules, meta issue catalog
	Step 5: Tailor model (Chapter 7, Section 7.1)		Decision filtering
Decision modeling	Step 2: Model individual decisions (Chapter 6, Section 6.1)	Metamodels in architectural knowledge management	Existing metamodels extended for reuse and collaboration in new context
Model structuring	Step 3: Structure model (Chapter 6, Section 6.2)	Decision ontologies, MDA model types	Model formalization, refinement levels, topic group hierarchy starting with layers, decomposition
Dependency management	Step 3: Structure model (Chapter 6, Section 6.2) Step 4: Add temporal decision order (Chapter 6, Section 6.3)	Pattern relations, ontologies in architectural knowledge management (for decisions made)	Formalization of logical and temporal dependency relations, integrity constraints, production rules (for decisions required)
Design method usage	Step 6: Make decisions (Chapter 7, Section 7.2)	Backlog (manual updates)	Managed issue list, macro and micro process
Decision enforcement	Step 7: Enforce decisions (Chapter 7, Section 7.3)	Agile development, governance models	Decision injection in model-driven development
Collaboration system	Tool support for seven SOAD steps (Chapter 8)	Plugins to rich client tools, standard wikis	Design of an application wiki for decision modeling

This section provided sufficient information to proceed with Chapters 5 to 7, which present the SOAD framework steps and concepts from the bottom up. In the remainder of this chapter, we give a framework overview in a top-down manner. We also demonstrate how to apply the framework to SOA design.

4.2 Framework Concepts in Architecture Design Context

In this section we first explain how SOAD advances from retrospective decision capturing to proactive decision modeling. Next we position SOAD in the software engineering process and the SOA design tool context introduced in Chapter 2.

4.2.1 Separating Design Issues from Decision Outcomes

In the current state of the art and the practice, architectural decisions are captured after they have been made on a particular project. Decision reuse as motivated in Section 4.1 is hard to achieve with such a retrospective approach [TAG+05]. To facilitate such reuse, we distinguish decisions *made* from decisions *required*:

Definition 4.3 (Outcome). A *decision* outcome *is the record of a decision actually made on a project and its* justification*. Outcomes may only appear in ADMs.*

Figure 3 on page 14 showed outcomes to appear in the decision log. Such a log is an architecture documentation artifact, providing rationale for a certain design. Decision logs with outcomes convey valuable, but project-specific information. To make decision knowledge reusable, we introduce the notion of issues:

Definition 4.4 (Issue). *A design* issue *informs the architect that a particular design problem exists and that an architectural decision is required. It presents* decision drivers *(e.g., quality attributes), and references potential design* alternatives *which solve the issue along with their* pros *(advantages),* cons *(disadvantages) and* known uses*. It may also make a* recommendation *about the alternative to be selected in a certain requirements context. Issues appear in RADMs and in ADMs.*

The design issues in the motivating case study, which we outlined in Section 2.2.5 in Chapter 2, are examples of such issues. Figure 12 zooms into RADM and ADM from Figure 11 and shows their issue, alternatives, and outcome content:

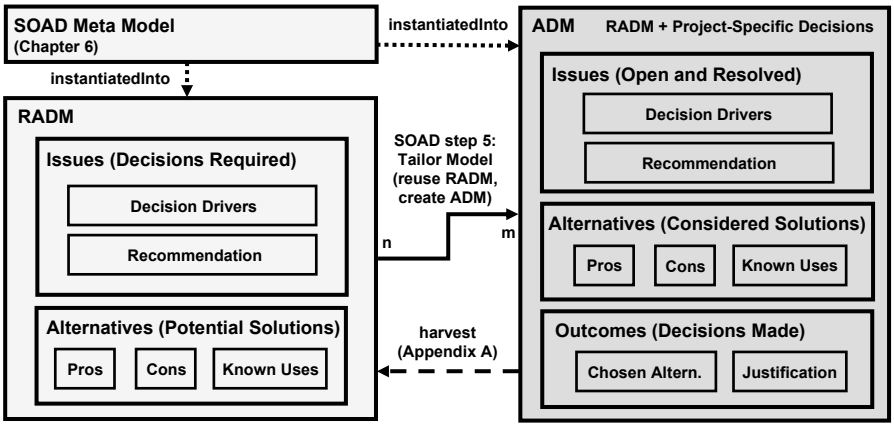


Figure 12. RADM and ADM elements

RADMs and ADMs are instances of the SOAD metamodel. Issues convey knowledge about design problems that may occur (RADM) or actually occurred (ADM). Outcomes document alternatives chosen; they are created by the architects to complete the architectural decision capturing on projects. An outcome refers to an issue, which may come from a RADM or may have been created in the ADM. Issues without outcomes are *open issues*. *Resolved issues* have outcomes; they comprise the decision log. Open and resolved issues form the *managed issue*

list for the architects. We will define these concepts in Chapters 6 (metamodel) and 7 (managed issue list).

Having introduced issues and outcomes, we can now investigate where SOAD framework, RADMs, and ADMs fit in the software engineering process.

4.2.2 The Framework in the Software Engineering Process

To support the envisioned extended usage of architectural decision models during design, our architectural decision modeling framework has to tie in with the software lifecycle and software engineering process introduced in Chapter 2. The integration points with concepts in existing work must be clarified, e.g., artifacts and process phases in software engineering methods and architectural viewpoints.

As a reusable asset, a SOAD RADM guides the architect through the design activities; it complements artifact- and process-centric software engineering methods such as Rational Unified Process (RUP) [Kru03] or IBM Unified Method Framework (UMF) [CCS07] with architectural decision knowledge. An ADM is a project-specific architecture documentation artifact; such artifact is known in many methods. The ADM is updated during the *architectural decision modeling* activities, which become part of the process defined by the method. Figure 13 shows this RADM and ADM positioning as an extension of Figure 3 on page 14:

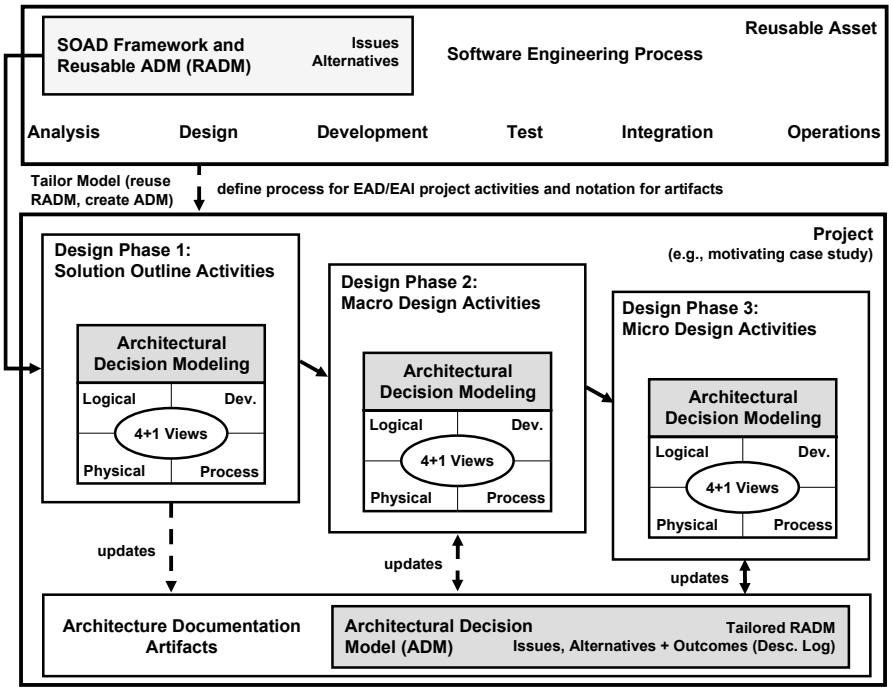


Figure 13. Decision modeling as a guide through the architecture design work

Figure 13 uses the same process as Figure 3: The architecturally relevant documentation artifacts are updated during the three UMF design phases. We use Kruchten's 4+1 view model [Kru95], introduced in Chapter 2, to structure the architecture design activities and documentation artifacts: Issues and outcomes pertain to architecture elements that appear in these views, e.g., *components* and *connectors* [BCK03] in the logical and *nodes* [YRS+99] in the physical view.

Having clarified when and by whom RADMs and ADMs are created and used, we propose the architecture of a tool supporting the SOAD framework next.

4.2.3 Tool Support for Framework Concepts

To make the architectural decision modeling activities efficient, a tool can support the SOAD framework. Such a tool must fit into the various tools already used during SOA design (shown in Figure 10 on page 39).

We use Object-Oriented Analysis and Design (OOAD) [Boo94] terms to introduce the SOAD tool context: The steps we introduced in Section 4.1 realize the design method *use case* for SOAD.[23] Knowledge engineers and software architects are the *primary actors* of SOAD; other stakeholders are *secondary actors*, e.g., developers and project managers.

Figure 14 introduces the tool architecture we propose from a logical viewpoint:

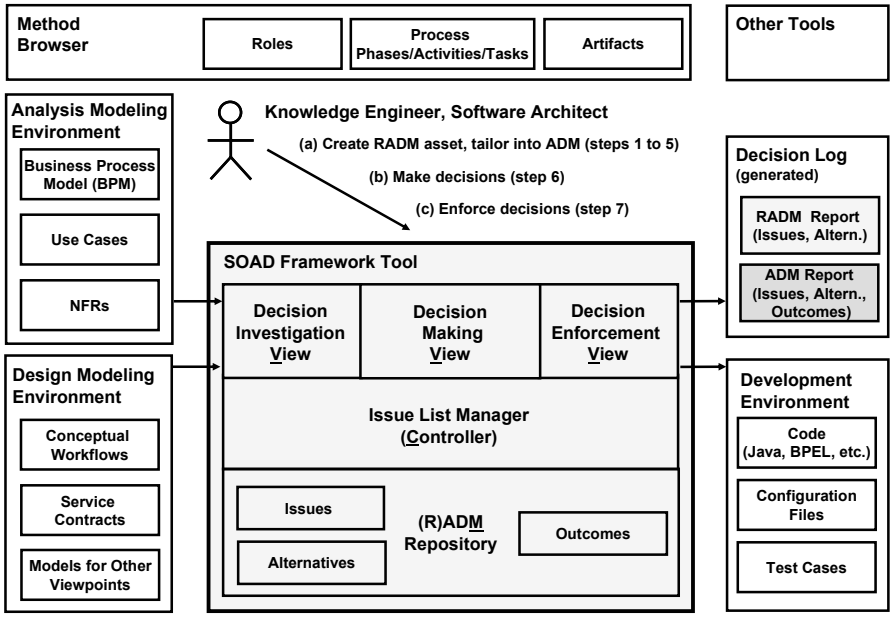


Figure 14. An architecture for a SOAD tool and its context

[23] As motivated in Chapter 1, secondary use cases are *education*, *knowledge exchange*, *review technique*, and *governance instrument*. We focus on the design method usage here.

A SOAD tool used during design has to provide interfaces with analysis modeling, design modeling, and development environments as well as method browsers. These tools contain artifacts that trigger or reflect architectural decisions. According to Figure 10, project management software, office suites and traceability management tools, as well as asset repositories also have to be interfaced with; these tools are not shown in Figure 14.

As a tool for SOAD must respond to user stimuli and be extensible, we define a component-oriented architecture [Eme03] and organize it according to the model-view-controller pattern [BMR+96]. The framework steps from Section 4.1 are supported by five components:

1. The *decision investigation view* supports the asset creation steps 1 to 4 and model tailoring step 5, taking analysis and design models as input.
2. There is a *decision making view* for step 6.
3. The *decision enforcement view* supports step 7 with decision log (report) generation and development environment integration capabilities.
4. An *issue list manager* controls the RADM and ADM processing. It is primarily used in step 6, but also connects the asset consumption steps 5 to 7.
5. The *RADM and ADM repository* is structured according to a common metamodel, which comprises the model elements introduced in Section 4.2.1. RADMs and ADMs are instantiated from this metamodel.

In contrast to the fragmented situation in Figure 10, our SOAD tool stores and manages architectural decision knowledge via dedicated, centralized components.

We now apply SOAD framework steps to SOA design, returning to the motivating case study and the artifacts and tools introduced in Chapter 2.

4.3 Application of the Framework to SOA Design

To give an example how to use SOAD in SOA design, we continue with a subset of the architecture design activities we initiated in Chapter 2. Steps 1 to 4 are independent of the motivating case study; steps 5 to 7 are case-specific.

Step 1: Identify decisions. Decision identification scopes a RADM and the SOA design work: The SOA patterns (Definitions 2.6 to 2.9) led to the exemplary design issues we motivated in Section 2.2.5 beginning on page 29. We categorized them into strategic design issues, conceptual design issues, and platform-related design issues. Let us recapitulate and name the issues now.

SERVICE COMPOSITION PARADIGM, WORKFLOW LANGUAGE, and BPEL ENGINE[24] are issues that refine the process manager component that appears in Definition 2.8 (see Figure 4 on page 16): First, a conceptual paradigm how to realize the process manager must be selected (e.g., WORKFLOW). Moreover, a language technology implementing the paradigm must be chosen (e.g., BPEL). Fi-

[24] From now on, issues and alternatives are set IN THIS FONT (SMALL CAPS).

nally, a middleware product or open source asset supporting this language must be picked (e.g., WEBSPHERE PROCESS SERVER). The three issues are related to each other; logical dependencies between their alternatives exist.

The issues INTEGRATION PARADIGM, INTEGRATION TECHNOLOGY, and SOAP ENGINE pertain to the ESB pattern from Definition 2.7, following the same refinement hierarchy. We explain these issues in Chapter 5.

IN MESSAGE GRANULARITY, OUT MESSAGE GRANULARITY, OPERATION-TO-SERVICE GROUPING, MESSAGE EXCHANGE PATTERN, TRANSPORT PROTOCOL BINDING, and INVOCATION TRANSACTIONALITY PATTERN decisions must be made for each operation realized by a service provider (Definition 2.6). WEB SERVICES API is one of many issues regarding service consumers (Definition 2.6). All these issues appear in the RADM for SOA; we will cover them in Chapters 6 and 7. Some of them are featured in separate publications [ZGT+07, ZZG+08, PZL08].

All these issues recur, which qualifies them for inclusion in a RADM such as the RADM for SOA we created during thesis validation. In this step 1, we only identify the issues by name to scope the RADM; in steps 2 to 4, we add detailed architectural decision knowledge to populate the RADM.

Steps 2, 3, and 4: Model individual decisions, structure model, and add temporal decision order. In these steps, architectural knowledge about the issues is added to the RADM, e.g., decision drivers, alternatives, and recommendations. For instance, a problem statement for INTEGRATION TECHNOLOGY is "which remoting technology should be used to let the activities in the business process communicate with Web services?". Selected decision drivers are quality attributes such as "interoperability", "reliability", and "tool support". Alternatives known to be used on projects are WS-* [WCL+05] and RESTFUL INTEGRATION [Fie00].

In step 1 above we already gave first examples of model structure and decision dependencies; more examples will follow in Chapter 6.

Step 5: Tailor model for SOA project. The system context diagram (Figure 2 on page 10) indicates which existing PremierQuotes Group systems are involved (e.g., customer care, contract, and risk management) and which mandatory distribution requirements are introduced by the user channels. The analysis-phase BPM (Figure 5 on page 23) captures functional requirements about one business process (customer enquiry). The business rules and NFRs including legacy constraints (see Section 2.2.3) provide us with decision drivers, e.g., concerning process integrity, interoperability, standards usage, and already existing backend interfaces.[25]

Let us assume that all issues identified in step 1 above are relevant in this case as they arise from the adoption of the service consumer-provider contract, ESB, and service composition patterns. Detailing the SOA from Figure 8 on page 28, the following Figure 15 assigns the issues from step 1 to logical components in the architecture. The issues are shown as questions. Several of them appear multiple times, e.g., those about the ESB and those dealing with the three atomic services

[25] Typically not all decision drivers are specified in explicit form. In practice, explicit NFRs may be incomplete and/or unrealistic as they are difficult to agree upon. Tacit knowledge advises the architect how to deal with this situation. The RADM can give related advice.

(customer care service, contract service, and risk management service). This is the case because the respective patterns are applied multiple times in the architecture. As the service registry pattern is not used, no related issues arise on this project.

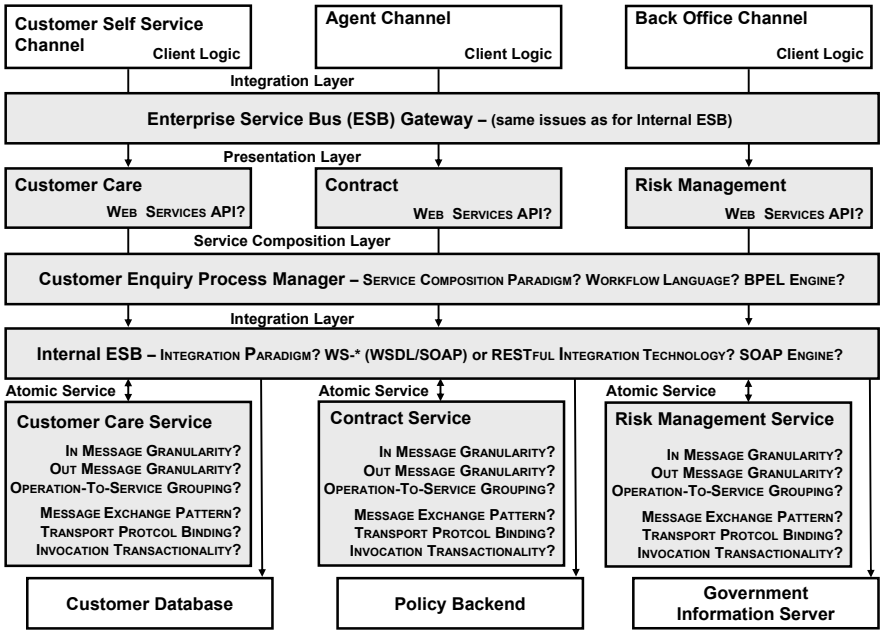


Figure 15. Decision identification in motivating case study

Step 6: Make decisions. During this step, the PremierQuotes Group architects select alternatives resolving the open issues based on project-specific requirements. During their SOA design and architectural decision modeling activities (shown in the middle of Figure 13 in Section 4.2.2), they capture the justifications for their decisions in outcomes, which refer to issues (see Section 4.2.1).

The SOAD tool proposed in Section 4.2.3 can support this step: The decision making view filters and orders issues by practitioner role, project phase, and architecture element as defined by method and viewpoint schema in use. To do so, the view is supported by the issue list manager and the (R)ADM repository.

Pattern selection decisions such as using WORKFLOW as SERVICE COMPOSITION PARADIGM (refining the abstract process manager from the service composition pattern, Definition 2.8) may be made during the solution outline phase. In macro design, implementation technologies such as BPEL as WORKFLOW LANGUAGE may be selected. Product selection (e.g., for a particular BPEL ENGINE such as WEBSPHERE PROCESS SERVER) would best be conducted in macro or micro design; however, practical challenges such as procurement processes often mandate that executives make (and/or approve) such decisions at an early stage, e.g., solution outline (or even prior to project start). Product configuration typically takes place in the micro design and in the later development, test, and integration phases.

Continuing the case study, let us assume the SOA project to be in the macro design phase; several key decisions have already been made during solution outline. This becomes apparent in Figure 15, e.g., a service composition layer and two ESBs have already been introduced in the architecture. Table 10 gives five more examples for decisions already made, captured as outcomes. The issues and alternatives come from the RADM for SOA scoped in step 1 and populated in steps 2 to 4. The sample justifications are specific to the case, referring or paraphrasing the requirements for the motivating case study which we stated in Chapter 2.

Table 10. Motivating case study: Architectural decisions made already

Resolved Issue	Alternative Chosen as Outcome (and Rejected Ones)	Examples of Justifications for Decisions Made (Rationale)
ARCHITECTURAL STYLE (not shown in Figure 15)	SOA MESSAGING (DEF. 2.7) (FILE TRANSFER, SHARED DATABASE, RPC [HW04])	Strategic initiative, cross platform integration required and desired, reliability needs (see Section 2.2.3)
LAYERING (sketched only in Figure 15)	LAYERS FROM DEFINITION 2.8 (POEAA LAYERING [Fow03])	Defined by enterprise architecture team; no industry standard
INTEGRATION PARADIGM	ESB [KBH+04] (TRADITIONAL EAI, CUSTOM CODE)	Integration needs (legacy constraints 1 to 3), service monitoring required
SERVICE COMPOSITION PARADIGM	WORKFLOW [LR00] (HUMAN USER, OBJECT-ORIENTED PROGRAMMING)	Long running process, central process manager can preserve integrity across channels (business rule 2)
SERVICE REGISTRY	NONE (UDDI, VENDOR PRODUCTS) [ZTP03]	Only a few services appear in BPM, no business case for a registry

Refining the previously made decisions, the ones in the following Table 11 proceed from conceptual to platform-specific design. A decision dependency is mentioned, relating the WORKFLOW LANGUAGE and BPEL ENGINE issues:

Table 11. Motivating case study: Architectural decisions made now

Resolved Issue	Alternative Chosen as Outcome (and Rejected Ones)	Examples of Justifications for Decisions Made (Rationale)
INTEGRATION TECHNOLOGY	WS-* WEB SERVICES [ZTP03] (RESTFUL INTEGRATION [PZL08])	Interoperability and standardization requirements (NFRs), tool support
WORKFLOW LANGUAGE	BPEL [OAS07] (PROPRIETARY LANGUAGES)	Standardized (NFR 2), used by BPEL ENGINE selected (see below)
SOAP ENGINE	IBM WEBSPHERE (APACHE AXIS2)	Comes with BPEL ENGINE
BPEL ENGINE	WEBSPHERE PROCESS SERVER (ORACLE BPEL PROCESS MANAGER, ACTIVE BPEL)	Operational procedures and enterprise license agreement in place (executive decision before project start)

So far, we merely captured decisions already made and their rationale. Table 12 lists additional issues, this time issues still open at the current project stage:

Table 12. Motivating case study: Architectural decisions still required

Open Issue	Alternatives	Decision Drivers (RADM for SOA)
IN MESSAGE GRANULARITY (see	DOT PATTERN DOTTED LINE PATTERN	Structure and amount of enterprise resources to be exchanged, message ver-

Chapter 7 for introduction of pattern alternatives)	BAR PATTERN COMB PATTERN	bosity, programming convenience and expressivity, change friendliness
OPERATION-TO-SERVICE GROUPING	SINGLE OPERATION MULTIPLE OPERATIONS	Cohesion and coupling in terms of security context and versioning
MESSAGE EXCHANGE PATTERN	ONE WAY REQUEST-REPLY	Consumer semantics and availability needs, provider up times
TRANSPORT PROTOCOL BINDING	SOAP/HTTP SOAP/JMS POX/ HTTP	Provider availability, data currency needs from consumer's perspective, systems management considerations
INVOCATION TRANSACTIONALITY	TRANSACTION ISLANDS TRANSACTION BRIDGE STRATIFIED STILTS	Resource protection needs, legacy system interface capabilities, process lifetime (see Chapter 6 for discussion of issue)

We will resolve these issues and create outcomes in Chapter 7 in Section 7.2.

Step 7: Enforce decisions. In this step, the PremierQuotes architects create reports about decisions made: The outcome content of Table 10 and Table 11 is exported to a decision log, e.g., an architectural decisions artifact in UMF [CCS07]. This artifact is then shared within the technical project team (e.g., other architects, developers, and system administrators) and other stakeholders. The made decisions are executed, e.g., through procurement, installation, and configuration of the selected BPEL ENGINE and through BPEL and Java development activities.

We will give more enforcement examples in Chapter 7. Appendix B provides a complete example of a resolved issue accompanied by an outcome instance.

4.4 Discussion and Summary

In this chapter, we motivated the concepts of decision reuse and decision modeling. We introduced the SOAD framework which comprises seven steps to scope and populate RADMs within a community and tailor them into ADMs used during architecture design on EAD and EAI projects. RADMs and ADMs are instantiated from and adhere to a common metamodel. We presented the architecture of a SOAD tool and applied framework and tool to SOA design.

Justification. The design of the SOAD framework and tool architecture is justified by the 31 method requirements we established in Chapter 3. RADM and ADM are instantiated from a metamodel for architectural decision knowledge (R1-2 and R1-3). The context shown in Figure 14 is justified by the integration needs (e.g., R1-4 and R1-5). The three tool views realize the design method use case for SOAD and address the usability requirement (R1-7). The decision making step 6 addresses the software architecture design method requirements (R2-1 to R2-7) such as the need for a managed issue list (R2-6) and the collaboration requirements (R5-1 to R5-5 and R5-7). The rationale for step 7 can be found in the collaboration needs (R5-5, R5-6).

R3-1 (patterns usage) is satisfied by step 1. The EAD and EAI requirements R3-2 to R3-5 and the SOA requirements (R4-1 to R4-5) are less architecturally significant than the previous ones; they are addressed by the RADM for SOA content and its organization into levels and layers. Hence, we can design the SOAD

framework steps and the supporting tool in such a way that other application genres and architectural styles can also be supported in addition to enterprise applications and SOA. Such generic design also satisfies the extensibility requirement R1-6. We discuss SOAD applicability and extensibility in Chapter 10.

A benefit of our decision-centric approach to method creation is that the target audience, software architects, knows the core metaphor, architectural decisions, from a different usage scenario, architecture documentation. Furthermore, the metaphor is easy to relate to: Making decisions is important in many fields, not just in software architecture design. It is also part of everyday life.

Our selection of presented issues might appear to be rather arbitrary or too SOA-specific. It is justified by several criteria: The examples must be realistic so that they motivate the value of decision reuse and modeling, but also simple to be understandable and self containing. We decided to present examples that deal with service contract design, ESB integration styles and technologies, and service composition using workflow concepts. Additional examples are featured in other publications [ZKL07, ZZG+08, PZL08]. Many more issues appear in the RADM for SOA developed during thesis validation (see Chapter 5 for an introduction).

Assumptions. SOAD assumes that many of the issues recur: If this assumption does not hold, the RADM asset will not provide sufficient value to justify its creation as the effort will outweigh the benefits. If multiple projects in the same application genre employ the same architectural style, there are good chances that this assumption holds; only the design *issue* must recur, not the actual decision *outcome*. We present several industrial case studies that verify this hypothesis in Chapter 9.

We assume that architectural knowledge for the chosen architectural style is already available, e.g., in the form of patterns or decision logs harvested from completed projects (see Appendix A for harvesting process and related guidance), and that a community is willing to make this knowledge explicit. If this is not the case, our decision identification technique (step 1) can also be applied during the design work on a project rather than to scope a reusable asset. Steps 2 to 4 can be reduced in their scope (or even skipped) if a lightweight knowledge sharing strategy is followed, e.g., personalization.

Consequences. To become adopted in practice, a RADM has to meet higher quality standards than project-specific, retrospective decision logs. We discuss this aspect in more detail in Chapter 9 when presenting the results from the validating industry case studies. A funding model as well as a review, approval, and maintenance process must exist.

Asset harvesting must be supported in the framework, e.g., concepts and tools to upgrade ADM information from completed projects to RADM knowledge. Appendix A covers such bottom-up knowledge engineering (asset harvesting).

Next steps. In Chapter 5, Chapter 6, and Chapter 7, we present the seven SOAD framework steps in depth and introduce all supporting concepts.

5 Scoping Reusable Architectural Decision Models

In this chapter, we present SOAD step 1: In Section 5.1 we introduce a pattern-centric technique which identifies the issues to be included in a Reusable Architectural Decision Model (RADM). Next we demonstrate how the technique can be applied to scope a RADM for SOA (Section 5.2) and discuss its rationale (Section 5.3). The technique addresses the decision identification problem from Chapter 3:

Which architectural decisions required (issues) recur during SOA design?
Can such decisions be identified systematically in patterns?

As motivated in Chapter 4, we split the solution to this problem into two steps to facilitate reuse and collaboration: Step 1 is to identify decisions during asset creation; like the following model population steps 2 to 4, it is performed by the knowledge engineer. Step 5 is to tailor a model for a project; this step is performed by the software architect to initiate the asset consumption on a project. Figure 16 shows all seven SOAD steps along with the responsible roles and the design artifacts involved, patterns (step 1) and analysis and design models (step 5):

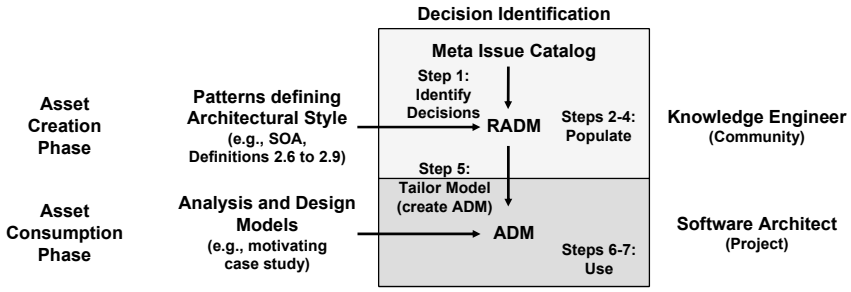


Figure 16. SOAD step 1 and step 5 in context

In this chapter, we focus on step 1; step 5 will be described later in Chapter 7 (Section 7.1). We introduce a generic, style-independent *meta issue catalog* which, together with the patterns defining the architectural style, serves as step 1 input. The output of step 1 is an initial RADM enumerating the names of the decisions required, which we call issues according to Definition 4.4, in a catalog. To populate the RADM, issues and alternatives are then modeled in detail in the subsequent steps 2 to 4.

5.1 Framework Step 1: Identify Decisions

This section briefly reviews the state of the art and the practice, gives an overview of our decision identification technique, and details its activities and concepts.

5.1.1 State of the Art and the Practice

State of the art. Pattern languages, genre- and style-specific extensions to software engineering methods, technical papers, and vendor documentation can be studied to identify issues. In principle, these sources of information provide deep coverage of all issues. However, a vast amount of information must be studied; architectural decisions are often hidden behind various other material not targeting architects and therefore not being presented adequately [ZKL06]. Patterns per se do not aim at guiding the architect through the architecture design activities required once a certain pattern has been selected. The core metaphor of a pattern is *solution*, not *problem*, even if pattern templates usually contain an intent section or a problem statement [Fow06]. Pattern authors often reverse engineer the problem statement from the solution they want to educate the readers about [Hoh07].

State of the practice. Decisions are often identified ad hoc based on personal experience, not via diligent literature studies, or systematic reuse of knowledge already gained. Independent of the technique in use, architects have to search for issues and *pull* the required knowledge from the literature and their experience today. As a consequence, much time is spent in the solution outline phase to identify issues and alternatives. This is particularly true for inexperienced architects.[26]

5.1.2 A Technique for Decision Identification and Model Scoping

To overcome the decision identification challenges, SOAD provides a decision identification technique. It is applied by knowledge engineers who are tasked with the creation of a RADM for an architectural style and comprises five activities:

1. For each pattern in the definition of an architectural style, *review the pattern description*s and enumerate the logical components and connectors [BCK03] referenced in the pattern.
2. *Apply identification rules*, which we will define in Section 5.1.3 below:
   a. Identify issues transcending a particular system context, e.g., business domain- and enterprise-wide ones [MB02, Pul06].
   b. Identify pattern-specific issues (see below).
3. During activity 2, *screen sources of architectural decision knowledge*:
   a. Screen supplemental design artifacts about the architectural style.
   b. Screen catalog of generic meta issues (see below).

[26] The assessment is subjective, drawing on input from practicing architects and personal experience (see Chapter 9 and 10). It is supported by the findings in [DFL+07, TAG+05].

This top-down identification activity is detailed in Section 5.1.4 below.

4. *Add architectural knowledge gained on projects* that have already applied the architectural style or the patterns (bottom-up identification).
5. *Add issues from activities 2 to 4 to RADM* if:
   a. they are architecturally relevant (i.e., satisfy the definition of an architectural decision from Chapter 1),
   b. they have a high potential to recur (i.e., they are not project-specific), and
   c. they are not already present in the RADM.

We focus on activities 2 and 3 in this section. Activities 1 and 5 are self explaining, and Appendix A presents an informal description of activity 4.

5.1.3 Technique Concept: Identification Rules

All architecture design methods introduced in Chapter 2 emphasize the need to refine and elaborate designs iteratively and incrementally. The importance of a global view is also stressed [HKN+07]. Following the same principles of stepwise refinement and separating such global view from that on individual design model elements, we introduce seven *Identification Rules (IRs)* to organize activity 2 in our decision identification technique (detailed explanations and rationale follow):

IR1. *Identify style-independent issues with project- or enterprise-wide scope*. We call issues identified with IR1 *executive decisions*, adopting a term from [KLV06]. IR1 is detailed below.

IR2. *For each pattern in the definition of the architectural style, add one issue to the RADM, deciding whether the pattern is used or not.* We call issues identified with this IR *Pattern Selection Decisions (PSDs)*. Issues selecting the SOA patterns in Definitions 2.6 to 2.9 are examples.

IR3. *Identify Pattern Adoption Decisions (PADs) in PSDs, already identified PADs, and the logical components and connectors comprising the patterns involved in these PSDs and PADs.* IR3 is explained below.

IR4. *For each logical component and connector that is part of a pattern referenced in a PSD or PAD, add one issue concerning its implementation technology.* Such issues may present alternatives regarding integration middleware and application servers as well as application and network protocols. We call issues identified with IR4 *Technology Selection Decisions (TSDs)*.

IR5. *Identify Technology Profiling Decisions (TPDs) in TSDs.* IR5 is explained below.

IR6. *For each technology appearing in a TSD, add one issue deciding which vendor asset is used to provide the technology.* Commercial, open source, and company-internal assets provide alternatives. We call issues identified with this IR6 *Asset Selection Decisions (ASDs)*.

IR7. *Identify Asset Configuration Decisions (ACDs) in ASDs.* IR7 is explained below.

Figure 17 illustrates the activities from Section 5.1.2 and the relations between the seven IRs. We place the IRs in four groups, *executive* (IR1), *conceptual patterns* (IR2, IR3), *technologies* (IR4, IR5), and *vendor assets* (IR6, IR7) and distinguish two types of relations between IRs: Relations between IRs in the same group are *decomposition relations*, relations between IRs in different groups *refinement relations*. Later we will organize the groups hierarchically and use the relations to structure RADMs and ADMs (see Section 5.2 and then Section 6.2 in Chapter 6).

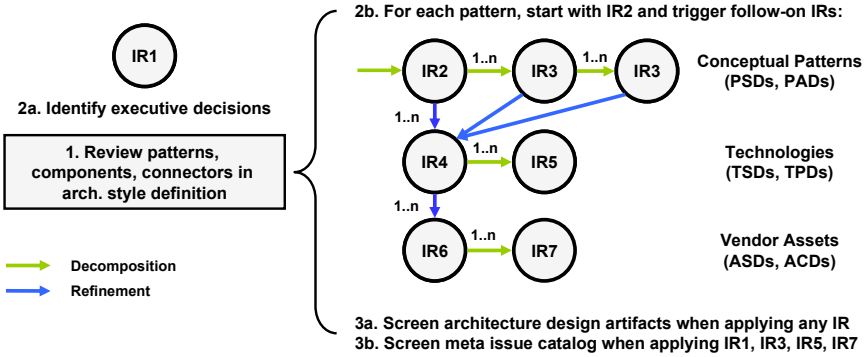


Figure 17. Identification rules in decision identification technique

Figure 17 indicates that additional architecture design artifacts and the generic, style-independent meta issue catalog are used in activities 3a and 3b. We will introduce these concepts shortly; before that, we elaborate on the IRs.

IR1. IR1 deals with executive decisions about strategic technical directions [KLV06] as well as business requirements analysis [Som95]. It pertains to the scenario viewpoint in Kruchten's 4+1 model [Kru95]. Examples of such strategic issues are platform directions (e.g., programming language, operating system, and hardware preferences) as well as strategic, cost-intensive decisions regarding network and server topologies (e.g., setup of geographically distributed data centers, standalone server versus high availability server cluster).

IR2, IR3. The need for PSDs is obvious if a pattern-centric approach is followed. Patterns can be found in all architectural viewpoints; many existing patterns take a logical one [Kru95]. PSDs identified with IR2 have a long lasting impact on project and solution health; many functional and non-functional decision drivers must be considered. The user channel, process and resource integrity, integration, and semantics challenges from Chapter 2 provide many of these decision drivers.

PADs then deal with selected patterns in a detailed way. Many pattern descriptions list variants; one or more variants have to be selected once a PSD has been made. For instance, the description of the "broker" pattern in [BMR+96] lists "direct communication" as a variant; deciding for or against this variant is a PAD. A bullet list in the solution part of a pattern text may also indicate variability, requiring a PAD. Many pattern books supply navigable diagrams or decision trees to show how composite and atomic patterns in a pattern language relate to each other

[Eva03]. Pattern grammars are emerging as well [Zdu07]. These design options may also lead to the identification of one or more PADs.[27]

IR4, IR5. When refining a conceptual, platform-independent design based on patterns into an implementable, platform-specific one, decisions about implementation technologies must be made: TSDs identified with IR4 select certain technologies that implement the patterns selected in PSDs and adopted in PADs. The development perspective of the SOA Definition 2.5 corresponds to this group.

TPDs identified with IR5 follow TSDs. They specify implementation details, e.g., which version or subset of a technology standard to employ or which design alternatives permitted by a standard to pick. XML SCHEMA (XSD) CONSTRUCTS is a TPD example recurring in SOA design: due to the large scope of the technology standard, the subset of the XSD language constructs used to model SOAP request and response messages must be decided (see Definition 2.6).

Technology-level decisions are more concrete than those pertaining to pattern selection and adoption; measurable decision drivers regarding interoperability, performance (i.e., response time and throughput), and scalability apply.

IR6, IR7. ASDs and ACDs identified with IR6 and IR7 pertain to assets that provide and support the technologies selected in TSDs and profiled in TPDs. In SOA design, commercial products, open source, and company-internal assets supply the alternatives. Discrepancies between abstract concepts and implementation reality can be expressed as ACDs: Vendor products may implement a conceptual pattern in an unusual way, have limitations, or offer proprietary extensions.

Having defined the IRs applied in activity 2 of our identification technique, we cover activity 3 next, which deals with the top-down identification of knowledge.

5.1.4 Artifact Screening and Meta Issue Catalog

Screen supplemental design artifacts (all IRs). Table 13 repeats the IR cardinalities from Section 5.1.3 and adds information about the artifacts in which architectural knowledge about the issues can be found, as well as additional follow-on issues (the dependencies are modeled later in step 3):

Table 13. Identification rules, cardinalities, and artifacts to be screened

Identification Rule	Cardinality (Section 5.1.3)	Artifacts to be Screened
IR1: Identify executive decisions	Apply *once* (specific for application genre, but not for architectural style)	Enterprise architecture documents [SZ92], project proposals, system context diagrams, meta issues (Table 14)
IR2: Identify PSDs	Apply *once* per pattern in style definition	Architectural style definition, table of content, overview diagrams, and cheat sheets in pattern books, e.g., [Fow03]

[27] If two patterns have similar or identical intent, context, or forces sections, they can be combined into a single PSD. This is a modeling decision of the knowledge engineer.

IR3: Identify PADs in PSDs and PADs	Apply *multiple times* per PSD/PAD and logical component and connector in pattern	Descriptions of architectural patterns (online, text books), pattern variants and grammars, meta issues
IR4: Identify TSDs in PSDs and PADs	Apply *once* per logical component and connector in pattern	Enterprise architecture documents, standards bodies (e.g., W3C, OASIS)
IR5: Identify TPDs in TSDs	Apply *one or more times* per TSD	Technology standards and primers, tutorials, meta issues
IR6: Identify ASDs in TSDs	Apply *once* per technology appearing in a TSD	External parties (analyst reports), enterprise architecture documents
IR7: Identify ACDs in ASDs	Apply *one or more times per* ASD	Vendor documentation, previous projects, existing systems, meta issues

The artifacts appearing in Table 13 are either referenced, e.g., [Fow03], come from the software engineering methods introduced in Chapter 2 (e.g., system context diagram), or are self explaining (e.g., vendor documentation). They can be part of the definition of the architectural style for which the RADM is created or originate from already completed projects which have applied the style.[29]

Screen catalog of generic meta issues (IR1, IR3, IR5, IR7). IR2, IR4, and IR6 are straightforward to apply. However, architecture design work does not stop when patterns, technologies, and vendor assets have been selected; hence, pattern adoption, technology profiling, and vendor asset configuration issues have to be identified as well. According to our knowledge engineering experience, pattern texts, technology specifications, and vendor documentation often do not provide detailed information about such issues; information about platform-dependent quality attributes such as performance and scalability remains tacit.[30] More knowledge is required to make IR1, IR3, IR5, and IR7 reproducible and scope the RADM in such a way that the issues are concrete and specific enough to be applicable during the design work on a project (i.e., in the asset consumption phase).

To provide such knowledge, we introduce the notion of meta issues: *Meta issues are architectural decisions that recur in the application genre, but are not specific to any architectural style, implementation technology, or vendor asset.* Like issues, meta issues have to meet the qualification criteria for architectural decisions from Chapter 1; for instance, they must pertain to the system as a whole or to its key components, and impact the quality attributes of the system. However, they are more abstract and generic than RADM issues, e.g., they do not reference any particular component or connector in the patterns defining the architectural style. Unlike patterns, they describe problems (design concerns) rather than solutions to them. Each issue references and instantiates one or more of the meta issues. To give an example: "system transactionality" is a meta issue because usage of the concept is common in many application genres and architectural styles. Fowler [Fow03] instantiates the meta issue into an issue giving concrete advice for

[29] In the latter case, the bottom up identification activity 4 (see Appendix A) can assist with the harvesting of the knowledge. Bottom up harvesting is also required for steps 2 to 4.

[30] For patterns, this is not the fault of the pattern author: By design, most patterns are "soft around the edges" [Hoh07] to make them broadly applicable and platform-independent.

enterprise application architectures and concurrency management in application servers that support a Web-based presentation layer.

A meta issue catalog makes formerly tacit knowledge explicit. Table 14 presents our meta issue catalog which, when combined with the patterns defining the SOA style, yields concrete executive decisions as well as PAD, TPD, and ACD issues when applying IR1, IR3, IR5, and IR7 to scope a RADM for SOA.

Table 14. Meta issue catalog for EAD and EAI

IR and Artifact	Decision Topic	Meta Issues
IR1: Enterprise architecture documentation [SZ92, ZTP03]	IT strategy	Buy vs. build strategy, open source policy
	Governance	Methods (processes, notations), tools, reference architectures, coding guidelines, naming standards, asset ownership
IR1: System context [CCS07]	Project scope	External interfaces, incoming and outgoing calls (protocols, formats, identifiers), service level agreements, billing
IR1: Other viewpoints [Kru95]	Development process	Configuration management, test cases, build/test/production environment staging
IR3: Architecture overview diagram [Fow03, CCS07]	Logical layers	Coupling and cohesion principles, functional decomposition (partitioning)
	Physical tiers	Locations, security zones, nodes, load balancing, failover, storage placement
	Data management	Data model reach (enterprise-wide?), synchronization/replication, backup strategy
IR3: Architecture overview diagram [Eva03, Fow03]	Presentation layer	Rich vs. thin client, multi-channel design, client conversations, session management
	Domain layer (process control flow)	How to ensure process and resource integrity, business and system transactionality
	Domain layer (remote interfaces)	Remote contract design (interfaces, protocols, formats, timeout management)
	Domain layer (component-based development)	Interface contract language, parameter validation, Application Programming Interface (API) design, domain model
	Resource (data) access layer	Connection pooling, concurrency (auto commit?), information integration, caching
	Integration	Hub-and-spoke vs. direct, synchrony, message queuing, data formats, registration
IR3: Logical component [ZTP03]	Security	Authentication, authorization, confidentiality, integrity, non-repudiation, tenancy
	Systems and network management	Fault, configuration, accounting, performance, and security management
IR3: Logical component [ZZG+08]	Lifecycle management	Lookup, creation, deletion, static vs. dynamic activation, instance pooling, caching
	Logging	Log source and sink, protocol, format, level of detail (verbosity levels)
	Error handling	Error logging, reporting, propagation, display, analysis, recovery
IR5 and IR7: Components and connectors [ZTP03, CCS07]	Implementation technology (IR5)	Technology standard version and profile to use, deployment descriptor settings (QoS)
	Deployment (IR7)	Collocation, standalone vs. clustered
IR7: Physical node [YRS+99]	Capacity planning	Hardware and software sizing, topologies
	Systems and IT service management	Monitoring concept, backup procedures, update management, disaster recovery

This particular catalog originates from project experience [ZMC+04, ZDG+05] as well as the literature, e.g., [Eva03, Fow03, HNS00, HW04]. The meta issues in this catalog are relevant and recurring in EAD and EAI as introduced in Chapter 2, and they address the genre-specific design challenges in Section 2.1.2 (i.e., user and channel diversity, process and resource integrity management, integration, and semantics). Solutions to them may exist in pattern form; these patterns then become alternatives resolving identified issues. The meta issues do not prerequisite or imply any architectural style such as SOA. When being combined with the SOA patterns from Section 2.1.4 (Definitions 2.6 to 2.9), the meta issues in the catalog are broad and deep enough to reproduce the 389 issues in our RADM for SOA (see Section 5.2).

The meta issue catalog merely serves as reference; it is not self explaining. To apply our technique, the knowledge engineer must be familiar with the subject matter and/or have project experience with the architectural concerns indicated by the meta issues. The referenced literature provides background information.

Termination. The RADM creation activities continue until the model is rich enough to support SOAD steps 5 to 7, ADM creation and usage on projects. No firm termination condition can be given for a technique targeting human knowledge engineers: According to our experience (see case study 3 in Chapter 9) and assuming a codification strategy for architectural knowledge management, up to a dozen issues should be added for atomic patterns and about 20 to 30 for composite patterns. Quality and accuracy have higher priority than quantity.

Extensibility. We do not claim the meta issue catalog to be complete; when applying SOAD, it is possible to add, update, and delete meta issues in the catalog as needed. For instance, the following sources of input can be taken into account when creating a custom meta issue catalog:

- Other architectural patterns [VKZ04], problem descriptions in intent, context, forces, and consequences sections in particular.
- Architectural tactics as defined in software architecture literature [BCK03] and other architectural knowledge that meets the definition of a meta issue.
- Design challenges explained in genre-specific literature, e.g., BPM tutorials, EAI handbooks, and industry reference models [IBM, Sup].
- SOA literature also presenting style-agnostic knowledge [Jos07, KBS05].

This completes the conceptual coverage of SOAD step 1, which scopes an initial RADM. We give a larger example for this step in the next section.

5.2 A Reusable Architectural Decision Model for SOA

To demonstrate that the concepts from Section 5.1 work for SOA, we now introduce a particular RADM, the *RADM for SOA* we created during thesis validation.

Structure of RADM for SOA. The RADM for SOA is organized into *levels* and *layers*: An overarching *executive level* comprises issues regarding requirements analysis and technical decisions of strategic relevance. Picking up the structure from Figure 17, a *conceptual level*, a *technology level* and a *vendor asset level* follow, taking inspiration from MDA model types [OMG03].[31] Architectural layers further structure the RADM; we adopted the layers from the SOA definitions in Chapter 2. Figure 18 shows the resulting model structure (each box represents one group of issues that deal with the same topic area on the same level) :

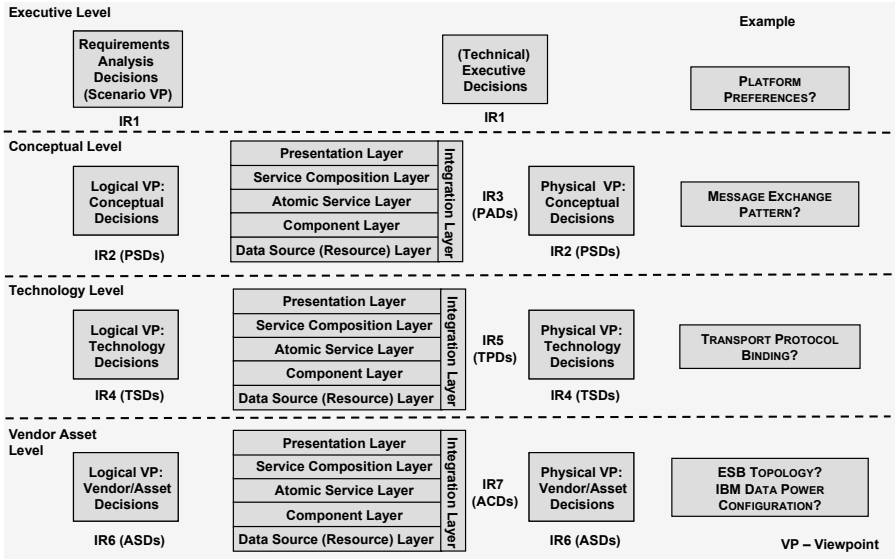


Figure 18. Structure of RADM for SOA (adapted from [ZKL+09])

With the help of IR1 and the meta issue catalog, we identified the issues on the executive level. The requirements analysis decisions are required to scope EAD and EAI project activities; they are related to the scenario viewpoint. In ADMs, their outcomes define the system context for the solution under construction. Technical executive decisions include those listed in Section 5.1.3 (page 72).

The second level from the top is the conceptual level, which in MDA terms is platform-independent [OMG03]. Architectural patterns appear as alternatives of conceptual decisions. A conceptual design helps to prepare an SOA design for future change. The conceptual level contains decisions identified with IR2 and IR3, called Pattern Selection Decisions (PSDs) and Pattern Adoption Decisions (PADs) according to the definition of the identification rule in Section 5.1.3.

The technology level comprises Technology Selection Decisions (TSDs) identified with IR4 and Technology Profiling Decisions (TPDs) identified with IR5. TSDs and TPDs are platform-specific, but do not deal with any particular SOA middleware products yet.

[31] The level concept will be formally introduced in Chapter 6 (Section 6.2).

The vendor asset level comprises Asset Selection Decisions (ASDs) identified with IR6 and Asset Configuration Decisions (ACDs) identified with IR7. It also is platform-specific. ACDs concern the various implementation, configuration, and deployment details in SOA middleware. These issues deal with commercial and open source assets in greater detail than ASDs.

Executive level decisions (IR1 applied). With IR1, we identified the executive decisions in the RADM for SOA. Two of the executive decisions that we introduced in Chapter 4 are: ARCHITECTURAL STYLE[32] with SOA MESSAGING as one of several alternatives, LAYERING (see Definition 2.8 in Chapter 2), and LANGUAGE AND PLATFORM PREFERENCES with alternatives such as MICROSOFT .NET/C#, JEE/JAVA, and LAMPP. TOOLING DIRECTIONS (e.g., OPEN SOURCE, SINGLE VENDOR) also recur. Identified with IR1, these are one-of-a-kind issues specific to the genre, but not to SOA. The identifying meta issues from Table 14 are "reference architectures" and "tools".

Two examples of business requirements decisions appeared in Section 2.2.2 on page 22: ANALYSIS-PHASE BPM vs. USE CASE MODELS or USER STORIES as FUNCTIONAL REQUIREMENTS NOTATION and using BPMN or UML ACTIVITY DIAGRAMS as BPM NOTATION. They were identified with IR1 as well; the meta issue is "methods (processes, notations)".

Conceptual level decisions (IR2 and IR3 applied). The identification rules advised us to add one PSD per pattern (IR2) and multiple PADs per PSD (IR3). The resulting PSDs and PADs in the RADM for SOA deal with the following topics:

- Selection and adoption of the SOA patterns from Chapter 2: service consumer-provider contract, ESB, service composition, and service registry.
- Design of abstract, non-technical part of service contract, corresponding to the WSDL 1.1 [W3C01] port type (interface in WSDL 2.0 [W3C03]).
- Definition of security and service management concepts, e.g., transport- or message-layer security and business process monitoring concepts. Unlike in the full RADM for SOA, these issues are out of our scope here.
- Selection of transaction management patterns (see Chapter 6).

Atomic service layer. A PAD related to the service consumer-provider contract pattern is to decide whether the IN MESSAGE GRANULARITY of the service operations should be coarse or fine in terms of the breadth and depth of the message parts (i.e., number of message parts, usage of scalar or complex data types). This decision is required for each service operation. A similar decision has to be made about the OUT MESSAGE GRANULARITY. Furthermore, a conscious decision for the OPERATION-TO-SERVICE GROUPING is also required. "API design" is the IR3 meta issue for both issues. We return to these three issues in Chapter 7.

A related PAD is MESSAGE EXCHANGE PATTERN, introduced in Chapter 4 and shown in Figure 18: A "service operation" appears in Definition 2.6, and an IR3 meta issue called "synchrony" appears in Table 14. Combining these two knowledge sources during activity 3b (page 70) identified this issue: For each service

[32] We set issues and alternatives IN THIS FONT in this thesis (SMALL CAPS).

operation invocation, it has to be decided how to invoke atomic services from the business activities in the service composition layer. Synchronous REQUEST-REPLY calls and asynchronous ONE WAY messaging are two of the alternatives.

INVOCATION TRANSACTIONALITY PATTERN is an issue we cover in Chapter 6.

Integration layer. INTEGRATION PARADIGM is the PSD that originates from the ESB pattern (see Section 4.3 in Chapter 4 for alternatives). The pattern text of the broker pattern in [BMR+96] supplies us with more knowledge about integration issues: (1) define an object model. (2) decide which type of component interoperability the system should offer, binary or Interface Description Language (IDL). (3) specify the APIs the broker component provides for collaborating with clients and servers. (4) use proxy objects to hide implementation details from clients and servers. (5) design the broker component. (6) develop IDL compilers. Step (5) has nine sub steps: (5.1) on-the-wire protocol, (5.2) local broker, (5.3) direct communication variant, (5.4) (un)marshalling, (5.5) message buffers, (5.6) directory service, (5.7) name service, (5.8) dynamic method invocation, and (5.9) the case in which something fails. All these steps qualify as PADs, following the INTEGRATION PARADIGM PSD according to IR3 (see Figure 17 on page 72).

Service composition layer. We already motivated SERVICE COMPOSITION PARADIGM with alternatives WORKFLOW and OBJECT-ORIENTED PROGRAMMING. Moreover, a PROCESS LIFETIME issue has to be decided for any executable process, with alternatives such as long running MACROFLOW and short running MICROFLOW [ZD06]. This is a conceptual abstraction of an engine-specific design issue not handled by the BPEL specification. This issue is out of our scope in this thesis.

"System transactionality" was one of the meta issues listed in Table 14. The RADM for SOA contains several issues dealing with this concern. For instance, it has to be agreed which RESOURCE PROTECTION STRATEGY should be taken, e.g., SYSTEM TRANSACTIONS or BUSINESS COMPENSATION (or a combination thereof). The SESSION MANAGEMENT approach also has to be decided in this context.

Technology level decisions (IR4 and IR5 applied). The identification rules instructed us to add one TSD per conceptual pattern in the RADM for SOA (IR4) and to add multiple TPDs per TSD (IR5). The issues deal with topics such as:

- Selection of technologies implementing the SOA patterns and profiling of the standards defining these technologies.
- Design of the technical part of the service contract (WSDL binding), and decisions about SOAP [W3C03], BPEL [OAS07], and UDDI [OAS04].
- Selection of protocols, algorithms, and data formats for security, e.g., authentication, authorization, and encryption with Transport Layer Security (TLS) [IETF] and/or WS-Security [WSI07] as well as service management, e.g., monitoring protocols and formats.
- Technology refinement of transaction management patterns, e.g., the decision to use WS-AtomicTransaction [OAS07a] (see Chapter 6).

Atomic service layer. For each service invocation, the following TSDs must be made (Figure 18): Which TRANSPORT PROTOCOL BINDING should be used to invoke atomic services from the processes in the service composition layer, e.g., HYPERTEXT TRANSFER PROTOCOL (HTTP) or JAVA MESSAGING SERVICE (JMS)? Which MESSAGE EXCHANGE FORMAT structures request and response messages in an interoperable manner, e.g., SOAP or JAVASCRIPT OBJECT NOTATION (JSON)?

SOAP COMMUNICATION STYLE with alternatives DOCUMENT/LITERAL or RPC/ENCODED is a related TPD, assuming that SOAP was decided for as MESSAGE EXCHANGE FORMAT. The WEB SERVICES API and JAVA SERVICE PROVIDER TYPE have to be decided per service consumer and service provider; JAX-RPC vs. JAX-WS and ENTERPRISE JAVABEAN (EJB) [SunEJB] vs. PLAIN OLD JAVA OBJECT (POJO) are Java alternatives. This issue and its alternatives are identified with IR4 in activity 3a (see Table 13 on page 73). Moreover, the subset of XML SCHEMA (XSD) CONSTRUCTS used to define message parts in WSDL contracts and SOAP messages must be decided. These issues are identified with IR5, following the IR3-related meta issues about integration and component-based development; the related meta issue is "API design" (Table 14).

Integration layer. A TSD following the PSD about an INTEGRATION PARADIGM is to decide for an INTEGRATION TECHNOLOGY such as WS-* WEB SERVICES or RESTFUL INTEGRATION [PZL08]. It is identified with IR4. TRANSPORT QOS is a related TPD identified with IR5; it is explained in detail in Chapter 6.

Service composition layer. Already motivated in Chapter 4, a TSD that is required for each process is the choice of WORKFLOW LANGUAGE, e.g., BUSINESS PROCESS EXECUTION LANGUAGE (BPEL). Some TPDs follow the TSD to use BPEL: Which BPEL VERSION and which COMPENSATION TECHNOLOGY to use? We refer the reader to [ZZG+08] for further explanations about these issues.

Vendor asset level decisions (IR6 and IR7 applied). ASDs are required for all technologies appearing in TSDs (IR6); ACDs follow ASDs (IR7). Supported by the IR7 meta issues in Table 14, we identified issues about the following topics:

- Issues pertaining to assets that implement the Web services standards, for instance, WSDL editors, SOAP engines, BPEL engines, and UDDI registries.[33]
- Design of the part of the service contract related to deployment, which corresponds to the service and port elements in WSDL 1.1.
- Configuration of the selected products to reflect the technology profiling choices made, including selection and customization of proprietary APIs.

Integration ASDs are the selection of a SOAP ENGINE, of an ESB PRODUCT, and of a BPEL ENGINE. For instance, the IBM DATAPOWER appliance [IBM] ap-

[33] Many of these decisions may be made as executive decisions in practice, e.g., if strategic partnerships with certain vendors or a single vendor policy have been established. This is often the case for middleware such as application servers or databases, with justifications such as direct and indirect costs (e.g., licenses, training, and systems management needs).

pearing in Figure 18 is an XML processing hardware which also implements several of the WS-Security specifications and can act as an ESB. ESB TOPOLOGY (IBM DATAPOWER CONFIGURATION) is a related ACD. The BPEL ENGINE decision has many vendor and open source alternatives, including, but not limited to IBM WEBSPHERE PROCESS SERVER and ORACLE BPEL PROCESS MANAGER. SOAP ENGINE has alternatives such as APACHE AXIS2. We will return to a subset of these issues in Chapter 7.

Table 15 summarizes the RADM for SOA issues we introduced in this section. The full model comprises 389 issues with close to 2000 alternatives. 86 topic groups and 683 relations are defined.

Table 15. Subset of RADM for SOA issues

Identification Rule	Layer	Issue (Decision Required)
IR1: (Technical) executive decisions, requirements analysis decisions	n/a	ARCHITECTURAL STYLE LAYERING LANGUAGE AND PLATFORM PREFERENCES TOOLING DIRECTIONS FUNCTIONAL REQUIREMENTS NOTATION BPM NOTATION
IR2 and IR3: Pattern Selection Decisions (PSDs), Pattern Adoption Decisions (PADs)	Atomic service layer	IN MESSAGE GRANULARITY OUT MESSAGE GRANULARITY OPERATION-TO-SERVICE GROUPING MESSAGE EXCHANGE PATTERN INVOCATION TRANSACTIONALITY PATTERN SERVICE PROVIDER TRANSACTIONALITY (ST)
	Integration layer	INTEGRATION PARADIGM COMMUNICATIONS TRANSACTIONALITY (CT)
	Service composition layer	SERVICE COMPOSITION PARADIGM PROCESS LIFETIME SESSION MANAGEMENT RESOURCE PROTECTION STRATEGY PROCESS ACTIVITY TRANSACTIONALITY (PAT)
IR4 and IR5: Technology Selection Decisions (TSDs), Technology Profiling Decisions (TPDs)	Atomic service layer	TRANSPORT PROTOCOL BINDING MESSAGE EXCHANGE FORMAT SOAP COMMUNICATION STYLE WEB SERVICES API JAVA SERVICE PROVIDER TYPE XML SCHEMA (XSD) CONSTRUCTS
	Integration layer	INTEGRATION TECHNOLOGY TRANSPORT QOS
	Service composition layer	WORKFLOW LANGUAGE BPEL VERSION COMPENSATION TECHNOLOGY
IR6 and IR7: Vendor Asset Selection Decisions (ASDs), Vendor Asset Configuration Decisions (ACDs)	Atomic service layer	SOAP ENGINE
	Integration layer	ESB PRODUCT ESB TOPOLOGY (IBM DATAPOWER CONFIGURATION)
	Service composition layer	BPEL ENGINE INVOKE ACTIVITY TRANSACTIONALITY

Issues in physical viewpoint. The examples presented so far dealt with the logical viewpoint introduced in Chapter 2. However, many issues pertain to the physical viewpoint. Such issues reside on all levels of refinement shown in Figure 18. For example, several PADs and PSDs are required to create a *conceptual* operational model, e.g., about clustering or a certain network topology. Follow on TSDs and TPDs are required to create a *technology* operational model, for instance selecting a certain data replication mechanism supporting backup or failover concepts appearing on the conceptual level. Even more detailed ASDs and ACDs are required to create a *vendor asset* operational model, e.g., concerning the proprietary system management scripts required to deploy the selected backup or failover technology, the installation of heartbeat and takeover protocols, and the configuration of servers and network equipment. Further details regarding decisions pertaining to the physical viewpoint are out of scope here, but present in the full RADM for SOA.

This concludes the RADM for SOA overview. We will return to some of these issues in Chapter 6, and tailor this RADM into a project ADM in Chapter 7.

5.3 Discussion and Summary

In this section we introduced SOAD step 1, which deals with RADM scoping in the asset creation phase. We introduced and demonstrated a technique leveraging identification rules and a meta issue catalog to define the boundaries of a RADM.

Justification. We propose a human-centric technique for decision identification, rather than an algorithm than can be implemented in a tool. This is adequate given the current state of the art and the practice. For further automation, it would be required to capture expert knowledge in machine-readable form and apply data mining techniques. This appears to be too ambitious, requiring strong assumptions regarding the formalization of input models and a highly stable application genre.

Our decision identification approach is pattern-centric: Principles and patterns such as those defined in Chapter 2 serve as anchor points for the RADM scoping. They provide conceptual alternatives in the RADM for SOA. Leveraging knowledge already captured in pattern form as conceptual alternatives is a key advantage of SOAD; it saves the knowledge engineer much documentation effort. Our technique can be applied even if patterns are not available yet: Logical components and connectors used on previous projects can be studied instead. The created RADM then serves as an intermediate step during the pattern harvesting.

As we could observe in one of the case studies presented in Chapter 9 (action research), the technique increases the productivity of the knowledge engineer.

Assumptions. A key assumption of SOAD is that the architectural decisions required during design (which we call issues) *recur*.[34] The feedback obtained during the validating industry case studies (discussed in detail in Chapter 9) indicates that

[34] The decision outcome (actual decision made and its justification) has reuse potential as well, but not as much as the background information. It is valuable knowledge, though.

this assumption is rather strong, but valid for SOA. The RADM for SOA also makes evident that the assumption holds: We have identified 35 issues in this chapter (see Table 15); the full RADM for SOA models 389 recurring issues.

Consequences. The issue names create a language for a problem domain, just like pattern names create one for a solution domain.

Our identification rules and meta issues leave many modeling choices to the knowledge engineer; this is deliberate. It is possible to combine or remove issues, e.g., when a pattern itself already resolves a meta issue or when the related knowledge can not be made reusable.

The presented top-down identification technique must be complemented with a bottom up knowledge *harvesting* method to ensure continuous content contributions from industry projects. This method must provide a process, criteria whether a decision qualifies for inclusion in a RADM, and decision modeling guidance. Such process, criteria, and guidance are informally described in Appendix A.

Next steps. The issue catalog produced in this step does not give any advice how to document and use the issues; so far, we have only named them and touched upon alternatives and dependencies in anecdotal form. In the following steps, we present how to model, structure, order, and use issues once they have been identified.

Related publications

We discuss the complementary and synergetic relationship between patterns and decision models in detail in [ZZG+08].

A RADM for SOA overview is also given in [ZKL+09].

6 Populating Reusable Architectural Decision Models

In this chapter, we present our concepts for SOAD steps 2 to 4, which are conducted during the asset creation phase: We introduce the SOAD metamodel supporting reuse and collaboration (Section 6.1), structure decision models statically with refinement levels, topic group trees and logical dependency relations (Section 6.2), and add a temporal decision order (Section 6.3).

6.1 Framework Step 2: Model Individual Decisions

A metamodel for architectural decision capturing and sharing is required for step 2 in the SOAD framework. Such metamodel solves the following problem:

Which information to model for each architectural decision required (issue)?

Once an issue has been identified as recurring, it has to be described and positioned in the RADM asset to be populated. This section deals with describing single issues; Section 6.2 will then cover issue positioning in the RADM. The input for this step is a linear list enumerating identified issues (issue catalog). Its output is an issue catalog containing elaborate descriptions of issues and alternatives.

The section starts with a brief review of the state of the art and the practice and then progresses to presentation of solution, application to SOA, and discussion.

6.1.1 State of the Art and the Practice

State of the art. As explained in Chapter 2, many templates and metamodels for decision capturing exist [Bre, DFL+07, JB05, TA05]. A decision log is a key artifact in many industrial methods, e.g., UMF [CCS07] (“architectural decisions”).

State of the practice. Many inhibitors for retrospective decision capturing exist, e.g., lack of time, immediate benefit, and tools [TAG+05]. Architectural decisions typically are captured in text documents; e.g., the UMF artifact description suggests a table format. Capturing dependencies and organizing decisions in this form is manual, time consuming work. The alignment with other artifacts is cumbersome. Scalability and collaboration challenges can be observed on larger projects: A large text document with many cross references is difficult to maintain manu-

ally [SZP07]. As a consequence, decisions are often captured in rudimentary form (e.g., as a spreadsheet or bullet list) or as part of other artifacts (e.g., as an appendix of a document describing the architecture from a logical viewpoint or in a project team wiki). They may even remain tacit or vaporize over time [Jan08].[35]

6.1.2 Concepts: Metamodel Extensions for Reuse and Collaboration

To overcome the inhibitors, we define a *metamodel* that extends existing templates for knowledge capturing to support active usage of decision models during design. We first introduce an informal template and then specify the metamodel precisely.

Architectural Decision (AD) template. We build on existing templates to describe issues (see Definition 4.4), outcomes (Definition 4.3), and supporting information. To satisfy the needs of our extended usage context, we add attributes as indicated in Figure 19. Our template is structured into *decision investigation*, *decision making*, and *decision enforcement* sections:

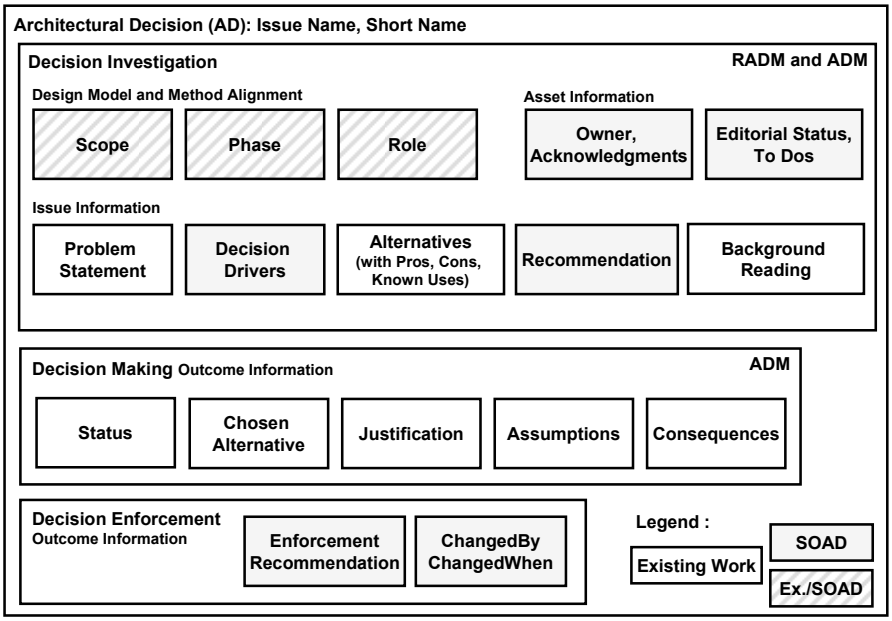


Figure 19. Architectural decision capturing template with SOAD extensions

Decision investigation. We model an issue *name*, e.g., SESSION MANAGEMENT. This name is the only information that is available after step 1; all other information is added during RADM population (steps 2 to 4). There is an abbreviating *short name* as well, e.g., "Scl-01". SCL is the abbreviation of Service Composi-

[35] The assessment is subjective, drawing on input from practicing architects and personal experience (see Chapter 9 and 10). It is supported by the findings in [DFL+07, TAG+05].

tion Layer, one of the architectural layers introduced in Definition 2.8 (in Chapter 2) and used to structure the RADM for SOA introduced in Chapter 5.

The *scope* links an issue to types of design model elements such as "business process", a component name used in the SOA definitions in Chapter 2. Method alignment is achieved via project *phase* ("macro design") and *role* ("application architect") information, which comes from the software engineering method adopted. These attributes are particularly useful in step 5 (tailor model) when filtering the RADM: It is possible to select only issues relevant in a particular context. Asset information such as *owner*, *acknowledgments*, *editorial status*, and *to dos* captures information about the origin and the maturity of the knowledge.

The *problem statement* motivates the design issue, often as a question (e.g., "How to correlate incoming user requests and server-side session objects?"). *Decision drivers* convey information about the factors that influence the decision making; the pattern community uses the term forces synonymously [ZZG+08]. Decision drivers may include genre-specific NFRs such as the user, process and resource integrity, integration, and semantics challenges from Section 2.1.2 in Chapter 2, but also general software quality attributes [ISO01] and environmental issues such as project budget, license costs, development efforts, and team skills (e.g., "size and amount of enterprise resource data to be exchanged, scalability needs from a service provider perspective"). We provided more examples in Chapter 4; a genre-specific decision driver categorization appears in Appendix A.

The *alternatives* element in the template lists available design options ("CLIENT STATE PATTERN") with their pros, cons, and known uses. Subjective information is conveyed in the *recommendation*, which depending on the decision type can be a simple rule of thumb ("avoid client-side state if the state information is large"), a weighted mapping of forces to alternatives, or a pointer to a more complex analysis process to be performed outside the decision model.[36] The recommendation should refer to decision drivers and pros and cons of alternatives. With the *background reading* attribute, supporting material such as primers and tutorials can be referenced ("Fowler [Fow03] describes issue and alternatives in detail").

Decision making. A *status* attribute captures the current state of processing (step 6); its values can come from existing ontologies such as that in [KLV06]. The other decision outcome attributes *chosen alternative*, *justification*, *assumptions*, and *consequences* are adopted from an existing capturing tool [ABK+06].

Decision enforcement. A decision *enforcement recommendation* for step 7 can be stated, informing the architect about suggested ways to educate developers and other project stakeholders about a decision made. Examples are "coaching", "architectural templates (code snippets)", and "code generation". Attributes such as *changedBy* and *changedWhen* convey decision authoring history and lifecycle management information to support collaboration.

We now specify the information in the template in a UML class diagram, adding several attributes and basic decision dependency information.

[36] A SOA design example is: "Follow the WS-I basic profile, which endorses the `document/literal` SOAP COMMUNICATION STYLE and bans `rpc/encoded`" [WSI06].

UML metamodel. Already present in the template, *ADIssue*, *ADAlternative*, and *ADOutcome* are the core entities in the SOAD metamodel. It is shown in Figure 20; each entity is represented by a UML class. Alternatives are represented as a separate ADAlternative class now, which has a physical containment relation with ADIssue (labeled *isSolvedBy*) Decision dependencies are explicitly modeled as associations between ADIssues. We introduce a single *dependsOn* association here; in Sections 6.2 and 6.3, we refine this link and define several different dependency relations both on the ADIssue and the ADAlternative level.

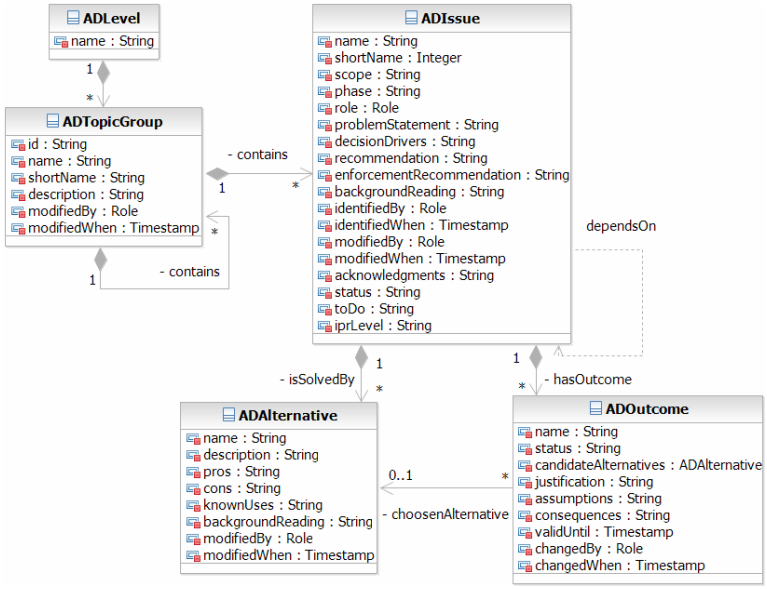


Figure 20. SOAD metamodel as UML class diagram (adapted from [ZKL+09])

Two structuring constructs appear in the metamodel: *ADLevel* and *ADTopicGroup* allow knowledge engineers to group closely related ADIssues and define topic group hierarchies. We discuss such model structuring in Section 6.2.

To facilitate reuse and distinguish issues and outcomes (as specified by Definitions 4.3 and 4.4), we separate ADOutcome information from ADIssue and ADAlternative knowledge. The rationale behind this modeling choice is that the same issue might pertain to multiple elements in a design model: Types of components and connectors are referenced via the *scope* attribute in the ADIssue. Multiple ADOutcome instances can be created, and refer to actual design model elements via their *name*. In SOA design, an order management process model might state that three business processes have to be implemented as a set of composed Web services; while the SESSION MANAGEMENT issue has to be resolved for all three processes, the chosen alternative might differ per process [ZKL+09].

ADIssue and ADAlternative instances appear in SOAD RADMs. ADOutcome instances are added to an ADM during RADM tailoring (step 5) and decision making (step 6). If issues recur, only the outcome has to be documented on each project (including its justification); the detailed issue description, for instance

pointers to pattern descriptions, is already present in the tailored RADM. The issue description can be modified in the ADM, e.g., if an alternative is chosen that is not defined or not described properly in the RADM. Issues can be added as well.

Let us now investigate a comprehensive ADIssue and ADAlternative example.

6.1.3 Sample Application to SOA: Invocation Transactionality Pattern

In this section, we instantiate the SOAD metamodel and capture a complex SOA design issue, the design of system transaction boundaries in *process-centric SOA* [ZHD07]. This issue and its alternatives were first presented in [ZGT+07].

We call this issue INVOCATION TRANSACTIONALITY PATTERN. It can be identified with IR3 from Chapter 5, combining the service composition pattern with the meta issue called "system transactionality" (see Table 14 on page 75). The issue appears in the RADM for SOA introduced in Section 5.2 because it meets the definition of an architectural decision from Chapter 1 and it recurs multiple times in each business process supported by an SOA. Figure 21 shows an excerpt from the issue description in the RADM for SOA (see Appendix B for full description):

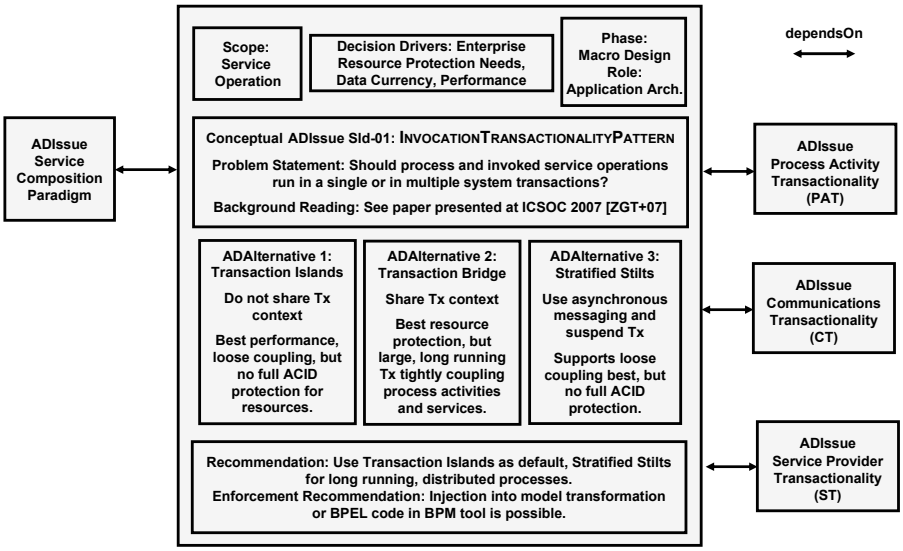


Figure 21. QOC+ diagram for INVOCATION TRANSACTIONALITY PATTERN

The notation introduced in Figure 21 is our variant of *Question, Option, Criteria (QOC)* diagrams [MYB+91]. QOC diagrams raise a design question which points to the available options for it; decision criteria are associated with the options. Option selection can lead to follow-on questions. QOC diagrams are an effective way of displaying design issues and possible solutions. QOC diagrams align well with SOAD: The questions can be found in the problem statements of ADIssues, the options correspond to ADAlternatives, and the criteria are the decision drivers. Unlike standard QOC diagrams, we do not link criteria to options

graphically. However, our QOC variant adds recommendations, as well as the scope, phase, and role attributes from the SOAD metamodel from Section 6.1.2.

The scope, phase, and role attributes express which design model element the ADIssue pertains to, when it should be made, and who is responsible: The scope attribute of the INVOCATION TRANSACTIONALITY PATTERN issue is set to "service operation", a term used by the service consumer-provider-contract pattern (Definition 2.6). This informs the architect that the decision must be made for each of the operations defined in a service contract and implemented by a service provider. It is typically taken in "macro design" phase, and the "application architect" is responsible. These terms originate from the software engineering method adopted.

Having been identified with IR3, the issue is classified to reside on the conceptual level of the RADM for SOA, as it deals with patterns, and not with technology- or product-specific design aspects. The problem statement is given in question form. It refers to terms from the SOA definitions in Chapter 2 to ensure that it is understandable. For architects who are not familiar with the problem and with possible solutions, a technical paper is referred to under background reading.

Figure 21 lists "enterprise resource protection needs, data currency, performance" as decision drivers [Fow03]. These decision drivers are related to the process and resource integrity challenge from Chapter 2.

One incoming and three outgoing dependencies with other issues are defined: This issue becomes relevant once WORKFLOW is selected as SERVICE COMPOSITION PARADIGM. We investigate the three depending issues on the right shortly.

A recommendation is also given. It is weak here due to the complexity inherent to this particular design issue: There is no single, one-size-fits-all solution to it.

Architectural patterns as alternatives of conceptual decisions. Figure 21 already listed three architectural patterns as ADAlternatives. Figure 22 illustrates these patterns on a platform-independent, conceptual level:

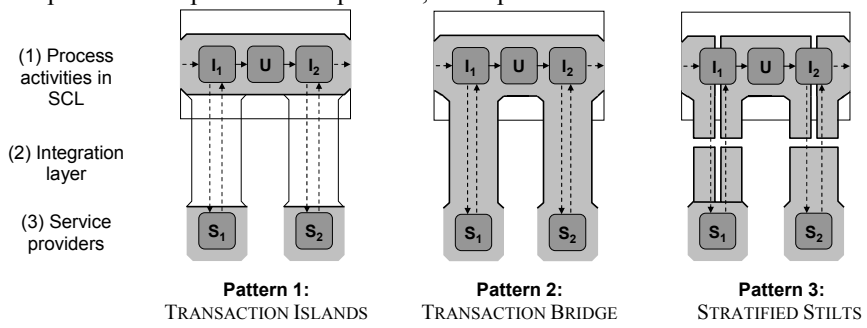


Figure 22. INVOCATION TRANSACTIONALITY PATTERN alternatives [ZGT+07]

TRANSACTION ISLANDS, TRANSACTION BRIDGE, and STRATIFIED STILTS are commonly used to address enterprise resource integrity requirements. To minimize the RADM creation effort, a RADM can reference already established patterns such as SOA patterns [HZ06], messaging patterns [HW04], and general architectural patterns [BMR+96]. In this case, we defined the patterns in [ZGT+07].

The Service Composition Layer (SCL) from the SOA definitions in Chapter 2 is represented by the white boxes. It implements the tasks from an analysis-phase BPM as *process activities* that are part of executable workflows which run in the process manager that appears in the service composition pattern; here, two invoke activities I_1 and I_2 enclose a third activity U, which correspond to a BPEL assign activity [OAS07] or another utility on the technology level.[37] S_1 and S_2 represent *service providers* exposing operations. Service operation invocations are displayed as dotted lines. A contiguous light grey area represents a single *global transaction* [LR00], which may be extended if it is not enclosed by a solid black line.

In the remainder of this section, we present the three patterns in detail; RADM population and coverage of the SOAD metamodel continues in Section 6.2.

Pattern anatomy. As composite patterns, TRANSACTION ISLANDS, TRANSACTION BRIDGE, and STRATIFIED STILTS comprise three types of *primitives* [ZAH+08] corresponding to several architectural layers from Definition 2.8 (Chapter 2):

1. *Process Activity Transactionality (PAT)* primitives on the SCL.
2. *Communications Transactionality (CT)* primitives modeling the transaction sharing capabilities of the integration layer.
3. *Service provider Transactionality (ST)* primitives stating the capability of service providers to join a transaction. Service providers may reside in the atomic service layer and in the SCL (see Definition 2.8).

The primitives are conceptual, platform-independent abstractions of concepts found in BPEL [OAS07] and SCA [OSOA] technology, and can be viewed as design time statements of architectural intent. From a decision modeling standpoint, each primitive type offers multiple design options. This requires us to represent the primitive types as Pattern Adoption Decisions (PADs), shown in Figure 23:

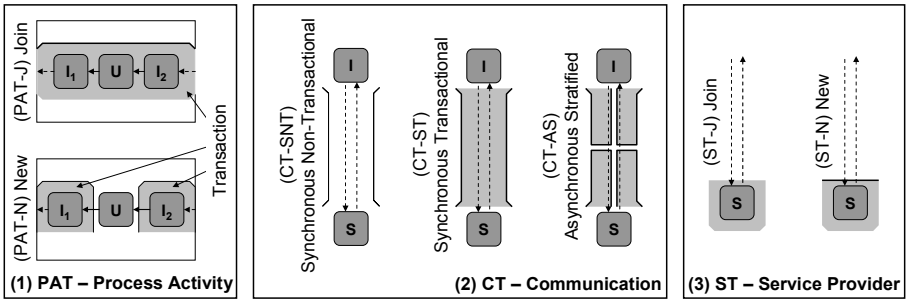


Figure 23. Pattern primitives in Pattern Adoption Decisions (PADs) [ZGT+07]

To elaborate upon the defining characteristics of the patterns and the primitives, we now present them in a format commonly used in the design patterns literature. This knowledge is paraphrased in the RADM (e.g., QOC diagram in Figure 21).

[37] An example of such process is the customer enquiry process in the motivating case study. In practice, the business activities from an analysis-phase BPM are not mapped to executable process activities directly; processes are often refactored during design.

Context. All patterns and primitives share common objectives: To protect enterprise resources (Definition 2.2) against integrity and correctness threats that may occur during concurrent process execution, e.g., when multiple processes and activities in the SCL invoke service operations via the ESB (Definitions 2.6 to 2.8).

Pattern 1. Decoupled TRANSACTION ISLANDS (PAT-J+CT-SNT+ST-N)

Problem. How to isolate SCL process activities from service operation execution?

Solution. Do not propagate the transaction context from the SCL to the service.

Forces and consequences. If a service operation fails, the process navigation in the SCL is not affected, and vice versa. If a service works with shared enterprise resources, its operations must be idempotent, as they may be executed more than once due to the transactional process navigation in the SCL. In many cases, the service provider must offer compensation operations, and higher-level coordination of the compensation activities is required (e.g., via business transactions [Fow03]; various models exist [LR00]). This pattern is often chosen as a default.

Pattern 2. Tightly coupled TRANSACTION BRIDGE (PAT-J+CT-ST+ST-J), with MULTIPLE BRIDGES variant (PAT-N+CT-ST+ST-J)

Problem. How to couple process activity execution in the SCL and service operation execution from a system transaction management perspective?

Solution. Configure process activities, communications infrastructure, and service providers so that the SCL transaction context is propagated to the service.

Forces and consequences. Process activities and the service operations invoked by them execute in the same transaction. As a result, several service operations can also participate in the same transaction. A natural limit for their response times exists ("tenths of seconds to seconds at most" [LR00]). If an operation-internal processing error occurs, previous transactional work, which can include process navigation in the SCL and the invocation of other operations, has to be rolled back.

This pattern meets resource protection needs well on the system level, but often is not applicable, e.g., when processes and operations run for days or months. A common variation of this pattern is to split a process up into several *atomic spheres*, creating MULTIPLE BRIDGES for selected process activity/service operation pairs. Executing the process activities in a small number of transactions (TRANSACTION BRIDGE) reduces the computational overhead for process navigation; splitting the process up into several atomic spheres (MULTIPLE BRIDGES) increases data currency (which is a decision driver appearing in Figure 21).

Pattern 3. Loosely coupled STRATIFIED STILTS (PAT-J+CT-AS+ST-J)

Problem. How to realize asynchronous, queued transaction processing in SOA?

Solution. Use message queuing [HW04] on the integration layer (ESB). I_1 and S_1 use *stratified transactions* during service invocation; unlike S_1, service S_2 reads the request message and sends the response message within a single transaction.

Forces and consequences. Services do not have to respond immediately; the delivery of the messages is guaranteed by the integration layer (ESB). If the execution of the service operation fails, the process may not get an immediate response; additional error handling is required, often involving compensation logic. This pattern often is the only choice when integrating legacy systems.

PAT primitives. The Process Activity Transactionality (PAT) issue defines two SCL alternatives, transaction context sharing or *Join (J),* and transaction context separation or *New (N)*. If PAT-J is chosen, a process activity executes in the same transaction context as the adjacent activities in the same process; it *joins* an existing context. As a consequence, the process activity's work might be rolled back if any other process activity or service operation that participates in the same transaction fails. With PAT-N, a process activity is executed in a *new* transaction context. PAT-J is a valid choice in all three INVOCATION TRANSACTIONALITY PATTERN alternatives and shown in Figure 22. In TRANSACTION BRIDGE, PAT-N models the MULTIPLE BRIDGES variant. PAT-N is justified if two process activities should be isolated from each other from a business requirement point of view.

CT primitives. We model the Communications Transactionality (CT) issue with alternatives *Synchronous Non-Transactional* (CT-SNT), *Synchronous Transactional* (CT-ST), and *Asynchronous Stratified* (CT-AS). These primitives deal with system transactions on the integration layer. CT-SNT is forced by the TRANSACTION ISLANDS pattern. It represents a synchronous service invocation from the process activity without propagation of the transaction context. As a consequence, the activity waits until the call to the service returns. Until then, the work conducted by the service can not be influenced. For example, the CT-SNT service invocation may cause the transaction to exceed its maximum duration which may result in a transaction timeout and a subsequent rollback. With CT-SNT, undoing the work of the service can not be included in this rollback.

CT-ST is forced by TRANSACTION BRIDGE. It models a synchronous service invocation with transactional context propagation. As a consequence, the process activity waits until the call to the service returns. A rollback may occur after the service execution has completed; the service participates in the SCL transaction.

CT-AS is forced by the STRATIFIED STILTS pattern. It represents an asynchronous service invocation without transaction context propagation. In CT-AS, long-running services can be invoked without loosing transactional behavior, as the process navigation is part of a *stratified transaction* [LR00]. At least three transactions are involved in the invocation of a long-running service: the request message is sent in a first transaction; in a second transaction, the message is received by the service provider and the response message is sent; in a third transaction, the process activity receives the response from the service. Depending on the service implementation, the second transaction (provider side) may be split up into several transactions, e.g., receive the message and commit, and later on, send the response in a new transaction. Such stratification details are described further in [LR00].

ST primitives. Two alternatives exist for the Service Provider Transactionality (ST) issue: *join* an incoming transaction (ST-J) or create a *new* one (ST-N). ST-J

is forced by TRANSACTION BRIDGE, ST-N by TRANSACTION ISLANDS. In ST-J, the service provider participates in the transaction of the caller (if a transaction exists). As a consequence, process activity execution in the SCL and the invoked service operation influence each other, e.g., when causing a rollback. In ST-N, the service provider does not participate in the incoming transaction. As a consequence, if the transaction in which the process activity runs is rolled back and the activity is retried later (e.g., due to process engine-specific error handling procedures), the service may operate on enterprise resources that have been modified in the meantime.

This completes coverage of the INVOCATION TRANSACTIONALITY PATTERN issue and depending PADs as illustration of SOAD step 1.

6.1.4 Discussion and Summary

In support of step 2, we introduced the SOAD metamodel for capturing individual issues and presented an SOA example demonstrating the relation with patterns.

Justification. Our metamodel draws on our own decision capturing experience [ZGK+07], existing assets [ABK+06], and the literature [Jan08, TA05]. It also takes inspiration from pattern templates [GHJ+95, BMR+96]. The standardization of the decision capturing template simplifies both asset creation and asset consumption. Only one template has to be learned; guidelines how to use the attributes can be established (e.g., regarding value ranges and semantics of content). Standardization also accelerates the knowledge exchange between architects.

Assumptions. We assume that attribute names and formats can be agreed upon. There is a conflict between flexibility and extensibility on one side and standardization and exchangeability on the other side. The latter two requirements have higher priority for us, as reuse is a design goal and key framework concept. The validation results show that practicing architects consider information such as problem statement, decision drivers, and pros and cons of alternatives useful; the attribute names and formats can indeed be agreed upon (see Chapter 9). Other architectural knowledge management work draws different conclusions [DFL+07].

Consequences. Creating a RADM and describing the issues according to the SOAD metamodel causes knowledge engineering efforts. Hence, a decision to create a fully documented RADM must be in line with the knowledge management strategy in place, e.g., codification as opposed to personalization [Jan08].

Next steps. As a next step of RADM population in the asset creation phase, we introduce refinement levels and architectural layers to structure decision models.

Related publications

An earlier version of the SOAD metamodel is described in [ZGK+07]; the version presented in this section is also featured in [ZSE08] and [ZKL+09].

The decisions and patterns in this section first appeared in [ZGT+07].

6.2 Framework Step 3: Structure Model

With issues identified and modeled (SOAD steps 1 and 2), step 3 can be taken:

How to organize decision models in an intuitive, use case-driven way?

Let us assume that several hundred issues have been identified and modeled individually. An issue catalog organized as a linear list or table that has to be studied from beginning to end (as produced in steps 1 and 2) can not improve the decision making as desired. Hence, the output of this third asset creation step is a hierarchically structured model that is easier to navigate than a linear list.

The section structure is the same as that we used for steps 1 and 2, starting with a short review of the state of the art and the practice, then progressing to solution, application to SOA and motivating case study, and brief discussion of rationale.

6.2.1 State of the Art and the Practice

State of the art. In the architectural knowledge management community, the ontology proposed by Kruchten et al. [KLV06] defines three types of decisions: executive, existence, and property decisions (with subtypes such as ban decision). Booch is in the process of defining a pattern classification taxonomy as part of his software architecture handbook project [Boo]. Model Driven Architecture (MDA) distinguishes platform-independent from platform-specific models [OMG03]. Panes in enterprise architecture frameworks such as TOGAF [OG07] also structure architectural domains. However, we are not aware of any usage of these concepts in the context of structuring reusable architectural decision models.

State of the practice. RADMs for the enterprise application genre and SOA are not broadly available yet. Hence, basic organizing principles are used when capturing decisions in spreadsheets, word processing templates, and wiki tables. The resulting decision logs often are ordered chronologically and/or by topic areas only. This makes them easy to create, but hard to read and maintain. Lack of structure and resulting maintenance effort are among the many reasons why such decision logs often are not kept up to date until project end. This inhibits reuse.

6.2.2 Concepts: Multi-Level Decision Model and Logical Constraints

To solve the model structuring problem, we complement the UML model from Section 6.1 with formal definitions. Basic concepts from set and graph theory are adequate to define the entities in the UML model and the relations between them.

We begin with representations for the three UML model elements ADTopicGroup, ADIssue, and ADAlternative from Figure 20 on page 88:

Definition 6.1 (Architectural Decision Topic Groups *T*). *Let T be a set of architectural decision topic groups T = {(n, s, d) | n, s, d ∈ Strings} where the tuple (n, s, d) represents the name, short name, and description of an architectural decision topic group.*[38]

An architectural decision topic group (short: topic group, topic) represents closely related design concerns. For instance, in our RADM for SOA, one topic group per architectural layer is defined on each refinement level (see Figure 18 on page 77). An example is "Atomic Service Layer Decisions" corresponding to the atomic service layer from Definition 2.8. It is worth noting that our topic groups do not represent individual design issues, but group such issues. Representing individual design issues is the purpose of the next entity:

Definition 6.2 (Architectural Decision Issues *I*). *Let I be a set of architectural decision issues I = {(n, s, p, r, {tt}) | n, s, p, r, tt ∈ Strings} where n is a name, s a scope, p a project phase, r a role attribute, and {tt} a set of topic tag strings.*

An architectural decision issue (short: issue) represents a single design concern. Name, scope, phase, and role are describing texts. The name is used to identify and list issues. The topic tags index the model content. This information can be used to locate issues by subject area keyword. The architect can query the model for all issues dealing with "security", "transaction management", "workflow", and so on.

In our RADM for SOA, the issue MESSAGE EXCHANGE PATTERN deals with the abstract protocol syntax and synchrony of operation invocations. A second issue is INVOCATION TRANSACTIONALITY PATTERN, dealing with system transactions protecting enterprise resources from invalid concurrent access, e.g., lost updates and phantom reads (see Section 6.1.3). A third issue is IN MESSAGE GRANULARITY, which concerns the syntactical structure of the in message parameters.

An architectural decision issue captures a single design concern without modeling possible solutions to it. Architectural decision alternatives do so:

Definition 6.3 (Architectural Decision Alternatives *A*, Chosen). *Let A be a set of architectural decision alternatives A = {(n, s, chosen) | n, s ∈ Strings, chosen ∈ {`undefined`, `true`, `false`}} where n is a name, s is a solution description, and chosen is a marking that is* `undefined` *initially and becomes* `true` *when the alternative is chosen by the architect and* `false` *when the alternative is rejected.*

An architectural decision alternative (short: alternative) presents a single solution to a design issue. For instance, MESSAGE EXCHANGE PATTERN decides between synchronous REQUEST-REPLY and asynchronous ONE WAY alternatives. As presented in Section 6.1.3, two alternatives for INVOCATION TRANSACTIONALITY PATTERN are TRANSACTION ISLANDS and TRANSACTION BRIDGE.

Definition 6.4 (*contains* Relations $\prec_T$, $\prec_I$, $\prec_A$, $\prec$). *Let $\prec_T \subseteq T \times T$ be a* contains *relation defined between topic groups, $\prec_I \subseteq T \times I$ be a* contains *relation defined between topic groups and issues, and $\prec_A \subseteq I \times A$ be a* contains *relation defined be-*

[38] The other attributes from the UML model are irrelevant for the model structure.

tween issues and alternatives. Subsequently, we will only speak of the contains relation $\prec = \prec_T \cup \prec_I \cup \prec_A$.

The contains relation $\prec$ allows us to define a hierarchical structure. One or more architectural decision alternatives solve a particular design problem (expressed as an issue). Related issues can be grouped into topic groups. Related topic groups can be placed in the same parent topic group. Figure 24 illustrates the tree structure resulting from the $\prec$ relation:

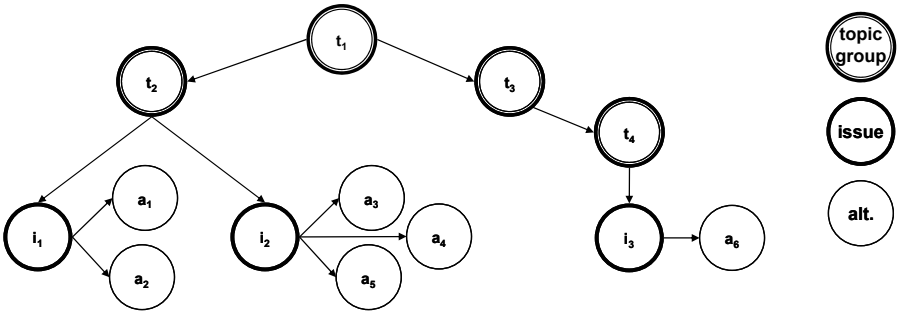


Figure 24. General organization of an architectural decision tree [ZKL+09]

In the UML metamodel in Section 6.1 (Figure 20 on page 88), the $\prec$ relation is represented by the three associations that express physical containment between ADTopicGroups, ADIssues and ADAlternatives, respectively (i.e., arrows filled with solid diamonds at originating end).

Definition 6.5 (Architectural Decision Tree $\mathbb{T}$, Root Topic). *Using T, I, A, and the $\prec$ relation, we can define an architectural decision tree $\mathbb{T} = (T \cup I \cup A, \prec)$ with a single root node $t_0 \in T$ called the* root topic. *In $\mathbb{T}$, a topic group contains zero or more other topic groups and issues, while an issue may contain zero or more alternatives. In this tree, each topic group $t \in T$ except the root topic is contained in exactly one other topic group $t_i \in T$:*

$$\forall\ t, t_i, t_j \in T: (t_i \prec t) \wedge (t_j \prec t) \Rightarrow t_i = t_j$$

Each issue $i \in I$ must be contained in exactly one topic group $t \in T$:

$$\forall\ i \in I\ \ \exists\ t \in T: (t \prec i)$$
$$\forall\ i \in I, t_i, t_j \in T: (t_i \prec i) \wedge (t_j \prec i) \Rightarrow t_i = t_j$$

Each alternative $a \in A$ must be contained in exactly one issue $i \in I$:

$$\forall\ a \in A\ \ \exists\ i \in I: (i \prec a)$$
$$\forall\ \ i_i, i_j \in I, a \in A: (i_i \prec a) \wedge (i_j \prec a) \Rightarrow i_i = i_j$$

Figure 25 instantiates the abstract tree structure for parts of our SOA example:

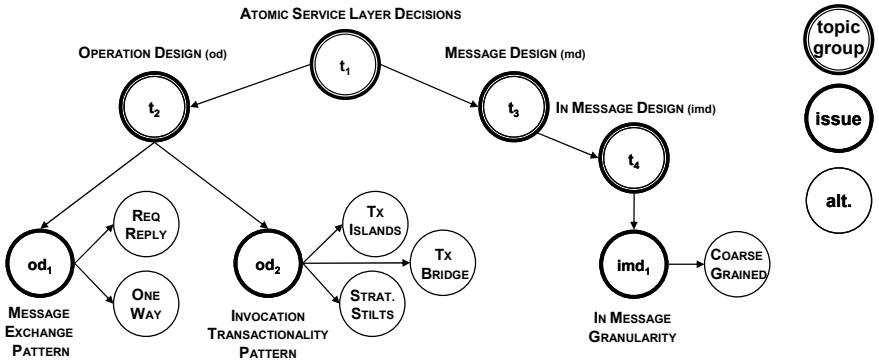


Figure 25. An instantiated example tree (RADM for SOA excerpt) [ZKL+09]

Definition 6.6 (Ordered Architectural Decision Tree $\mathbb{T}$). *We define an* ordering *among the child nodes of identical type (topic group, issue, alternative) contained in a node in order to be able to enumerate sibling nodes of the same type sharing one parent node, i.e., we introduce* $<_T$, $<_I$, $<_A$.

An ordering relation defines a recommended reading and decision making sequence, and can be used to express integrity constraints on architectural decision trees (which we will define later). In the simplest case, the $<_T$, $<_I$, and $<_A$ relations can be the alphanumeric sorting of the topic group, issue, and alternative names. Note that a topic group may contain other topic groups and issues. In this case, we order all topic group siblings before all issue siblings. This yields an ordered tree $\mathbb{T}$; we refer to its total order relation as $<$.

The elementary definitions allow knowledge engineers to capture decisions and organize the knowledge in a topic group hierarchy. However, the resulting ordered architectural decision tree does not yet support the envisioned design method usage of architectural decision models, in which a managed issue list takes an active, guiding role. More relations between topic groups, issues, and alternatives must be defined.[39] We now introduce multi-level models and logical constraints.

Definition 6.7 (Architectural Decision Model $\mathbb{M}$, Root Topic, Initial Issue). *An* architectural decision model $\mathbb{M}$ *is a partially ordered set of architectural decision trees* $\mathbb{T}_{00}, \ldots, \mathbb{T}_{10}, \ldots, \mathbb{T}_{km}$ *arranged in levels* $L_0, \ldots, L_k$. *Each tree belongs to exactly one level and each level must contain at least one tree, i.e., no empty levels exist. A tree* $\mathbb{T}_{ki}$ *is the i-th tree in level k. If k < l, we speak of tree* $\mathbb{T}_{ki}$ *having a higher level than tree* $\mathbb{T}_{lj}$ *and* $\mathbb{T}_{lj}$ *having a lower level than* $\mathbb{T}_{ki}$. *Each architectural decision model* $\mathbb{M}$ *has exactly one distinguished* root topic*, which is the root topic of the initial tree* $\mathbb{T}_{00}$ *in the highest level* L_0. *Accordingly, the first issue in the distinguished root topic (according to* $<_I$*) is identified as the* initial issue*.*

Architectural decision models define the multi-level structure required for decision models such as the RADM for SOA introduced in Chapter 5. The partial order assigns topic groups and issues to different levels of abstraction and refine-

[39] Note that the UML model in Section 6.1 only defined a generic "dependsOn" association.

ment. For example, as motivated in Chapter 5, a *conceptual level* issue about an INTEGRATION PARADIGM can be identified by Identification Rule (IR) 2: Should the services be integrated via SOA MESSAGING (Definitions 2.6 and 2.7), REMOTE PROCEDURE CALLS (RPC), FILE TRANSFER, or a SHARED DATABASE [HW04]? A related *technology level* issue is to agree on the TRANSPORT PROTOCOL BINDING such as SOAP OVER HTTP. Finally, a SOAP ENGINE asset can be selected on the *vendor asset level*, e.g., APACHE AXIS2 or IBM WEBSPHERE.

We now define several additional relations. They formally capture how issues residing in different levels and trees of a model $\mathbb{M}$ can be combined in order to express that an abstract, conceptual design is elaborated upon on the same or on a lower, more concrete level of design refinement.

Definition 6.8 (*refinedBy, decomposesInto, influences* Relations). *Let $\mathbb{M}$ be an architectural decision model with levels $L_0, \ldots, L_k$ and trees $\mathbb{T}_{00}, \ldots, \mathbb{T}_{km}$ belonging to levels $L_0, \ldots, L_k$. The following relations are defined between issues $i_{00\,0}, \ldots, i_{km\,n}$ where an issue $i_{km\,n}$ is the n-th issue in the m-th tree $\mathbb{T}_{km}$ contained within level L_k of a model $\mathbb{M}$.*

- influences($i_{jl\,n}$, $i_{km\,o}$) *with j, k, l, m, n, o arbitrary. The* influences *relation captures cross-cutting concerns between issues. It adds additional undirected edges to the model that must not necessarily form a connected graph. The relation is symmetric, i.e., if i_i influences i_j, then i_j influences i_i. In addition, the* influences *relation is not reflexive, but transitive. An issue can influence several other issues and it can also be influenced by several other issues.*
- refinedBy($i_{jl\,n}$, $i_{km\,o}$) *with $j < k$ and l, m, n, o arbitrary. The* refinedBy *relation links issues that have to be investigated at several levels. It adds additional directed edges to the model that must always lead from an issue in a higher level to an issue in a lower level of the model, i.e., no cycles can occur. The relation is transitive, but neither reflexive nor symmetric. If $k = j + 1$, i.e., the refinement of an issue is contained within the next lower level, we speak of a* strict refinedBy *relation. Issues in the highest level L_0 can not refine any other issue, while an issue in the lowest level L_k can not be refined by any issue. If (i_1* refinedBy *i_2), i_1 is also referred to as having an outgoing refinement relation and i_2 is also referred to as having an incoming refinement relation.*
- decomposesInto($i_{jl\,n}$, $i_{km\,o}$) *with $j = k$ and l, m, n, o arbitrary. The* decomposesInto *relation expresses functional aggregation. It adds additional directed edges between issues within the same level. The relation is transitive, but neither reflexive nor symmetric. No cycles are permitted.*

The *influences* relation can be used to express cross-cutting concerns without making any assumptions about the level ($\prec$) and order ($<$) of the related issues. For instance, the choice of a WORKFLOW LANGUAGE also has to do with the WEB SERVICES API, but the relation type is neither refinement (the two issues belong to the same refinement level, the technology level) nor decomposition because different design model elements are affected (workflow and service consumer). The

influences relation is often used in rapid decision capturing efforts and replaced by a more elaborate form such as *refinedBy* and *decomposesInto* as the decision model matures during subsequent knowledge engineering iterations.

The *refinedBy* relation allows us to model that the same design issue typically has to be investigated at several stages of the software engineering process. A level can correspond to a Model Driven Architecture (MDA) model type such as platform-independent model and platform-specific model [OMG03], to a development milestone, e.g., an elaboration point defined in RUP [Kru03], or to a TOGAF pane [OG07]. A conceptual pattern such as SERVICE COMPOSITION PARADIGM abstracts away from any particular technology. Consequently, a WORKFLOW LANGUAGE like BPEL has to be selected in refinement of the conceptual decision to adopt the WORKFLOW pattern. Next, a particular BPEL ENGINE vendor asset has to be selected if BPEL is the selected WORKFLOW LANGUAGE.

The *decomposesInto* relation expresses functional aggregation of issues. When following the separation of concerns principle, complex design problems are often broken down into to smaller, more manageable units of design work. These units can then be investigated independently of each other. The decomposition of the transaction management patterns into layer-specific primitives in the Section 6.1.3 was an example of such an approach.

Table 16 summarizes the main properties of the relations.

Table 16. Logical relations between architectural decision issues

Relation	Set(s)	Reflexive/ Symmetric/ Transitive	Cardinality	Other Properties
influences	I × I	no/yes/yes	n:m	–
refinedBy	I × I	no/no/yes	0..1:0..1	Introduces one or more additional Directed Acyclic Graphs (DAGs), i.e., no cycles permitted; only from higher to lower level (next lower if strict)
decomposesInto	I × I	no/no/yes	0..1:n	No cycles permitted. Only within the same level.

With these relations introduced, we can define two logical constraints on architectural decision models $\mathbb{M}$.

Integrity Constraint 1. *The* refinedBy *and* decomposesInto *relations are mutually exclusive.*

$$\forall i_i, i_j: i_i \, \textit{refinedBy} \, i_j \Rightarrow \neg (i_i \, \textit{decomposesInto} \, i_j)$$
$$\text{and } \forall i_i, i_j: i_i \, \textit{decomposesInto} \, i_j \Rightarrow \neg (i_i \, \textit{refinedBy} \, i_j)$$

This follows from our basic definitions, because the *refinedBy* relation is defined between issues residing on different levels, while the *decomposesInto* relation is only defined between issues on the same level.

Integrity Constraint 2. *If two issues are related via* refinedBy *or* decomposesInto *relations, they can not be related via an* influences *relation and vice versa.*

$$\forall\ i_i, i_j\colon i_i\ \textit{refinedBy}\ i_j \vee i_i\ \textit{decomposesInto}\ i_j \Rightarrow \neg\,(i_i\ \textit{influences}\ i_j)$$
$$\forall\ i_i, i_j\colon i_i\ \textit{influences}\ i_j \Rightarrow \neg\,(i_i\ \textit{refinedBy}\ i_j \vee i_i\ \textit{decomposesInto}\ i_j)$$

Figure 26 adds the three levels we introduced in Figure 18 on page 77 (Section 5.2) to our example, the design of transactional workflows in SOA. The patterns and primitives are a subset of those shown in Figure 22 (page 90) and Figure 23 (page 91) in Section 6.1.3, now represented as issues that appear in an architectural decision model. The topic group hierarchy is now shown: three SOA layers, the atomic services layer, the service composition layer, and the integration layer, are represented by separate topic groups. As explained in Section 6.1.3, INVOCATION TRANSACTIONALITY PATTERN (ITP) is an example for the decomposition of a complex conceptual issue into two more primitive ones residing on the same level (here: conceptual). The transactionality of a service operation is a non-functional design concern. It affects design model elements in the atomic services, service composition, and integration layers; therefore, ITP has *decomposesInto* relations with issues in topic groups for two other SOA layers, PROCESS ACTIVITY TRANSACTIONALITY (PAT) and COMMUNICATIONS TRANSACTIONALITY (CT). PAT is an issue that pertains to the service composition layer, CT to the integration layer. Note that SERVICE PROVIDER TRANSACTIONALITY (ST) (also from Section 6.1.3) is not shown in the interest of readability.

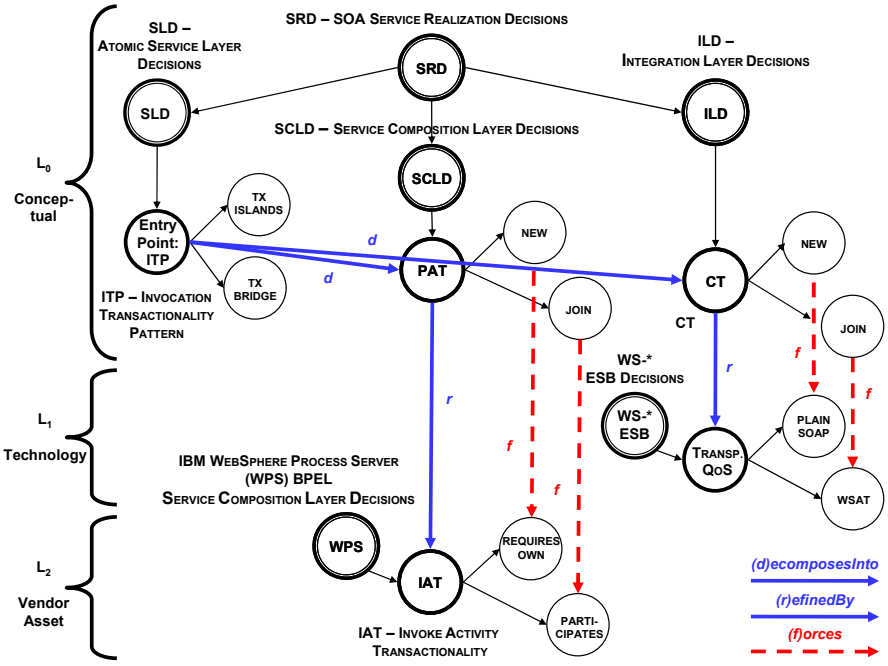


Figure 26. Architectural decision model with logical relations [ZKL+09]

Furthermore, there are two examples of *refinedBy* relations: A strict one runs from the conceptual to the technology level (outgoing issue: CT, incoming issue:

TRANSPORT QOS). Another one goes from the conceptual to the vendor asset level: The outgoing issue is PAT, the incoming is INVOKE ACTIVITY TRANSACTIONALITY (IAT).[40]

Figure 26 also introduces a new type of relation, *forces*, expressing that certain alternatives for the conceptual issues PAT and CT mandate the alternatives for the refining issues on lower levels. This is one of three relations to be defined next, formally capturing the relationships that may exist between alternatives.

Definition 6.9 (*forces, isIncompatibleWith, isCompatibleWith* Relations). *Let* $\mathbb{M}$ *be an architectural decision model. Let* a_i, a_k *be architectural decision alternatives within* $\mathbb{M}$*. Several relations can be defined between alternatives within the same or across different levels and trees of* $\mathbb{M}$*:*

- forces(a_i, a_k) *with* $i \neq k$ *and* $i_i \prec a_i$, $i_k \prec a_k$ *implies* $i_i \neq i_k$*. The* forces *relation expresses that selecting an alternative* a_i *in one issue necessarily means to select an alternative* a_k *in another issue. It adds additional directed edges between alternatives. It is not reflexive and not symmetric, but transitive. The relation must not form any cycles.*
- isIncompatibleWith(a_i, a_k) *with* $i \neq k$*. The* isIncompatibleWith *relation expresses that certain combinations of alternatives do not work together. It adds additional undirected edges to the graph. The relation is symmetric, but neither reflexive nor transitive.*
- isCompatibleWith(a_i, a_k) *with i, k arbitrary. The* isCompatibleWith *relation expresses that certain combinations of alternatives work together. The relation defines an equivalence relation, i.e., it is reflexive, symmetric, and transitive and thus identifies classes of compatible alternatives*.

Table 17. Logical relations between architectural decision alternatives

Relation	Set(s)	Reflexive/ Symmetric/ Transitive	Cardinality	Other Properties
forces	A × A	no/no/yes	n:m	Forms a DAG, which does not have to be connected
isIncompatibleWith	A × A	no/yes/no	n:m	–
isCompatibleWith	A × A	yes/yes/yes	n:m	Default if no other relation exists between two alternatives

Our next two integrity constraints pertain to these three relations.

Integrity Constraint 3. *A* forces *relation implies that an alternative in one issue is incompatible with all other alternatives in that issue:*

$$\forall\ i, a_i, a_j, a_k, i \prec a_j, i \prec a_k, j \neq k\text{: } a_i \textit{ forces } a_j \Rightarrow a_i \textit{ isIncompatibleWith } a_k$$

[40] As we will explain in Section 6.2.3, this must be a vendor asset level issue because the transactionality of invoke activities is not specified by the BPEL technology standard.

Integrity Constraint 4. *The* forces*,* isIncompatibleWith*, and* isCompatibleWith *relations between alternatives are mutually exclusive; one of them must exist. If nothing is defined,* isCompatibleWith *is the default.*

$$\forall a_i, a_j: a_i \mathit{forces}\ a_j \wedge a_i \mathit{isIncompatibleWith}\ a_j \equiv \texttt{false}$$
$$\forall a_i, a_j: a_i \mathit{isIncompatibleWith}\ a_j \wedge a_i \mathit{isCompatibleWith}\ a_j \equiv \texttt{false}$$
$$\forall a_i, a_j: a_i \mathit{forces}\ a_j \wedge a_i \mathit{isCompatibleWith}\ a_j \equiv \texttt{false}$$
$$\forall a_i, a_j: a_i \mathit{forces}\ a_j \vee a_i \mathit{isIncompatibleWith}\ a_j \vee a_i \mathit{isCompatibleWith}\ a_j \equiv \texttt{true}$$

The *isIncompatibleWith* relation expresses that certain alternatives do not work with each other, for instance a nontransactional service provider (primitive ST-N from Section 6.1.3) must not be called from a service consumer that has been decided to share transaction context with its provider (primitive PAT-J). A *forces* relation specifies that an alternative can only be combined with one alternative in a different issue. For example, a conceptual primitive ST-J requires the ENTERPRISE JAVABEAN TRANSACTION ATTRIBUTE (technology) to be set to `TX_MANDATORY`.

In addition to the four formally defined integrity constraints, several heuristics can also be defined for an architectural decision model $\mathbb{M}$.

Definition 6.10 (Balanced Architectural Decision Model). *An architectural decision model* $\mathbb{M}$ *is balanced if and only if the following informally defined heuristics regarding its structural properties hold:*

1. $\mathbb{M}$ *has at least two, but not more than five levels.*
2. *Topic groups do not contain more than nine other topic groups and twelve issues.*
3. *On all but the lowest level, there is at least one issue that has an outgoing refinement relation.*
4. *On all but the highest level, there is at least one issue that has an incoming refinement relation.*
5. *The maximum path length to get from the initial issue to any issue via the contains relation* $\prec$ *and to get from the initial issue to any issue via* refinedBy *and* decomposesInto *relations is ten.*

Quality attributes such as usability and consumability for humans justify these heuristics: An unbalanced model is difficult to maintain (for the knowledge engineer) and consume (for the software architect) due to the many elements per topic group and lengthy reasoning paths. We provide more rationale in [ZKL+09].

We now continue the SOA design example and provide more details about the refinement of the exemplary patterns on the technology and the vendor asset level.

6.2.3 Sample Application to SOA: Transaction Management

Figure 26 showed a balanced architectural decision model with several dependency relations between issues. We now cover the alternatives of the issues residing on the technology level and the vendor asset level in more detail.

Technology-level refinement. We now map the PAT, CT, and ST primitives to Business Process Execution Language (BPEL) [OAS07] and Service Component Architecture (SCA) [OSOA]. We expect that BPEL engines allow configuring the transactional behavior at least for *invoke activities*, which correspond to the abstract process activities we introduced in Section 6.1.3. Invoke activities communicate with services via protocols such as SOAP/HTTP [WSI06], Internet Inter-ORB Protocol (IIOP) [OMG04], and JMS [SunJMS], which differ in their support for transaction context propagation and (a)synchrony. The transactional behavior of SCA components is defined by SCA *qualifiers*. Qualifiers specify the behavior desired from the point of view of the service consumer (SCA reference, SCA import) and the service provider (SCA interface, SCA implementation) [ZGT+07].

1. The PAT primitives do not have a direct BPEL realization; typically, BPEL engine vendors add proprietary support for it. The exact semantics of PAT are BPEL engine-specific. For example, during a rollback an engine may let the entire process fail, request resolution by a human operator, or retry one or more activities at a later point in time (potentially with a different transactional scope). While this is engine-specific behavior outside of the scope of the BPEL specification, the architect must be aware of it when selecting between PAT-J and PAT-N.
2. CT-SNT as a synchronous invocation not propagating the transactional context maps to native SOAP/HTTP or IIOP as transport protocol. CT-ST maps to SOAP/HTTP with WS-AtomicTransaction (WSAT) [OAS07a] support or to IIOP. CT-AS can be implemented with JMS [SunJMS]; however, no standardized WSDL binding exists at present. CT also determines the SCA qualifiers on reference, import, and interface level, e.g., `SuspendTx` and `JoinTx`.
3. ST can be mapped to the SCA qualifier `Transaction` on SCA component implementation level.

Table 18 maps the three conceptual patterns from Section 6.1.3 to CT and ST primitives and corresponding SCA qualifiers exemplarily. This mapping implies *forces* relations, e.g., CT-SNT forces `SuspendTx=true`.

Table 18. Mapping of conceptual patterns to primitives and SCA qualifiers

Primitive (PADs)	**CT**	**CT**	**CT**	**ST**
TransportQoS / **TPDs** / **Pattern**	**SCA reference** (BPEL process as SCA component invoking other components)	**SCA import** (reference to external service provider)	**SCA interface** (service provider)	**SCA implementation** (service provider)

TRANSACTION ISLANDS	CT-SNT
	`DeliverAsyncAt=n/a`
	`SuspendTx=true`
		CT-SNT
		`JoinTx=false`
			CT-SNT
			`JoinTx=false`
				ST-N
				(or ST-J)
				`Transaction=local|global|any`

TRANSACTION BRIDGE	CT-ST
	`DeliverAsyncAt=n/a`
	`SuspendTx=false`
		CT-ST
		`JoinTx=true`
			CT-ST
			`JoinTx=true`
				ST-J
				`Transaction=global`

STRATIFIED STILTS	CT-AS `DeliverAsyncAt =commit SuspendTx=false`	CT-AS `JoinTx =n/a`	CT-AS `JoinTx =n/a`	ST-J `Transac- tion =global`

The decision to use SCA is a Technology Selection Decision (TSD) as per IR4 from Chapter 5; each SCA qualifier is an example of a Technology Profiling Decision (TPD) as per IR5 (Chapter 5).

Refinement to vendor asset level. IBM WebSphere Process Server (WPS) [IBM], to be selected in an ASD as per IR6 from Chapter 5, provides a BPEL engine which exposes processes and services as SCA components. In WPS, a BPEL-based SCL connects to the lower architectural layers via SCA. The SCA qualifiers from Table 18 govern the transactional context propagation. Furthermore, PAT translates into a proprietary invoke activity configuration attribute called `transactionalBehavior` which can be set to `requiresOwn` (PAT-N) and `participates` (PAT-J) as shown in Figure 26. Two additional vendor-specific values exist, which we did not model as primitives, `commitBefore` and `commitAfter`. The proprietary attribute is modeled as an ACD as per IR7 from Chapter 5.

We implemented this PAT mapping in a decision injection tool prototype which will be introduced in Chapter 7 (Section 7.3). The tool analyzes the conceptual pattern selection decision and configures the BPEL process model in WPS accordingly.

6.2.4 Discussion and Summary

In this section, we formalized the entities in the SOAD metamodel with the objective to structure decision models as SOAD step 3. Our primary concepts were refinement levels and topic group hierarchies starting with architectural layers.

Justification. When designing enterprise applications, the technical discussions often circle around detailed features of certain vendor products or the pros and cons of specific technologies, whereas many highly important strategic decisions and generic concerns are underemphasized. While these discussions are related, they should not be merged. Hence, our level structure is inspired by Model Driven Architecture (MDA) model types: Practitioners in roles such as business analyst, architect, and developer are involved in SOA design. They create a Platform-Independent Model (PIM) of the design based on a Computing-Independent Model (CIM) of requirements analysis results and transform the PIM into one or more Platform-Specific Models (PSMs) and eventually into code [OMG03].[41]

Going through at least two refinements steps is good practice, e.g., Fowler [Fow00] and RUP [Kru03] recommend such an approach for UML class diagrams used as design models. IBM UMF [CCS07] defines three levels of refinement for logical component models and physical operational models.

[41] With this model structure, we do not imply that MDA concepts such as Meta Object Facility (MOF), metamodels and model transformations [OMG03] are adopted.

Using layers as a second organizing principle is a natural choice, projecting the SOA principle of logical layering into the decision models. We introduced SOA layers in Chapter 2 to motivate the service composition pattern (Definition 2.8).

An explicit representation of logical dependency relations helps uncovering implicit assumptions, contradictions, and implementation limitations so that a more objective technical discussion becomes possible (see our example).

Assumptions. The motivating examples came from the SOA domain; however, the concepts presented in this section can also be applied to other application genres and architectural styles; extensibility is a design goal for SOAD. It is possible use other structuring schemes, for instance, other refinement levels such as elaboration points from software engineering methods like RUP [Kru03] or panes from enterprise architecture frameworks like TOGAF [OG07].

In the SOA design example dealing with transaction management, it is possible to map the primitives to other vendor assets, requiring a different set of ASDs and ACDs. Furthermore, a non-SOA transaction management attribute refinement is presented in [WJ05].

Consequences. Comprehending the level structure requires certain skills. Not all members of the target audience see the benefit of separating concepts and technologies during design if a single technology or vendor dominates the design.

Next steps. The next section in Chapter 6 presents SOAD step 4, completing the formalization of the SOAD metamodel with temporal decision dependencies.

Related publications

This part of our metamodel formalization is also described in [ZKL+09].

The SOA design example first appeared in [ZGT+07].

6.3 Framework Step 4: Add Temporal Decision Order

As SOAD step 4, we investigate:

How to represent temporal dependencies between decisions required (issues)?
How to order the decisions in a model to prepare for decision making?

Making this step, we already identified issues, documented them individually, and structured the model in refinement levels and topic group hierarchies starting with architectural layers. In this last step in the asset creation phase, we enhance the decision models with temporal dependency relations.

To structure the section, we evolve from a brief review of the state of the art and the practice to presentation of our concepts to brief discussion.

6.3.1 State of the Art and the Practice

State of the art. Kruchten et al. introduce dependency types in their ontology [KLV06]. Some of these dependency types have temporal semantics. However, the dependencies are not used to define a decision making process. Jansen views software architecture as a set of decision decisions [Jan08]. His focus is on changes in the architecture. However, he does not consider how to model temporal dependencies and when in the design process to make which decision.

State of the practice. As already mentioned in Chapter 2, decision capturing is often based on text templates and conducted as an after-the-fact documentation activity. In such retrospective practices, dependency management and model organization often have low priority. The ordering of the decision making process is inherited from the general software engineering or architecture design method adopted. The most common ordering approach is intuition: "Worst first" in terms of external dependencies, effort, and impact on technical risk and project plan is a common rule of thumb. The methods presented in [HKN+07] give some advice.

6.3.2 Concepts: Temporal Relations and Production Rules

We add a relation to an architectural decision model $\mathbb{M}$ (see Section 6.2) to order the decision making process. It is defined between nodes of different types.

Definition 6.11 (*triggers* Relation). *Let $\mathbb{M}$ be an architectural decision model. Let a_i, a_j be architectural decision alternatives in $\mathbb{M}$, let i_k be an architectural decision issue in $\mathbb{M}$, and let t_l be an architectural decision topic group in $\mathbb{M}$.*

- triggers(a_i, i_k, t_l) *with $\neg(i_k \prec a_i)$ and $t_l \prec i_k$. An architectural decision alternative a_i can trigger another issue i_k and with this it triggers the topic group t_l which contains the issue. Indirectly, with the issue, all possible*

alternatives are triggered to direct the architect in the decision making process to the next recommended focus point, i.e., an issue to be resolved next. The relation adds additional directed edges to the model. The relation must not form any cycles when combined with $i_k \prec a$. If triggers(a_i, i_k, t_l) *we also say that a_i triggers i_k and that i_k is triggered by a_i.*

Table 19. Temporal relation in architectural decision models

Relation	Set(s)	Reflexive/ Symmetric/ Transitive	Cardinality	Other Properties
triggers	A × I × T	n/a	n:m:1	Forms one or several DAGs, but not a tree.

The *triggers* relation expresses a temporal ordering during the decision making process. For example, when a certain INTEGRATION TECHNOLOGY such as RESTFUL INTEGRATION is decided for, a topic group containing follow-up issues such as URI DESIGN and HIGH OR LOW REST is triggered, while all issues in a WSDL PORT TYPE topic group become irrelevant and can be pruned [PZL08]. Note the suggestive nature: It is permitted to resolve issues that have not been triggered (yet) and multiple triggers may exist per issue. It is possible that an alternative, an issue, and a topic group do not have any *triggers* relation. It would be far too restrictive for the architect to define a strictly enforced decision ordering based on these relations. These *triggers* must satisfy the following integrity constraints:

Integrity Constraint 5. *If an issue i_i is refined by or decomposes into another issue i_j then any alternative a_i in i_i triggers i_j:*

$$\forall\, i_i, i_j, a_i, i_i \prec a_i \colon i_i \mathit{\ refinedBy\ } i_j \vee i_i \mathit{\ decomposesInto\ } i_j \Rightarrow a_i \mathit{\ triggers\ } i_j$$

Integrity Constraint 6. *The* forces *relation between alternatives implies a* triggers *relation:*

$$\forall\, i, a_i, a_j \colon i \prec a_j \wedge a_i \mathit{\ forces\ } a_j \Rightarrow a_i \mathit{\ triggers\ } i$$

In the next step, we define two more integrity constraints regarding the *triggers* relation. The logical implications caused by integrity constraints 5 and 6 allow us to define these solely on *triggers* relations (i.e., it is not required to include *refinedBy*, *decomposesInto*, and *forces* in the definitions):

Integrity Constraint 7 (Trigger Compatibility). *Let a_i* triggers *i_j hold. Let $I(a_i)$ be the set of issues that can be reached from a_i following* triggers *relations and the contains relation $\prec$ within one tree $\mathbb{T}_{km}$ starting with alternative a_i. Note that $I(a_i)$ can reach into other trees $\mathbb{T}_{ln}$.*[42]

[42] $I(a_i)$ can be calculated like this: Initialize $I(a_i)$ with all issues triggered by a_i. Iterate: For any issue i added in the last iteration, follow the *triggers* relations originating in alternatives contained in i and add the target issues. Re-iterate if any new members were added in this iteration.

Then a_i must either have an isCompatibleWith *relation with at least one alternative a_x or a* forces *relation with exactly one a_x for every $i_j \in I(a_i)$ and $i_j \prec a_x$:*

$$\forall\ a_i, a_x \in A, \forall\ i_j \in I(a_i):$$
$$i_j \prec a_x \Rightarrow a_i\ \textit{isCompatibleWith}\ a_x \vee a_i\ \textit{forces}\ a_x$$

Integrity Constraint 8 (Top-Down Progression). *Let $i_i \prec a_i$ and a_i* triggers *i_j hold. i_j must then reside on a lower level than i_i or, if i_i and i_j reside on the same level, i_j must be greater than i_i according to $<$.*

Certain combinations of *forces*, *triggers*, and *isIncompatibleWith* relations should not occur. For example, an alternative must not trigger the issue in which it is contained ($\prec$ relation). Less obvious consistency problems can occur when chaining more issues and alternatives together.

Definition 6.12 (Valid and Strictly Valid Architectural Decision Model). *An architectural decision model $\mathbb{M}$ is called* valid *if integrity constraints 1 to 7 hold. If integrity constraint 8 also holds, $\mathbb{M}$ is called* strictly valid*.*

The model of the transaction management issues (Figure 26 on page 101) meets all constraints. It is a strictly valid architectural decision model.

Finally, we can define how architectural decision models can be traversed:

Definition 6.13 (Entry Points, EP). *The* Entry Point (EP) *set is a set of architectural decision issues in an architectural decision model $\mathbb{M}$ that do not have any incoming* triggers *relations:*

$$EP = \{\ i \in I \mid \nexists\ a \in A: (a\ \textit{triggers}\ i)\ \}$$

An entry point is a natural starting point for architecture design activities in a given decision making context. There can be multiple ones. The INVOCATION TRANSACTIONALITY PATTERN issue is the only entry point in Figure 26, which is marked as such.

Definition 6.14 (Open Issue, Made Decision). *An* open issue *is an issue which does not have any chosen alternative. A* made decision *(a.k.a. resolved issue) is an issue with exactly one chosen alternative, i.e., where chosen =* `true` *(recall Definition 6.3). We do not allow multiple alternatives to be chosen per issue.*

We can further classify issues with the help of the *triggers* relations:

Definition 6.15 (Eligible Issue). *An* eligible issue *is an open issue whose incoming* triggers *relations (if existing) originate from alternatives in made decisions*.

Definition 6.16 (Pending Issue). *A* pending issue *is an open issue which has one or more incoming* triggers *relations and at least one of these relations originates from an alternative in an open issue*.

All open issues are either eligible or pending. Eligible issues can be resolved in the next decision making step, while pending ones have to wait until the ones they depend on (due to an incoming *triggers* relation) have been made. Note that issues can be eligible or pending because of *triggers* relations implied by *refinedBy*, *decomposesInto*, or *forces* relations.

In some cases, issues no longer have to be considered because of other decisions already made and existing *forces* or *isIncompatibleWith* relations:

Production Rule 1 (Alternative Pruning). *If two alternatives have an* isIncompatibleWith *relation and one of them is chosen during the decision making process, then it prunes the other:*

$$\forall\ a_i, a_j \in A:$$
$$a_i\ \textit{isIncompatibleWith}\ a_j \wedge \texttt{chosen}(a_i) \equiv \texttt{true} \Rightarrow \texttt{chosen}(a_j) = \texttt{false}$$

Production Rule 2 (Outcome Implication). *If one alternative is chosen and it forces another, then the second one must be chosen as well:*

$$\forall\ a_i, a_j \in A:\ a_i\ \textit{forces}\ a_j \wedge \texttt{chosen}(a_i) \equiv \texttt{true} \Rightarrow \texttt{chosen}(a_j) = \texttt{true}$$

Integrity Constraint 9. *Only alternatives that do not have an* isIncompatibleWith *relation can be chosen within the same decision making process (i.e., either an* isCompatibleWith *or a* forces *relation must exist due to integrity constraint 4):*

$$\forall\ a_i, a_j \in A:\ \texttt{chosen}(a_i) \equiv \texttt{true} \wedge \texttt{chosen}(a_j) \equiv \texttt{true}$$
$$\Rightarrow (a_i\ \textit{isCompatibleWith}\ a_j \vee a_i\ \textit{forces}\ a_j)$$

Production Rule 3 (Outcome Instance Status Update) and Definition 6.17 (Implied Decision). *An* implied decision *is an issue with:*

Case 1) All but one alternative have been pruned by production rule 1, i.e., `chosen` ≡ `false`*. The remaining alternative is set to chosen* ≡ `true`*. The open issue becomes a resolved issue (a.k.a. made decision).*
Case 2) One alternative has been selected by production rule 2, i.e., `chosen` ≡ `true`*. All other alternatives can be set to* `chosen` ≡ `false`*.*

We can verify whether additional decision making is still required.

Definition 6.18 (Decided and Correct Architectural Decision Model). *A valid architectural decision model is called* decided *if all decisions are made (all issues are resolved), i.e., have exactly one of their alternatives marked as* `chosen`*. If integrity constraint 9 holds, a decided architectural decision model is called* correct*.*

When the decision making process completes, all decisions must have been made, i.e., neither eligible nor pending issues exist. Each issue now has one alternative with `chosen` ≡ `true` (and all other alternatives are `chosen` ≡ `false`) or all alternatives are `chosen` ≡ `false`. All integrity constraints should be satisfied.

Definition 6.18 completes the formalization of our architectural decision metamodel supporting reuse and collaboration.

6.3.3 Sample Application to SOA: Transaction Management

The concepts introduced in this section can be applied to SOA design. We will give an SOA decision making example in Chapter 7 (Section 7.2), continuing to

use the excerpt from the RADM for SOA created during thesis validation which we already used in steps 2 and 3 (transactional workflows in SOA).

6.3.4 Discussion and Summary

In this section, we covered SOAD step 4, the final step in the asset creation phase. We introduced temporal decision relations as well as integrity constraints, an issue status classification, and production rules to the SOAD metamodel.

Justification. Active issue management leads to a more dynamic knowledge base than one provided by static asset repositories and method browsers. This helps to cope with the challenges in enterprise application development and integration. For instance, entire topic group trees can be pruned based on the outcome of a decision just made, which reduces the decision making effort. We present an example of such pruning in [ZKL+09].

Assumptions. We assume only one outcome instance to be present per ADM issue as we did not formalize ADOutcome instances in this section. Hence, the issue classification and the production rules do not take the existence of multiple outcome instances into account. To do so, the formalism is extended in [ZKL+09].

Consequences. While a top-down approach to architecture design is generally recommended and useful, it can not always be applied in practice. When modernizing enterprise applications, many technology- and vendor asset-level decisions have already been made prior to project start (e.g., those pertaining to legacy systems). When procuring a software package, the procurement decision implies the interface, transaction, and session management design chosen in the package. When deciding for a certain application server strategically, a vendor asset level decision is upgraded to the executive level. An architectural decision model for such a setting does not satisfy integrity constraint 8 (top-down progression). Different integrity constraints and production rules must be defined to reflect such bottom-up approach to design. We will discuss such applicability and extensibility issues in the decision making step 6 (Section 7.2 in Chapter 7).

Next steps. With design issues identified (step 1) and modeled according to the SOAD metamodel (steps 2, 3, and 4), the asset creation phase ends. The RADM is ready for reuse, i.e., it can now serve as input to the creation of project ADMs (i.e., asset consumption, described in steps 5 to 7).

Related publications

An extension of the metamodel formalization and additional examples are presented in [ZKL+09].

7 Creating and Using Architectural Decision Models on Projects

So far, we focused on the creation of a reusable asset comprising architectural decision knowledge. We now progress to the consumption of such asset (Figure 27):

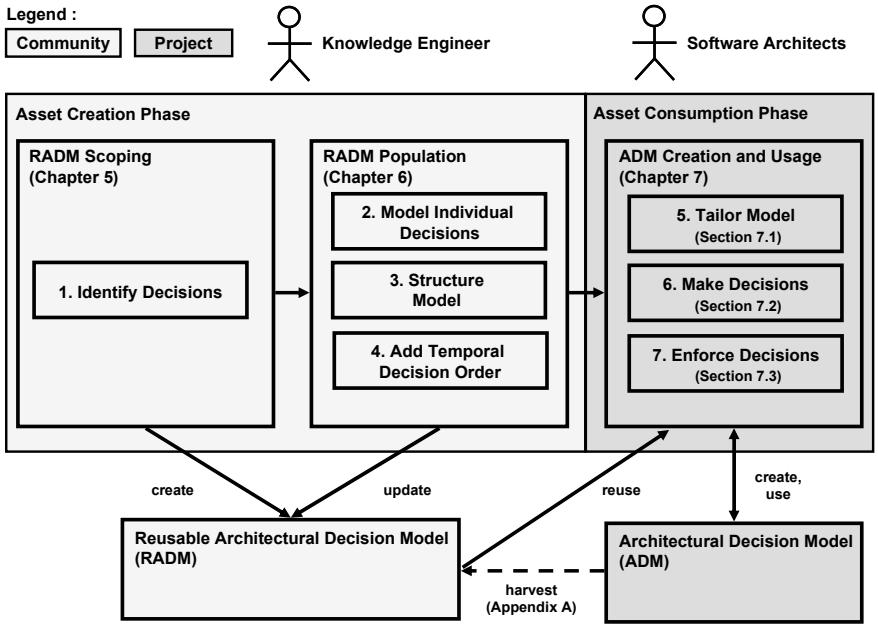


Figure 27. SOAD framework steps during asset consumption on projects

Each section in this chapter presents one of the steps in the asset consumption phase: Section 7.1 explains how to tailor a RADM into an ADM, Section 7.2 how to use an ADM when making decisions, and Section 7.3 how to enforce decisions.

7.1 Framework Step 5: Tailor Model

SOAD step 5 deals with reuse of RADMs scoped and populated in steps 1 to 4:

How to tailor a Reusable Architectural Decision Model (RADM) for a project?

This step takes a RADM asset (see Definition 4.1) with issues and alternatives (Definition 4.4) as input; project requirements provide further tailoring input. Its output is an ADM (Definition 4.2), which does not include any outcomes (Definition 4.3) yet.

We begin with a brief review of state of the art and the practice and then present our solution. Next we return to the motivating case study and tailor the RADM for SOA for it. The section concludes with a brief discussion.

7.1.1 State of the Art and the Practice

State of the art. Model tailoring is part of the unsolved decision identification problem. Concepts for method tailoring exist in situational method engineering, e.g., method chunks and method fragments [HGR08]. However, existing methods are process- and artifact-centric; hence, these concepts are not applicable here.

State of the practice. Leveraging RADMs during design is not common as of today; hence, no tailoring other than copy-paste of document fragments is practiced.

7.1.2 Tailoring Technique and Decision Filtering Concept

RADMs developed with SOAD do not aim to be complete: On the one hand, it will always be required to capture project-specific issues and outcomes not covered in a RADM. Examples are issues concerning the integration of proprietary technologies and legacy systems, issues related to environmental conditions (e.g., regarding budget and team), and issues dealing with out-of-line situations. On the other hand, it is not likely that all issues present in a RADM are relevant in a particular project or phase, e.g., if only a subset of the style-defining patterns is used. Furthermore, certain issues may have to be resolved more than once. In SOA design, this is the case if a pattern such as ESB is applied several times or if multiple business processes and Web services appear in the architecture.

In response to these customization and adoption needs, we provide a model *tailoring technique*. It leverages the SOAD metamodel from Chapter 6 (for an overview, see template in Figure 19 on page 86 and example in Figure 21 on page 89).

Technique overview. The SOAD tailoring technique works in the following way:

1. *Select RADMs* to be tailored, having reviewed the project context. We assume that one or more architectural styles have already been chosen for the project. If RADMs for these styles are not available, our decision identification technique (step 1) can be applied to scope an ADM now.
2. *Use decision filtering* (explained below) to eliminate unnecessary issues. Requirements and existing architectural artifacts drive this activity.
3. *Update issue information*, e.g., with project-specific decision drivers or alternatives not present in the RADM. This additional information is structured according to the SOAD metamodel introduced in steps 2 to 4.

4. *Add issues* known to occur, but not covered by the RADM.
5. *Create outcome instances* for issues that apply multiple times.

In this section, we focus on the concept supporting activity 2, decision filtering. The other activities are supported by the concepts we introduced in steps 2 to 4.

Decision filtering. We use *decision filtering* to select issues from the RADM that are relevant in a particular project context. All issue attributes defined in the SOAD metamodel can be used to select relevant issues from a RADM. Three attributes are particularly relevant and were introduced specifically for this purpose: *scope*, *phase*, and *role*. The *scope* attribute references an architecture element or organizational units such as "enterprise", "domain", or "project" [Pul06]; the *phase* and *role* attributes link issues to the process defined by a software engineering or architecture design method. It is also possible to use the *level* and *topic group hierarchy* introduced in Section 6.2. For instance, in SOA design the architect can select the entire conceptual level or all issues related to the atomic service layer. The *decision dependency* relations can be leveraged as well, e.g., selecting the INVOCATION TRANSACTIONALITY PATTERN issue and all issues it decomposes into or it is refined by. Finally, *topic tags* that annotate issues with subject area keywords such as "transaction management" or "security" can be used (if defined). Figure 28 illustrates the four filtering options (which can be combined):

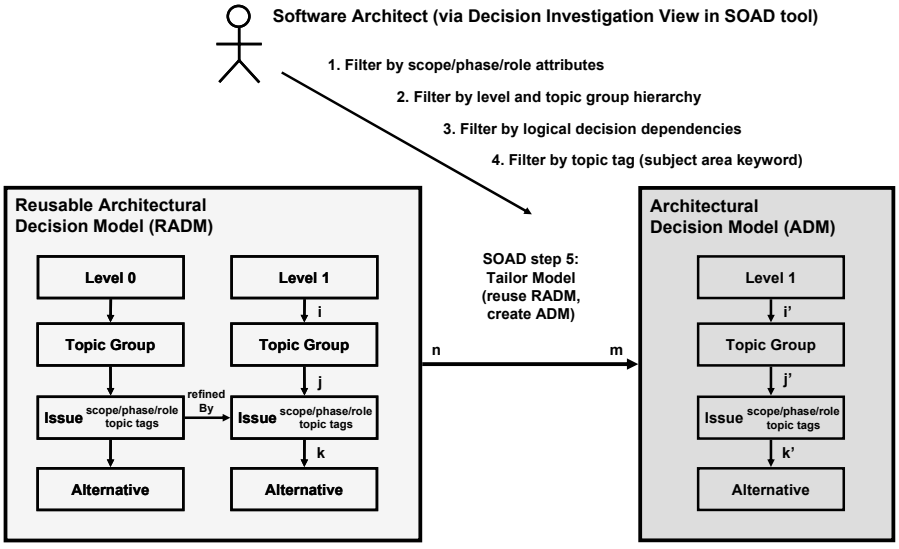


Figure 28. SOAD step 5: Decision filtering

The level filtering is applied here: Only one of two levels is promoted from the RADM to the ADM. The appropriate subset of the logical and temporal decision dependencies (modeled in steps 3 and 4) has to be projected from the RADM to the ADM. In the example, *decomposesInto* relations in level 1 are preserved in the ADM, while *refinedBy* relations between level 0 and level 1 disappear. As a result, new entry points into the decision making become available (see Definition 6.13).

We do not mandate any scope, phase, and role taxonomy; selecting one is a decision of the knowledge engineer. Table 20 suggests scope, phase, and role settings for the seven decision types we defined in step 1 (see Table 13 on page 73).

Table 20. Decision types and exemplary scope, phase, and role attributes

Decision Type	Scope	Phase	Role
Executive decisions	Enterprise- or project-wide	Solution outline	Project manager, lead architect, business analyst
Pattern Selection Decisions (PSDs)	Per project or per pattern in style	Solution outline or macro design	Application or integration architect (for SOA)
Pattern Adoption Decisions (PADs)	Component or connector in pattern	Macro or micro design	Application or integration architect (for SOA)
Technology Selection Decisions (TSDs)	Per project or identifying pattern	Macro or micro design	Application or integration architect (for SOA)
Technology Profiling Decisions (TPDs)	Per usage of technology	Micro design	Application or integration architect (for SOA)
Asset Selection Decisions (ASDs)	Enterprise- or project-wide	Solution outline or macro design	Lead architect, supported by platform specialist
Asset Configuration Decisions (ACDs)	Per physical node or deployment unit	Micro design	Platform specialist, infrastructure architect

The phases were introduced in Chapter 2; the role names are among those used by the architect profession program of IBM. Referencing the SOA patterns from Chapter 2, our RADM for SOA introduced in Chapter 5 defines scopes such as *service consumer*, *operation invocation*, *service provider*, *operation*, *ESB*, and *process*.

7.1.3 Sample Application to SOA and Motivating Case Study

We now tailor the RADM for SOA for the PremierQuotes project, following our tailoring technique and using decision filtering by phase (filter option 1).

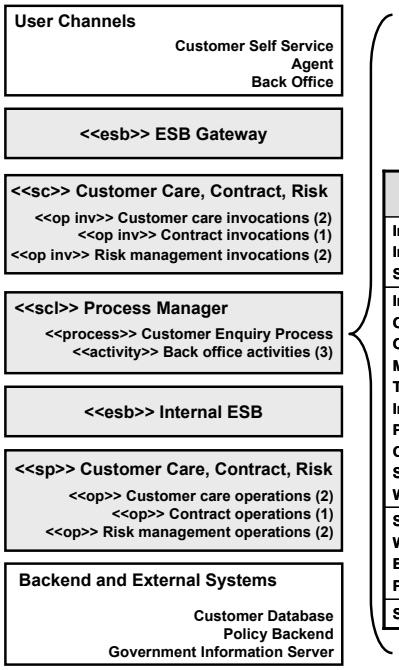
Select RADMs. Let us assume that the RADM for SOA introduced in Chapter 5 is selected because SOA has been decided to be the architectural style.

Use decision filtering. Let us assume that the PremierQuotes architects qualify all 389 decisions in the RADM for SOA (see Table 15 on page 81 in Chapter 5 for an excerpt) to be relevant. We further assume the project to be in the macro design phase. If the RADM for SOA is queried with this decision filtering information, 148 out of 389 decisions are returned and transferred to the ADM.

Update issue information and **add issues.** These activities are also part of SOAD step 2 described in Chapter 6. They do not require any further explanations here.

Create outcome instances. Three service providers appear in Figure 8 (page 28), as well as two instances of the ESB pattern. There is one process, which originates from the single application of the service composition pattern. The service registry pattern is not applied. Figure 29 shows how the architectural elements in Figure 8 and the issues from the RADM for SOA are combined to create the project ADM.

The decision scoping information comes from the RADM for SOA; in line with Table 20, these scopes refer to the SOA patterns from Chapter 2 (e.g., `<<sp>>` for service provider). Three outcome instances are created for the service provider issues, two for the ESB issues, one for the process manager issues, and 2+1+2=5 for the issues pertaining to the operations implemented by service providers.



<<sc>> service consumer, used 3x
<<op inv>> operation invocation, used 2+1+2=5x
<<sp>> service provider, used 3x
<<op>> operation, used 2+1+2=5x
<<esb>> Enterprise Service Bus (ESB), used 2x
<<scl>> Service Composition Layer (SCL), used 1x
<<process>> process, used 1x
<<activity>> process activity in business process, used 3x

Issues in ADM (tailored from RADM for SOA)	scope Attribute	Outcome Instances
Integration Paradigm (PSD)?	<<esb>>	2
Integration Technology (TSD)?	<<esb>>	2
SOAP Engine (ASD)?	<<esb>>	2
In Message Granularity (PAD)?	<<op>>	5
Out Message Granularity (PAD)?	<<op>>	5
Operation-To-Service Grouping (PAD)?	<<sp>>	3
Message Exchange Pattern (PAD)?	<<op>>	5
Transport Protocol Binding (TSD)?	<<op>>	5
Invocation Transactionality Pattern?	<<op>>	5
Process Activity Transactionality?	<<activity>>	3
Communications Transactionality?	<<op>>	5
Service Provider Transactionality?	<<op>>	5
Web Services API (TSD)?	<<op inv>>	5
Service Composition Paradigm (PSD)?	<<scl>>	1
Workflow Language (TSD)?	<<scl>>	1
BPEL Engine (ASD)?	<<scl>>	1
Process Lifetime (PAD)?	<<process>>	1
Service Registry (PSD)?	<<project>>	0

Figure 29. SOAD step 5: RADM tailoring in motivating case study

7.1.4 Discussion and Summary

In this section, we presented our tailoring technique and decision filtering (step 5).

Justification. As explained in Chapter 2, method tailoring, also known as method adoption, is a common concept in commercial methods such as RUP [Kru03] and UMF [CCS07]. RADM tailoring satisfies the usability requirement from Chapter 3 (R1-7) and makes SOAD applicable to a broad range of projects.

Assumptions. To make the decision filtering efficient, the issues in the RADM must be attributed with accurate scope, phase, and role information, organized in a balanced topic group hierarchy, and/or annotated with expressive topic tags.

Consequences. Decision filtering not only helps during tailoring; it also can be leveraged during decision making to reduce the number of decisions displayed.

Next steps. Having demonstrated model tailoring, we cover decision making next.

7.2 Framework Step 6: Make Decisions

Having identified, modeled, and ordered architectural decisions and structured and tailored decision models in steps 1 to 5, we can now realize the primary use case for SOAD as step 6:

How to use an Architectural Decision Model (ADM) as an SOA design method?

This step operates on an ADM created in step 5. The project requirements provide additional input. When the decision making completes, the ADM contains project-specific outcome instances capturing decision making rationale.

We begin with a brief review of state of the art and the practice and evolve to presentation of solution, SOA application, and discussion. We continue with the design activities in the motivating case study and resolve the open issues.

7.2.1 State of the Art and the Practice

State of the art. We presented five industrial architecture design methods in Chapter 2, including Attribute-Driven Design (ADD) and Siemens 4 Views (S4V). Techniques such as Cost-Benefit Analysis Method (CBAM) [BCK03] also support certain architecture design tasks. These methods and techniques do not order the decision making process in an application genre- and architectural style-specific way. While a backlog has been suggested [HKN+07], we did not find any concepts that support a semi-automatic population of the backlog with genre- and style-specific issues or an active, dependency-based issue management.

Decision Support Systems (DSS) can be leveraged during architectural decision making [DC07, SWL+03]. Such existing work helps to make one or more decisions; however, it does not focus on organizing the decision making process.

State of the practice. A major gap exists between research and practice.[43] Architectural decision making is often seen as an art rather than part of an engineering process. Many architects do not follow a design method, but their personal experience and intuition ("gut feel"). Issue lists are maintained manually if at all.

Personal preferences have a large impact on the decision ordering and making. Frequently, a single decision driver (e.g., quality attribute) or issue (e.g., technology selection) is overemphasized. Phrases like "we have always done it like that" or "this is the industry trend" justify decisions rather than sound technical judgment backed by evidence gained in tradeoff analysis activities or technical evaluations. Consequently, the technically best solution is not always selected. Ill-motivated and poorly organized decision making often is a root cause for project failure: Too much focus on less relevant issues and suboptimal alternatives may degrade the quality of the resulting software architecture, or cause unnecessary design and development efforts which delay the project.

[43] This is a subjective assessment; see footnote in Section 5.1.1 for sourcing information.

7.2.2 Concepts: Managed Issue List and Decision Making Processes

To overcome the gap between the state of the art and the state of the practice, we leverage an ADM to steer the decision making activities. A *managed issue list* orders the issues so that only those that are currently relevant are presented to the architect. For each of these issues, architectural knowledge required for the decision making is presented, which originates from previous project experience with the issue and the alternatives captured in the RADM tailored in the previous step.

In the role of a decision-centric architecture design method, SOAD extends the software engineering method(s) employed. It adds a *macro process* for the decision making on the project. This process is based on the decision ordering concepts from Chapter 6, i.e., logical *refinedBy* and *decomposesInto* relations modeled by the knowledge engineer and resulting temporal *triggers* relations. It launches a *micro process* for each issue. This micro process leverages attributes such as problem statement, decision drivers, and recommendation, which are defined in the SOAD metamodel, to investigate, make, and enforce individual issues.

Definition 7.1 (Managed Issue List). Adopting and adapting the concept of a backlog suggested in [HKN+07], *we define the set of open and resolved issues (Definition 6.14) in the ADM as our managed issue list and the resolved issues (a.k.a. made decisions, also Definition 6.14) as our decision log.*

Figure 30 zooms into Figure 14 from page 62 in Chapter 4 to introduce the use cases and components of the issue list manager, including the managed issue list:

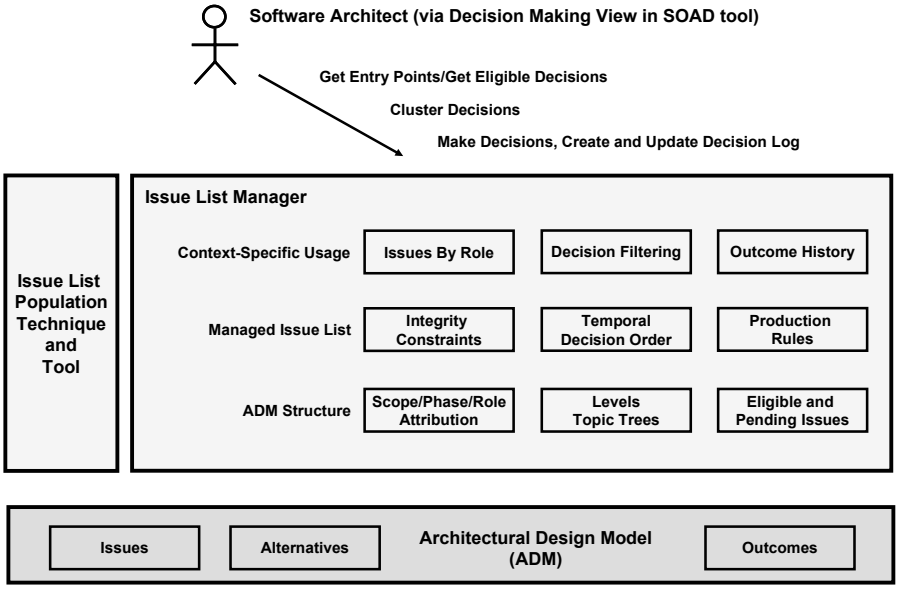


Figure 30. SOAD step 6: Issue list manager with managed issue list

Architects interact with the managed issue list via the decision making view component of a SOAD tool: When following the macro process defined below,

architects use the get entry points/get eligible decisions, cluster decisions, and make decisions and create/update decision log features provided by the issue list manager. The issues by role, decision filtering, and outcome history components (in the context-specific usage part of the issue list manager) support these operations with ADM querying capabilities. During decision making, issue states and temporal decision order are updated continuously, with the help of the *triggers* relations and production rules defined in Section 6.3. The managed issue list can check the validity and correctness of the ADM with the help of the integrity constraints from Section 6.2 and Section 6.3 (e.g., alternatives of made decisions must be compatible with each other or force each other). The managed issue list can be populated semi-automatically with the help of an issue list population technique and tool.

We demonstrate the capabilities of the managed issue list in an example in Section 7.2.3. Chapter 8 provides further information on SOAD tool support.

Macro process (project level). The macro process works with the managed issue list. We use the phases from the IBM Unified Method Framework (UMF) in this macro process. As explained in Chapter 2, it comprises three design phases, *solution outline*, *macro design* and *micro design*. Figure 31 shows the activities to be conducted in these three phases:

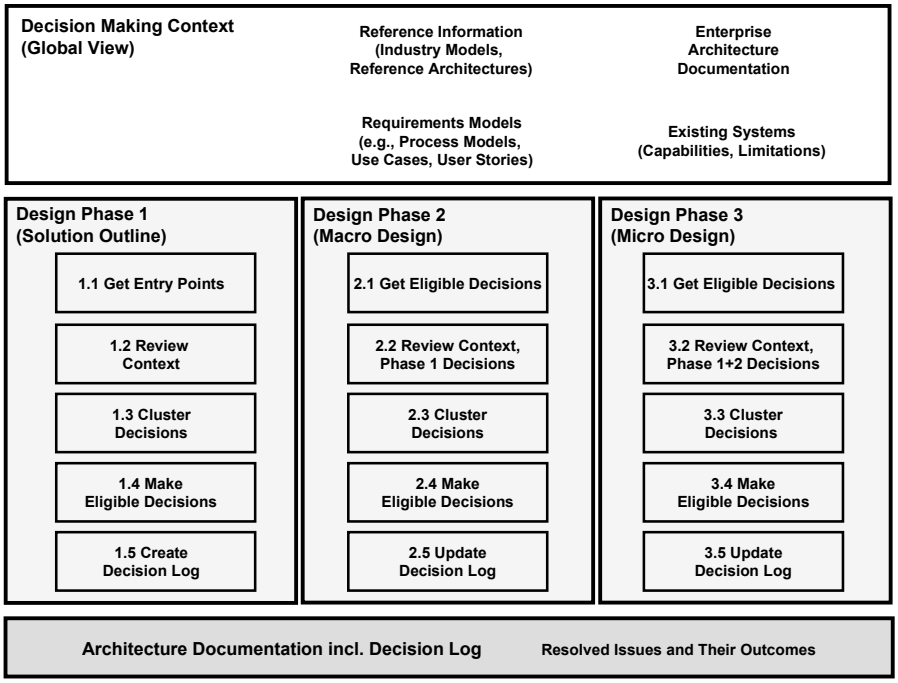


Figure 31. SOAD step 6: Macro process for decision making on projects

The *decision making context* [HKN+07] includes reference information, requirements models, and documentation of the enterprise architecture [SZ92] as

well as existing systems, e.g., legacy systems (Chapter 2). The output of the macro process is the decision log which becomes part of the architecture documentation.

Activity 1.1, 2.1, 3.1. Activities 1.1, 2.1, and 3.1 in our macro design process can be characterized as follows, showing mixed initiatives by the architect A and a decision support system S implementing the SOAD concepts:

```
getEntryPoints (inout: adm, out: entryPoints)

   [S: Initialize managed issue list mil with adm]
   getEligibleDecisions(mil, entryPoints)

getEligibleDecisions (inout: mil,
                      out: eligible decisions)
   [S: Calculate implied decisions as per Def. 6.17]

   If (adm in mil is not decided as per Def. 6.18)
      [S: Calculate eligible/pending issues in mil adm]
      [S: Return eligible issues]
   Else
      [S: Inform architect: decision making terminated]
   End if
```

Activities 1.2, 2.2, 3.2. The second activity in each phase of our macro process is a review activity conducted by the architect. It includes a review of requirements and architectural documentation already available in the decision making context. In solution outline, the review includes *legacy decisions* (i.e., decisions made in a previous project or pertaining to a different enterprise application). The previous project might have been a presales activity or the development of a legacy system a long time ago. In macro and micro design, the decisions made in previous phases of the macro process are reviewed. These activities are standard analysis and decision preparation activities that do not require any further explanation here.

Activities 1.3, 2.3, 3.3. These activities deal with decision clustering. Decisions are rarely made in isolation due to their amount and due to the many dependencies between them. However, it is not obvious how to group and order the decisions that are eligible in a particular macro process phase. Grouping decisions into clusters is typically part of the tacit knowledge of an architect; mature software engineering and architecture design methods provide related advice. Disciplines and elaboration points in RUP [Kru03] are examples of such groupings.

The decision filtering concept introduced in the tailoring step 5 can be leveraged in addition to tacit knowledge about decision clustering:

```
clusterDecisions (in: adm)

   [S: Suggest grouping of issues by scope/phase/role,
       by dependencies, by topic tag (subject area)]
   [A: Group issues as suggested or by tacit knowledge]
    For (each group)
        [A: Assign group to performing team member]
    End for
```

Due to the formalization of the SOAD metamodel, tools can give clustering advice. However, the architect drives the activity. In SOA design, the tool might suggest to assign all issues about an “ESB router” to be made in the “macro de-

sign" phase to an "integration architect". The actual grouping depends on the project setup (e.g., methods adopted, human resources available) and on the architects' experience. The literature provides related criteria, e.g., [HKN+07, RK96].

Activities 1.4, 2.4, 3.4. These activities instruct the architect to make the decisions that were classified to be eligible in the respective phase. The micro process is launched from this activity once per issue (see below).

```
makeEligibleDecisions (in: admGroups)

    For (each open issue oi in each group in admGroups)
        [A: Launch micro process for oi]
    End for

    [A: Consolidate and review decisions from each group]

    If (decision model is not correct as per Def. 6.18)
       [A: Reset alternatives to undefined as needed]
       [A: Repeat decision making for one or more groups]
    End if
```

Activities 1.5, 2.5, 3.5. As the last activity on the macro level, the decision log is created or updated with the outcome instances created during the execution of the micro process. It becomes part of the project deliverables. We will describe decision injection as an additional concept for this step in Section 7.3.

Micro process (issue level). Figure 32 illustrates the micro process:

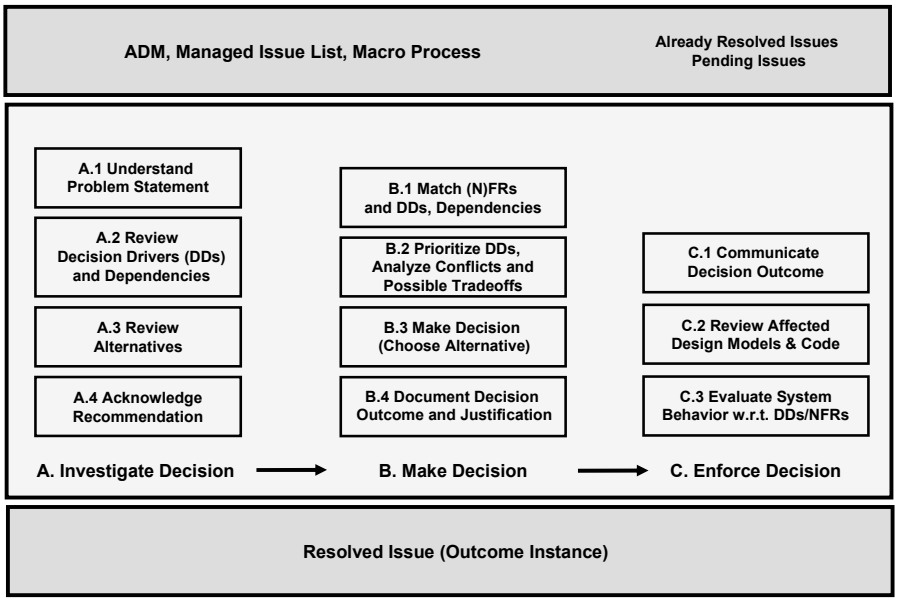


Figure 32. SOAD step 6: Micro process for making single decision

When performing the micro process activities, architects make use of the architectural knowledge in the ADM, which is structured according to the SOAD

metamodel, e.g., listing decision drivers and decision dependencies (see template in Figure 19 on page 86 and example in Figure 21 on page 89).

Step A: Investigate decision. As a first step, the information about an issue in the ADM must be analyzed; the architects can add missing information. We performed this step for the INVOCATION TRANSACTIONALITY PATTERN issue in Chapter 6. It also supports the education and knowledge exchange use cases of SOAD.

In this step, the problem statement, defined in the SOAD metamodel (Chapter 6), must be understood first; if the motivation for the issue remains unclear, the referenced background reading can be consulted (activity A.1).

Next, the decision driver attribute is studied (activity A.2). Like the problem statement, it is an issue attribute; it is reusable, but not project-specific (unless information about actual requirements has been added during tailoring). Hence, it can only list *types* of decision drivers.[44] Still in activity A.2, decision dependencies, particularly those to and from already resolved issues (but also open ones) are investigated. Decisions are rarely made in isolation; the decision maker has to ensure that the resolution of issues that have dependency relations does not introduce conflicts, i.e., that the ADM remains correct. This can later be verified by checking Integrity Constraint (IC) 9 defined in Chapter 6 (Section 6.3).

The available alternatives have to be considered next (activity A.3). The pros and cons information is particularly relevant; when studying it, the decision drivers and project requirements studied in A.1 and A.2 are revisited.

The final investigation activity A.4 is to review and acknowledge the recommendation. This does not mean that the recommendation should always be followed. The decision making context determines whether this is possible.

Step B: Make decision. The second step of the micro process is the actual decision making. In activity B.1, the architect matches the actual (N)FRs on the project against the decision drivers and decision dependencies investigated in activity A.2. Activity B.2 advises the architect to prioritize decision drivers according to their importance and to analyze potential conflicts and tradeoffs. Before an alternative can be selected, its consequences must be assessed. In many cases, an alternative which may appear to be suited on the micro process level can not be selected due to certain constraints which are only visible at the macro process level (e.g., limitations of legacy systems). Activity B.3 is to actually make the decision, based on the insight gained during the previous step A and step B activities.

Activities B.1 to B.3 are already supported by existing architecture design assets such as ADD [BCK03]; further details are therefore out of scope of SOAD.

In activity B.4, the chosen alternative and the justification for the decision are documented in outcomes. Decision drivers, pros and cons of alternatives, and the recommendation should be referenced in the justification. The justification should not only quote reusable background information such as the types of decision drivers coming from the RADM, but refer to actual project requirements as well [ZSE08].

[44] Appendix A provides a classification of types of decision drivers particularly relevant for enterprise application development and integration and SOA design.

Step C: Enforce decision. The third step of the micro process deals with enforcing the decision. The three activities in this step are to communicate the decision outcome (activity C.1), to review affected design model elements and code (activity C.2), and to compare the behavior of the emerging implementations of the system under construction with the decision drivers and actual NFRs including project-specific quality attributes (activity C.3). It is necessary to re-evaluate on the macro level, as decisions often unveil their full consequences in combination.[45]

Termination of macro and micro process. Macro process and, in turn, micro process continue as long as architectural decision making is still required and the ADM is not decided (according to Definition 6.18). More than three phases can be required. It may take a long time to complete the decision making; the managed issue list can be continued to be used during operations and *maintenance* [Som95].

Extensibility. We designed the macro process to be customizable for different software engineering methods. A project adopting an agile process iterates through all design activities rapidly, e.g., within one day (notion of *daily stand ups* [Yip]). Our issue management (step 6), decision filtering (step 5), *triggers* relations (step 4), and model heuristics (step 3) concepts must work in such setting. This is the case if RADM (and consequently ADM tailored from it) are documented in a compact form and well attributed with decision filtering information.[46]

7.2.3 Sample Application to SOA and Motivating Case Study

To demonstrate how the macro and the micro process work with the managed issue list, we now apply these concepts to SOA design and the motivating case study. Due to space constraints, we can only demonstrate a subset of the activities.

Managed issue list and macro process. We presented a subset of the RADM for SOA issues in detail in Chapter 6 (see Figure 21 on page 89 and Figure 26 on page 101): INVOCATION TRANSACTIONALITY PATTERN is the entry point in that ADM which comprises five issues. As an initial issue, it resides on the highest level, the conceptual level. It is eligible initially. Table 21 illustrates the initial decision making status, returned by `getEntryPoints`:

Table 21. Entry points, eligible, and pending decisions in example (1)

Eligible Issues	Pending Issues	Made Decisions
INVOCATION TRANSACTIONALITY PATTERN	COMMUNICATIONS TRANSACTIONALITY (CT) PROCESS ACTIVITY TRANSACTIONALITY (PAT) TRANSPORT QOS INVOKE ACTIVITY TRANSACTIONALITY	none

[45] Hofmeister et al. see activities C.2 and C.3 as part of architecture evaluation [HKN+07].

[46] Further information how to configure SOAD for an agile project is out of scope of this thesis; such configuration of the SOAD processes requires future work.

We now choose TRANSACTION ISLANDS for all three service consumers because process and backend interactions can run for days; it is not affordable to keep the transaction context open [LR00]. Making this decision does not violate any integrity constraints; however, the model is not decided yet. Once this issue has been resolved, the related pattern adoption decisions are triggered and become eligible. Table 22 shows the new status as returned by `getEligibleDecisions`:

Table 22. Entry points, eligible, and pending decisions in example (2)

Eligible Issues	Pending Issues	Made Decisions
COMMUNICATIONS TRANSACTIONALITY (CT) PROCESS ACTIVITY TRANSACTIONALITY (PAT)	TRANSPORT QOS INVOKE ACTIVITY TRANSACTIONALITY	INVOCATION TRANSACTIONALITY PATTERN (see Table 23 for outcome)

Due to *forces* relations (see Section 6.2.3), outcomes for one eligible and one pending issue, the CT and TRANSPORT QOS issues, can be implied by the resolved INVOCATION TRANSACTIONALITY PATTERN issue. If tool support for the presented concepts is available, this logical implication can be detected automatically and presented to the architect as a decision making proposal. This example shows that modeling decomposition and refinement relations and implying certain outcomes can accelerate the decision making process (via triggers and production rules) and improve the quality of the decision making (via integrity constraint checks).

Decision log. In Section 4.3, we left the motivating case study from Chapter 2 at the end of the solution outline phase, with nine decisions made (see Table 10 and Table 11 on page 66). Table 23 shows the decision log at the end of the macro design phase. The decisions from Table 12 on page 66 have now been made as well:

Table 23. SOA decisions in motivating case study made in macro design

Issue (Outcome Instance)	Chosen Alternative	Justification Examples
IN MESSAGE GRANULARITY (all service operations, default)	COMB PATTERN (see below)	API convenience, verbosity not a problem (low volumes)
IN MESSAGE GRANULARITY (assess risk operation)	DOTTED LINE PATTERN (see below)	Legacy system constraint in government information server interface (see Chapter 2)
OUT MESSAGE GRANULARITY (all service operations, default)	COMB PATTERN (see below)	API convenience, verbosity not a problem (low volumes)
OUT MESSAGE GRANULARITY (assess risk operation)	DOTTED LINE PATTERN (see below)	Legacy system constraints (see Chapter 2)
OPERATION-TO-SERVICE GROUPING	SINGLE OPERATION	Command pattern followed (tacit knowledge, experience)
MESSAGE EXCHANGE PATTERN	REQUEST-REPLY	Consumer semantics (see Chapter 2)
INVOCATION TRANSACTIONALITY PATTERN	TRANSACTION ISLANDS	Long running process, backend slow and not transactional
INTEGRATION PARADIGM	ESB (BROKER)	Heterogeneity (see Chapter 2)
SERVICE COMPOSITION PARADIGM	WORKFLOW	Already decided in Section 4.3
RESOURCE PROTECTION STRATEGY	SYSTEM TRANSACTIONS, BUSINESS COMPENSATION	See [Fow03] and [LR00]

SERVICE PROVIDER TRANSACTIONALITY (ST)	ST-N	Implied by TRANSACTION ISLANDS choice
COMMUNICATIONS TRANSACTIONALITY (CT)	CT-N	Implied by TRANSACTION ISLANDS choice
PROCESS LIFETIME	MACROFLOW	Business rules from Chapter 2
PROCESS ACTIVITY TRANSACTIONALITY (PAT)	PAT-J	No business or technical need for MULTIPLE BRIDGES
TRANSPORT PROTOCOL BINDING (ESB gateway, internal ESB upstream to process)	SOAP/HTTP	Ubiquity, Internet self service
TRANSPORT PROTOCOL BINDING (internal ESB, downstream to backend systems)	PROPRIETARY DATABASE, MQ INTERFACE, FILE TRANSFER	Legacy system constraints
SOAP COMMUNICATION STYLE	DOCUMENT/LITERAL	Recommended by WS-Interoperability organization
XML SCHEMA (XSD) CONSTRUCTS	CUSTOM SUBSET	Sufficient for the domain data model to be exposed
MESSAGE EXCHANGE FORMAT	SOAP	Standardized, interoperable, supported by open source engines
INTEGRATION TECHNOLOGY	WS-* WEB SERVICES	Interoperability proven, standardized, tool supported
WEB SERVICES API	JAX-WS [SunWS]	Standardized, flexible, tools available
JAVA SERVICE PROVIDER TYPE	PLAIN OLD JAVA OBJECT (POJO)	Simplicity
TRANSPORT QOS	PLAIN SOAP (WITHOUT WSAT)	Implied by TRANSACTION ISLANDS choice
SCA QUALIFIERS	See Table 18 in Chapter 6 (page 104)	Implied by TRANSACTION ISLANDS choice
WORKFLOW LANGUAGE	BPEL	Standardized, tools available
BPEL VERSION	2.0	OASIS specification, used by engine selected in Section 4.3
COMPENSATION TECHNOLOGY	BPEL COMPENSATION HANDLER	Using standards is required as per NFR 2 (see Chapter 2)
ESB PRODUCT	CUSTOM	No license costs, available development skills, availability of an in-house command interface
ESB TOPOLOGY	CLUSTERED	High availability and failover requirements
INVOKE ACTIVITY TRANSACTIONALITY	See Figure 26 in Chapter 6 (page 101)	Implied by TRANSACTION ISLANDS choice

We now walk through the table, resolving selected issues.

Micro process. We demonstrate the steps of the micro process by resolving the IN MESSAGE GRANULARITY issue (see Chapter 2 for requirements and Chapter 4 for motivation).

Activity A.1 is to understand that the issue deals with the structure of the message parts and the data types in the operation signature (see Chapter 4). Let us assume that decision drivers stated in the RADM for SOA and studied in activity A.2 include "service consumer API convenience, request message verbosity, and interoperability between Java and .NET". Also in activity A.2, the architect studies decision dependencies. For instance, the technology-specific issue of creating

XML SCHEMA (XSD) CONSTRUCTS for the operations defined in the WSDL contracts of the customer case, contract, and risk management service providers (see Figure 15 on page 65 as well as Figure 29 on page 117) has to be considered.

Let us further assume that the alternatives listed in the RADM for SOA include a "deeply nested complex type structure, representing the business domain model accurately (which we call COMB PATTERN)" and "several flat, serialized strings (which we call DOTTED LINE PATTERN)".[47] The architect reviews them in activity A.3. We skip activity A.4 in this example, assuming that no recommendation exists. Such recommendation can be added to the RADM over time once sufficient experience with chosen alternatives has been gained on industry projects.

Now switching from RADM content to requirements analysis (activity B.1 and activity B.2), let us assume that the architect investigates the business rules, NFRs, and legacy constraints from Chapter 2 and concludes that a rich domain model has to be exposed, that API convenience has a high priority, and that the verbosity concerns can be resolved. This is an example of decision rationale that can not be linked to the requirements in Chapter 2 exclusively, but is also justified by the experience of the architect. The architect decides for the COMB PATTERN as a default for all service operations (activity B.3). An exception is the assess risk operation: It uses the government information server interface which only works with scalar data (integers); this means that either the DOTTED LINE PATTERN has to be selected or that an ESB mediation must be introduced. The architect documents the two different decision outcomes and their rationale in activity B.4.

Having made this decision, the architect still has to communicate the decision outcomes to the development team (activity C.1) and to verify that the decision is actually implemented (activity C.2) and that this implementation is workable (activity C.3).

This walkthrough completes the coverage of the case study. In reality, the decision making would continue until project closure; many more decisions would have to be made. The industrial case studies featured in Chapter 9 are such projects. In [ZZG+08], we present another industrial case study, resolving and giving rationale for 35 decisions, e.g., about the integration issues listed in Table 23.

7.2.4 Discussion and Summary

In this section we presented SOAD step 6, which works with a managed issue list and a macro and a micro process for architectural decision making.

Justification. Unlike [HKN+07], we see an opportunity to support the managed issue list in a tool rather than a simple spreadsheet or wiki page: The decision making process enactment can be automated this way. Chapter 8 presents such tool. Macro and micro process, however, are designed for human consumption; they are not executable in some machine without human intervention.

[47] These patterns have not been published yet; we intend to do so in a future publication. This demonstrates that a RADM can serve as an intermediate step of pattern harvesting.

Assumptions. We assume decision dependencies and filtering information to be modeled so that entry points can be identified and clustering advice can be given.

Our macro and micro processes are not specific to SOA design; they fit into any method that defines roles and phases. Software engineering methods also introduce macro-level phases, but only define activities coarsely, e.g., "define software architecture" in RUP. Hence, the relation between method *phases* and refinement *levels* as introduced in Section 6.2 is worth investigating. In the SOA design example used in step 4 and in our definition of strict validity (Definition 6.12), we assumed that entry points reside on the conceptual level and that *refinedBy* relations with issues on lower levels imply *triggers* relations. This leads to a top-down macro process. This strong assumption has to be reconsidered when creating RADMs for other genres and styles, but also other project types and communities in EAD/EAI and SOA design. In such a case, vendor preferences, software procurement strategy, and quality of legacy code determine whether a top-down approach is feasible; often a meet-in-the-middle approach is required [ZKG04]. An example is a bottom-up design method starting from legacy decisions and technology and asset capabilities rather than patterns and requirements. In [ZKL+09] we show that the primary change required in such case is the modification of Integrity Constraint (IC) 5 from step 4, which implies *triggers* relations from logical *refinedBy* and *decomposesInto* relations (Section 6.2).

Consequences. Depending on the type of decision to be made in an instance of the micro process, we can select from a continuum of complementary techniques: Architecture Tradeoff Analysis Method (ATAM) [BCK03] can be leveraged in activity B.2. In B.3, simple recommendations, semi-structured Strengths, Weaknesses, Opportunities, Threats (SWOT) tables, Question, Option, Criteria (QOC) diagrams [MYB+91], Attribute-Driven Design (ADD) [BCK03], Decision Support Systems (DSS) [SWL+03], hands-on evaluations and formal scoring algorithms can be used. Templates, metamodels, and tools from the architectural knowledge management community can be leveraged in activity B.4 and in activity C.1. Various review and evaluation techniques supporting C.2 and C.3 have been proposed [BCK03]. With this integrative approach, SOAD refers the architect to existing techniques suited for particular issues and decision making activities.

Rather often, it will not be possible to make certain decisions that would be required, for instance if the related requirements have not been captured sufficiently. This is unavoidable (or even desired when following an agile process [Bec00]); however, deferring the resolution of an issue should be a conscious decision. It is possible to overrule the ordering proposed by the issue list manger and to backtrack if it turns out that a certain design does not work. Note that we did not define any what-if analysis capabilities yet; this would require metamodel extensions.

Next steps. The decision model is now complete and has been applied to the case study; it remains to be shown how SOAD supports decision enforcement (step 7).

7.3 Framework Step 7: Enforce Decisions

Step 7 in the SOAD framework introduces decision-aware model transformations as a novel solution to the decision enforcement problem:

How to enforce that made architectural decisions are respected during subsequent design activities and during development?
How to update design models and code according to outcome information in an architectural decision model?

Input to this step are the decisions made in step 6, i.e., outcome instances; its output comprises modified design models and/or generated code.

We present this step in the same way as the previous ones, starting with a brief review of the state of the art and the practice, and then progressing to presentation of solution, application to SOA design, and discussion of rationale.

7.3.1 State of the Art and the Practice

State of the art. Software engineering processes like RUP [Kru03] advise architects to enforce decisions by refining the design in small and therefore actionable increments. The agile community emphasizes the importance of face-to-face communication and team empowerment [Bec00]. This advice is human-centric.

Model-Driven Development (MDD) can help to automate decision enforcement partially. MDD pertains to multiple project phases from analysis to design, development, and test. One of its objectives is to ensure consistency between the artifacts created throughout the phases: Model definitions and transformations become key elements of the development process. The standard *Model Driven Architecture (MDA)* approach is based on the Meta Object Facility (MOF) [OMG03].[48] Model-Driven Software Development (MDSD) as introduced by Stahl and Voelter [SV06] uses modeling and code generation in a flexible and pragmatic way. Brahe et al. present a model transformation chain for process-enabled SOA starting from business process models created by domain experts and then incrementally refined during development [BB06].

State of the practice. Informal techniques such as coaching, architectural templates, and code reviews dominate today. Maturity models such as the Capability Maturity Model Integration (CMMI) [SEI] and genre-specific governance models [AH05] recommend rigid approaches to ensure that decision outcome materializes, e.g., formal reviews. All of these techniques are valid and relevant in our SOA design context. However, applying them takes time and their success depends on the architects' coding and leadership skills. Some of the techniques are difficult to apply in distributed teams.

[48] From now on, we use the generic term MDD in this section. We require a metamodel for design models to exist; hence, our technique works with MDA and other forms of MDD.

We are not aware of any MDD implementations that respect project phases and roles defined by software engineering processes like RUP. As a consequence, it is not clear when in the process to apply which transformation and who is responsible for doing so. Furthermore, model transformations often are hard to adjust according to project-specific architectural decisions [ZKL06]. For example, many commercial BPM-to-BPEL transformation tools allow the user to make simple decisions, e.g., regarding activity naming, but use fixed defaults for architectural concerns, e.g., system transaction management boundaries [ZGT+07]. Consequently, development resources have to be invested to change the defaults to the settings required in a particular design.

OpenArchitectureWare [OAW] is an MDSD framework targeting developers. OpenArchitectureWare transformations are configurable. However, architectural decisions are not a genuine modeling concept and transformation input at present.

7.3.2 Concept: Decision Injection in Model-Driven Development

As an additional form of decision enforcement[49] in an MDD context, we bridge the gap between design and decision models. We provide a concept to *inject* outcome information into design, code, and deployment artifacts created by model-to-model and model-to-text transformations. This unidirectional and therefore partial automation helps to ensure that design models reflect the architectural decisions made. It is complementary to the existing practices introduced in Section 7.3.1.

Two concepts are required to realize decision injection: First we establish an MDD concept for SOAD decision models. This allows us to let SOAD decision models and MDD design models interact with each other in a second step.

MDD concept for SOAD Architectural Decision Models (ADMs). Two existing concepts make it possible to develop model transformations, *metamodeling* and *platform model(s)* [OMG03]. The SOAD formalization from Chapter 6 qualifies as a metamodel (although it is not based on MOF). Platform models for SOA as an architectural style exist, e.g., SoaML [OMG], as well as platform models for technology platforms such as Java Web services [SunWS].

Figure 33 illustrates a two-step refinement hierarchy and model transformation chain for SOAD decision models. In SOAD, the conceptual level in the RADM for SOA introduced in Chapter 5 and formally defined in Chapter 6 serves as *Platform-Independent Model (PIM)*; technology level and vendor asset level are two types of *Platform-Specific Models (PSMs)* [OMG03]. Unlike in MDD, these models are decision models capturing architectural knowledge rather than design models comprising components and connectors: The decision model elements (i.e., issues, alternatives, and outcomes) are instantiated from the SOAD metamodel.

Figure 33 also shows two model transformations. Exemplary SOA *marks* [OMG03] are shown as well; in MDA, such marks steer the transformations. Se-

[49] Although enforcement sounds authoritative, we do not imply any leadership style here.

lecting a mark is making a decision. Our exemplary marks map concepts from the SOA Definitions 2.6 to 2.9 to selected Web services platform elements.

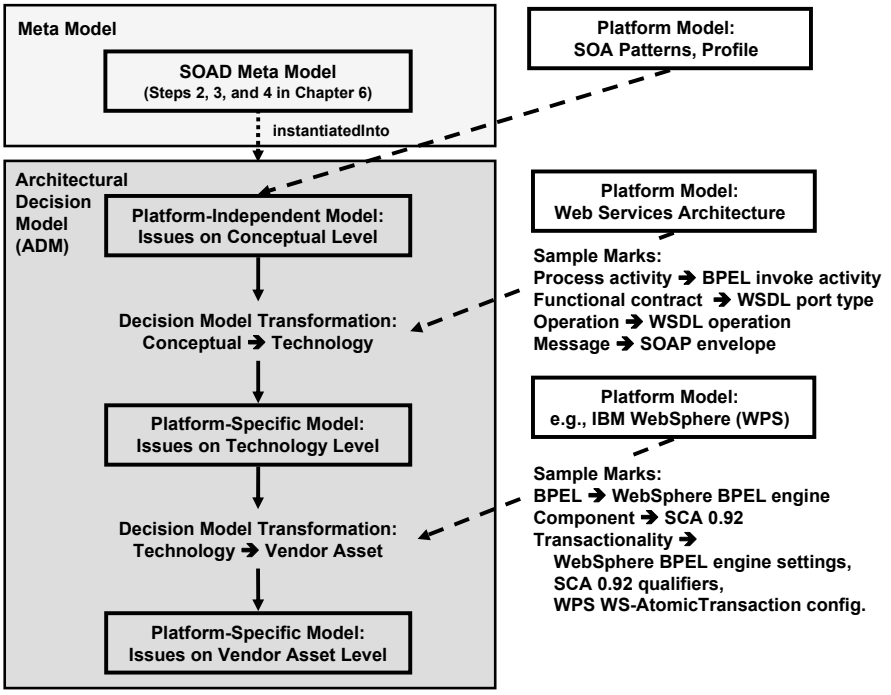


Figure 33. Model transformations in ADM

The *refinedBy* relations defined in Section 6.2 can be leveraged to implement certain transformations between outcomes on different levels, possibly requiring additional input from the architect. If this is the case, automation is partial. Section 6.2.3 provides an exemplary specification of a transformation related to transactional workflows in SOA. As demonstrated in this example, lower-level decisions can be implied in some cases, e.g., leveraging *forces* and *isIncompatibleWith* relations (production rule 2 in Section 6.3.2). Feasibility depends on the modeling choices made by the knowledge engineer and the characteristics of the application genre a decision model is created for: The more variability exists on a lower level (e.g., technology level, vendor asset level), the fewer opportunities exist to imply decisions and the stronger is the need for configurable model transformations.

Having aligned SOAD and MDD, we can connect decision and design models.

Decision injection (conceptual design). We formalized ADMs in Chapter 6 and provided an MDD concept for them above; hence, ADMs can now be used in model transformations and code generation via *decision injection*. Figure 34 extends Figure 33 with decision-aware design model transformations operating on a platform-specific level (i.e., technology level, vendor asset level). This is a realization of the conceptual decision enforcement component in Figure 14 on page 62 in Chapter 4. It works for ADMs only as it uses outcome instances as input.

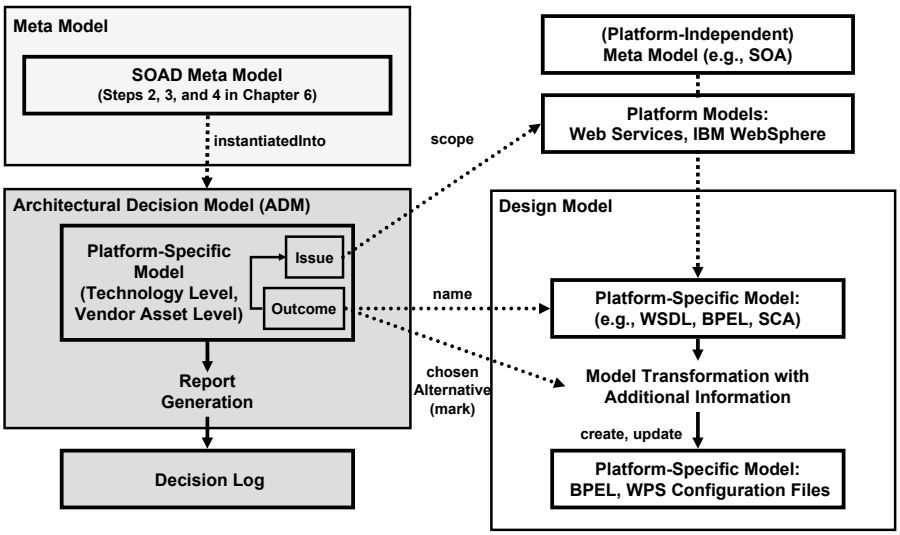


Figure 34. SOAD step 7: Decision injection into design models and code

Two types of models appear in the figure, SOAD ADMs (shown on the left side) and MDD design models (shown on the right side). The design model notation can be UML or a domain-specific language. The *name* attribute of outcome instances in the SOAD metamodel is leveraged to identify design model elements (e.g., UML classes); the *scope* attribute of an issue specifies the design model element type (e.g., UML *stereotypes* defined in a platform model or UML profile). The outcome instance parameterizes a design model transformation as desired.

One important form of model-to-text transformation is *report generation*. Also refining Figure 14, this transformation creates a *decision log* as an architecture documentation artifact. With this generation capability, existing decision capturing templates such as those discussed in Section 6.1 can be supported by SOAD. In this context, the decision log is a text page that reports on the ADM content. An example of such report is shown in Appendix B. The report is an HTML decision log that adheres to the guidelines in the artifact description of IBM UMF [CCS07].

The decision injection concept can be implemented in many modeling environments and transformation frameworks. We outline an SOA example now.

7.3.3 Sample Application to SOA

To demonstrate that the presented SOAD-MDD alignment and decision injection work for SOA design, we implemented a demonstrator for decision injection that works with resolved INVOKE ACTIVITY TRANSACTIONALITY issues. According to Chapter 6, this is an architectural decision dealing with a non-functional aspect of executable business processes. The scope of the issue is “invokeActivity”, a term from the BPEL specification. The demonstrator injects an alternative chosen in an

outcome (e.g., “requiresOwn” or “participates” from Section 6.2.3) into a BPEL file [OAS07], which is a technology-specific development artifact:

```
injectTransactionAtttribute(inout: bpelXmlFile, in: outcome)
```

The demonstrator is called from a SOAD tool; it implements the above function, which injects chosen alternatives (see above) into BPEL code, which may have been generated by a BPM tool that supports BPEL. The BPEL specification takes the role of the platform model; the sample marks were shown in Figure 33.

7.3.4 Discussion and Summary

In this section we presented a novel concept for decision enforcement (SOAD step 7), connecting SOAD decision and MDD design models via decision injection.

Justification. As pointed out in step 3, there is a natural affinity between our refinement levels and MDA model types. We let architects regain control over the model transformations, which is faithful to the original vision of MDA, (e.g., “additional information” in [OMG03]). Unnecessary development can be avoided and architectural consistency can be ensured this way. To preserve the decision rationale in the design, excerpts from the decision model such as the value of the justification attribute can be injected into the design model in addition to the chosen alternative. Traceability from requirements to design models and code can also be realized. However, our solution does not prevent design models from being updated erroneously after a decision has been injected as it is unidirectional (i.e., injection from decision model into design model, but not in the opposite direction).

Assumptions. We assume an MDD approach to be followed: A design metamodel, a platform model, and marks must be available. For SOA and Web services, this is the case. Moreover, decisions must be scoped (see step 2 and step 5).

Consequences. Our approach seems to violate the separation of concerns principle for tool design as we let decision and design models interact; these models represent different viewpoints on software architecture (decision model: knowledge viewpoint, design model: traditional viewpoints, e.g., 4+1 [Kru95]). In the current state of the art and the practice, these models are often isolated from each other, with the consequence that inherently existing relations remain tacit. In our approach, they are made explicit and can therefore be managed by tools.

We believe that our approach has a better chance to succeed than traditional MDD implementations: It increases flexibility and configurability and integrates architectural knowledge and design process information into the model transformations. However, the practicality of decision injection remains to be shown: Unlike the previous SOAD steps, we implemented the described concepts in demonstrator form, but did not validate them in industrial case studies (see Chapter 9).

Next steps. All SOAD framework steps are now introduced conceptually. In the next chapter, we present a collaboration system providing tool support for them.

8 A Collaboration Tool for Architectural Decision Modeling

In this chapter, we present the conceptual architecture of a collaboration system supporting the SOAD framework steps and concepts. Design and implementation of the collaboration system form the final contribution of this thesis:

Which logical building blocks comprise a tool that supports architects when they investigate, make, and enforce architectural decisions?
How to support collaborative creation and usage of decision models?

The collaboration system works with the input to steps 1 and 5, patterns (step 1) and analysis and design models (step 5). It produces a partially or fully decided Architectural Decision Model (ADM), which is instantiated from the SOAD metamodel supporting steps 2 to 4. The collaboration system can be used to document decisions made in step 6 and to generate reports and inject decision outcomes into design models as described in step 7.

The chapter starts with a brief review of state of the art and the practice, and then progresses to design, implementation, example, and discussion of rationale.

8.1 State of the Art and the Practice

State of the art. Active usage of decision models in the design process is a novel approach; therefore, no tools specifically designed for this purpose exist. However, more general architectural decision capturing tools have been proposed.

In the 1990s, Knowledge-Based Software Engineering (KBSE) tools such as Argo stressed that tools for designers should support their cognitive needs such as reflection in action, opportunistic design, comprehension and problem solving [RHR96]. To achieve this, a *managed to do list* was seen as one of several key features. At that time, regulatory compliance and team collaboration forces were not as dominating in software engineering as today; aspects specific to these forces were not addressed explicitly. KBSE did not provide any support specific to our particular knowledge domain (recurring architectural decisions), application genre (enterprise applications), or architectural style (SOA).

Architects' Workbench (AWB) [ABK+06] is an Eclipse plugin that supports the IBM Architecture Description Standard [YRS+99] in its metamodel. AWB provides two UMF-conformant viewpoints for architectural decision modeling. Due to its powerful refactoring capabilities, AWB is well suited for architectural

decision knowledge capturing. It can generate reports. However, it was not designed for knowledge exchange and team collaboration.

PAKME [AGJ05] is the prototype of an architecture knowledge management system implemented on top of an existing groupware platform. It uses 25 tables to capture various forms of architectural artifacts, including design rationale. PAKME is populated from patterns repositories and the literature. Jansen [Jan08] and Falessi [FBC06] present several other tools for the management of architectural knowledge. Being passive knowledge repositories, these tools do not support the SOAD concepts (see Table 9 on page 59 for an overview of these concepts).

As potential building blocks for our solution, we also evaluated related assets such as UML tools, native HTML, and standard wiki technologies. None of these assets meets all requirements from Chapter 3: UML tools specialize on capturing analysis and design models such as use cases, class, activity, and sequence diagrams [RJB99] graphically in the form of diagrams. They fall short when it comes to modeling knowledge comprising text, often semi-structured and combined with other formats, e.g., images and URLs, to capture design intent and rationale. Native HTML and standard wikis provide flexible human user interfaces when designed and configured appropriately. Many development project teams already use standard wikis for collaboration and information sharing. However, standard wikis store their content unstructured and/or blended with presentation elements (which are defined in HTML or a wiki language). Typically there is no communication or programming interface allowing other tools to access the content apart from the HTML data sent to the browser via HTTP. Thus, it is difficult to populate the system from third party software or to extract any well-structured content for automatic processing. This is required to support an active issue management as defined in Chapters 6 and 7.

State of the practice. Eclipse plug-ins [Ecl] and standard wikis represent the state-of-the-practice in decision modeling and knowledge exchange tools. Text-based approaches to designing architectures and sharing rationale are common as well: Templates defined in Word processors, HTML forms, or groupware databases are frequently used for decision capturing. Much of the knowledge remains tacit.

8.2 Conceptual Design of an Application Wiki for SOAD

We believe that a lack of collaboration and reuse features and a lack of active guidance during the design as envisioned by Argo are two of the deficiencies of existing approaches. We already outlined the architecture of a tool that provides such features in Chapter 4. We now refine this architecture and add collaboration and issue management capabilities: Unlike passive knowledge management repositories and templates designed for decision capturing, we facilitate the decision making process and, faithful to Argo's vision of a managed to do list, make context-specific architectural knowledge available during the design process.

Architecturally significant requirements. The architectural knowledge management requirements from Chapter 3 (obtain, tailor, delegate, involve, make, enforce, and share, R5-1 to R5-7) are the primary use cases for the tool; the concepts we developed for the seven SOAD steps provide detailed functional requirements. The tool must focus on the design phase [Som95]. Discussion and interaction support, e.g., via email, comments and issue tracking, document management, and versioning are important functional requirements shared with existing wiki-like collaboration systems. The tool must integrate with others as outlined in Figure 10 on page 39. A communication or programming interface should be provided so that import and export mechanisms can automatically populate the tool with issues and outcome instances, e.g., those identified in analysis and design tools. The system must be user friendly: Practitioners do not appreciate having to work with yet another tool to fulfill additional obligations such as decision capturing. It must be intuitive to browse the content, and users should be attracted to contribute new knowledge (R1-6, R1-7). User management including simple workflow and basic security support (i.e., authentication, authorization) is required if decision making responsibilities are shared within and between teams (R3-5). A thin client eases deployment and remote access. The tool must support frequent and incremental updates of RADMs and ADMs so that knowledge engineers can keep the knowledge about issues up to date, e.g., by adding rationale gathered on successful and failed projects that completed after a RADM or ADM was created.

Conceptual architecture (logical viewpoint). Our key concept is to use an *application wiki*[50] as the collaboration system, realizing the tailor, delegate, make, and enforce use cases specified in Chapter 3 (Section 3.1.5) in dedicated application logic. Standard wiki features such as user-generated content and comments (discussion forums) realize the involve use case. Providing import and export capabilities, such an application wiki can also facilitate an exchange of architectural decision knowledge, which realizes the remaining use cases, obtain and share.

Architectural Decision Knowledge Wiki is such a Web-centric collaboration system, providing explicit support for sharing architectural decision models. Its architecture combines the benefits of a rich Web 2.0 [SZP07] front end with those of the domain model pattern [Eva03] and a Relational Database Management System (RDBMS) [SKS02].

To refine the functional view from Figure 14 into a logical component model we use *layers* as our governing architectural pattern [BMR+96]. This allows us to evolve the layers independently of each other, and to integrate our solution with other tools. The three layers of Architectural Decision Knowledge Wiki are: *presentation layer*, *domain layer*, and *persistence layer* [Fow03]. The metamodel from Chapter 6 affects all layers: the metamodel elements topic group, issue, alternative, and outcome (for an overview, see UML metamodel in Figure 20 on page 88 and example in Figure 21 on page 89) are represented by presentation layer (user

[50] An application wiki combines a wiki engine with an application server [SZP07]. It extends the user and page management capabilities of standard wikis with application server extensibility and a mash-up (composition) interface. This allows creating and managing page content programmatically, e.g., with the help of a custom database.

interface) components, related domain layer logic, and corresponding database tables.

The tool architecture provides components that support the SOAD steps and concepts defined in this thesis. Figure 35 refines Figure 14 on page 62 and illustrates the architecture and tool context of Architectural Decision Knowledge Wiki:

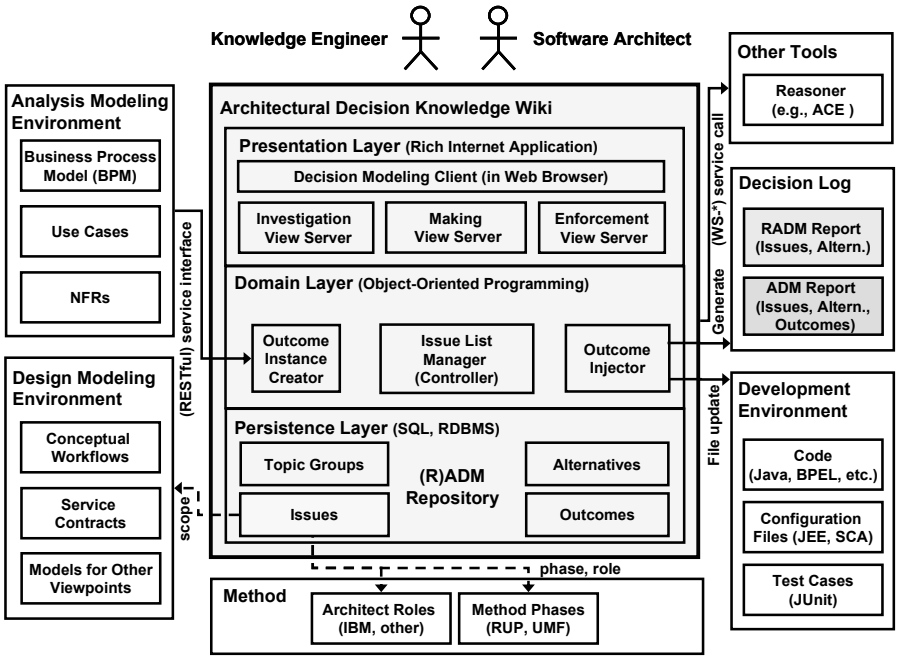


Figure 35. Component model of Architectural Decision Knowledge Wiki

Via a common decision modeling client, which runs inside the users' Web browser, the presentation layer exposes three server-side views for steps 1 to 5 (which we jointly refer to as investigation), 6 (making), and 7 (enforcement). These views refine those introduced in Figure 14. The presentation layer also provides collaboration features such as page editing and versioning (allowing users to create and update content). Attachments can be added to wiki pages to explain decision drivers, alternatives, and other aspects in more detail than in the (R)ADM content. The presentation layer is realized as a Rich Internet Application (RIA), which combines the advantages of a thin client (e.g., no installation effort on the client side, all users have access to the same ADM on the server) with those of a rich client (e.g., usability). With this presentation layer design, an entire project team can share one ADM and participate in the decision making.

Dependency management is a key domain layer responsibility: The domain layer component *issue list manager* implements the concepts from Chapter 6 and 7, Section 7.2 in particular (e.g., managed issue list based on dependency relations such as *refinedBy*, *decomposesInto*, *forces*, and *triggers*). It also provides import and export capabilities: A service interface organized according to the service layer pattern [Fow03] allows clients to create outcome instances via an outcome

instance creator. Decision injection (Section 7.3) is supported with a file update interface (outcome injector). Additional integrations can be provided by this layer as well.

The persistence layer implements the RADM and ADM repository as a relational database. Hence, the extensive capabilities of a RDBMS can be leveraged, e.g., to create decision logs as database reports, to ensure the integrity of a model, and to query it. The RADM and ADM repository supports create, read, update, delete, and search operations for the RADM and ADM tables whose table definitions are derived from the SOAD metamodel (Chapter 6).

Furthermore, the architecture allows integrating Web services available on the Internet. One example is a reasoning service that accepts and returns Attempto Controlled English (ACE), a natural controlled language that is formally defined, but human readable [FS96]. Such reasoning service could implement the production rules defined in Chapter 6 (using *forces* and *isIncompatibleWith* relations).[51]

8.3 Implementation of the Conceptual Design

We implemented Architectural Decision Knowledge Wiki on top of a situational application and Web 2.0 mashup environment called QEDWiki [SZP07]. QEDWiki is an application wiki, i.e., a hybrid wiki engine and PHP application server, providing access to incoming HTTP request data via a command interface. QEDWiki is based on the Zend PHP Framework, which extends and runs inside the Apache HTTP server. It uses relational databases managed by IBM DB2. HTTP server and QEDWiki provide the required user authentication and authorization. Through predefined commands, QEDWiki provides support for adding comments, attachments, and email threads. We extended these commands to provide native support for the SOAD metamodel, using the Dojo JavaScript library to provide a user experience as attractive as that of rich clients (in the decision modeling client). An issue and all its alternatives are displayed in a single, composite QEDWiki page comprising multiple tabs (one tab per view server from Figure 35). The domain layer is implemented in object-oriented PHP. It accesses the persistence layer via the active record pattern [Fow03], requiring little coding effort.

We released a base version of Architectural Decision Knowledge Wiki on IBM alphaWorks [SZ08]. The tool has been used in several industrial projects and training classes. More than 200 users are registered in a company-internal hosted instance. 630 users downloaded Architectural Decision Knowledge Wiki in the first twelve months of public availability.

To demonstrate automatic outcome instance creation, an additional requirements management tool interface was implemented in demonstrator form (i.e., it was not released): This interface comprises an IBM WebSphere Business Modeler

[51] If SOAD decision drivers and recommendations and project requirements are articulated in ACE as well, the reasoning engine can also suggest certain alternatives to the architect.

(WBM) [IBM] to Architectural Decision Knowledge Wiki model transformation, demonstrating decision identification in analysis-phase business process models.

The decision injection concept (see Section 7.3) was also implemented in demonstrator form (and not released either): An interface from Architectural Decision Knowledge Wiki to IBM WebSphere Integration Developer (WID) shows decision injection in a BPEL editor via a local file update interface and XPath.

Finally, the macro process from Section 7.3 and the ACE interface were only implemented in experimental form (i.e., not released either).

User interface. We now present Architectural Decision Knowledge Wiki from the user's point of view, briefly describing the most important features of its user interface. Figure 36 shows the main page of the decision modeling client from Figure 35, displaying a decision modeling project overview in the Workspace explorer on the left and a single issue in the main part of the page:

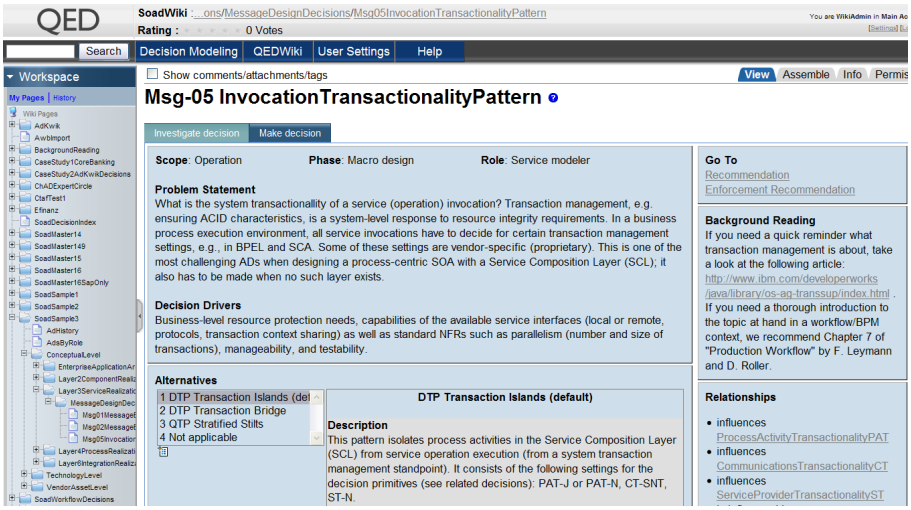


Figure 36. Architectural Decision Knowledge Wiki screen caption

The issue is the conceptual decision INVOCATION TRANSACTIONALITY PATTERN, which we introduced in Chapter 6. Several of the SOAD metamodel elements, e.g., issue attributes such as short name ("Msg-05"), name ("Invocation Transactionality Pattern") and problem statement ("What is the system transactionality …") are visible at first glance. Decision drivers are displayed in another text field ("Business-level resource protection needs …"); alternatives and their attributes are also displayed ("Transaction Islands"). In support of the involve use case from Chapter 3, literature links are provided under background reading. More detailed documentation about this and other attributes can be attached to the page.

Issues can influence each other; their dependency relationships are displayed as hyperlinks, e.g., from the currently displayed issue to "Process Activity Transactionality (PAT)" and to "Communications Transactionality (CT)". These issues were also introduced in Chapter 6.

The decision models are organized and displayed in a hierarchical structure and tagged to enable searches. Applying the master-details pattern, issues can be located with the Workspace explorer: Clicking on a Workspace explorer entry displays the details of a topic group or of an issue and its alternatives in the main part of the page. At the top of the page (to the right of the label "SoadWiki"), a link list provides an additional means of orientation, flattening the topic group hierarchy according to the breadcrumbs pattern. The list ends with the name of the topic group in which the issue is contained ("Message Design Decisions") and the issue name ("Invocation Transactionality Pattern").

The same user interface is used for decision investigation, decision making, and decision enforcement. Architects can not only identify issues from scratch, but also import an initial set, which supports the obtain use case from Chapter 3. Export features also exist, supporting the share use case also introduced in Chapter 3. Issues carry owner and status information to further facilitate collaboration.

8.4 Discussion and Summary

In this chapter we presented the design and implementation of Architectural Decision Knowledge Wiki. This application wiki and collaboration system supports the steps and concepts in the SOAD framework. Its model-driven design centers on a managed issue list and decision investigation, making, and enforcement views, which are exposed to users via a common decision modeling (browser) client.

Justification. Using a wiki as a presentation layer, Architectural Decision Knowledge Wiki makes decision models available via Web protocols. We combine existing concepts such as wiki, domain layer, and relational database in a way that is sound in our requirements context and novel in the tool development genre.

Assumptions. We assume a stable and agreed upon metamodel to exist. As discussed in Chapter 6, this is a realistic assumption. The SOAD metamodel has been in use since September 2006 and the architectural decisions template provided by the IBM Global Services Method (now called UMF) has been stable since 1998.

The design of Architectural Decision Knowledge Wiki does not assume that enterprise applications are developed using the SOA style. The tool has already proven to be able to model issues from other application genres (see Chapter 9).

User validation. Early adopters reported the combination of a wiki, a domain layer, and a relational database to be innovative and appealing. The decision investigation page design was appreciated. However, only few users returned and used the tool continuously throughout their projects. One explanation can be found in the non-functional challenges encountered during development of the prototype: Our decision to extend an already existing wiki engine caused a rather long installation procedure: Two hours are required to install the prerequisite software; up to two Gigabyte disk space is consumed. Moreover, there is no support for Linux and MySQL. As an application wiki, Architectural Decision Knowledge

Wiki requires users to be connected to the server; there are no offline capabilities. This was criticized by users working in professional services firms, which do not always have access to project- or company-wide server infrastructures from their laptops when traveling.

We identified several change cases due to the feedback of the early adopters. For instance, the hierarchical level and topic group tree shown in the Workspace explorer appearing on the left side of Figure 36 was not well received by inexperienced practitioners. They reported orientation problems in large models. Furthermore, the display of model elements that are formatted according to the SOAD metamodel requires the user to read a lot of text in the current implementation. In response, we designed an additional *ADIssue Status Overview* view (not shown in Figure 36). This view provides one window element (tab) for each decision status type from Section 6.3, i.e., entry points, eligible, pending, and implied issues and outcome instances. The user navigates from issue to issue via *refinedBy* and *decomposesInto* relations.

Early adopters also requested better integration with other tools used by architects, for example the analysis and design modeling environments and development platforms shown in Figure 10 on page 39 and Figure 14 on page 62, as well as emerging team collaboration platforms such as Jazz [Jaz]. Due to the positive overall reactions and the confirmation that the realized use cases are valuable, we consider implementing such features in our future work.

Next steps. In the following Chapter 9, we present how we validated SOAD.

Related publications

An earlier version of our decision modeling tool is described in [SZP07].

The domain metamodel, decision processing steps, and use cases of Architectural Decision Knowledge Wiki are also described in an article targeting practitioners [ZSE08].

9 Validation of Research Results

In this chapter, we demonstrate how we validated the SOA Decision Modeling (SOAD) framework, the Reusable Architectural Decision Model (RADM) for SOA, and Architectural Decision Knowledge Wiki regarding their practical value and usability. First we clarify objectives and scope of the validation and present our approach in a validation overview (Section 9.1). Next, we assess whether the requirements for SOA design methods stated in Chapter 3 are satisfied (Section 9.2). After that, we present five industrial case studies; two of these case studies involved action research (Section 9.3). We also feature supplemental evaluation techniques such as self experiments, industry workshops, teaching, and implementation (Section 9.4). Finally, we summarize the validation results (Section 9.5).

9.1 Validation Overview

In this section, we clarify objectives and scope of the validation activities. We outline our approach and its rationale, and give an overview of the validation results.

9.1.1 Objectives

Research contributions in software engineering must be validated. Important validation objectives are to demonstrate *technical feasibility*, to confirm the *practical value* for the target audience, and to evaluate the *usability*. A validation of the monetary value and business benefits such as opportunities to increase revenue or reduce cost would be required when creating a business case for the development of a commercial version of our solution. While we touched upon such aspects occasionally, such an analysis was not a primary goal of the thesis validation.

To demonstrate the technical feasibility of the SOAD concepts, we created the RADM for SOA introduced in Chapter 5 and implemented Architectural Decision Knowledge Wiki, the collaboration system (tool) presented in Chapter 8. Practical value and usability remain to be evaluated, i.e., whether practicing architects are willing and able to apply SOAD and whether such application is beneficial.

9.1.2 Approach and Rationale

The problem solved in this thesis is the creation of a decision-centric SOA design method. Due to the design nature of this problem, *validation by experience* is an adequate validation type, as opposed to analysis with formal proofs or controlled experiments [Sha03].

Validation activities. A requirements self assessment, industrial case studies, and implementation served as our primary validation activity types. Supplemental activities were self experiments, teaching, and industry workshops.

Our *requirements self assessment* is based on the method requirement catalog we established in Section 3.1. The assessment is presented in Section 9.2.

We validated SOAD in five *industrial case studies* which are featured in Section 9.3. The case studies primarily focused on confirming our key hypothesis that architectural decisions recur and can be modeled according to a metamodel. The requirements catalog served as a source of validation criteria for these case studies. We also conducted a user survey. Particularly relevant quality attributes were:

- *Functionality*, e.g., are the issues and alternatives relevant and accurately described? Is the captured architectural knowledge useful during design?
- *Usability*, e.g., is the RADM for SOA well organized so that issues can be located easily, is the collaboration system (tool) straightforward to work with? Is the tailoring effort manageable?

We applied *action research* [ALM+99] in two of the case studies. This is a concept with roots in pedagogical research: The researcher joins a project and influences it actively, for instance as coach, pacemaker, or technical reviewer. This is different from exposing selected research results to users and merely observing them (this would be done in a controlled experiment). Applying action research allowed us to experience the practical applicability of our concepts ourselves and to interact with and learn from other architects while they used SOAD.

We hosted a company-internal test instance of the *implementation* of Architectural Decision Knowledge Wiki and made the tool available for external download [SZ08]. For several advanced concepts, we implemented prototypical tool support without reaching out to practitioners.

Another validation activity was to conduct *self experiments*. For instance, we applied SOAD to our own SOA projects retrospectively [ZMC+04, ZDG+05]. We also used framework, RADM for SOA, and Architectural Decision Knowledge Wiki for *teaching*. Additional practitioner feedback was collected regularly through active participation at various *industry workshops*. These activities helped us confirm the state of the practice, to evaluate SOAD in several states of evolution, and to ensure that the developed concepts are applicable for software architects independent of their area of expertise, experience, and company affiliation.

Organization. Given the rather broad scope of the thesis, the validation activities had to be decomposed into controllable and observable parts. We organized the validation activities by contribution type (i.e., SOAD framework steps, RADM for SOA, and tool) and by use case (i.e., education, knowledge exchange, design

method, review technique, and governance instrument). This approach made the communication with the early adopters in the industry efficient and allowed us to confront them with the detailed research questions from Chapter 3. Table 24 summarizes the validation objectives and activities conducted.

Table 24. Validation overview

Validation Criterion	Primary Validation Activities	Secondary Validation Activities
Technical feasibility	Concepts used to create RADM for SOA content (389 SOA decisions); other knowledge also modeled (see Section 9.4); several tool implementations (see Chapter 8 for tool design and Chapter 10 for evolution)	
Practical value	Gaps between state of the art and the practice identified in Chapter 2 and assessed in Chapter 3; requirements self assessment conducted and presented in Section 9.2; practical value of SOAD concepts indicated in Chapters 5 to 7 and validated in industrial case studies (Section 9.3)	
Framework step 1: *Identification technique* *RADM for SOA*	 Case study 3 Case studies 1, 2, 3, 4, 5	 Self experiments Teaching, industry workshops
Framework step 2: *SOAD metamodel and capturing template*	 Case studies 1, 2, 3, 4, 5	 Knowledge from other domains modeled, industry workshops
Framework step 3: *Level/layer structure*	 Case studies 2, 3, 4	 Self experiments
Framework step 4: *Ordering concepts*	 Case study 3	 Prototypical implementation
Framework step 5: *Decision filtering*	 Case study 3, 4, 5	 Prototypical implementation
Framework step 6: *Managed issue list* *Macro process* *Micro process*	 Self experiment, case study 4 Case study 4, 5 (basic form) Case studies 1, 2, 4, 5	 Prototypical implementation Prototypical implementation Teaching
Framework step 7: *Decision injection*	 Prototypical implementation	 –
Collaboration system (tool)	Case studies 3, 4, hosted wiki instance, public release	Walkthrough with practicing architects, classroom training
Additional use cases: *Education* *Knowledge exchange* *Review technique* *Governance instrument*	 Teaching (practitioners) Hosted wiki instance (tool) Case study 5 Additional case study	 Guest lectures, industry workshops Self experiment (Section 9.4) Smaller cases (Section 9.3.6) Demonstrations to target audience

Table 24 shows that all framework steps except for step 7 were validated by experience. Design method usage (step 6) was only partially validated because adopting a new, immature method has a significant impact on the technical project risk, and industry projects operating under tight economical constraints can not be expected to make such strong commitment. The only validation activity for decision injection (step 7) was the implementation of a prototype due to the limited adoption of model-driven development on projects suited to apply SOAD (see discussion in Chapter 10). The additional use cases such as review technique and instrument were validated partially.

Rationale for validation approach. Our validation approach is in line with Shaw's recommendations: Our validation type is "experience", with the objective to show "correctness, usefulness, and effectiveness" of our concepts [Sha03].

Our validation activities gave us direct access to the target audience and had a short feedback loop, which allowed us to employ an *iterative and incremental* concept development and validation approach. This had the objective to verify (or falsify) the hypotheses expressed by the research questions from Section 3.2 continuously throughout the project: Six months after project initiation, we developed a demonstrator, followed by the implementation of a working tool prototype. This prototype was enhanced over a two year time span. This allowed us to expose new concepts to practicing architects rapidly throughout the project. The preliminary validation results were used to improve subsequent versions of SOAD framework, RADM for SOA, and tool prototype.

We continued to validate until we had sufficient evidence that the fundamental hypothesis that issues recur holds true and that the core concepts such as a common metamodel and refinement levels work in practice. The justification for conducting five case studies is that the selected projects yielded a reasonable coverage (in terms of breadth and depth) without causing unmanageable validation efforts for the involved researchers and the case study participants from the industry.

Overview of validation results. All five use cases of the SOAD framework and RADM for SOA were seen to be relevant and not covered by existing assets properly. The asset creation phase (steps 1 to 4) and model tailoring steps 5 were seen to be useful and practical. SOAD was used in design method support role successfully, but not as a standalone method (step 6). The implementation of the decision injection concepts demonstrated technical feasibility (step 7). The core functionality of the tool (see Chapter 8) was well received. For instance, the display of issues and alternatives in a single page was considered useful. Usability challenges were reported for large models and when accessing the system from remote.

We present the validation results in detail in the remainder of the chapter, starting with the requirements self assessment, followed by industrial case studies, and additional validation activities such as self experiments and teaching.

9.2 Method Requirements Coverage

In this section we assess SOAD with regards to the SOA design method requirements established in Chapter 3. This *fit-gap analysis* is structured like the requirements catalog: General software engineering method requirements are evaluated first, followed by software architecture design method requirements, requirements specific to enterprise application development and integration, and those specific to SOA design. Requirements for capturing and sharing architectural knowledge come last.

9.2.1 General Requirements for Software Engineering Methods

Table 25 assesses whether SOAD meets the general software engineering method requirements stated in Section 3.1.1.

Table 25. Software engineering method requirements coverage

Requirement	SOAD	Assessment
R1-1: Method = process + notation + supporting techniques and content	Micro and macro process (step 6) + QOC diagram variant (step 2) + identification technique (step 1) + tailoring technique (step 5) + RADM for SOA content	Met, can be combined with existing methods to fill their gaps described in Section 3.3. See Chapter 10 for detailed comparison and positioning.
R1-2: Provide standard description format, metamodel, or formal underpinning	UML metamodel (step 2) and formalization (steps 3 and 4)	Met for architectural decision knowledge, extending existing work for new usage context
R1-3: Be broadly applicable and actionable, e.g., provide templates and examples	All SOA layers covered, template available (step 2), examples given (steps 2 to 4, step 6)	Met
R1-4: Provide link between requirements engineering (analysis) and design work	Scope attribute in metamodel, outcome instances	Met conceptually and in tool; instance creation only implemented as demonstrator
R1-5: Provide link to project management methods	Phase and role attribute in metamodel	Met conceptually and in content; basic implementation
R1-6: Ease method content authoring (extensibility)	Identification technique, meta issues, integrity constraints, heuristics, decision capturing advice	Met conceptually; basic support in tool: create and update operations, decision modeling guidance
R1-7: Be consumable and comprehensible, provide tailoring means (usability)	Existing templates extended, step 3 model structure (levels, layers), decision filtering (step 5)	Partially met (improvements required to make tool more user friendly, e.g., graphical views)

The main objective of SOAD is to complete existing general purpose and SOA design methods. Hence, it does not propose a new software engineering process or architecture design notation (R1-1); it rather defines two decision making processes (step 6). The feedback from practicing architects and method creators confirms that this integrative approach works and is beneficial (see Sections 9.3 and 9.4). On the case studies, SOAD was used in combination with other methods.

The other requirements are met by SOAD concepts and supporting information. For instance, the scope, phase, and role attributes of issues in the RADM for SOA refer to types of analysis and design model elements and method elements, respectively (R1-4). Content authoring is simplified with the pattern-centric identification technique (step 1), the integrity constraints and heuristics (step 3), and supporting documentation such as decision capturing advice (R1-6). The case studies demonstrated that the consumability (R1-7) goal can not be fully met if an unstructured catalog of decisions or simple topic group tree is provided; practitioners reported not to have enough time to study lengthy documents during project initiation. Model structure (step 3) and content tailoring features such as decision filtering (described in step 5) solve this problem conceptually.

9.2.2 Software Architecture Design Method Requirements

Table 26 shows how SOAD meets the architecture design method requirements stated in Section 3.1.2 (when being combined with existing work):

Table 26. Software architecture design method requirements coverage

Requirement	SOAD	Assessment
R2-1: Refine general purpose methods: Provide multiple architectural viewpoints	Topic groups for viewpoints can be defined, but viewpoints are not a genuine concept in our metamodel	Met with help of level formalization (step 3), met in RADM for SOA content
R2-2: Be driven by quality attributes and stakeholder goals	Text-based decision driver attribute in ADIssue in metamodel (step 2)	Partially met, quality attributes not modeled as genuine concept in metamodel
R2-3: Support decomposition of complex design issues (architectural analysis)	ADTopicGroup hierarchy, *refinedBy* and *decomposesInto* relations (step 3)	Met, rather fine grained decomposition approach based on the dependencies
R2-4: Support composition of resolved design issues (architectural synthesis)	Managed issue list, macro and micro process serve as SOA design method (step 6)	Partially met, but decision log is not a complete design (other artifacts still required)
R2-5: Define relationships between design issues and leverage them in method design	Rich dependency modeling capabilities, several types of relations defined in metamodel (steps 3 and 4) and used in managed issue list	Met, on issue and on alternative level
R2-6: Provide a managed to do list	Concepts for managed issue list (step 4, step 6)	Met, pre-populated with SOA-specific design issues
R2-7: Support architecture evaluation, feedback loops, and backtracking	Basic support via review use case for framework, report creation	Partially met, additional concepts and integration into existing methods required

Multiple viewpoints can be provided by defining topic group hierarchies (R2-1). We decided to make the viewpoint support customizable rather than define a single one in our metamodel because each architecture design method has its own viewpoint scheme.[52]

The decision driver attribute allows capturing quality attributes (R2-2); however, they play a less central role in our metamodel than in other methods. This does not mean that they are less important in our method; see discussion in Chapter 10 for further rationale.

R2-3 to R2-7 are met or partially met; dedicated SOAD concepts provide related support as indicated in Table 26.

9.2.3 Requirements Specific to the Enterprise Application Genre

Table 27 assesses whether SOAD meets the method requirements specific to the development and integration of enterprise applications (see Section 3.1.3):

[52] The IEEE 1471 standard suggests viewing architectures from multiple viewpoints and gives many examples, but does not norm any particular viewpoint scheme [IEEE07].

Table 27. EAD/EAI method requirements coverage

Requirement	SOAD	Assessment
R3-1: Refine architecture design methods for EAD and EAI: Support pattern-based architecture design	Patterns as conceptual alternatives (steps 1 and 3)	Met, core concept and benefit
R3-2: Align with analysis methods (e.g., BPM, OOA), enterprise architecture frameworks, and maturity models	Outcome instances in metamodel, service interface in tool, governance use case	Met conceptually, automatic population implemented in demonstrator only (not released)
R3-3: Cover integration of legacy systems and software packages	ADM can be used to analyze and assess existing assets	No legacy modernization decisions captured so far
R3-4: Support Model-Driven Development (MDD) concepts, use industry models	Metamodel, refinement levels separating concerns, decision injection (steps 3 and 7)	Met, makes MDD transformations configurable
R3-5: Align with contemporary commercial EAD and EAI project delivery and procurement practices	ADM can take governance role and can have contractual relevance; collaboration system supports multiple users	Met conceptually, advanced use cases (such as governance) only partially validated

A key strength of SOAD is its usage of and alignment with patterns (R3-1). R3-2 is addressed by the component-oriented tool architecture that provides a service interface to BPM, OOAD, and other tools; the metamodel design takes the required alignment into account. Additional architectural knowledge must be harvested from projects to meet R3-3. The existing RADM for SOA content does not focus on legacy modernization specifically; however, the SOAD modeling concepts are generic enough so that decisions related to legacy system evolution can be captured. R3-4 is met with the level organization from step 3 and the decision enforcement concept from step 7. SOAD extends the usage of MDD and MDA concepts such as separation of platform-independent from platform-specific design concerns and model transformations from design models to *decision* models. In response to R3-5, the Web-centric collaboration system is designed to support distributed teams. The governance use case of SOAD and the role and phase attributes can be leveraged when coordinating the outsourcing and offshoring of design and development activities.

9.2.4 SOA-Specific Design Method Requirements

Moving from EAD and EAI method requirements to SOA design, Table 28 assesses whether SOAD meets the requirements from Section 3.1.4:

Table 28. SOA design method requirements coverage

Requirement	SOAD	Assessment
R4-1: Refine previous three categories: Support *service* engineering process	RADM for SOA content covers all phases of service lifecycle; main focus is on realization	Met in method and content; collaboration system is SOA agnostic
R4-2: Define notation for multiple service contract dimensions	Not in scope, already met by existing assets such as UML profiles for software services, SOMA, and SCA	Not applicable, met in combination with other assets
R4-3: Integrate SOA princi-	Service consumer, provider, contract,	Met

Requirement	SOAD	Assessment
ples and patterns (Definitions 2.6 to 2.9)	ESB, service composition, service registry issues in RADM for SOA	
R4-4: Give advice regarding granularity and other SOA-specific design issues	Such issues exist in RADM for SOA, see examples given in Chapter 5, 6, and 7 (steps 1 to 6)	Met
R4-5: Cover service lifecycle management, e.g., ownership and versioning	Such issues exist in RADM for SOA (not featured in this thesis)	Met

SOAD does not introduce a new service model artifact, service lifecycle model, or business modeling and service identification techniques (R4-1, R4-2); it rather complements and completes existing SOA design methods such as SDLC [Pap08] and SOMA [AGA+08] with service realization advice (see discussion in Chapter 10 for detailed positioning). Architectural knowledge for all composite SOA patterns introduced in Chapter 2 (Definitions 2.6 to 2.9) is present in the RADM for SOA (R4-3). In addition to the examples used in Chapters 5, 6, and 7, the RADM for SOA covers executive and requirements engineering decisions, logical and physical architecture design, deployment, and governance (R4-4, R4-5).

9.2.5 Requirements for Architectural Knowledge Management

Table 29 assesses if and how SOAD framework and its tool support meet the knowledge capturing and sharing requirements from Section 3.1.5:

Table 29. Architectural decision capturing and sharing requirements coverage

Requirement	SOAD and Architectural Decision Knowledge Wiki	Assessment
R5-1: Obtain required knowledge	Importers provide basic support for obtaining knowledge; decision identification technique	Met in basic form (only one RADM at a time)
R5-2: Adopt identified knowledge	Create/read/update/delete, search features in tool, decision filtering concept (step 5)	Met
R5-3: Delegate decisions	Issue owner, outcome status in SOAD metamodel and collaboration system (tool)	Met, basic lifecycle implementation
R5-4: Involve community	Collaboration features, references in background reading attribute, acknowledgments	Met, loosely coupled with core features
R5-5: Document decisions	Decision log report generation capability in collaboration system (tool)	Met, several report formatting options
R5-6: Align with other models	Decision enforcement step 7 in framework, service interface in collaboration system	Met, implemented as demonstrator only
R5-7: Share gained knowledge	Basic exporters in collaboration system, no content cleansing, no feedback into RADM	Partially met, basic support in tool

Architectural Decision Knowledge Wiki, our application wiki for architectural decision knowledge sharing (Chapter 8), has been designed specifically to support these use cases. R5-1 and R5-7 could only be addressed in a basic form. For instance, the current tool implementation provides a filtered export of ADMs from ongoing or completed projects. Content cleansing is not supported, and the tool does not provide any dedicated features to upgrade newly gained knowledge from an ADM into a RADM.

9.2.6 Overall Fit-Gap Assessment

We leveraged existing work from software architecture research, the pattern community, architectural knowledge management, and model-driven development to solve the decision identification, decision modeling, model structuring, dependency management, design method usage, decision enforcement, and collaboration system problems from Chapter 3.

In summary, SOAD satisfies 25 of the 31 method requirements; another four are partially met (e.g., conceptually, but not supported in tool). The service modeling notation requirement (R4-2) was out of scope; it is met when SOAD is combined with existing methods or modeling languages, e.g., SOMA and UML. We also did not capture issues dealing with legacy modernization (R3-3).

Evidence for this overall fit-gap assessment can not only be found in the self assessment conducted in this section, but also in the industrial case studies.

9.3 Industrial Case Studies

We validated SOAD in five commercial projects, as well as several smaller industrial cases. Table 30 introduces the case studies and indicates which SOAD steps were validated in which case:

Table 30. Overview of industrial case studies

Case Study	Project	SOAD Usage	Key Validation Results
Case study 1	SOA coaching	Step 2, 6 RADM for SOA	Issues recur; metamodel and RADM for SOA content practical
Case study 2	SOA design	Step 2, 3, 6 RADM for SOA	Issues recur; metamodel and RADM for SOA content practical
Case study 3 (action research)	Reference architecture creation	Step 1, 2, 3, 4, 5 RADM for SOA, tool	Issues recur; metamodel and RADM for SOA content practical; additional concepts introduced to manage large models
Case study 4	SOA design and development	Step 2, 3, 5, 6 RADM for SOA tool	Issues recur; concepts and RADM for SOA content practical, tool requires usability and installability improvements
Case study 5 (action research)	Web services design	Step 2, 5, 6 RADM for SOA, review technique	Issues recur; concepts and RADM for SOA content practical, review technique works and speeds up workshop preparation and execution
Other cases	Miscellaneous	All concepts	Results from five case studies confirmed

Case study 1 was a knowledge transfer to an SOA consultant with five years IT experience, providing *SOA coaching services* for the design and development team in a German *government institution*.[53] Case study 2 was *SOA design* for a German *multi-channel retailer*, supporting project scoping and executive decision making. The case study was conducted by an experienced IT architect (eight years

[53] Some case study participants asked not to be referenced by name. Hence, we present the case studies in this sanitized form.

as architect, twelve years in the IT industry). The character of case studies 1 and 2 was similar; the scope of case study 2 was broader than that of case study 1.

Shortly after case studies 1 and 2 completed, case study 3 was initiated in a professional *information technology services* firm, developing a world-wide *SOA infrastructure reference architecture* over two project stages each lasting one year. The architects had up to 13 years of IT architect experience and were located in Germany and Switzerland. We participated in the project via action research.

Case study 4 dealt with SOA projects in the professional services arm of a software firm; two client projects and one internal project used SOAD. The client projects were *full lifecycle SOA development projects* for *two government clients* (in an East European and an Arabian country). The main user was an IT architect with two years of experience. Case study 5 was *Web services design in a telecommunications company* in a Benelux country. This was a professional services engagement, which was in the process of developing a work breakdown structure for a second phase of Web service design. We were involved in the project via action research. The results of case studies 1 and 2 were already available when case study 5 was conducted; case studies 3 and 4 were ongoing. Unlike the other case studies, this case study comprised a technical review as well as design activities.

We also conducted *SOA education workshops* for a Northern European government agency and a major American bank. Another professional services firm was involved. As an additional small case study, we also captured the architectural *decisions made during development of Architectural Decision Knowledge Wiki*.

We present the five main case studies in detail now. The presented information originates from the user survey, short oral interviews with the practitioners conducted to clarify certain answers in some cases, as well as our own project and workshop protocols. Architects involved in case studies 1, 2, and 4 reviewed our representation of their projects to make sure they agree with observations made and conclusions drawn from their SOAD usage reports.

Template. The following template is used to present the case studies:

Project scope and set up. This part of the case study template characterizes the case by project phase, team size, SOA design problem to be solved, and project responsibilities of the firm in which the case study was conducted.

Motivation to use SOAD. This part of the template summarizes the status quo of the case before the architects decided to apply SOAD. It gives the rationale for the decision to use SOAD, e.g., commenting on skill levels and pain points. It also qualifies the SOAD use case applied (i.e., education, knowledge exchange, design method, review technique, and governance instrument).

Actual SOAD usage. In this part, the SOAD concepts and thesis contributions are investigated. Did the project team use the SOAD metamodel, RADM for SOA content, and/or tool? Which RADM for SOA content was utilized?

Validation results and action points. This part reports on the projects status after the SOAD usage: Was the project successful? Did the SOAD concepts work and

provide value? Did the architects miss any features or did they encounter any inhibitors? Did any action points result, and if so, how were they followed up upon?

9.3.1 Case Study 1: Professional Services Firm, SOA Coaching

The most important result from this early case study was evidence for the central hypothesis that architectural decisions recur, which is imperative for SOAD to work. We could also verify that the issues and alternatives in the RADM for SOA have an adequate level of detail and are useful during SOA design.

Project scope and set up. This case was a project for a social security agency of the German government. Selected RADM for SOA content was used by a consultant working for a professional services firm which had been contracted by the agency to provide SOA coaching in an early design phase of the modernization of a pension plan profile management system. Information from several backend systems had to be collected, consolidated, processed, and displayed. These systems were technically heterogeneous for historical reasons.

A mixed team was put in place in the first quarter of 2007, with technical roles being staffed from the government agency, the professional services firm, as well as a database vendor and its partners.[54]

Motivation to use SOAD. The SOA coach was responsible for establishing architectural principles and directions for the mixed team, leveraging the broad experience of the professional services firm. The principles and directions were supposed to be reviewed by the entire project team and approved by the agency. A list of important design concerns for which guidance was needed had already been compiled by the agency.

The SOA coach had five years experience working for a professional services firm, mainly as a developer. He had hands-on experience with Web services technologies from previous projects, but not worked in lead architect role on these projects. The motivation to apply SOAD was to enable the coach to bring in a broader set of experience when establishing the architectural principles and giving the requested guidance. The traditional approach for skill and experience transfer would have been to leverage a coaching or mentoring relation between community leaders in the professional services firm and the SOA coach (and the other members of the mixed team, in turn). SOAD was seen to add a systematic approach to knowledge sharing and design (metamodel, issue catalog, decision drivers, pros and cons of alternatives), ensuring a seamless and complete skill and experience transfer. To facilitate architectural workshops efficiently was another motivation to use SOAD.

Actual SOAD usage. SOAD was used as an education and knowledge transfer instrument, and to frame the design work (steps 2 and 6). The RADM for SOA was

[54] The database vendor was a different company than the software vendor involved in case study 4 (presented in Section 9.3.4).

used to prepare architectural workshops; translated excerpts became part of the project deliverables.

Validation results and action points. From a research perspective, the project provided an early opportunity to validate the key hypothesis that architectural decisions recur if the application genre and architectural style are known, that the SOAD metamodel with its issue-level decision drivers and explicitly modeled alternatives is an adequate way of representing such knowledge, and that the already modeled issues are understandable and useful. The initial list of issues compiled by the agency had 15 entries. For 13 of these issues (e.g., use PLAIN OLD JAVA OBJECTS or ENTERPRISE JAVABEANS as a JAVA SERVICE PROVIDER TYPE?), detailed advice could be found in the RADM for SOA, which at that time had about 100 entries. The effort required to create an SOA principles deliverable decreased from eight estimated to five actual person days. For instance, the architect on that project reused the issue JAVA SERVICE PROVIDER TYPE. The decision drivers in the RADM for SOA, particularly "transactionality needs and ease of deployment", matched with the project requirements, so that the SOAD recommendation to "use ENTERPRISE JAVABEANS if leveraging the declarative EJB transaction model is adequate, and to use PLAIN OLD JAVA OBJECTS otherwise", was directly applicable. The architect also reported that he found several issues in the RADM for SOA that he had not identified yet, but which turned out to be required: for instance, the decision to use a SERVICE CATEGORIZATION SCHEME to distinguish technical utility services and business services became a key element of the SOA design [ZGK+07].

The SOA coach reported a significant productivity increase and quality improvements. External feedback for his work was mostly positive, with some criticism coming from the database vendor. The critique turned out to be a political issue; one of the technical recommendations was in conflict with the Web service middleware design of that vendor. The critique did not pertain to the SOAD approach, but one particular issue in the RADM for SOA.

The main action point for further development of SOAD was to add the two missing issues to the RADM for SOA, following the method for content harvesting which we present in Appendix A. Another conclusion was that a translation of content from English to other languages (in this case German) is required to make the asset directly usable in project deliverables. Clear rules have to be established regarding intellectual property rights; users can be granted a non-exclusive right to use the RADM for SOA content, but can not expect to become owners of such material via a professional services engagement.[55]

9.3.2 Case Study 2: Professional Services Firm, SOA Design

On this project, the RADM for SOA content was reviewed in depth and used to make recommendations to the client and to prepare project deliverables. The two

[55] Legal terms and conditions may request such copyright ownership transfer for all project deliverables including reusable assets that are brought into the project.

key validation results were the confirmation that architectural decisions required (issues) recur and the appropriateness of the SOAD metamodel presented in Chapter 6 (e.g., attributes of issues and alternatives, level and layer structure).

Project scope and set up. This case concerned the project initiation and solution outline phase of an enterprise-wide SOA redesign of the existing enterprise applications at a German multi-channel retailer. Having been convinced about benefits of SOA such as flexible (re-)configuration of application components and reuse of services, the retailer approached the professional services firm (the same as that involved in case 1) and asked for a three-week SOA workshop to define a roadmap for SOA adoption, which was supposed to be organized in multiple stages (rationale: cost control and risk management). The roadmap therefore had to suggest several design options for high priority design issues.

Motivation to use SOAD. High client expectations, time pressure, and insufficient experience with the SOA style (despite eight years experience as IT architect at that time and twelve in IT projects in several firms) prompted the architect of the professional services firm to look for a reusable asset. Being familiar with the notion of architectural decisions, he requested access to the RADM for SOA, which at that time (April 2007) had grown to 268 decisions.

Actual SOAD usage. In this case, the lead architect used the SOAD content (in HTML form) to structure the architecture design activities, to educate the team on SOA, and, later on, to present the project results to the client. This affected the project initiation phase (scoping); the expected benefit was that the existing RADM content could give the practitioner a fast start into the design work (he was facing a tight project schedule). The recommendations in the RADM for SOA, e.g., about layering and a stepwise SOA adoption, were used during decision making.

The lead architect reviewed 46 decisions in depth and provided detailed feedback. This gave us the opportunity to validate SOAD steps 2 and 3, as well a subset of the step 6 processes.

Validation results and action points. The desired reuse effect occurred; the project orientation phase indeed could be accelerated. The lead architect marked 28 out of 46 reviewed issues as relevant (note that this is a different validation element than that provided in case 1; in case 1, a list of issues was already available). The feedback also reported a checklist ("safety net") effect. The SOAD metamodel was confirmed to be appropriate; the representation of issues and alternatives was assessed to be adequate in terms of depth and breadth. Many detailed comments concerned the level and topic group positioning of individual issues.

The architect was skeptic whether SOAD can serve as a full design method (step 6), as there will always be project-specific issues. Technology evolves rather rapidly, which makes it hard to keep the model content up to date. However, SOAD was assessed to be a suitable technique supporting existing design methods. The architect also did not see MDD decision injection (step 7) to be immediately practicable. SOA was reported not to be mature and stable enough to justify an investment in MDD, which is a prerequisite for our decision injection concept.

The architect proposed to use the SOAD approach not just on individual projects, but also for an SOA reference architecture asset to be released globally.

One main action point from this case study was to improve the content to address the comments and concerns from the in-depth review. It was required to perform many editorial changes, to refactor and reposition some issues, and to add missing ones.

Secondly, the RADM contains some managerial decisions, e.g., about project scoping and team setup. While such issues can be documented in the SOAD metamodel, using the term *architectural* decision for them mislead the architect and caused terminology discussions. We introduced the concept of decision types to mitigate this problem.

A third important point was to make the prerequisites for SOAD clear, such as the existence of an SOA reference architecture or pattern language that defines architectural layers, service types, and possible values of the scope attribute. Such prerequisites must be easy to relate to and generally available to the target audience. Getting started with SOAD must be easy, e.g., a matter of minutes. To be self-containing without reinventing the wheel and stating the obvious, SOAD must provide an issue summary and links to detailed information, e.g., text books. These rather practical concerns resemble general lesson learned for knowledge management and asset creation. They are key factors to ensure practical applicability.

9.3.3 Case Study 3: Professional Services Firm, Development of an SOA Infrastructure Reference Architecture

This project used SOAD concepts and RADM for SOA content most intensively (all steps). It allowed us to create and validate the RADM for SOA asset. Many of the SOAD concepts such as identification rules, dependency relations, and decision filtering originate from practitioner feedback gained on this project.

Project scope and setup. In March 2007, another line of business in the professional services firm[56] initiated the development of an SOA infrastructure reference architecture. An architectural decisions artifact was defined as one of the key architecture design artifacts besides a logical component model and a physical operational model. The technical project sponsor and the SOA lead architect (17 years of IT industry experience, 13 of which in architecture roles) decided to apply SOAD concepts to model the content of this architectural decisions artifact.

The first project phase lasted one year. There were four team members working part time on this firm-internal project. We conducted *action research*, joining the project team and taking ownership of the architectural decisions artifact.

Motivation to use SOAD. A model-driven approach leveraging UML was followed for all other artifacts. Hence, the decision modeling approach of SOAD was

[56] This line of business focuses on deployment, integration, maintenance, and hosting services, the first one on business consulting and application development services.

seen to be superior to text template-based decision capturing. Filtered report generation was an important requirement (easing reviews and reference architecture customization). Finally, an advantage of SOAD was that it documents the decisions *required* during adoption of the reference architecture (issues). Previous reference architectures only captured decisions *made* during reference architecture development (outcomes), which was seen as an inhibitor for their adoption.

Actual SOAD usage. We performed the knowledge engineering and created the architectural decisions artifacts in Architects' Workbench [ABK+06] and Architectural Decision Knowledge Wiki (see Chapter 8); the RIHA method (described in Appendix A) was developed to process the large number of incoming architectural decisions artifacts from more than 30 projects systematically. A 320-issue decision model was compiled in 2007; all content featured in Chapter 4 is contained in that model. Every three months, there was an intermediate milestone. ESB integration and security-related decisions were particular focus areas. Systems management was investigated as well. The SOA lead architect reviewed 160 issues in depth and made the content available to an extended set of reviewers in the professional services firm. The RADM for SOA content syndication continued throughout 2008 (see discussion in Chapter 10 for evolution of the content).

Validation results and action points. This case again made evident that architectural decisions recur. Another SOA reference architecture team had already created a draft architectural decisions artifact, which we received in January 2007. It contained 50 entries, some as early drafts. 42 of these issues were already covered by our RADM for SOA which at that time had about 100 entries.

Depth, breadth, and quality of RADM for SOA content were appreciated by the reviewers. The decision to create a standalone tool was criticized. Integrations with UML and requirements management tools were requested to improve traceability and usability and to reduce installation and learning efforts.

One early action point resulting from this case study was to explain the level and layer concepts (step 3) better. To do so, we added the topic group hierarchy to the architectural decisions reports generated by Architectural Decision Knowledge Wiki and improved the documentation of the tool. To make decision type and position in the level hierarchy clear in the issue name, we defined naming conventions, e.g., using the suffix PATTERN for pattern selection decisions. We also created class material, e.g., a three-hour lecture accompanied by hands-on exercises.

Early users appreciated the knowledge captured in every single issue, but struggled to stay orientated when being confronted with several hundred such issues. This was the case despite already existing concepts such as the scope, phase, and role attributes as well as the topic group hierarchy. In response, we provided additional search, filter, and export capabilities for ease of orientation and consumption. Finally, we added the entry point concept (Section 6.3).

The project continued throughout 2008. The reference architecture was released successfully. A broader SOA practitioner community was coached via a Web conference (SOA infrastructure community of practice, 25 attendees) and a teach-the-teachers class. The asset was announced commercially after business and technical executives had become convinced of the SOAD vision and value. A rollout to ad-

ditional, non-SOA application domains such as archiving solutions and systems management has been initiated.

9.3.4 Case Study 4: Software Vendor, SOA Design for Clients

In this case study, even more evidence could be gained that design issues recur, as well as detailed feedback about RADM content and writing style. Several practical inhibitors that must be overcome became apparent. The collaboration system, Architectural Decision Knowledge Wiki, was deployed in a production environment.

Project scope and set up. This case study was conducted by an advanced technologies group in a software company with middleware focus (process engines, message brokers) in the second half of 2007. This group performs early adoption engagements at clients. These engagements range from short-term, often unbilled proof-of-concept projects to full-scope, fixed-price projects running for several years. Hence, there is a wide range of methods applied; decision making and capturing rigor varies. Two teams in that group decided to apply SOAD concepts, content, and tool in three projects:

1. SOA project at a government agency in Eastern Europe.
2. SOA project for a municipality in an Arabian country.
3. Company-internal design and implementation of an end-to-end SOA reference solution for the telecommunications industry. More than 20 team members were expected. The project was foreseen to run for several years and to be staffed with a distributed team with members in Canada, France, Israel, Great Britain, and the USA. Two architects evaluated Architectural Decision Knowledge Wiki and decided for it.

Motivation to use SOAD. In projects 1 and 2, the motivation of the architects was to benefit from already gained experience, to train and govern project teams in emerging countries, and to follow a more rigorous decision making approach in a multi-company, -country, and -culture setup. The primary SOAD user had two years experience as an architect and eight years IT industry experience.

The architects in project 3 were less interested in reusable content and modeling concepts, but looking for a knowledge creation and sharing platform, and guidance regarding decision capturing. A wiki with an underlying relational database was seen as the right solution for the international team.

Actual SOAD usage. The first two projects had the same lead architect and SOA subject matter expert; he used the RADM for SOA content intensively. Using the collaboration system would have required offline capabilities, which were not available in the prototype we had developed. The third project mainly used the collaboration system; the RADM for SOA supplied examples that were used during team enablement (which was conducted as a one hour telephone conference).

Project 1 used several SOAD issues and alternatives directly on the engagement, including those about system transaction management patterns presented in Sections 6.1 and 6.2. Project 2 also started to use RADM for SOA content, as well

as knowledge from project 1; however, due to a negative team-internal review (see below), the reuse was not as significant as in the first project. Project-specific decision model content was developed. Project 3 used a shared, hosted instance of Architectural Decision Knowledge Wiki and begun to capture decisions about backend integration (e.g., access to software packages).

The three projects in this case gave us the opportunity to validate steps 2 and 3 (in full scope), as well as 5 and 6 (in basic form).

Validation results and action points. Decision capturing advice, metamodel, and organization of content were appreciated on all three projects.

Projects 1 and 2 successfully used RADM for SOA content (architectural decision knowledge). Project 3 used the tool initially, but struggled with non-functional issues (usability, response times). The main validation result was the insight that while the tool promised to be valuable, its prototypical implementation was not ready for production use.

The main action point from projects 1 and 2 was to clarify scope and objective of the content (what the RADM for SOA should and should not be used for): No matter how well the content is documented and how sound the given architectural advice is, it will always be required to adopt it for the project context (i.e., requirements, architectural principles, and decisions already made). It is not sufficient to transfer generic recommendations into outcome instances and state "this is the SOAD recommendation" in the justification attribute. Another action point was to add an *editorial status* as an attribute in the metamodel to indicate which issues and alternatives are not yet ready for consumption on industry projects.[57]

Another conclusion was to make clearer that the level and layer structure is configurable and to show how to customize the collaboration system. A final lesson learned was that in a commercial version of a RADM, the content has to have publication quality; professional editing is required to achieve such quality.

9.3.5 Case Study 5: Telecommunications Firm, Web Service Design

This case study allowed us to investigate several of the SOAD use cases, e.g., education, design method usage (steps 2, 5, and 6), and review technique. It also reconfirmed many of the validation results from the previous case studies.

Project scope and set up. This case study was conducted at a mobile phone service provider in a Benelux country. It is the second case of action research: We joined the consultants working for professional services firm 1 (the same firm as in case studies 1 and 2) for two workshops in the beginning of a second phase of Web services design (conducted in the second half of 2007). The first phase had been completed; hence, the objective was to compare the already existing design with the industry "best practices" captured in the RADM for SOA and to define a work breakdown structure for the second project phase.

[57] Apart from that change, the metamodel did not have to be modified on any case study.

Motivation to use SOAD. Demonstrating thought leadership and supplementing general project management techniques with SOA design-specific elements were the drivers for SOAD usage. The client and the consultant team also welcomed the opportunity for best practices sharing and receiving a formal technical quality assurance review in a short timeframe. The project had a limited budget; therefore, efficient use of resources was a seen as a key benefit of SOAD.

Actual SOAD usage. In two on-site workshops and following technical review activities conducted remotely, SOAD was used as an education tool, design method, and review instrument. A work breakdown structure for the Web service design activities was created, drawing upon experience already gained and captured in the RADM for SOA. The relevant decisions in the RADM for SOA dealt with service contract design, granularity issues, as well as Java Web service provider design, e.g., parameter validation, provider type, and transactionality.

Our involvement in the case was limited due to budget and scheduling constraints: Two one day workshops were conducted, as well as two document walkthroughs with following telephone conferences to present findings and recommendations. The main focus was on RADM for SOA reuse and, from a method perspective, on creating a work breakdown structure for the design activities.

Validation results and action points. The project manager and six IT architects and IT specialists from the professional services company participated in both workshops. On site the feedback was very positive: The workshop was considered a success, i.e., prepared and conducted efficiently. The recommendations about service granularity and other issues in the RADM for SOA were welcomed. The project manager appreciated the notion of open issues as an opportunity to clearly communicate client obligations such as defining an enterprise data model.

Being involved for a limited amount of time was sufficient for our validation purposes. As the tool was not self explaining yet, continued interactions with the team would have been required to ensure a sustainable use (e.g., active project participation after completion of the on-site workshops). Our main action point was to invest in the accessibility of the RADM for SOA content and to produce supporting material (e.g., tutorials, packaging, and getting started tips); the SOAD concepts and RADM for SOA content did not require any further changes. Some SOA knowledge gained on this project (role of enterprise data model, service identification in business use cases) could be fed back into the RADM for SOA.

9.3.6 Other Cases

We conducted several additional education, technical review, and method coaching activities for various companies in America and Europe as action research.

An SOA and Web services education event, also serving as an informal design review, was held for a Norwegian government agency in September 2007. The main focus was to share best practices regarding general SOA design and service composition issues. Feedback was obtained in writing both from the government agency and a professional services firm involved (a competitor of that involved in

cases 1, 2, and 3) and from a software vendor (the same as in case study 4, but different from that in case study 1). We could validate SOAD steps 1, 2, and 3 here.

Upon invitation from a large American bank, we presented on SOA best practices at an education event with focus on industry reference models (October 2007). There are several connections between industry reference models and architectural decisions: Reference models standardize a problem domain and/or solution space; hence, many reference model selection and adoption decisions recur, and reference model content can provide alternatives on the conceptual and on the technology level. Steps 1, 2, and 3 were validated at this education event.

We also captured decisions made during design and implementation of Architectural Decision Knowledge Wiki in our own metamodel and tool.

The feedback from these small cases resembled that of the larger case studies presented in the previous sections. Applicability of SOAD concepts, RADM for SOA content, and tool was confirmed in several companies and countries.

9.3.7 Survey and Summary

SOAD users on the five large industry case studies presented in Sections 9.3.1 to 9.3.5 as well as the SOAD tool developers that used SOAD to capture their design decisions were asked to fill out a questionnaire. The objective of this survey was to understand who the users are, why they decided to use SOAD (rationale), whether SOAD concepts could successfully be applied and what has to be improved to make framework, RADM for SOA, and tool usable on a broader scale.

The survey first enquired about demographics, existing practices, and project characteristics (e.g., job role/profession, IT architecture experience, typical projects, and current decision capturing and sharing practices). Questions about practical value and usability of SOAD followed. This second part was structured by contribution: SOAD framework step, RADM for SOA, and tool (Architectural Decision Knowledge Wiki). At least one question about each validated SOAD step and concept was asked. Survey participants were given the opportunity to assess SOAD generally, both retrospectively (as used throughout the project) and forward looking. Intending not to overburden busy practitioners, but also to be able to analyze the answers systematically, we decided to offer a choice between highly decisive yes/no and multiple choice questions on the one hand and open questions with free text forms for answers on the other hand. The justification is ease of processing without losing precision: The yes/no questions are simple to answer; the open questions give participants the opportunity to comment on more complex aspects, articulate concerns and request additional features. The questionnaire was tested with two early adopters and improved based on their feedback about clarity and processing time.

We only asked architects to fill out the survey that used concepts, content, and/or tool on actual projects. A 100% return rate could be achieved, and a total of eight responses. Three of the eight responses originate from SOAD team members due to the action research conducted and the SOAD usage during the tool design.

The detailed information from the survey was used to fill out the template for the cases (see Sections 9.3.1 to 9.3.5).

Table 31 summarizes the survey results:

Table 31. Overview of SOAD framework user survey results

Question	Summary of Responses	Comments
Role	IT architect, consultant, developer, middleware product expert	Diverse for role, experience; few countries; only one company affiliation
Experience	From 0 to 13 years as architect	Broad spectrum
Project type	Several different engagement types (consulting, design, development)	Three long running projects, several small services engagements
Existing practices	Mostly text-based decision capturing and sharing, following a method	Participants indicated to be rather rigorous and active users of a method
SOAD framework steps	All steps practical and useful except for SOAD step 7 (not a SOAD, but an MDD issue)	Difference between issues and outcomes not clear initially; SOA domain not seen to be stable and mature enough for application of MDD
RADM for SOA content	Very useful; unclear skills prerequisites and immature editorial quality of some issues at early stages	Some misunderstandings originating from our relaxed interpretation of term *architectural* decision
Collaboration system (tool)	Capability: good; practicability: only for small team or single user	Negative comments affected implementation limitations, not concepts
General comments	Value of model and content appreciated; no consensus whether tool should be integrated with other ones	Assumptions and prerequisites to be clarified, pitfalls to be avoided (intellectual property rights, languages)
Summary	*Value and usability of solution largely proven in practice*	*More requirements identified, as well as non-functional adoption challenges*

The practitioners on all case study projects confirmed the SOAD problem statement (see Chapter 3), and appreciated the framework steps and the RADM for SOA content they worked with. Architectural decisions such as those compiled in the RADM for SOA (see Chapters 4 and 5) recur indeed.

Measurable benefits could be observed in one project situation, in case study 1; case studies 2, 4, and 5 also reported project acceleration and decision making quality improvements. Case study 3 would not have been feasible without SOAD due to the vast amount of knowledge to be processed. The large number of issues to be managed in the reference architecture called for a systematic harvesting approach, the metamodel extensions introduced in Chapter 6, the decision filtering concept from Chapter 7, and the tool support presented in Chapter 8. Case study 4 demonstrated that the RADM for SOA content formatted according to the SOAD metamodel can even become part of the project deliverables. Case study 5 confirmed that SOAD can be used as a review technique, while the other case studies focused on usage of SOAD during SOA design (including education and knowledge exchange).

Due to these informative validation results, we did not conduct further case studies.

9.4 Additional Industrial Validation Activities

Practical value and usability of SOAD also became evident in several self experiments, in workshops with industry participation, and in teaching activities.

Self experiments. The decision-centric design style presented in this thesis originates from our architectural decision making practices on industry projects 1999 to 2005 [ZMC+04, ZDG+05]. To validate the SOAD framework on these projects, we revisited them and applied SOAD to one of them retrospectively, leveraging the RADM for SOA, which at that time comprised 130 issues. In a controlled self experiment, we replayed the architectural decision making and capturing. The sole decision base was the context and high-level architecture presented in [ZMC+04]. Two hours were sufficient to capture 120 outcomes because the recurring issues had already been documented in the RADM for SOA. In the walkthrough, the correct alternative was chosen and a one-sentence justification given, referring to actual project requirements. The validated SOAD steps were 2, 3, and 4 (metamodel usage), as well as 6 (macro and micro process) [ZZG+08].

In a second self experiment, we revisited the 26 Web services decisions from a text book we had co-authored [ZTP03], modeling them with the objective to review and update them if needed.[58] Originally, they had been captured in text only (e.g., in form of bulleted lists). This experiment showed that modeling reusable decisions previously captured in free form is feasible and improves the quality of the knowledge. We detected several missing attributes, could perform consistency checks, and leveraged the SOAD levels to improve the structure of the knowledge. Service registry decisions turned out to be a topic group for which the book content was incomplete and had to be updated. The issues in this topic group dealt with the selection of PROVIDER LOOKUP TIME, SERVICE REGISTRY TECHNOLOGY, and UDDI REGISTRY ASSET; however, detailed pattern adoption decisions, technology profiling decisions, and asset configuration decisions were missing. We could also verify that the 26 issues compiled in 2003 were still valid even if technology had evolved (in several cases, new technology level alternatives had to be added and obsolete vendor asset level alternatives had to be replaced). This is a strong indicator that the decision reuse effect is sustainable.

In a third self experiment, we captured selected knowledge from popular patterns books such as [Fow03, BHR+96, BHS07, HW04] in SOAD to validate the decision identification technique described as step 1. This turned out to be feasible, with good overlap with already existing content (e.g., logical layering, session management, and concurrency), but also new insight (e.g., business patterns, data access patterns, and presentation layer design). Cheat sheets from inside book covers and pattern language diagrams were helpful to identify and model decision dependencies. The formatting of the knowledge according to the SOAD metamodel added structure to the knowledge, e.g., cross-language dependencies.

[58] The book had been used on an industry project in 2004 to follow a decision-centric, methodical approach to Web services design (we were not involved in this project).

Industry workshops and invited talks. At an early project stage, we hosted two open space sessions [Fow05] at an invitation-only "European Software Architects Workshop" (Arosa, Switzerland, January 2007). Participation at this event was diverse, including many business partners of an operating system and personal productivity software vendor, as well as independent, self-employed consultants. About 30 attendees participated in the two sessions. None of them had the same company affiliation as the thesis author. The sessions confirmed the problem statement, the state of the practice, and the solutions developed until that point in time, e.g., decision identification, making, and enforcement steps and SOAD metamodel [ZGK+07].

SOAD was presented to two enterprise architects from a Northern European oil company in February 2007 and September 2007. This thought exchange led to the identification of the governance use case. Architectural Decision Knowledge Wiki was evaluated to be promising and suitable to facilitate a knowledge exchange.

We presented how SOAD can be used as a company-wide knowledge exchange asset at the IIR conference "Enterprise Architecture Management" (Wiesbaden, Germany, May 2008). At the conference we discussed SOAD with enterprise architects from a large logistics carrier, a bank, and a chemical company, as well as representatives of enterprise architecture management tool vendors. The discussions confirmed our assessments of the state of the practice (see Chapters 5 to 8).

Interactions with more than 100 practicing architects. To verify that the SOAD concepts are not limited to SOA as the primary architectural style, we cooperated with a product architect and a consultant in a software firm who specialize on information management. They documented their expertise with information integration and data-centric architectures with SOAD in September 2006. They appreciated the refinement level and the dependency management concepts. The study results were presented at a company-internal conference and in a workshop paper [ZKL07]. This helped to validate SOAD steps 1, 2, and 3 at an early stage.

In January 2007, SOAD was presented to twelve members of a regional community of J(2)EE architects. The architects in the group specialize on application server technologies, component-based development, and message-based integration. The session confirmed the value of SOAD. One concrete suggestion was to model decision drivers as separate entity in the SOAD metamodel. This change was not implemented due to backward compatibility and flexibility concerns.

A half day workshop was requested by six practicing architects of a professional services firm in April 2008; the leader of an international SOA center of excellence also participated. The architects assessed the metamodel to be well designed and nearly complete. It was suggested to capture the organizational reach of a decision (not just its design model scope) as an issue attribute and to add a direct link to actual requirements to the outcome entity. The value of a collaboration system (tool) for decision capturing and sharing was acknowledged; the broad and deep scope of SOAD appreciated. All use cases implemented in Architectural Decision Knowledge Wiki were assessed to be valid. Several additional use cases were identified, for instance, clustering related decisions for joint processing and the ability to compare the architectural decisions made on different projects.

Teaching. We used SOAD and RADM for SOA for teaching at public conferences, guest lectures at universities, and company-internal events such as OOPSLA tutorials 2005 to 2008, ECOWS 2006, and ECOWS 2007. Each event had between 10 and 45 students and software engineers attending; overall, more than 120 practitioners were educated with the help of SOAD project results. For instance, we educated 24 practitioners at a four-hour lab at a company-internal technical leadership exchange event. They were presented four lectures accompanied by hands-on exercises using Architectural Decision Knowledge Wiki. Later on, we presented the material in a one hour Web conference (30 attendees).

At these events, we confronted the attendees with our research questions, e.g., enquiring whether they agree that issues recur, and whether reuse is desirable and possible. The validation results resembled those reported on the case studies.

9.5 Summary of Validation Results

We structure the summary of the validation results by SOAD framework steps and concepts, RADM for SOA content, and collaboration system (tool).

Framework steps and concepts. The fundamental hypothesis that *architectural decisions recur* if the same architectural style is employed on multiple projects in an application genre was confirmed multiple times (step 1). We interacted with several hundred architects. Only one of them disagreed openly, which turned out to be misunderstanding: We do not claim that the decision *outcome* always is the same; only the *issue*, expressing the need for a decision and the related background information (e.g. alternatives, decision drivers) has to recur. We could demonstrate this in the industrial case studies.

The attributes in the metamodel (step 2) were rated well. They were seen to be understandable intuitively, conveying useful information, and giving enough information about the aspects of a decision that matter during decision making. A few additional attributes were suggested (see Section 9.3 and Section 9.4). While the concept of refinement levels (step 3) was acknowledged, the levels in the RADM for SOA were not seen to be the only required structuring means. Other content organization schemes such as panes as defined by The Open Group Architecture Framework (TOGAF) [OG07] were suggested, which is supported by our formalization. Decision dependency management (steps 3 and 4) was seen as an important differentiator of decision modeling in comparison to text-based decision capturing.

Regarding tailoring (step 5) and design method usage (step 6), practitioners pointed out that many methods exist already and that any additional method must be aligned with these. SOAD was seen to take a method support role (i.e., as a technique for decision making embedded in a general purpose method), rather than a standalone method. Decision filtering was seen to be useful. A standardization of the decision processing order was considered to be difficult.

The MDD integration (step 7) was not received well. The skeptical reaction was a general MDD critique not caused by our decision injection concept. The immaturity of SOA was given as an explanation (see case study 2).

After the validation completed, practitioners started to apply SOAD to technical domains such as software package customization and integration, security, systems management, as well as server and storage infrastructure design.

RADM for SOA content. The selection of content and level of detail on which individual issues are represented in the RADM for SOA (steps 2 to 4) was appreciated and seen as appropriate (i.e., not obvious, relevant on SOA industry projects, and documented in an understandable way). Acceleration of decision identification and improved decision making quality were reported in the case studies.

Several times users commented that some issues present in the RADM for SOA do not qualify as architectural decisions according to their interpretation of the term. Examples are executive decisions dealing with project initiation and enterprise architecture and issues dealing with architectural principles. We made a conscious decision to stretch the usage of the term *architectural* decision to the limits of its definition given in Chapter 1 because senior architects often are confronted with a wide range of decisions. Decision types (step 1), model structure (step 3), and decision filtering (step 5) were introduced to avoid misunderstandings.

Some confusion regarding proactive versus retrospective decision modeling occurred; one user simply copied the issue descriptions and the recommendation attribute in the RADM for SOA to outcomes in the client deliverable (an ADM). This caused negative comments from a senior architect in a team-internal technical quality assurance review. We can conclude that the writing style (clarity, objectiveness) has a significant impact on RADM adoption. User expectations must be managed; SOAD is not designed to make architectural thinking obsolete.

Collaboration system (tool). The user feedback regarding the value of Architectural Decision Knowledge Wiki was encouraging: users appreciated that all knowledge required during architectural decision making can be conveniently located in a single place and that the tool comes with a set of initial content. The realized use cases were seen to be meeting practitioner wants and needs. The HTML presentation of issues, alternatives, and outcomes on a single page (with separate tabs for decision investigation, making, and enforcement) received positive reactions. However, users reported that they found it rather difficult to orient themselves and to navigate in large models. In early versions, the static topic group hierarchy was the only order defined; the dependency relations defined in Section 6.3 were not fully leveraged at that point. Additional visual elements were requested, as well as additional views and a tighter integration with other tools.

SOAD framework, Reusable Architectural Decision Model (RADM) for SOA, and Architectural Decision Knowledge Wiki were validated successfully in industrial case studies. Two of these case studies involved action research. Additional validation forms were self experiments, teaching, industry workshops, and implementation of advanced concepts.

10 Discussion of Research Approach and Results

In this chapter, we reflect upon the research challenges we encountered and the research approach we selected to overcome these challenges (Section 10.1). Interpreting the validation results from Chapter 9, we discuss applicability criteria, benefits, and liabilities of our SOA Decision Modeling (SOAD) framework, Reusable Architectural Decision Model (RADM) for SOA, and their tool support (Section 10.2). We also compare SOAD with existing work and outline how the SOAD concepts can be supported in commercial tools (Section 10.3). The chapter closes with a short summary (Section 10.4).

10.1 Research Challenges, Approach, and Evolution of Results

In this section, we discuss the conceptual challenges we encountered, our research approach, and evolution of SOAD concepts, RADM for SOA content, and tool.

10.1.1 Challenges

The creation of a decision-centric SOA design method is an ambitious undertaking. The requirements and research problems from Chapter 3 scoped the required design work from a functional and non-functional perspective. Many additional challenges had to be overcome, including scoping and terminology issues, finding the right level of model depth and breadth, domain complexity and change dynamics, practical adoption challenges, as well as validation challenges.

Scoping and terminology issues. IT and software engineering still are emerging and relatively immature fields. IT in general and SOA in particular suffer from a terminology ambiguity and overload problem: Many vocabularies exist, which are neither well defined nor aligned with each other. At present, there is no commonly agreed reference model for SOA although standardization has been attempted by W3C [W3C04], OASIS [OAS06], and Open Group [OG]. Hence, it is unclear which SOA concepts to consider and how to name issues that have been identified. Such common understanding simplifies RADM scoping and population; it also helps users to locate relevant model content during decision filtering.

To overcome these challenges, we documented SOA patterns ourselves and adopted the layering scheme from one reference architecture [Ars04]. We developed criteria for inclusion of issues in the RADM. These decision capturing guidelines are part of the decision identification technique described as SOAD step 1.

Finding an adequate model depth and breadth. Another challenge is to find the right depth and breadth for the captured knowledge. If, on the one hand, the captured knowledge is rather generic, it may be considered to be common sense and not delivering enough value. If only a few issues are present, prospective users might not find any relevant advice. If, on the other hand, the RADM content is very specific, the reuse effect might not be strong enough to justify the creation of a reusable asset because the captured knowledge is not applicable to multiple projects. If many issues are present, users might struggle to find the relevant ones.

To overcome this challenge, we developed the step 1 identification rules and step 2 decision capturing template, as well as supplemental decision capturing guidelines that complement the metamodel. A subset of the guidelines is presented in the form of heuristics in Chapter 6 and in Appendix A.

Domain complexity and change dynamics. The enterprise application genre is complex and faces a large amount of change. Business models and IT strategies are modified over time. The relevant technical background information also keeps on changing. For instance, new alternatives arise and further experience with technology and products is gained continuously. Hundreds, if not thousands of decisions are required on real-world projects; the dependencies between them are both manifold and subtle. This complexity can not be argued away or hidden by methods and tools. Making it explicit and manageable is important; however, any reusable asset doing so runs the risk of being seen as part of the problem rather than the solution.

In response, we introduced the separation of issues and outcomes in step 2 and the refinement level structure in step 3. The validation results demonstrate that our concepts indeed help to manage complexity and change.

Practical adoption challenges. Enterprises, projects, and people are different. For instance, there is no consensus how to organize the education and decision making activities on a project. As a design method based on reusable knowledge, SOAD may face the critique that that highly capable architects are method-agnostic and incapable ones do not become capable even if supported by such method.

The writing style used to document issues and alternatives in step 2 helped to mitigate this problem: We phrase the advice in a suggestive tone which would also be chosen by a technical mentor. This is opposed to an official design authority as defined in certain governance frameworks and maturity models. Constructive advice is easier to accept than firm rules. During our validation activities, both junior and senior architects gave positive feedback; we reached a fairly large community.

Validation challenges. Software engineering theses must select their validating case studies carefully. There is a conflict between *significance and fidelity* and *number of influencing factors*: The projects must be representative for real-world projects in terms of their scope and complexity, but also observable. Action re-

search (i.e., active project participation of the primary investigators of the research problem) is an efficient validation form. It must not be the only one, however: It has to be ensured that the validation results are reproducible and that the developed solutions are broadly applicable. Action research alone can not do so.

The architects on validating case studies must be willing and able to apply a design method following a novel paradigm, decision centricity. They should be experienced so that the application of SOAD does not overlap with learning activities (e.g., regarding software engineering, design method, and software architecture fundamentals). They must also be able to reflect on experience gained despite busy schedules. Working with volunteers that believe in reusable assets and methods may compromise the quality of the validation: Their feedback tends to be more positive than that of skeptic practitioners.

In our specific case, we also had to ensure that the problems identified and solved are not germane to a single company or region: The survey participants and case study architects are affiliated with one company. Geographical distribution was limited as well, as most of the involved architects work in central Europe.

To mitigate these risks, we worked with architects in different professions with varying experience, and clarified the purpose of the validation. To broaden our reach, we interacted with more than 100 architects from many countries and companies in various industry workshops (see Section 9.4). The feedback for SOAD from these interactions resembled that from the five industrial case studies.

10.1.2 Selected Research Approach and Notations

To overcome the challenges discussed in Section 10.1.1, we selected a research approach, notations, and tools that increased the productivity of researcher and knowledge engineer (asset creator) and software architects (asset consumers).

The problem context and challenges outlined above required an *engineering approach*: We built the SOAD framework starting from real-world requirements originating from industrial projects. Intermediary results were exposed to members of the target audience early on and throughout the project; the RADM for SOA content was gathered iteratively and incrementally. This research approach can be compared to that followed by Cockburn, the creator of the Crystal family of agile methods. Cockburn's dissertation is also concerned with method design: it focuses on people and how they cooperate on development projects [Coc03]. Action research is the primary validation type used in that thesis.

The SOAD concepts range from the conceptual design of a method framework to a pattern-centric decision identification technique to a metamodel to a decision making process to a prototypical tool implementation. Hence, we followed a *best-of-breed* approach to introduce our concepts and selected the notation most adequate for each step: In Chapters 2, 4, and 7, we used various diagram types, e.g., informal rich pictures, component-and-connector diagrams, BPMN process models, UML models, and other standard notations defined in commercial software engineering methods such as the system context diagram and the component model artifacts defined in the IBM Unified Method Framework (UMF). Basic set

and graph theory was applied to formalize the SOAD metamodel in Chapter 6. The rationale for this decision is the generality, expressivity, and precision of this mathematical notation which allowed us to specify the relations accurately. The Object Constraint Language (OCL) [OMG06] would have been an alternative. Lack of experience was one reason for not choosing OCL.

10.1.3 Evolution of Framework Concepts, Model Content, and Tool

In this section, we show how SOAD concepts and implementation evolved over time.

Metamodel evolution. Our two primary knowledge capturing approaches, patterns and decisions, differ in their maturity and adoption rate. On the one hand, patterns are used frequently on projects and in many ways, from pattern catalogs serving as a design reference to patterns becoming architectural templates in model-driven SOA development. On the other hand, architects capture architectural decisions retrospectively so far (if at all). Therefore, existing templates for retrospective decision capturing had to be extended to serve the SOAD use cases. The resulting SOAD metamodel has been stable since September 2006; the validation results did not make any significant changes necessary since then. SOAD continued to be used on industry projects after our validation activities completed.

RADM for SOA content evolution. The initial content of our RADM for SOA originates from successful large-scale SOA projects conducted since 2001. In the meantime, we have refactored the content several times, which led to the fine-grained level and layer structure introduced in Chapter 5. We also incorporated input from a practitioner community (see Appendix A for more information on decision harvesting). Table 32 shows how the RADM for SOA evolved during project duration. When the thesis validation activities completed, it consisted of 389 issues. About 200 of these issues are fully modeled according to the SOAD metamodel; the remaining ones are documented in shorter forms.

Table 32. Evolution of RADM for SOA over time and project phases

SOAD Project Phase	Issues	Comment
Idea (2004)	~10	Captured selected Web services and enterprise application architecture decisions in proof-of-concept
First demonstrator (12/2006)	~100	Identified SOA decisions, e.g., regarding transactional workflows; only small subset fully modeled
First shipment (06/2007)	268	Modeled security, ESB integration, and other issues
Second shipment (12/2007)	320	Added operational modeling and other issues
Third shipment (06/2008)	389	Detailed granularity decisions, presentation layer

Tool implementation. We used Architect's Workbench (AWB) [ABK+06], an Eclipse plugin implementing a model-driven approach based on the Architecture Description Standard (ADS) [YRS+99] for the syndication of the RADM for SOA. The rationale for this decision was that AWB supports a "Grouping ADs By Topic" viewpoint and has strong refactoring and dependency management capa-

bilities. Viewpoint and refactoring capabilities accelerated content creation and maintenance. The reminders feature warned about modeling errors.

We also supervised a diploma thesis to implement the SOAD concepts in Architectural Decision Knowledge Wiki (see Chapter 8 for rationale). We started to expose the system to practitioners in April 2007. Based on early adopter feedback and our own experience working with the tool, we added features en route to version 1.0 (March 2008) and version 1.2 (September 2008). Version 1.0 supported 55 uses cases; version 1.2 about 70. The tool is a reference implementation (proof of concept) not ready for production use (see Chapters 8 and 9 for details).

10.2 Strengths and Weaknesses of Solution

In this section we discuss applicability of SOAD and its benefits and liabilities.

10.2.1 Suited Projects, Application Genres, and Architectural Styles

Several criteria apply when considering an adoption of SOAD, e.g., *target audience* and design *variability vs. standardization* of problem and solution domain.

SOAD targets software architects with some experience, working on full-scope projects in a stable application genre which is characterized by few alternatives to be considered during early design stages and many variation points later in the design work. These two preconditions ensure that issues will recur and make it possible to use pattern-centric identification rules for RADM scoping in step 1.

SOAD is less suited for first-of-a-kind projects in emerging domains in which no reuse effect can be expected yet. The same is true for small projects in which few architectural decisions must be made or in which the consequences of making the wrong ones are not critical. Small, experienced teams with comprehensive tacit knowledge and a personalization strategy [Jan08] are less likely to benefit from explicit, modeled knowledge as promoted by SOAD. In a lightweight setup, RADM population (Chapter 6) can be reduced to a minimum (or skipped) and only decision identification (Chapter 5) and model tailoring (Section 7.1) be performed: The RADM then merely lists issues by name to start design discussions.

The SOAD framework can be adopted in other application genres and architectural styles if project experience with that style has already been gained. *The main adoption task is the creation of a RADM asset for the architectural style*, following SOAD steps 1 to 4 and the harvesting technique presented in Appendix A. When doing so, the SOAD concepts have to be reviewed for applicability; extension points are available to modify the concepts as indicated in the respective steps. It is required to support other sources of decision identification in step 1 if the architectural style is not defined via patterns, but *reference architectures* [BCK03] or other architecture documentation formats. The meta issue catalog might have to be extended for application genres facing different design challenges (Chapter 5). As discussed in Section 6.2 and in Section 7.2, it might also be

required to structure the model differently in step 3 (i.e., to use another refinement level and topic group organization). A different customization of the decision making order, which is based on temporal decision dependencies, may also be required (steps 4 and 6). The process presented in Section 7.2 works with *triggers* implied by *refinedBy* and *decomposesInto* relations, which may not always be appropriate (e.g., on legacy modernization and software package customization projects). Our formalization provides a foundation for defining other processes.

10.2.2 Benefits

Table 33 shows how common design activities are supported in the SOAD steps, particularly those defined in the macro and in the micro processes from Chapter 7:

Table 33. Architectural decision making without and with SOAD

Activity (SOAD Step)	State of the Practice (Chapters 2 and 3)	SOAD (Chapters 4 to 8)
Identify issues (1 to 5)	One-of-a kind, on project	Recurring issues in RADM asset, tailoring technique and meta issues
Find alternatives (1 to 5)	Tacit, personal experience	Already modeled, can be extended
Establish criteria (1 to 5)	Tacit, ad hoc (gut feel)	Decision driver attribute in RADM
Background research (1)	Search Web, repositories	Links to relevant literature in RADM
Consult subject matter expert, delegate (6)	Personal contacts, email, forums, escalations	Best practices recommendations from community captured in RADM
Assess alternatives (6)	Consulting techniques	Same, starting from decision drivers
Review earlier decisions, predict consequences (6)	Tradeoff analysis methods	Modeled logical and temporal decision dependencies can be leveraged
Make decision (6)	Tacit knowledge, architecture design methods, decision support systems	Integrative approach, access to previous decisions (RADM), completeness and error check, managed issue list
Document decision and assumptions (6)	Word processing, wiki tables, groupware databases	Only outcomes have to be captured (for issues present in RADM asset)
Inform project team (6)	Send decision log to team (text document)	Entire team has access to collaboration system (tool); report generation
Enforce and evaluate decision (7)	Manual: Coaching, coding, architectural templates	Partial enforcement automation (decision injection), collaboration tool

Let us now walk through the table, using the views introduced in Chapter 4.

Decision investigation view (steps 1 to 5). As demonstrated in the case studies, usage of a SOAD RADM as created in steps 1 to 4 and tailored in step 5 *increases productivity* during the early project activities such as team orientation and candidate asset screening. From the validation results, we estimate that on average one third of the early project phases is spent on the identification of issues and alternatives, as well as on establishing criteria. Some of this effort will always be required to give new team members an opportunity to familiarize themselves with the project context, for instance the business problem to be solved and the project logistics (tools, build environment, etc.). However, productivity gains can be accomplished.

Junior architects and developers can use a RADM as a *training* mechanism to develop their architectural thinking capabilities. While this is a welcome side effect, it is not the main usage scenario of SOAD; it can not substitute a software architecture curriculum. Due to its reference character, a decision model is rather dense and therefore tiring to read from beginning to end; architecture overviews, component interaction diagrams, and code snippets are required to illustrate the alternatives. However, SOAD can assist with education planning, e.g., help to identify classroom trainings or online courses (via the background reading attribute).

Decision making view (step 6). The managed issue list and its supporting concepts (decision classification by eligibility status and dependencies, decision clustering) *make the decision making more efficient* and *improve the decision making quality*. Detecting and disabling combinations that do not work before a design error is even made improves software quality and reduces technical project risk. The managed issue list can simplify the preparation of architecture design reviews and other architectural workshops when serving as a questionnaire. The tailored RADM content gives the architects access to architectural knowledge already gained in a community, e.g., information about certain decision drivers as well as pros, cons, and known uses of alternatives.

Decision enforcement view (step 7). Decision injection into design models *makes model transformations more flexible* with decision outcomes serving as MDA marks. This leads to less manual development and configuration efforts, which simplifies the model-code *reconciliation* and provides *traceability* between decisions and design. Decision *logs can be generated* as reports (model excerpts).

SOAD tools can implement a feedback loop between roles, which improves team *communication*. With Architectural Decision Knowledge Wiki, decision capturing becomes a shared responsibility; decisions that are openly created, discussed, and justified are easier to accept than dictated ones. A positive impact on team communication and climate can be expected.

10.2.3 Liabilities

Constructive criticism obtained during validation concerned content quality and provenance aspects, as well as complexity and change dynamics.

Content quality and provenance. A challenge is to agree on RADM content and assure its quality. For instance, recommendations must be correct and up to date. Depending on the reuse culture in a company, a diligent review and approval process may have to be established; a self-governing approach is the other extreme. A design method based on reusable architectural decision knowledge will only be successful if practitioners are motivated to contribute high quality knowledge. Collaborative ownership of the model partially solves this maintenance problem; the refinement level and layering structure introduced in Chapter 6 and the basic harvesting method presented in Appendix A help knowledge engineers to determine where input from projects is needed and how it can be incorporated.

Complexity and change dynamics. The challenges of the enterprise application genre lead to a rather complex decision model structure. On the technology and vendor asset levels, thousands of possible solutions exist. New alternatives arise almost daily; issues also change. Alternatives residing on the vendor asset level have to be updated whenever a vendor releases a new product version with enhanced features or with different non-functional characteristics. If we aimed for completion, the RADM for SOA would have to contain thousands of issues with numerous dependencies and alternatives. While this complexity is inherent to the problem domain, SOAD could be criticized for exposing it. However, according to our validation results practitioners prefer to be made aware of this complexity.

While the developed concepts were able to solve the complexity and consumability challenges partially (i.e., in the metamodel and in RADMs), the validation results indicate that further tool innovations are required to fully overcome them.

10.3 Comparison with Related Work

In this section, we compare our work with contributions from the software engineering, software architecture, enterprise application development and integration, SOA design, and architectural knowledge management fields. We introduced this related work in Chapter 2 and assessed its strengths and weaknesses in Chapter 3.

10.3.1 Software Engineering

Software engineering and design methods. Our approach complements general purpose processes such as the Rational Unified Process (RUP). Such assets cover the entire software lifecycle, but do not focus on the design issues in a specific application genre. While they instruct the architect which artifact has to be produced in which activity, they do not state which design issues must be addressed, which alternatives are available, and what the pros and cons of these alternatives are with respect to the decision drivers (e.g., requirements and constraints) germane to an application genre.

Furthermore, such process- and artifact-centric software engineering methods have a passive reference character once they have been adopted on a project in a manual or tool-assisted step; there is no notion of a *managed* issue list.

SOAD extends such methods in a genre- and style-specific way: The RADM for SOA focuses on a particular application genre and can therefore draw on knowledge gathered on previous projects. Using an ADM as a managed issue list (as shown in step 6) becomes possible if the scope, phase, and role attributes as well as decision dependencies are set to meaningful values.

Patterns. Patterns primarily have educational character. Using patterns as a design method has been proposed and is practiced successfully, e.g., by Buschmann and Henney [BHS07]: Patterns can be applied in an incremental refinement process. The decision making then is based on the forces. Applying patterns in such a way

requires a broad view on how to select from a large body of patterns. The reason is that patterns do not provide solutions for a particular application genre, but generic design knowledge. For instance, the INVOKER pattern in [VKZ04] describes how a middleware invokes remote objects in general. The pattern applies to all kinds of middleware, but does not explain the specifics of an SOA INVOKER in an enterprise application. Platform-specific implementation aspects of the pattern are not covered either.

As a primary source of architectural knowledge, patterns play a pivotal role in SOAD. Applying a pattern *is* making a decision; the consequences of applying a pattern engender more decisions. Our step 1 identification rules use the patterns defining an architectural style (jointly with principles); pattern selection and adoption decisions are identified in these patterns. In the RADM for SOA, patterns are positioned as alternatives on the conceptual level. Additionally, SOAD also covers technology and vendor asset level design issues. As a consequence, the strengths of patterns and decision models complement each other in SOAD. A patterns-based RADM can reference the pattern text, which makes it easier to create and maintain than a self-containing one. Outcomes can be captured in much less detail because they only record the adoption of the patterns and can reference the patterns for further detail [ZZG+08].

10.3.2 Software Architecture

Software architecture in general. Software quality attributes and viewpoints are represented in the SOAD metamodel and the RADM for SOA structure. Software architecture literature introduces these concepts, but does not cover how to satisfy a set of quality attributes in a given application genre and architectural style. SOAD complements existing methods with such advice.

Software architecture design methods. Existing architecture design methods are process- and artifact-centric; a backlog is introduced in [HKN+07]. The existing methods offer techniques to resolve general architecture design issues, but do not provide method content that identifies possible solutions based on already gained knowledge; backlogs are populated and maintained manually. Neither EAD and EAI challenges nor SOA principles and patterns are addressed explicitly. SOAD takes inspiration from these methods, for instance the ASR to design decision linkage in ASC and the global analysis activity and issue cards in S4V (see Chapter 2 for introduction). In contrast to these methods, SOAD treats issues and outcomes as first class citizens in its metamodel and integrates genre- and style-specific knowledge: An open issue is an architecture design task. As issues recur when application genre and architectural style are known, SOAD can populate the managed issue list (backlog) during decision identification and model tailoring, e.g., with pattern selection and adoption decisions and related issues residing on the technology and the vendor asset levels of the RADM for SOA.

We did not propose yet another technique to support the making of an individual decision, but focused on finding the decisions relevant in a particular genre,

style, and project context; existing decision making techniques can be integrated into our framework as discussed in step 6 in Chapter 7.

We put less emphasis on quality attributes than the methods described in the literature. This is not to say that quality attributes are not important (as we pointed out in Chapters 1 and 2): In our approach, the advice how to deal with certain decision drivers including quality attributes is a key part of the architectural knowledge captured for issues and their alternatives. However, it is not created anew using some technique, but originates from projects that already encountered and resolved a similar or the same design problem.

10.3.3 Enterprise Application Development and Integration

Genre-specific design methods. The genre-specific design methods such as those introduced in Chapter 2 have the same characteristics as software engineering and software architecture design methods. Hence, SOAD complements them and can be integrated into them in the same way (see Sections 10.3.1 and 10.3.2).

Enterprise architecture frameworks. While not targeting the design of individual applications or services, enterprise architecture frameworks such as Zachman [SZ92] and TOGAF [OG07] influence the SOA design. Enterprise architecture frameworks cover both logical and physical aspects. Typically, they define structural viewpoints such as process and data to display entire application landscapes and system-to-system relations. This helps to position an application under development and to avoid unnecessary parallel development; reuse opportunities and integration needs can be identified. Like software engineering methods, enterprise architecture frameworks are complementary to SOAD; they can be used to structure decisions models (step 3). In return, SOAD RADMs can provide enterprise architecture frameworks with genre- and style-specific architectural knowledge.

10.3.4 SOA Design and Service Modeling Methods

SOA design and service modeling methods cover all phases of SOA design; they are particularly strong in early phases such as business modeling and service identification. Typically, they provide less technical advice than our SOAD framework and RADM for SOA. Architectural decisions are mentioned in SOMA [AGA+08], but not modeled and managed as first class method elements. The relationship between these methods and our approach is complementary. For instance, a SOMA service model can serve as a starting point for RADM tailoring (step 5). To do so, the scope attribute in the RADM for SOA may refer to a “service” instance present in the SOMA service model. Moreover, SDLC [Pap08] and SOMA phases can be referenced in the SOAD phase attribute to indicate which issues should be resolved in a certain phase of SOA design and service modeling. In return, the existing methods can be used to populate a RADM with style-specific knowledge.

10.3.5 Architectural Knowledge Management

In the industry, many templates for architectural decision capturing exist. Practitioners perceive the documentation of made decisions to be an unwelcome, time consuming obligation. There are many real-world inhibitors such as lack of immediate benefits, incentives, budget, and tools [TAG+05]. Hence, a retrospective approach is hard to implement, even if seen to be beneficial in the long term.

None of the existing approaches supports decision identification in patterns and requirements, and there is little support for active reuse, i.e., no separation of decisions required (issues) and decisions made and no asset creation phase. Platform-independent issues are not separated from platform-specific ones. Predefined decision documents are contained in certain reference architectures used in the industry [TA05]; however, we did not find any concepts for bringing issues into the design process to provide active guidance. As a consequence, the decision view typically remains isolated and disconnected from the other architectural views.

Our work enhances the existing modeling approaches in these directions, which helps to overcome the inhibitors. Unlike existing work, we introduce an asset creation phase and a decision identification technique to facilitate collaboration and reuse. In doing so, we apply the ontology and the use cases defined by Kruchten et al. [KLV06] to EAD, EAI, and SOA design. The pattern, technology, and vendor asset selection decisions in SOAD map to existence decisions (chosen alternatives) and ban decisions (rejected alternatives); pattern adoption, technology profiling, and vendor asset configuration decisions in SOAD map to property decisions. We discuss this aspect and more related work in detail in [ZKL+09].

10.3.6 Commercial Products

We are not aware of any commercial or open source decision modeling method or tool that supports decision reuse and modeling or an active issue management (as opposed to capturing issues and outcomes for documentation purposes).

The SOAD concepts can be implemented in many products, leveraging our requirements catalog (Chapter 3), the conceptual tool architecture (Figure 14 on page 62 in Chapter 4), and the SOAD metamodel (Chapter 6). Architectural Decision Knowledge Wiki (Chapter 8) then serves as reference implementation.

A commercial version of SOAD must be highly configurable to accommodate multiple decision making processes and decision maker preferences. For instance, additional ways to order decisions must be provided. The SOAD steps are designed to be extensible; the formalization from Chapter 6 helps tools to provide the required flexibility without compromising other architectural qualities. A critical success factor is to find an appealing visualization of the managed issue list.

One option is to integrate SOAD concepts into *tools for software architects* such as IBM Rational Software Architect [IBM], Telelogic System Architect [IBM], or ArcStyler from Interactive Objects [IO]. Most of these tools are analysis and design model-centric; however, support for architectural decision modeling can be added with a combination of product customization and programming.

Another possibility is to leverage *configurable, metamodel-driven requirements engineering and traceability tools* such as IBM Rational RequisitePro [IBM]. Depending on the flexibility of the metamodels and the provided interfaces, it is possible to customize them to support the concepts presented in this thesis. It is required to integrate the entities defined in the SOAD metamodel. The cardinalities of the relations defined in Chapter 6 are a critical success factor (e.g., multiple outcomes per issue). Powerful model tailoring and managed issue list processing capabilities as defined in steps 4, 5, and 6 (Chapters 6 and 7) are required.

10.4 Summary

We summarize the discussion in this chapter as follows:

Overcoming a number of challenges, SOAD framework, Reusable Architectural Decision Model (RADM) for SOA, and Architectural Decision Knowledge Wiki complement existing methods and tools with genre- and style-specific architectural decision knowledge (method content) and active issue management capabilities.

Benefits of SOAD are an acceleration of decision identification, improved decision making quality, and additional decision enforcement opportunities; managing complexity and change are liabilities and critical success factors.

The SOAD concepts can be applied to other application genres and architectural styles. It is possible to integrate SOAD concepts into several existing tools.

11 Conclusions and Outlook

In this chapter we summarize the thesis and its contributions (Section 11.1) and answer the research questions (Section 11.2). We discuss future work (Section 11.3) and present a vision for an extended usage of our solution (Section 11.4).

11.1 Thesis Summary

In this thesis, we created a decision-centric architecture design method for enterprise application development and integration projects employing SOA as their architectural style. Our method consists of an architectural decision modeling framework, which we call SOA Decision Modeling (SOAD) framework, and a Reusable Architectural Decision Model (RADM) for SOA. It is tool supported. SOAD, RADM for SOA, and tool have five use cases: education, knowledge exchange, design method, review technique, and governance instrument. The RADM for SOA is style-specific; framework and tool usage is not limited to SOA design.

Introduction. In Chapter 1, we introduced problem context and related work. We defined the term architectural decision and outlined the research problems to be solved. We gave an overview of our solution and the structure of this thesis.

State of the art and the practice. In Chapter 2, we defined enterprise applications as an application genre and SOA as an architectural style based on principles and patterns such as service consumer-provider contract, enterprise service bus, service composition, and service registry. After that, we introduced a motivating case study. We demonstrated that software architects encounter numerous design issues during SOA design; they have to make many related architectural decisions to satisfy functional and non-functional requirements. One reason for the size and complexity of this design space is that many technologies and implementation assets are available for the SOA patterns. Finally, we presented a selection of methods and supporting assets in five categories: Software engineering and design, software architecture design, enterprise application development and integration, SOA design and service modeling, and architectural knowledge management.

SOA design method requirements and research problems. In Chapter 3, we established 31 requirements for SOA design methods from personal experience, practitioner input, and the literature. We distilled seven research problems from the requirements as the focus area for this thesis: decision identification, decision modeling, model structuring, dependency management, design method usage, deci-

sion enforcement, and collaboration system. In an analysis of existing methods, we demonstrated that none of these problems has been properly solved so far.

Architectural decision modeling framework for SOA design and tool support. In Chapters 4 to 8, we introduced the SOAD framework and tool support for it. SOAD consists of seven steps, which are organized in an asset creation and an asset consumption phase. The seven steps and supporting concepts are:

1. Identify decisions (concepts: identification rules, meta issue catalog).
2. Model individual decisions (SOAD metamodel).
3. Structure model (logical relations, levels and layers, integrity constraints).
4. Add temporal decision order (temporal relations, production rules).
5. Tailor model (decision filtering).
6. Make decisions (managed issue list, macro and micro processes).
7. Enforce decisions (decision injection).

In Section 11.2, we summarize how the concepts solve the research problems.

Validation of research results. A design method is difficult to validate due to the large number of influencing factors on real projects and the limited informative value of classroom experiments. In Chapter 9, we described how we overcame these validation challenges: We presented a requirements self assessment and five industrial case studies. In two of the cases, we conducted action research. The case studies demonstrated that the developed concepts are valuable and work in practice. As supplemental validation activities, we applied SOAD retrospectively to our own SOA projects, performed several more self experiments, and used excerpts from the RADM for SOA for teaching purposes and industry workshops.

Discussion of research approach and results. In Chapter 10, we reflected on our research approach and the strengths and weaknesses of SOAD. We also compared SOAD with related work: SOAD extends existing proposals for retrospective architectural decision capturing, which in turn are based on existing work in design decision rationale. We added one assumption: multiple projects must apply the same architectural style (SOA) in the same application genre (enterprise applications). This makes it possible to extend the usage of architectural decisions from architecture documentation to design method support: Our architectural decision models do not serve as passive knowledge repositories, but take an active, guiding role during the design work. Because SOA is specified and standardized openly, it is possible to start from knowledge already captured as patterns. This allows us to anticipate the decisions required when adopting and refining the patterns and to reuse related design rationale gathered by communities of practicing architects.

Benefits of SOAD are an acceleration of decision identification, improved decision making quality, and additional enforcement opportunities. Dealing with the complexity of the application genre and keeping the RADM for SOA up-to-date, consistent, and easy to navigate are key challenges for a broader adoption. A related success factor is to incent users to contribute, not only consume, architectural knowledge (method content). This has been a challenge for many industrial knowledge management approaches in the past.

11.2 Answers to Research Questions

In this thesis, we showed that SOA design requires more than a straightforward transformation from analysis-phase business process models to executable workflows and Web services. Many SOA-specific architectural decisions have to be made, starting with pattern selection and adoption, followed by technology- and vendor asset-level decisions. Our overall focus area was (Chapter 1 and Chapter 3):

How to facilitate the architectural decision making in SOA design, starting from functional and non-functional requirements and already gathered architectural knowledge captured in SOA principles and patterns?

Research questions regarding seven research problems had to be answered: Decision identification, decision modeling, model structuring, dependency management, design method usage, decision enforcement, and collaboration system.

Decision identification. *What are the architectural decisions required during SOA design (issues)? Do these issues recur? If so, can the issues be identified systematically in patterns? Can this systematic approach be transferred to other application genres and architectural styles?*

The patterns that define SOA as an architectural style determine which issues arise; additional architectural knowledge originates from projects that applied the SOA patterns. Decisions recur and can be identified systematically: To support step 1, we provided a novel technique for decision identification. It works with architectural patterns, style-independent meta issues, and additional sources of architectural knowledge. Appendix A provides a basic process for harvesting architectural knowledge from projects, as well as related guidance. Following this process, we synthesized a RADM for SOA from our own project experience, contributions from practicing architects, and the literature. It comprises 389 issues. This RADM for SOA is in use in a company-internal community of architects. 20 of these decisions were published as samples that come with Architectural Decision Knowledge Wiki [SZ08]; 35 decisions serve as examples in this thesis. Others are introduced in separate publications [PZL08, ZZG+08, ZGT+07]. Our RADM for SOA is the first reusable architectural decision model for any architectural style. While we did not give any decision modeling examples from other domains in this thesis, we validated that SOAD can be applied to other genres and styles (see Chapter 9). In Chapter 10, we discussed applicability criteria such as suited target audiences and design variability.

Decision modeling. *Which information to model for each issue (and its alternatives)? Which level of detail is appropriate so that the given advice is detailed enough to be actionable and generic enough to be broadly applicable and not subject to overly frequent, unmanageable changes? Which aspects are not covered by existing templates and metamodels used to document architectures and capture decisions made?*

We model individual decisions, rather than capture them in text form: Issues, alternatives, and outcomes are instantiated from a common metamodel, which we in-

troduced in step 2. This metamodel extends existing work to facilitate reuse and collaboration: Each issue describes a single, concrete design problem that recurs or has been solved. The issue is separated from the available alternatives and the chosen outcomes. The metamodel also introduces decision driver and decision lifecycle (owner, status) attributes. Phase and role attributes integrate SOAD into software engineering methods; a scope attribute links decision models and design models. The knowledge exchange is facilitated via a recommendations attribute.

Model structuring. *Assuming that a large number of issues recurs, how can a decision model be organized in an intuitive, use case-driven way? How to separate rarely changing conceptual knowledge from rapidly evolving technology information and platform-specific know how? How to leverage existing problem solving concepts such as architectural layers and viewpoints in the decision models?*

As there are several hundred recurring SOA decisions, we structured decision models with the help of refinement levels in step 3. The level structure is inspired by MDA principles, separating executive, conceptual, technology, and vendor asset levels. Architectural patterns serve as alternatives of issues residing on the conceptual level. Such knowledge changes less rapidly than that on the technology and the vendor asset level. Within the levels, logical layering serves as a proven structuring principle. The resulting decision model structure is extensible.

Dependency management. *Which logical and temporal dependencies exist between decisions? How can such dependencies be represented in decision models? Can these dependencies be used to detect design errors, to organize the decision making process, and to prune irrelevant decisions? If so, how to order the decisions to prepare for decision making?*

RADMs can be fairly complex, issues and alternatives are intertwined heavily. Hence, we formally defined logical dependency relations in step 3. Logical relations such as *decomposesInto* and *refinedBy* as well as integrity constraints help to ensure the soundness and usability of models. In step 4, we added temporal *triggers* relations so that decisions can be ordered for usage during design. We proposed to imply these *triggers* relations from logical *refinedBy* and *decomposesInto* relations to create a top down design process. These concepts allow a SOAD tool to actively manage the design work: Only issues that are relevant in a given context are displayed. Production rules prune irrelevant issues from the model and imply certain outcomes based on decisions already made. This saves the architect unnecessary work. Existing work handles temporal relations informally if at all; there is no active issue management.

Design method usage. *How to use an architectural decision model as an SOA design method? Can a process be defined that considers only the decisions required by a particular role in a certain project phase and design context? What is the relation to software engineering and design methods?*

With the identification step 1 and the metamodeling steps 2 to 4 completed, a decision model can fulfill its envisioned purpose, usage as a design method. We introduced a model tailoring step 5 and a macro and a micro decision making process supporting step 6. These processes take a decision-centric view, making use of

a managed issue list. The RADM for SOA with its issues and alternatives harvested from SOA industry projects makes concrete, genre- and style-specific architectural knowledge available during process execution.

Our decision-centric design method complements and completes existing software engineering methods as well as genre- and style-specific methods: It positions architectural decisions (open issues) in the software engineering process. Using decision models conforming to a metamodel in a design method context is a new paradigm for method design. For the first time, a managed issue list can be populated and maintained semi-automatically with the help of tool support for the metamodel formalization, the decision identification technique, and the model tailoring technique (decision filtering). Unlike any other modeling approach or method we are aware of, SOAD pushes this managed issue list including issues, available alternatives, pros and cons, known uses, and literature references to the architect. In the current state of the art, such knowledge must be pulled from a repository. Furthermore, we only display currently relevant decision knowledge to the architect, based on the current decision making context and status information.

Decision enforcement. *How to enforce that made architectural decisions are respected during subsequent design activities and during development? How to update design models and code according to outcome information in an architectural decision model? What is the relation between decision models and Model-Driven Development (MDD)?*

We described MDD alignment and a novel concept for decision injection into model transformations in step 7. Our decision models are structured according to a metamodel and are machine-readable; hence, decision logs can be generated in this step. Furthermore, decision outcome information (e.g., chosen alternative, justification) can be injected into design models and code. Unlike the concepts developed for the previous steps, we did not validate this concept in practice yet: We could not locate an SOA project willing to apply both SOAD and MDD. Existing techniques such as coaching and code reviews continue to be essential in this step.

Collaboration system. *Which logical building blocks comprise a tool that supports architects when they investigate, make, and enforce architectural decisions? How to support collaborative creation and usage of decision models? How to integrate such tool with other tools used during SOA design?*

We designed Architectural Decision Knowledge Wiki, a collaboration system (tool) realizing the SOAD concepts in a novel architecture combining Web 2.0 concepts, a logically layered architecture, and a relational database. This tool is an application wiki for architecture knowledge capturing and exchange. It supports about 70 use cases in its current implementation. The use cases allow architects to obtain, tailor, manage, and share architectural decision knowledge and to involve the project team and subject matter experts during these steps. Decision dependency and state management as well as model tailoring and report generation are supported. Use cases and display of single decisions were appreciated by early users. However, they encountered usability problems when working with large models, which were caused by the absence of a graphical overview making use of the

rich amount of decision dependency information modeled. The tool is in use in a company-internal community of architects and also available for download.

In summary, we solved the decision identification, decision modeling, model structuring, and decision dependency management problems. We also provided solutions to the design method usage and decision enforcement problems. The collaboration system problem requires further investigations to make the SOAD tool more user-friendly and to integrate it better with other analysis and design tools.

11.3 Future Work

With the SOAD concepts and one reusable architectural decision model defined, many opportunities for future work arise. Our main directions are *metamodel enhancements*, improving the *decision identification, making, and enforcement* steps, providing additional *tool* support, and integration with *other disciplines*.

Decision identification (step 1, step 5). SOA patterns are only one source of input for decision identification; architecture description languages, reference architectures, and other codifications of architectural styles can also support steps 1 and 5. Such support makes SOAD applicable in additional application genres and communities which do not follow a pattern-centric knowledge sharing and design approach. We do not expect additional concepts to be required if the architectural descriptions are available as models (or at least as structured texts), which can serve as starting points for our technique and enable partial automation.

Metamodel enhancements (steps 2 to 4). Future work regarding the SOAD metamodel is to formalize the interlock between decision models and other model types. Such formalization is required to integrate decision modeling tools with other design tools, e.g., UML modeling environments. Such integration was requested by some of the architects involved in the industrial case studies.

Decision making (step 6). Our formalization makes it possible to model two types of decision making orders, top down refinement and technology- or vendor-led design. In top down refinement, an executive decision is refined by one or more pattern selection decisions which decompose into pattern adoption decisions to be refined on the technology level and on the vendor asset level. We assumed such top down design process and decision making order in Chapter 6 when we stated that *triggers* relations are implied by *decomposesInto* and *refinedBy* relations. Another process is required for legacy system modernization and software package customization (as examples of vendor led design): A technology or product selection decision then implies the selection of conceptual patterns. Additional integrity constraints and production rules are required to define a decision making order in such cases.

To make the decision making process even more efficient, decision outcomes can be propagated along containment, refinement, and decomposition relations in a decision model. For instance, a decision that pertains to a composite architectural

component (e.g., business process) can also be applied to its comprising building blocks (e.g., invoke activities in the process). Examples are non-functional properties such as transaction boundaries and security settings. Furthermore, sequences of outcomes which were successfully used earlier in a project (or on a previous project) can be captured and applied to additional parts of the architecture without having to investigate all issues and alternatives in detail again (i.e., to iterate through the complete macro and micro process multiple times).

Decision enforcement (step 7). Our decision injection concept provides a partial solution to the decision enforcement problem; each injection deals with a single decision. To ensure that the injected decision outcome is not overwritten accidentally or deliberately, additional concepts are required (decision governance).

Tool support. A Web-based thin or rich client as provided by Architectural Decision Knowledge Wiki is only one of several alternatives that can be used to implement the presentation layer of a SOAD tool. Another alternative is to use Eclipse [Ecl]. Decision enforcement can be implemented with a design and development work item component in the emerging Jazz collaboration platform [Jaz], which complements Eclipse with team development support. This allows shifting work between practitioner roles (e.g., architects and developers).

Other disciplines. Finally, the interdisciplinary aspects of architectural decisions are worth studying: Software architects interface with many other project roles and stakeholders; there are mutual dependencies. Our decision identification and decision enforcement concepts can be improved when taking these dependencies into account and communicating related information from and to decision models.

Project managers can use decision models to create work breakdown structures and effort estimation reports for planning purposes: Open issues correspond to required activities. Health checks become possible: If there are many frequent design changes (e.g., switches between alternatives), or important conceptual issues are still open late in the process, the project is likely to be in a critical situation.

Moreover, it would be worth studying the role of architectural decision models in software product lines and feature-oriented design [Jan08]. In practice, these approaches are used to cope with functional variability (although non-functional features can be modeled), whereas our work focuses on managing non-functional design variability. There is a strong connection between the two variability types.

Legacy system analysis is an advanced usage scenario for the SOAD framework and the RADM for SOA: When modernizing legacy systems, not only the functional behavior has to be analyzed; quality attributes have to be considered as well. SOAD can serve as an analysis instrument during such bottom-up SOA design if the architectural decisions once made for a legacy asset are captured retrospectively. The resulting legacy decision model can help to assess whether an already existing function is suited to implement a certain business process activity.

Finally, asset selection and configuration decisions define which software licenses are required, and on which hardware nodes the required software has to be installed. Hence, the outcome of certain product-specific asset selection and configuration decisions can serve as input to software configuration management.

11.4 Extended Usage Scenario and Summary

Due to the positive, sometimes enthusiastic reactions from the target audience we received during thesis validation, we believe the concepts presented in this thesis have a significant potential to benefit *communities in professional services firms* and *software product documentation*.

In response to regulatory compliance requirements, the need for collaboration during architectural decision making, and the desire to reuse architecture design rationale, decision models can serve as fine-grained units of knowledge exchange within and between project teams in professional services firms. The efficiency of teams can be improved and delivery excellence achieved if issue modeling and outcome capturing are standardized. If rationale from previous projects is available in this form, unnecessary design discussions can be avoided. The existence of a company-wide RADM becomes a competitive advantage for the services firm.

We also envision reusable architectural decision models to improve the documentation of software products, for example software packages and middleware with many variation points: Such products could ship with a predefined architectural decision model, which elaborates upon the issues, decision drivers, and possible alternatives occurring during customization and deployment of the product. Early users of a new product then complete the architectural decision model by documenting their lessons learned as outcomes with justifications. Assuming that the product is successful in the market, the following mainstream projects follow the advice given and enrich the decision models further, e.g., with known uses and background references. Over time, the lessons learned evolve into best practices commonly agreed and captured as issue recommendations. The architectural decision models from the projects are fed back to development, informing the product architects how the product was used and how it performed.

We believe that such active usage of architectural decision knowledge promises to greatly improve the design and integration of enterprise applications.

Our research results summarize as follows:

Defining SOA as an architectural style based on principles and patterns allowed us to advance the state of the art regarding architecture design methods and propose a decision-centric SOA design method, which comprises an architectural decision modeling framework, a reusable architectural decision model for SOA, and related tool support.

Additional use cases for framework, reusable architectural decision models, and tool, which we validated in practice, are education, knowledge exchange, review technique, and governance instrument. Our concepts are designed to work for other application genres and architectural styles.

Future work concerns metamodel extensions, providing more comprehensive tool support for the framework steps, integration with other methods and tools, and a broader use in professional services firms and software product documentation.

12 Appendix A: Harvesting Architectural Decision Knowledge

In this appendix, we give an overview of the architectural decision knowledge engineering activities we conducted to create the Reusable Architectural Decision Model (RADM) for SOA introduced in Chapter 5. We present a basic process and related guidance to harvest architectural decision knowledge from industry projects. This appendix has the character of an experience report; it targets knowledge engineers in the industry that apply SOAD.

12.1 Overview of Knowledge Engineering Activities

Receiving input from practicing software architects, we studied a rich set of artifacts capturing SOA decisions and other architectural aspects. The first source of input for our RADM for SOA was personal SOA project experience [ZMC+04, ZDG+05]. We documented the issues encountered on these and other projects according to the SOAD metamodel. As a second step, we integrated input from other industry projects, leveraging a company-wide SOA and Web services practitioner community with more than 3500 members. We processed several hundred architectural decisions from more than 30 projects in several geographies and industries. A third type of input was systematic literature screening, e.g., technology introductions, vendor white papers, and technical project reports not necessarily (and not exclusively) focusing on architectural decisions. Pattern books [Fow03, HW04] provided particularly valuable architectural decision knowledge.

Originally, we had employed an ad hoc approach to incorporate these sources of input. This approach failed to produce high quality models and it was not efficient. For instance, it caused duplicate entries. Furthermore, the origins of the knowledge and the dependencies between issues became blurred over time.

To overcome these challenges, we defined a basic harvesting method. It consists of a four step *knowledge harvesting process* and related *decision modeling guidance*. The process is complementary to the top down decision identification technique described as SOAD step 1 (Chapter 5). Knowledge engineers applying the SOAD framework to create a reusable ADM for an architectural style and a community can follow process and guidance independently of each other. Our process and guidance are informal; they can be combined with existing, formally defined knowledge engineering and management approaches. Usage of process and guidance is not limited to the enterprise application genre and the SOA style.

12.2 Bottom Up Knowledge Harvesting Process

To overcome the limitations of our original ad hoc approach, we defined four basic knowledge harvesting steps to be performed by a knowledge engineer (i.e., a software architect working for a community as defined in Chapter 4). Figure 37 illustrates these steps, which we call *Review, Integrate, Harden*, and *Align (RIHA)*:

Figure 37. Four step decision model content syndication process

It is worth noting that it is possible to iterate and harvest knowledge incrementally, although Figure 37 suggests a linear process. We now present the steps:

Review. The first step is to review raw input from completed projects. The objective of this step is to assess the relevance and the quality of the input. Three *qualification criteria* determine whether a candidate decision is included in a RADM:

1. The first criterion is *technical quality*: Is a real architecture design problem described? Is the input an architectural decision according to the characterization of the term given in Chapter 1 (i.e., does it impact a system as a whole or one of its core components, does it have an impact on the non-functional characteristics of the system)? Are the presented alternatives technically sound, particularly the chosen one? Did the contributing project succeed? Is the architect still content with the decision?
2. The second criterion is the *reuse potential*: Does the candidate issue pertain to one of the principles and patterns defining the architectural style? Will it recur? Does it have sustainable, long lasting character or is it a tactical or temporary decision? Does it avoid to reference proprietary design elements or other information that can not be shared?
3. The third criterion is *editorial quality*: Does the issue description read well? Is established terminology used, e.g., are the referenced design model elements defined in Enterprise Application Development (EAD) and Enterprise Application Integration (EAI) literature or SOA patterns? Can issue and outcome be separated from each other?

The first two criteria are mandatory, the third one is optional: If the editorial quality is poor, it can be improved with reasonable editing effort if there is a strong need for the decision, e.g., high reuse potential (criterion 2).

Integrate. The second step is to integrate a decision that passes the qualification criteria into an already existing RADM. An identification rule must be selected (see Chapter 5). It has to be decided which level (as defined in Section 6.2) to add the decision to; if a single decision spawns several levels, e.g., covering conceptual, technology, and vendor-specific aspects, it has to be split. Furthermore, the

decision must be placed into an existing or an additional architectural decision topic group (without breaking the usability heuristics from Definition 6.10).

In this step, the decision made becomes a decision required (issue): The motivation for the decision becomes the problem statement. Assumptions and justification of the decision made become decision drivers of the issue. The chosen alternative becomes a recommended alternative; any rejected ones are also included.

A meaningful name for the issue must be found. The patterns community advises us that finding good names is essential when creating a pattern language, but also hard [Fow06]; the same holds for issue names. The name should be compact, but also expressive: A generic name is broadly understandable and does not to have to be modified often; a concrete one serves well as an issue identifier. In any case, the name must be self explaining, e.g., when appearing in a tool that does not display any other attributes in a particular view (e.g., an issue explorer).

Harden. In the hardening step, the issue and alternative descriptions are improved editorially. Issue and alternative information not present in the raw input, but required according to the SOAD metamodel is added (for an overview, see template in Figure 19 on page 86 and example in Figure 21 on page 89).

For instance, decision dependency information is often missing in the output of the review and the integrate steps. If missing, it is added in this step; if present, it is reviewed and improved. The integrity constraints and heuristics from Section 6.2 should be respected when doing so. Scope, phase, role, and subject area information is also added in this step, as well as the asset information such as owner, editorial status, and acknowledgments.

The knowledge engineer should not assume, guess, or strive for premature completion. It might be required to contact the contributor of the issue in this step to obtain missing information. This also is an opportunity to enquire about additional lessons learned if the knowledge engineer is uncertain about one or more of the three qualification criteria from above (if this has not been done in the review step already). The justification of a decision given on a project should only be upgraded to a recommendation if the architect still is content with the decision once made and if an agreed upon *quality assurance gate* is passed (e.g., the project completed successfully and the solution has been running in production successfully for a certain amount of time).

If the quality assurance gate is not passed, the issue can be kept if it is relevant (even if the outcome from the project is not used). It is possible to leave certain attributes empty, or use placeholders to indicate that additional knowledge from the community has to be obtained at a later stage. A disclaimer should be added to the model in such a case, indicating that the issue has not been fully modeled and quality assured yet and that additional contributions are welcome. Asset information such as the editorial status and the to do attribute can be used for that purpose.

Align. Finally, the alignment step adds dependencies to and from already existing issues and removes undesired redundancies.

In this step, already existing decisions in the RADM may have to be modified. It is important to observe the editorial status during such decision model *refactoring*, e.g., if certain issues have already been approved, it might not be possible for

the knowledge engineer to rephrase them freely without causing additional, undesired review efforts.

Note that some redundancies are desired: Due to the introduction of the refinement levels, many architecture design concerns are first presented conceptually, then on the technology level, and finally on the vendor asset level. Such controlled redundancies serve didactical purposes, and they also make a design future proof.

It is also required to review the writing style and general editorial maturity in this step: As reported in Chapter 9, consumers of the RADM expect publication quality. The standard guidelines for professional writing apply (e.g., to introduce all acronyms, to use them consistently, etc.). It is also important to manage expectations: a RADM does not intend take over the decision making responsibilities on a project as a knowledge engineer creating an asset for a community can not be aware of the project-specific requirements that apply when the RADM is reused.

To complete the four step process, a third path over all attributes is performed, as well as a final alignment of the decision dependencies (defined in Chapter 6).

If the resulting issue description does not yet meet the quality goals that have been established (e.g., review and approval by members of the target audience or by other knowledge engineers), it may be required to return to the harden, the integrate, or even the review step now.

This completes the description of the RIHA process in this thesis. Obviously it leaves many choices to the knowledge engineer. We successfully applied it when creating the RADM for SOA, which was well received in practice (see Chapter 9).

12.3 Experience and Decision Modeling Guidance

In this section, we present several lessons learned during asset harvesting and consolidate them into initial decision capturing advice for knowledge engineers.

12.3.1 Experience With the Review Step

Due to the practical inhibitors for retrospective decision capturing reported in the literature [TAG+05], the incoming knowledge can not be expected to be consistent, complete, and correct, or to adhere to any particular metamodel. According to our experience, its quality varies from poor to solid. For instance, it often is too abstract to be useful. Very few decision logs already have the editorial quality that is required in a reusable asset; e.g., practitioners can not be expected to provide detailed references to literature or other assets. This does not mean that the architects are poor technical writers or inexperienced in their profession, but can be explained by the practical decision capturing inhibitors (see coverage of state of the practice in Chapter 2 and SOAD step descriptions throughout Chapters 5 to 8).

Our particular input particularly lacked consistency and issue categorization. Dependency relations were captured only in a few exceptional cases. Conceptual aspects were sometimes blended with technology and vendor asset level rationale.

Many capturing styles were used. For instance, the assumptions and justification fields in the architectural decisions artifact in IBM UMF [CCS07] were used in several ways (e.g., to trace a decision back to requirements, to express uncertainty, or to make tacit context information explicit). Many decisions were hiding, e.g., in other technical or in project management artifacts. The decision logs often used other component names than other artifacts, e.g., method technique papers and design models. However, in a few cases, the decision logs already had publication quality; a remaining harvesting task was to remove client- and project-specific details.

Supported by the identification rules from Chapter 5, we detected missing issues. For instance, a pattern selection decision is usually accompanied by pattern adoption decisions and must be refined by a technology selection decision.

To screen the raw input rapidly and mark architecturally relevant parts for later processing, we used a color coding for the metamodel elements defined in Chapter 6: Blue marks indicated issues, purple marks alternatives. Yellow marks indicated decision drivers and recommendations; green stood for pros of alternatives, red for cons. The initial issue catalog can be derived from such preprocessed input.

12.3.2 Guidance for the Integrate, Harden, Align Steps

In Chapter 6, we defined integrity constraints and quality heuristics for architectural decision models, which advise on the number of nesting levels and how to work with the dependency relations (Definition 6.10). We now present several additional guidelines. All of these guidelines are suggestive rather than normative as this appendix is an experience report, not a validated research contribution.

Names. Issue and alternative names must be free of vendor jargon. They should be nouns which reference terms from a pattern language or other definition of the architectural style, e.g., MESSAGE EXCHANGE PATTERN in SOA design. The names should already indicate the SOAD refinement level, identification rule, and/or topic group so that they are self explaining when seen in isolation, e.g., in an index. Such naming conventions also simplify decision filtering in SOAD step 5. For instance, the terminology in our RADM for SOA references the enterprise application patterns and the SOA literature [Fow03, HW04, Ars04, KBS05].

Alternatives. All alternatives listed for an issue must solve the same problem. All alternatives of an issue must reside on the same level of refinement; conceptual and technology alternatives are assigned to different, but related issues so that the level structure introduced in Chapter 6 is adhered to. The alternatives in an issue should catch all known "mainstream" solutions as well as a few more exceptional ones that have been applied in practice. A "good enough" approach is followed; capturing all potential solutions, including theoretical options, is not a goal of the asset harvesting process ("if in doubt, leave it out"). By convention, the alternatives are ordered from common and recommended to exceptional; if present, fallback alternatives such as CUSTOM CODING and OTHER LANGUAGE appear last.

Decision drivers. The information about decision drivers should use a consistent vocabulary. It may originate from enterprise architecture guidelines or an industry standard such as [ISO01]. The more homogeneous and consistent the vocabulary is, the easier it becomes to tailor the model and to use it during the decision making processes described in Chapter 7. For instance, decision drivers can be searched for easily if the vocabulary is standardized. Supporting tradeoff analysis and decision making techniques such as ATAM and ADD [BCK03] can be applied more easily as well.

It is also worth nothing that decision drivers change over project phases and refinement levels; there is a trend from strategic and abstract to tactical and concrete. In early phases and higher levels, the decision drivers should be of strategic, long lasting nature, whereas later in the process and the level structure they become more concrete and tactical.[59]

Recommendations. The recommendations attribute of an issue should refer to the decision drivers. The same holds for the pros and cons information in an alternative. Justifications in outcome instances appearing in ADMs, which are added on projects, should then reference recommendations and decision drivers in the referred issue description.

General advice. The description of a decision and its alternatives should not exceed 1000 to 1200 words or one to three HTML pages in a decision log, which may have been generated by a SOAD tool (this is the case for the decision shown in Table 34 in Appendix B). Longer descriptions are difficult to display in a user-friendly way and time consuming to study. If more information is required, the RADM entry should summarize the issue and refer to a separate document via the background reading attribute.

The feedback from SOAD users on industrial case studies must be taken into account (see Chapter 9): Subjective information must be clearly separated from objective information. The SOAD metamodel has been designed to facilitate this separation (e.g., objective decision drivers vs. subjective recommendation). Furthermore, the editorial status should indicate the maturity of the knowledge.

The writing style and editing quality must meet professional standards, e.g., be informative and neutral (e.g., avoid marketing jargon), but also keep the reader interested. If strong claims are made, evidence for them must be provided. Intellectual property rights must be stated clearly; contributors should be acknowledged.

A suggestive, mentoring tone has higher chances to succeed than an authoritative one: The asset creator (knowledge engineer) should give the asset consumer (architect) the impression that the RADM intends to help and provide orientation, not to create additional effort or unnecessary technical complexities.

Further information exceeds the scope of thesis. Additional decision capturing advice is available in the documentation of Architectural Decision Knowledge Wiki introduced in Chapter 8 [SZ08] as well a practitioner article [ZSE08].

[59] A detailed analysis of the relation between the levels in our RADM for SOA and the decision driver categorization in Section 12.4 requires further study (future work).

12.4 Decision Drivers in EAD, EAI, and SOA Design

In Chapter 2, we motivated that many requirements in EAD and EAI are specific to the genre, such as the integration needs of heterogeneous systems and the business rules ensuring the integrity of enterprise resources over long periods of time (see discussion of EAD and EAI challenges in Section 2.1 and requirements in motivating case study in Section 2.2). Furthermore, many non-technical factors have an impact on enterprise application development and deployment, for example legacy system constraints and organizational issues such as regulatory compliance rules and legal constraints, cost, and available skills.

To organize the decision drivers and ensure we model relevant information, we developed a simple decision driver categorization. It can be used when screening existing architectural artifacts during SOAD step 1 and when applying the RIHA process from Section 12.2. It can also be used when evaluating whether the system under construction meets its design goals. Figure 38 introduces the categorization.

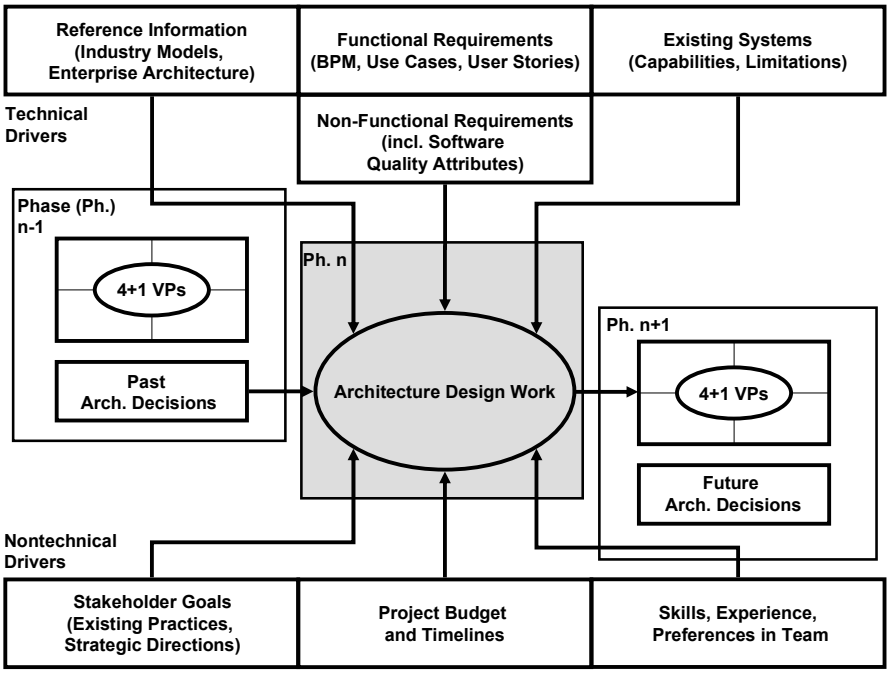


Figure 38. Decision driver categorization for EAD and EAI

Technical drivers form the top row. They include reference information, for instance industry models and legislative regulations (e.g., accessibility acts and auditing compliance rules), enterprise architecture standards, functional requirements, Non-Functional Requirements (NFRs), and the results of existing system analysis activities. The user channel diversity, process and resource integrity, integration needs, and semantics challenges (Section 2.1.2) and the requirements

stated for the insurance SOA case study, e.g., the analysis-phase BPM from Section 2.2.2 and business rules, NFRs, and legacy constraints from Section 2.2.3 all fall in this category.

Results from earlier projects and project phases (middle row) also influence the decision making. It may make sense to make such dependencies explicit not only via dependency relations (as defined in Chapter 6), but also as decision drivers. The first project phase can be a "phase 0", i.e., an unbilled presales phase.

There are many *non-technical drivers* (bottom row): Stakeholder goals, preferences, disabilities, project budget and timelines, as well as available design and development skills and experience fall into this category. When conducting professional services engagements, contractual obligations regarding education, maintenance, and support also have to be taken into account. Many of these drivers remain tacit; i.e., they are not specified in requirements documents. In practice, these drivers often dominate the decision making: The executive-level argument "there is no budget" is stronger than the technical argument "full portability and standards compliance is a mandatory NFR".

13 Appendix B: Excerpt from RADM for SOA

This appendix is a full report of the RADM content for INVOCATION TRANSACTIONALITY PATTERN, the issue that served as example in Chapter 6. One outcome instance for the motivating case study has been added.

Table 34. INVOCATION TRANSACTIONALITY PATTERN (RADM for SOA)

AD ID	Sld-01	AD name	InvocationTransactionalityPattern		
Topic group hierarchy	ConceptualLevel - SoaServiceRealizationDecisions – AtomicServiceLayerDecisions – OperationDesignDecisions				
Subject area	Transaction Management, PSD				
Scope	Service Operation	Phase	Macro Design	Role	Application Architect
Problem statement	What is the system transactionality of a service (operation) invocation? Transaction management, e.g., ensuring ACID characteristics, is a system-level response to resource integrity requirements. In a business process execution environment, all service invocations have to decide for certain transaction management settings, e.g., in BPEL and SCA. Some of these settings are vendor-specific (proprietary). This is one of the most challenging ADs when designing a process-centric SOA with a Service Composition Layer (SCL); it also has to be made when no such layer exists.				
Decision drivers	Business-level resource protection and data currency needs, capabilities of the available service interfaces as well as standard NFRs such as parallelism (number and size of transactions), manageability, and performance.				
Alternatives	**[1] Transaction Islands (default)**				
	Description	This pattern isolates process activities in the Service Composition Layer (SCL) from service operation execution (from a system transaction management standpoint). It consists of these settings for the primitives (see related decisions): PAT-J or PAT-N, CT-SNT, ST-N.			
	Pros	Transaction Islands is often seen as the only alternative that is faithful to the SOA vision of loosely coupling consumer and provider. This alternative decouples the service composition layer from the invoked services (from a system transaction management point of view). The transactions therefore are rather fine grained and often short running (see related decision, dealing with the PAT primitive, for a discussion of the size and duration of the SCL transactions).			
	Cons	If a service operation has to be rolled back, the transaction in which the process navigation in the SCL runs is not affected. If a service works with shared enterprise resources, the service operations must be idempotent, as they may be executed more than once due to the transactional process navigation in the SCL. In many cases, the service provider must offer a compensation operation, and higher-level coordination is required (e.g., via business transactions).			
	Known uses	In practice, this pattern is often selected as a default choice.			
	Background reading	The paper "Architectural decisions and patterns for transactional business process in SOA" has detailed explanations. It can be found			

		here: http://soadecisions.org/download/ICSOC2007_4749_0081_0093.pdf
	[2] Transaction Bridge	
	Description	Via context sharing, this pattern couples process activity execution in the service composition layer and service operation execution from a transaction management perspective. Transaction Bridge consists of the following primitives: PAT-J or PAT-N, CT-ST, ST-J. See related decisions for detailed information regarding these primitives.
	Pros	Covers resource protection needs well on the system level.
	Cons	Process activity and invoked service operation execute in the same transaction. Several service operations can participate in the same transaction. Therefore, there is a natural limit for the response times (tenths of seconds to seconds at most). If a service operation has to be rolled back, e.g., due to a service-internal processing error, previous transactional work, which can include process navigation in the SCL and the invocation of other services, has to be rolled back as well.
	Known uses	This pattern often is not applicable, e.g., when processes and operations are long running or communicate over a slow or unreliable Wide Area Network (WAN). However, in certain short-running micro flow scenarios, on subprocess level, it can be the most straightforward way to meet the resource protection needs.
	Background reading	The paper "Architectural decisions and patterns for transactional business process in SOA" has detailed explanations [ZGT+07].
	[3] Stratified Stilts	
	Description	Use message queuing as SOA communication infrastructure to realize Queued Transaction Processing (QTP) in SOA. The SCL transaction is suspended during service invocation. Stratified Stilts consists of the following primitives: PAT-J or PAT-N, CT-AS, ST-J. See related decisions for detailed information regarding these primitives.
	Pros	Process activities are loosely coupled with the services, distributing work asynchronously. Services are not forced to respond in a timely fashion; message delivery is guaranteed by the messaging infrastructure. This pattern is well suited in long running process integration scenarios that have to use unreliable networks or slow or unreliable service providers.
	Cons	However, if the service operation execution fails, the process may not get an immediate response; additional error handling is required, often using timeout and compensation logic. Significant testing and systems management efforts are required.
	Known uses	Most message-based workflow solutions. This pattern often is the only choice in process-enabled SOA, e.g., when integrating legacy systems.
	Background reading	Stratification is explained in depth in "Production Workflow" by F. Leymann and D. Roller [LR00].
	[4] Not applicable	
Recommendation	Transaction Islands as default, Stratified Stilts for long running, distributed processes.	
Decision outcomes	Default for Customer Enquiry process in PremierQuotes SOA	
	Status	decided
	Chosen Alternative	Transaction Islands
	Justification	Legacy system constraints force us to address resource protection needs with business transactions (compensation).
Background	If you need a quick reminder what transaction management is about, take a look at the	

reading	following article: http://www.ibm.com/developerworks/java/library/os-ag-transsup/index.html . If you need a thorough introduction to the topic in a workflow context, we recommend Chapter 7 of "Production Workflow" by F. Leymann and D. Roller.
Related decisions	influences Cmd-04 ProcessActivityTransactionalityPAT influences Crd-05 ServiceProviderTransactionalityST influences Ird-08 CommunicationsTransactionalityCT influences Ser-05 OperationCompensation is influenced by Cmd-01 ResourceProtectionStrategy is influenced by Msg-01 MessageExchangePattern
Editorial information	Acknowledgments: original SOAD content contributed by Olaf Zimmermann, harvested from projects 1999-2005. Joint work with Jonas Grundler and Stefan Tai. Last modification on 2009-03-17 11:48:59.557000 Status: published as alphaWorks sample (minor edits) Todo: to be reviewed semi-annually IPR level: COPYRIGHT-PROTECTED ASSET © Olaf Zimmermann and IBM Research GmbH, 2009. All rights reserved.

References

External References

[ABK+06] Abrams, S., Bloom, B., Keyser, P., Kimelman, D., Nelson, E., Neuberger, W., Roth, T., Simmonds, I., Tang, S., and Vlissides, J., Architectural Thinking and Modeling with the Architects' Workbench. IBM Systems Journal, Volume 45, Number 3, 2006. Pages 481-500.

[ACK+03] Alonso G., Casati F., Kuno H., Machiraju V., Web Services: Concepts, Architectures and Applications. Springer, 2003.

[ACM03] Alur D., Crupi J., Malks D., Core J2EE Patterns. Prentice Hall, 2003.

[AFM+05] Akkiraju R., Farell J., Miller J. A., Nagarajan M., Sheth A., and Verma K., Web Service Semantics – WSDL-S, W3C Submission. Available online: http://www.w3.org/2005/04/FSWS/Submissions/17/WSDL-S.htm

[AGA+08] Arsanjani A., Ghosh S., Allam A., Abdollah T., Ganapathy S., Holley K., SOMA: A Method for Developing Service-Oriented Solutions. IBM Systems Journal, Volume 47, Number 3, 2008. Pages 377-396.

[AGJ05] Ali Babar M., Gorton I., Jeffery R., Capturing and Using Software Architecture Knowledge for Architecture-Based Software Development. Proceedings of Fifth International Conference on Quality Software (QSIC), IEEE Computer Society, 2005. Pages 169-176.

[AH05] Arsanjani A., Holley K., Increase Flexibility with the Service Integration Maturity Model (SIMM), IBM developerWorks, 2005.

[ALM+99] Avison D., Lau F., Myers M., Nielsen P. A., Action Research. Communications of the ACM, Volume 42 Number 1, 1999. Pages 94-97.

[Ars04] Arsanjani, A., Service-Oriented Modeling and Architecture, IBM developerWorks, 2004.

[BB06] Brahe S., Bordbar S., A Pattern-Based Approach to Business Process Modeling and Implementation in Web Services. Proceedings of IEEE ICSOC Workshops, 2006. Pages 166-177.

[BC89] Beck K., Cunningham W., A Laboratory For Teaching Object-Oriented Thinking. Proceedings of OOPSLA 89. ACM, 1989. Pages 1-6.

[BCK03] Bass, L., Clements, P., Kazman, R., Software Architecture in Practice, Second Edition. Addison Wesley, 2003.

[Bec00] Beck K., Extreme Programming Explained. Addison Wesley, 2000.

[Bec02] Beck K., Test-Driven Development. Addison Wesley, 2002.

[BHS07] Buschmann F., Henney K., Schmidt D., Pattern-Oriented Software Volume 4 – A Language for Distributed Computing. Wiley, 2007.

[BMR+96] Buschmann F., Meunier R., Rohnert H., Sommerlad P., and Stal M., Pattern-Oriented Software Architecture – a System of Patterns. Wiley, 1996.

[Boe88] Boehm B., A Spiral Model of Software Development and Enhancement, IEEE Computer Volume 21(5), 1988. Pages 61-72.

[Boo] Booch G., Handbook of Software Architecture. Available online: http://www.booch.com/architecture

[Boo94] Booch G., Object-Oriented Analysis and Design with Applications. Addison Wesley, 1994.

[Bre] Bredemeyer Consulting, Key Architecture Decisions Template. Available online: http://www.bredemeyer.com/papers.htm

[CBD+06] Everware-CBDI Inc, CBDI Service Architecture & Engineering: A Framework and Methodology for Service-Oriented Architecture (SOA). CBDI Report, 2006.

[CCS07] Cook D., Cripps P., Spaas P., An Introduction to the IBM Views and Viewpoints Framework for IT Systems. IBM developerWorks, 2007.

[CLK07] Chang S. H., La, H. J., Kim S. D., A Comprehensive Approach to Service Adaptation. Proceedings of SOCA'07, IEEE Computer Society, 2007. Pages 191-198.

[CK07] Chang S. H., Kim S. D., A Systematic Analysis and Design Approach to Develop Adaptable Services in Service-oriented Computing. Proceedings of SCC'07, IEEE Computer Society, 2007. Pages 375-378.

[CNP+06] Capilla R., Nava F., Perez S., and Duenas J.C., A Web-based Tool for Managing Architectural Design Decisions. Proceedings of 1st ACM Workshop on SHaring Architectural Knowledge (SHARK), SIGSOFT Software Engineering Notes 31, 5, 2006.

[Coc03] Cockburn A., People and Methodologies in Software Development. Ph. D. Thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, Norway, 2003.

[DC05] Duenas, J. C., Capilla R., The Decision View of Software Architec-

ture. Proceedings of 2nd European Workshop on Software Architecture (EWSA), Springer LNCS Volume 3527/2005, Pages 222-230.

[DC07] Diaz-Pace J. A., Campo M. R., Using Planning Techniques to Assist Quality-driven Architectural Design Exploration. Proceedings of QoSA 2007, Springer LNCS Volume 4880/2008. Pages 33-52.

[DFL+07] de Boer R.C., Farenhorst, R., Lago P., van Vliet H., Clerc V., and Jansen A. Architectural Knowledge: Getting to the Core. Proceedings of QoSA 2007, Springer LNCS Volume 4880/2008. Pages 197-214.

[EAK06] Erradi A., Anand S., Kulkarni N., SOAF: An Architectural Framework for Service Definition and Realization. Proceedings of SCC'06, IEEE Computer Society, 2006. Pages 151-158.

[Ecl] Eclipse Foundation, Eclipse – An Open Development Platform. Available online: http://www.eclipse.org

[Eme03] Emerich W., Konstruktion von Verteilten Objekten. Dpunkt, 2003.

[Erl04] Erl T., Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services. Prentice Hall, 2004.

[Erl05] Erl T., Service-Oriented Architecture: Concepts, Technology & Design. Prentice Hall, 2005.

[Erl08] Erl T., SOA Principles of Service Design. Prentice Hall, 2008.

[Erl09] Erl T., SOA Design Patterns. Pearson, 2009.

[Eva03] Evans E., Domain-Driven Design. Tackling Complexity in the Heart of Software. Addison Wesley, 2003.

[FBC06] Falessi D., Becker M. Cantone G., Design Decision Rationale: Experiences and Steps Towards a More Systematic Approach. Proceedings of 1st ACM Workshop on SHaring Architectural Knowledge (SHARK). SIGSOFT Software Engineering Notes 31, 5, 2006.

[Fie00] Fielding R. T., Architectural Styles and the Design of Network-based Software Architectures, Ph. D. Thesis, University of California, Irvine, 2000.

[Fow97] Fowler M., Analysis Patterns: Reusable Object Models. Addison Wesley, 1997.

[Fow00] Fowler M., UML Distilled. Addison Wesley, 2000.

[Fow03] Fowler M., Patterns of Enterprise Application Architecture. Addison Wesley, 2003.

[Fow05] Fowler M., Open Space. Available online: http://martinfowler.com/bliki/OpenSpace.html

[Fow06] Fowler M., Writing Software Patterns. Available online: http://www.martinfowler.com/articles/writingPatterns.html

[Fow07] Fowler M., AltNetConf (OOPSLA school of software development). Available online: http://martinfowler.com/bliki/AltNetConf.html

[FS96] Fuchs, N. E. and Schwitter, R., Attempto Controlled English (ACE). Proceedings of CLAW 96, First International Workshop on Controlled Language Applications, University of Leuven, Belgium, 1996. Pages 124-136.

[GHJ+95] Gamma E., Helm R., Johnson R., Vlissides J., Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[GR93] Gray J., Reuter A., Transaction Processing: Concepts and Techniques. Morgan Kaufman Publishers, 1993.

[GR01] Gongla P., Rizzuto C. R., Evolving Communities of Practice: IBM Global Services Experience, IBM Systems Journal, Volume 40, Number 4, 2001. Pages 842-862.

[HAZ07] Harrison N., Avgeriou P., and Zdun U.. Using Patterns to Capture Architectural Decisions. IEEE Software, IEEE Computer Society 2007. Pages 38-45.

[HGR08] Henderson-Sellers, B., Gonzalez-Perez, C., Ralyte, J., Comparison of Method Chunks and Method Fragments for Situational Method Engineering. Proceedings of 19th Australian Conference on Software Engineering. IEEE Computer Society, 2008. Pages 479-488.

[HKN+07] Hofmeister C., Kruchten P., Nord, Obbink J. H., Ran A., America P., A General Model of Software Architecture Design Derived from Five Industrial Approaches. Journal of Systems and Software 80(1), Elsevier, 2007. Pages 106-126.

[HNS00] Hofmeister C., Nord R., Soni D., Applied Software Architecture. Addison Wesley, 2000.

[Hoh07] Hohpe G., SOA Patterns: New Insights or Recycled Knowledge? Keynote at Fifth International Workshop on SOA and Web Services Best Practices (at OOPSLA), Montreal, Canada, October 21, 2007.

[HW04] Hohpe G., Woolf, B., Enterprise Integration Patterns. Addison Wesley, 2004.

[HZ06] Hentrich C., Zdun, U., Patterns for Process-Oriented Integration in Service-Oriented Architectures. Proceedings of 11th European Conference on Pattern Languages of Programs (EuroPLoP 2006), Irsee, Germany, July 2006.

[IBM] IBM Software Group, http://www.ibm.com/software

[IEEE07] ISO/IEC IEEE Std 1471-2000, Systems and Software Engineering – Recommended Practice for Architectural Description of Software-Intensive Systems, ISO/IEC, 2007.

[IETF] Internet Engineering Task Force, Transport Layer Security. Available

online: http://www.ietf.org/html.charters/tls-charter.html

[IO] Interactive Objects, ArcStyler, http://www.interactive-objects.com/products

[ISO01] International Standards Organization (ISO), ISO/IEC 9126-1:2001, Software Quality Attributes, Software Engineering – Product Quality, Part 1: Quality Model, 2001.

[Jan08] Jansen A., Architectural Design Decisions. Ph. D. Thesis, Groningen University, Netherlands, 2008.

[Jaz] Jazz Community Site, https://jazz.net/pub/index.jsp

[JB05] Jansen A., Bosch, J., Software Architecture as a Set of Architectural Design Decisions, Proceedings of the 5th Working IEEE/IFP Conference on Software Architecture (WICSA'05), IEEE Computer Society, 2005. Pages 109-120.

[Joh05] Johnston, S., RUP Plug-In for SOA V1.0, IBM developerWorks, 2005.

[Jos07] Josuttis N., SOA in Practice – The Art of Distributed Systems Design. O'Reilly, 2007.

[KBH+04] Keen M., Bishop S., Hopkins A., Milinski S., Nott C., Robinson R., Adams J., Verschueren P., and Acharya A., Patterns: Implementing an SOA Using an Enterprise Service Bus. IBM Redbook, 2004.

[KBS05] Krafzig D., Banke K., Slama D., Enterprise SOA, Prentice Hall, 2005.

[KLV06] Kruchten P., Lago P., van Vliet H., Building up and Reasoning about Architectural Knowledge. Proceedings of QoSA 2006, LNCS 4214, Springer 2006. Pages 43-58.

[Kru95] Kruchten P., The 4+1 View Model of Architecture, IEEE Software, Volume 12, Number 6, November 1995. Pages 42-50.

[Kru03] Kruchten P., The Rational Unified Process: An Introduction. Addison-Wesley, 2003.

[LL91] Lee J., Lai, K., What's in Design Rationale?, Human-Computer Interaction, 6(3&4), 1991. Pages 251-280.

[LL07] Ludewig, J., Lichter H., Software Engineering: Grundlagen, Menschen, Prozesse, Techniken. dPunkt, 2007.

[LR00] Leymann F., Roller D., Production Workflow – Concepts and Techniques. Prentice Hall, 2000.

[MB02] Malan R., Bredemeyer D., Less is More with Minimalist Architecture. IT Pro, IEEE Computer Society, October 2002.

[Mey00] Meyer, B., Object-Oriented Software Construction, 2nd edition. Prentice Hall, 2000.

[Mit05] Mitra T., Business-Driven Development. IBM developerWorks, 2005.

[MS07] Microsoft Corporation, Microsoft Project 2007. http://office.microsoft.com/en-us/project/FX100487771033.aspx

[MYB+91] MacLean A., Young R., Bellotti V., and Moran T., Questions, Options, and Criteria: Elements of Design Space Analysis, Human-Computer Interaction, 6 (3&4), 1991. Pages 201-250.

[OAS04] OASIS, UDDI Version 3.0.2. UDDI Spec Technical Committee Draft, Dated 20041019. Available online: http://uddi.org/pubs/uddi_v3.htm

[OAS06] OASIS, Reference Model for Service Oriented Architecture Version 1.0. Committee Specification 1, August 2, 2006. Available online: http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.html

[OAS07] OASIS, Web Services Business Process Execution Language (WS-BPEL), Version 2.0, April 2007. Available online: http://www.oasis-open.org/committees /tc_home.php?wg_abbrev=wsbpel

[OAS07a] OASIS, Web Services Atomic Transaction (WS-AtomicTransaction) Version 1.1, OASIS Standard Incorporating Approved Errata, 12 July 2007. Available online: http://docs.oasis-open.org/ws-tx/wstx-wsat-1.1-spec/wstx-wsat-1.1-spec.html

[OAW] openArchitectureWare Version 4.3. Available online: http://www.openarchitectureware.org

[OG] Open Group, SOA Reference Architecture Working Group, Available online: http://www.opengroup.org/projects/soa/doc.tpl?CALLER=index.tpl&gdid=13577

[OG97] Open Group, Distributed Computing Environment (DCE) Version 1.2.2, 1997 to 2005. Available online: http://www.opengroup.org/dce

[OG07] Open Group, The Open Group Architecture Framework, Version 8.1.1, 2007. Available online: http://www.opengroup.org/togaf

[OHE99] Orfali R., Harkey D., Edwards J., Client/Server Survival Guide, Third Edition. Wiley, 1999.

[OMG] Object Management Group, Service oriented architecture Modeling Language (SoaML) – Specification for the UML Profile and Metamodel for Services (UPMS) Revised Submission, August 2008. Available online: http://www.omg.org/docs/ad/08-08-04.pdf

[OMG03] Object Management Group, MDA Guide Version 1.0.1, June 2003.

[OMG04] Object Management Group, CORBA Version 3.0.3, March 2004.

[OMG05] Object Management Group, Reusable Asset Specification, Version 2.2, November 2005.

[OMG06] Object Management Group, Object Constraint Language Version 2.0, May 2006.

[OMG08] Object Management Group, Software & Systems Process Engineering, Metamodel Specification (SPEM), Version 2.0, April 2008.

[OSOA] Open SOA Alliance. Service Component Architecture. Available online: http://www.osoa.org/display/Main/Service+Component + Architecture+Specifications

[Pap08] Papazoglou, M., Web Services: Principles and Technology. Pearson/Prentice Hall, 2008.

[Pul06] Pulkkinen, M., Systemic Management of Architectural Decisions in Enterprise Architecture Planning. Four Dimensions and Three Abstraction Levels. Proceedings of the 39th Annual Hawaii International Conference on System Sciences, Volume 08. IEEE Computer Society, Washington, DC, 2006. Page 179.1.

[PV06] Papazoglou M., van den Heuvel W. J., Service-Oriented Design and Development Methodology, International Journal of Web Engineering and Technology (IJWET) Volume 2 No 4. Inderscience Enterprises, 2006. Pages 412-442.

[RHR96] Robbins J. E., Hilbert D. M., and Redmiles D. F.: Extending Design Environments to Software Architecture Design. Proceedings of the 11th Knowledge-Based Software Engineering Conference (KBSE). IEEE Computer Society, 1996. Page 63.

[RJB99] Rumbaugh, J., Jacobson, I., Booch, G., The Unified Modeling Language Reference Manual. Addison-Wesley, 1999.

[RK96] Ran A., Kuusela J., Design Decision Trees. Proceedings of 8th International Workshop on Software Specification and Design, International Workshop on Software Specifications & Design. IEEE Computer Society, 1996. Pages 172-175.

[SEI] Software Engineering Institute, Capability Maturity Model Integration (CMMI). Available online: http://www.sei.cmu.edu/cmmi

[SG96] Shaw M., Garlan D., Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, 1996.

[Sha03] Shaw M., Writing Good Software Engineering Research Papers: Minitutorial. Proceedings of the 25th International Conference on Software Engineering. IEEE Computer Society, 2003. Pages 726-736.

[SKS02] Silberschatz A., Korth H. F., Sudarshan S., Database System Concepts. McGraw-Hill, 2002.

[Som95] Sommerville I., Software Engineering, Fifth Edition. Addison Wesley, 1995.

[SunEJB] Sun Microsystems, Java Platform, Enterprise Edition (Java EE), Enterprise JavaBeans Technology, http://java.sun.com/products/ejb

[SunJEE] Sun Microsystems, Java EE 5, http://java.sun.com/javaee/technologies/javaee5.jsp

[SunJMS] Sun Microsystems, Java Message Service, http://java.sun.com/products/jms

[SunWS] Sun Microsystems, Metro Web Services Overview, http://java.sun.com/webservices, JAX-WS, https://jax-ws.dev.java.net

[Sup] Supply Chain Council, Supply-Chain Operations Reference Model (SCOR) Version 9.0. Available online: http://www.supply-chain.org/cs/root/scor_tools_resources/scor_model/scor_model

[SV06] Stahl T., Völter M., Model-Driven Software Development. Wiley and Sons, 2006.

[SVQ06] Shishkov B., van Sinderen M., Quartel D., SOA-Driven Business-Software Alignment. Proceedings of the IEEE International Conference on e-Business Engineering (ICEBE 2006). IEEE Computer Society, 2006. Pages 86-94.

[SVT07] Shishkov B., van Sinderen M., Tekinerdogan, Model-Driven Specification of Software Services. Proceedings of the IEEE International Conference on E-Business Engineering (ICEBE 2007). IEEE Computer Society, 2007. Pages 13-21.

[SWL+03] Svahnberg M., Wohlin C., Lundberg L., Mattsson M., A Quality-Driven Decision Support Method for Identifying Software Architecture Candidates, International Journal of Software Engineering and Knowledge Management, Volume 13, No. 5, World Scientific, 2003. Pages 547-573.

[SZ92] Sowa J. F., Zachman, J. A., Extending and Formalizing the Framework for Information Systems Architecture, IBM Systems Journal, Volume 31, Number 3, 1992. Pages 590-616.

[TA05] Tyree, J., Ackerman, A., Architecture Decisions: Demystifying Architecture. IEEE Software Volume 22, Issue 2, 2005. Pages 19-27.

[Ta07] Tang A., A Rationale-Based Model for Architecture Design Reasoning, Ph. D. Thesis, Swinburne University of Technology, 2007.

[TAG+05] Tang, A., Ali Babar, M., Gorton, I., and Han, J. 2005. A Survey of the Use and Documentation of Architecture Design Rationale. Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture. IEEE Computer Society, 2005. Pages 89-98.

[TV03] Tanenbaum A. S., van Steen M., Distributed Systems. Principles and Paradigms, International Edition. Prentice Hall, 2003.

[VKZ04] Völter M., Kircher M., and Zdun U., Remoting Patterns – Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware. Wiley, 2004.

[VT] v.d. Aalst W.M.P., ter Hofstede A., Workflow Patterns. Available online: http://www.workflowpatterns.com

[W3C00] Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation 6 October 2000. Available online: http://www.w3.org/TR/REC-xml

[W3C01] W3C. Web Services Description Language (WSDL) 1.1. March 2001. Available online: http://www.w3.org/TR/wsdl

[W3C03] W3C. SOAP Version 1.2, W3C Recommendation 24 June 2003. Available online: http://www.w3.org/TR/2003/REC-soap12-part0-20030624

[W3C04] W3C, Web Services Architecture, W3C Working Group Note 11 February 2004. Available online: http://www.w3.org/TR/2004/NOTE-ws-arch-20040211

[W3C07] W3C, Web Services Policy 1.5 – Framework, W3C Recommendation September 2007. Available online: http://www.w3.org/TR/2007/REC-ws-policy-20070904

[Wah08] Wahler M., Using Patterns to Develop Consistent Design Constraints. Ph. D. Thesis, Swiss Federal Institute of Technology Zurich, 2008.

[WCL+05] Weerawarana S., Curbera F., Leymann F., Storey T., Ferguson D. F., Web Services Platform Architecture. Prentice Hall, 2005.

[WJ05] Witthawaskul W., Johnson R., Transaction Support Using Unit of Work Modeling in the Context of MDA. Proceedings of the Ninth IEEE International EDOC Enterprise Computing Conference. IEEE Computer Society, 2005. Pages 131-141.

[WSI06] Web Services Interoperability. WS-I Basic Profile 1.1, April 2006. Available online: http://www.ws-i.org/Profiles/BasicProfile-1.1.html

[WSI07] Web Services Interoperability. WS-I Basic Security Profile 1.0, March 2007. Available online: http://www.ws-i.org/Profiles/BasicSecurityProfile-1.0.html

[Yah] Yahoo Inc., Pipes: Rewire the Web. Available online: http://pipes.yahoo.com/pipes

[Yip] Yip, J., It's Not Just Standing Up: Patterns of Daily Stand-up Meetings. Available online: http://www.martinfowler.com/articles/itsNotJustStandingUp.html

[You89] Yourdon E., Modern Structured Analysis. Yourdon Press Computing Series, 1989.

[YRS+99] Youngs R., Redmond-Pyle D., Spaas P., and Kahan E., A Standard for Architecture Description, IBM Systems Journal, Volume 38, Number 1, 1999. Pages 32-50.

[ZAH+08] Zdun U., Avgeriou P., Hentrich C., and Dustdar S., Architecting as Decision Making with Patterns and Primitives. Proceedings of the 3rd International Workshop on Sharing and Reusing Architectural Knowledge (SHARK '08). ACM, 2008. Pages 11-18.

[ZD06] Zdun, U., Dustdar, S., Model-Driven and Pattern-Based Integration of Process-Driven SOA Models, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany. Available online: http://drops.dagstuhl.de/opus/volltexte/2006/820

[Zdu07] Zdun U., Systematic Pattern Selection using Pattern Language Grammars and Design Space Analysis. Software: Practice & Experience, 2007.

[ZHD07] Zdun U., Hentrich C., and Dustdar S., Modeling Process-Driven and Service-Oriented Architectures Using Patterns and Pattern Primitives. ACM Transactions on the Web (TWEB), Volume 1, No. 3, ACM, 2007.

Refereed Papers Co-Authored by Thesis Author (in Reverse Chronological Order)

[ZKL+09] Zimmermann O., Koehler J., Leymann F., Polley R., Schuster N., Managing Architectural Decision Models with Dependency Relations, Integrity Constraints, and Production Rules. Accepted for The Journal of Systems and Software, Special Issue on Design Decisions and Rationale in Software Architecture. Elsevier, 2009.

[PZL08] Pautasso C., Zimmermann O., Leymann F., RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision. Proceedings of WWW 2008, ACM, 2008. Pages 805-814.

[ZZG+08] Zimmermann O., Zdun U., Gschwind T., Leymann F., Combining Pattern Languages and Architectural Decision Models into a Comprehensive and Comprehensible Design Method. Proceedings of IEEE WICSA 2008, IEEE Computer Society, 2008. Pages 157-166.

[FCZ07] Fernandez E. Colmondeley P., Zimmermann O., Extending a Secure System Development Methodology to SOA. Proceedings of the 18th International Conference on Database and Expert Systems Applications (DEXA 2007). IEEE Computer Society, 2007. Pages 749-754.

[SZP07] Schuster N., Zimmermann O., Pautasso C., AD_{kwik}: Web 2.0 Collaboration System for Architectural Decision Engineering. Proceedings of the Nineteenth International Conference on Software Engineering & Knowledge Engineering (SEKE 2007), Knowledge Systems Institute

Graduate School, 2007. Pages 255-260.

[ZGK+07] Zimmermann O., Gschwind T., Küster J., Leymann F., Schuster N., Reusable Architectural Decision Models for Enterprise Application Development. Proceedings of QoSA 2007, LNCS 4880/2008, Springer, 2008. Pages 157-166.

[ZGT+07] Zimmermann O., Grundler J., Tai S., Leymann F., Architectural Decisions and Patterns for Transactional Workflows in SOA. Proceedings of ICSOC 2007, LNCS 4749/2007, Springer, 2007. Pages 81-93.

[ZKL07] Zimmermann O., Koehler J., Leymann F., Architectural Decision Models as Micro-Methodology for Service-Oriented Analysis and Design. Proceedings of the Workshop on Software Engineering Methods for Service-oriented Architecture (SEMSOA 2007). Available online: http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-244

[ZKL06] Zimmermann O., Koehler J., Leymann F., The Role of Architectural Decisions in Model-Driven Service-Oriented Architecture Construction. Proceedings of OOPSLA 2006 Workshop on Best Practices and Methodologies in Service-Oriented Architectures, Unipub, 2006. Pages 143-149.

[ZDG+05] Zimmermann O., Doubrovski V., Grundler J., Hogg K., Service-Oriented Architecture and Business Process Management in an Order Management Scenario: Rationale, Concepts, Lessons Learned. Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05). ACM, 2005. Pages 301-312.

[ZSW+05] Zimmermann O., Schlimm N., Waller G., Pestel M., Analysis and Design Techniques for Service-Oriented Development and Integration, INFORMATIK 2005 – Informatik LIVE! Band 2, Beiträge der 35. Jahrestagung der Gesellschaft für Informatik e.V. (GI), Bonn, 2005. Pages 606-611.

[ZMC+04] Zimmermann O., Milinski S., Craes S., Oellermann F., Second Generation Web Services-Oriented Architecture in Production in the Finance Industry, Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '04). ACM, 2004. Pages 283-289.

[BCO+04] Brandner, M., Craes, M., Oellermann, F., Zimmermann, O., Web Services-Oriented Architecture in Production in the Finance Industry, Informatik-Spektrum 02/2004, Springer-Verlag, 2004. Pages 135-145.

Other Publications Co-Authored by Thesis Author (in Reverse Chronological Order)

[SZ08] Schuster N., Zimmermann O., Architectural Decision Knowledge Wiki. Available online: http://www.alphaworks.ibm.com/tech/adkwik

[TMR+08] Tai S., Mikalsen T., Rouvellou I., Grundler J., Zimmermann O., Transactional Web Services. Invited Book Chapter, in: Georgakopoulos D. and Papazoglou M. P. (eds.), Service-Oriented Computing. MIT Press, 2008

[ZSE08] Zimmermann O., Schuster N., Eeles P., Modeling and Sharing Architectural Decisions, Part 1: Concepts. IBM developerWorks, 2008.

[ZKG04] Zimmermann O., Krogdahl, P., Gee C., Elements of Service-Oriented Analysis and Design. IBM developerWorks, 2004.

[ZM04] Zimmermann O., Müller F., Web Services Project Roles. IBM developerWorks, 2004.

[ZTP03] Zimmermann O., Tomlinson M., Peuser S., Perspectives on Web Services: Applying SOAP, WSDL, and UDDI to Real-World Projects. Springer Professional Computing, 2003.

[WTZ+02] Wahli U., Tomlinson M., Zimmermann O., Deruyk W., Hendricks, D., Web Services Wizardry with WebSphere Studio Application Developer. IBM Redbook, 2002.

Index