universität
wien

MASTERARBEIT

Titel der Masterarbeit

„Web-basiertes Tool zur Modellierung und Unterstützung von wiederverwendbaren Architectural Design Decisions“

Verfasser

Dian HAN, BSc

angestrebter akademischer Grad
Diplom-Ingenieur (Dipl.-Ing.)

Wien, 2014

Studienkennzahl lt. Studienblatt:	**A 066 926**
Studienrichtung lt. Studienblatt:	**Masterstudium Wirtschaftsinformatik**
Betreuer:	**Univ.-Prof. Dr. Uwe Zdun**

Danksagung

An dieser Stelle möchte ich meinen persönlichen Dank ganz herzlich an alle Personen aussprechen, die mir diese Masterarbeit ermöglicht haben. Mein erster Dank geht an Univ. -Prof. Dr. Uwe Zdun und Dipl. -Ing. Ioanna Lytra für die hervorragende Betreuung, die mir während des gesamten Erstellungsprozesses mit Wissen, Rat und Tat zur Seite stand. Des Weiteren bedanke ich mich bei meinen Eltern und ganz speziell meiner Frau YingJie Zhou, die mich mit Geduld und Unterstützung durch meine Studienzeit begleitet haben.

Abstract

In der heutigen Zeit wird die Softwarearchitektur immer mehr als eine Reihe von *Architectural Design Decisions* (ADDs) anstatt als *Components and Connectors* (C&C) betrachtet. Die effektive und effiziente Verwendung von ADDs kann die zwei wichtigen Gefahren bzw. Probleme architektonischer Drift und architektonische Erosion während der Dokumentation einer Softwarearchitektur vermeiden. Viele ausgereifte Dokumentationsmethoden und –ansätze wurden von diversen Wissenschaftlern und Organisationen entwickelt, um die ADDs zu entscheiden. Zum Beispiel ATAM, CBAM, SAAM und ArchPad. Im Rahmen der vorliegenden Arbeit werden viele *Achitectural Decision Models* (ADMs) und dafür geeignete Tools zur Erfassung der ADDs betrachtet. Im praktischen Teil dieser Arbeit wird das Web-basiertes Tool WebADvISE zur Modellierung und Unterstützung von wiederverwendbare ADDs entwickelt. Die meisten Funktionalitäten des WebADvISE-Tools stammen vom ***A****rchitectural* ***D****esign dec****I****sion* ***S****upport fram****E****work* (ADvISE). Es basiert auf den musterbasierten *Architectural Decision Model* (ADM) und *Questions, Options, Criteria* (QOC) Methoden. WebADvISE bietet auch eine Kollaborationsmöglichkeit für verschiedene Softwarearchitekten an. Als eine Fallstudie werden fünf architekturmusterbasierte ADMs und einige dafür geeignete ADDs für ein Universitätsinformationssystem mittels WebADvISE entwickelt.

Inhaltverzeichnis

1. Einleitung 1
    1.1 Problemstellung und Motivation 1
    1.2. Zielsetzung 3
    1.3 Aufbau der Arbeit 4
2. Related Work 6
    2.1 Background 7
        2.1.1 Wiederverwendbare Softwarearchitekturdokumentation 7
        2.1.2 Softwarearchitekturschichten 9
        2.1.3 Probleme in der Softwarearchitektur 11
    2.2 Architectural Design Decisions 13
        2.2.1 Historische Evolution der ADDs 13
        2.2.2 Dokumentationsmethoden und -ansätze der ADD 18
        2.2.3 Methoden zum Auffinden von ADDs 21
        2.2.4 Erfassung der ADDs von Architekturmustern 26
        2.2.5 Lösungsansätze für die Probleme in der Softwarearchitektur 28
    2.3 Modelle und Tools der ADDs 31
        2.3.1 Modelle der ADDs 31
        2.3.2 Existierende Tools für ADD 39
        2.3.3 Der Vergleich zwischen unterschiedlichen Tools 46
3. Web-basiertes Tool WebADvISE 50
    3.1 Grundlagen 51
        3.1.1 ADM für WebADvISE 51
        3.1.2 ADvISE 52
        3.1.2 QOC Methode und Fragebogen 54
    3.2 Implementierung 55
        3.2.1 Architektur und Architekturentscheidungen von WebADvISE 55
        3.2.2 Gebrauchstechnologien 61
        3.2.3 WebADvISE als Plug-Ins von Eclipse RAP 61
        3.2.4 MVC Entwurfsmuster für UI Design 64
    3.3 Funktionalitäten und Handbuch 64
        3.3.1 Grundsätzliche Funktionalitäten 66
        3.3.2 Verwaltung eines ADMs 67
        3.3.3 Bearbeitung eines ADMs 68
        3.3.4 Erfassung einer Architekturentscheidung 72
    3.4 Kollaborative Arbeit 75
        3.4.1 Konflikte der Ressourcen des Editors 75
        3.4.2 Atomisierung der Änderungszustände und Rollback 77
        3.4.3 Technische Implementierung und Log View 77
4. Fallstudie mittels WebADvISE 79
    4.1 Grundlagen 79
    4.2 Wiederverwendbare Muster-basierte ADMs 79
        4.2.1 Applikationsstruktur 80

4.2.2 Verteilte Komponenten....83
4.2.3 Taskverarbeitung....87
4.2.4 UI Design....90
4.2.5 Datenspeicher....92
4.3 Herstellung der entwickelten ADMs in WebADvISE....94
4.4 Herstellung der ADDs für Universitäts-IS mittels entwickelter ADMs in WebADvISE....94
4.4.1 Applikationsstruktur....96
4.4.2 Verteilte Komponenten....97
4.4.3 Taskverarbeitung....99
4.4.4 UI Design....100
4.4.5 Datenspeicherung....101
4.5 Bearbeitung der entschiedenen ADDs....101
4.6 Kollaborative Arbeit....102
5. Zusammenfassung....105
5.1 Ergebnisbetrachtung....105
5.2 Zukünftige Erweiterungen....106
6. Anhang....108
7. Literaturverzeichnis....111

Abbildungsverzeichnis

Abbildung 1: Entscheidungsschicht eingebettet in das „4+1“ Schichtenmodell.....10
Abbildung 2: Workflow Diagramm einer Entscheidung für Dokumentation-Template der Architekturentscheidung von Tyree [vgl. Tyree and Akerman, 2005] 18
Abbildung 3: Ein klassischer konzeptioneller Ablauf des ATAM [vgl. Kazman et al., 2000] 24
Abbildung 4: Archium Modell für Architectural Design Decisions [Quelle: Figur 1, Jansen and Bosch, 2005] 33
Abbildung 5: SOA Prozessablauf für Architectural Design Decisions [Quelle: Figur 1, Zimmermann et al., 2006].....34
Abbildung 6: Prozessablauf des DAMSAK für Architectural Design Decisions [Quelle: Figur 3, Babar et al., 2005].....35
Abbildung 7: Beziehungen zwischen architektonischer Rationale und architektonischen Elementen [Quelle: Figur 2, Tang et al., 2006].....35
Abbildung 8: Der Prozess einer Entwurfsentscheidung für das AREL Modell [Quelle: Figur 1, Tang et al., 2006].....36
Abbildung 9: Die grafischen Elemente des ADPM Diagramms [Quelle: Figur 2, Zalewski et al., 2011] 37
Abbildung 10: Überblick über den Prozess des STREAM-ADD [Quelle: Figur 2, Dermeval et al., 2012] 38
Abbildung 11: Übersicht des FuzzyDS [Quelle: Figur 3, Lytra and Zdun, 2013].....39
Abbildung 12: Konfigurationsstruktur des Archium Tools [Quelle: Figur 4, Jansen, 2007].....40
Abbildung 13: Übersicht des ArchiTech [Quelle: Figur 1, Ameller et al., 2012].....44
Abbildung 14: Ablaufsprozesse von ArchPad.....45
Abbildung 15: ADMM in ADvISE sowie in WebADvISE [Quelle:ADvISE, 2014] 53
Abbildung 16: QOC Darstellung (Phase 1) 57
Abbildung 17: QOC Darstellung (Phase 2) 57
Abbildung 18: QOC Darstellung (Phase 3) 59
Abbildung 19: Betriebsumgebung der WebADvISE [vgl. RAP, 2014] 62
Abbildung 20: Ein typischer Editier-Bereich von WebADvISE 63
Abbildung 21: MVC Struktur für UI Design von WebADvISE 64
Abbildung 22: Use-Case Diagramm zur Beschreibung aller möglichen Use-Cases der Benutzer .66
Abbildung 23: Design Patterns Editor 68
Abbildung 24: Design Solutions Editor 69
Abbildung 25: Die bezogenen Muster hinzufügen 69
Abbildung 26: Architectural Decisions Editor 70
Abbildung 27: Editieren Fragen (Questions) 71
Abbildung 28: Eine Frage mit einer erwarteten Freitext-Antwort 71
Abbildung 29: Eine Frage mit den erwarteten alternativen Optionen.....72
Abbildung 30: Generieren eines Fragebogen-Files.....73
Abbildung 31: Fragebogen-Editor 74
Abbildung 32: Lese-Schreib-Konflikte und Schreib-Schreib-Konflikte.....76
Abbildung 33: Change Logs in Log View 78
Abbildung 34: OSI-Modell 81

Abbildung 35: Die Baumstruktur für Applikationsstruktur 82
Abbildung 36: Baumstruktur für verteilte Komponenten 87
Abbildung 37: Pipes-and-Filters 88
Abbildung 38: eine Struktur des Compilers für die Programmiersprache Java 88
Abbildung 39: Baumstruktur für die Verarbeitungsaufgaben 90
Abbildung 40: Baumstruktur für das UI Design 92
Abbildung 41: Baumstruktur für Datenspeichner 93
Abbildung 42: Die Entscheidungen in Baumstruktur für die Applikationsstruktur 96
Abbildung 43: Die Entscheidungen in der Baumstruktur für Verteilte Komponenten 99
Abbildung 44: Die Entscheidungen in der Baumstruktur für die Taskverarbeitung 100
Abbildung 45: Die Entscheidungen in der Baumstruktur für das UI Design 101
Abbildung 46: Die Entscheidungen in der Baumstruktur für die Datenspeicherung 101
Abbildung 47: Inhalte eines Architekturentscheidungseditors 102
Abbildung 48: Kollaborative Arbeit der zwei Benutzer 103
Abbildung 49: Rollback zur Vermeidung der Konflikte (Alte Version wieder editieren) 104

Tabellenverzeichnis

Tabelle 1: Dokumentation-Template der Architekturentscheidung von Tyree [vgl. Tyree and Akerman, 2005] 16
Tabelle 2: Sieben Methoden zur architektonischen Entscheidungsfindung 22
Tabelle 3: Prozessschritte des ATAM [vgl. Kazman et al., 2000] 24
Tabelle 4: Vergleich unterschiedlicher Methoden zur Entscheidungsfindung [vgl. Xu et al., 2007] 26
Tabelle 5: Vermeidung der Probleme in der Softwarearchitektur mittels Use-Cases Modell. 31
Tabelle 6: Der Vergleich zwischen unterschiedlichen Modellen, die acht bekannte ADD Tools unterstützen[vgl. Shahin et al., 2009a]..... 46
Tabelle 7: Der Vergleich zwischen unterschiedlichen Modellen, die acht bekannte ADD Tools unterstützen[vgl. Shahin et al., 2009a]..... 47
Tabelle 8: Der Vergleich zwischen unterschiedlichen ADD Tools[vgl. Tang et al., 2010]..... 49
Tabelle 9: Die Gebrauchstechnologien von WebADvISE..... 61
Tabelle 10: Architekturentscheidungen, Muster, Lösungen und Fragen von Applikationsstruktur 82
Tabelle 11: Architekturentscheidungen, Muster, Lösungen und Fragen von den verteilten Komponenten..... 86
Tabelle 12: Architekturentscheidungen, Muster, Lösungen und Fragen hinsichtlich der Verarbeitungsaufgaben..... 90
Tabelle 13: Architekturentscheidungen, Muster, Lösungen und Fragen bezogen auf das UI Design 91
Tabelle 14: Architekturentscheidungen, Muster, Lösungen und Fragen bezogen auf den Datenspeicher..... 93
Tabelle 15: Anforderungen des Universitäts-IS [vgl. UIS, 2013] 96
Tabelle 16: Die Ergebnisse dieser Fallstudie 105

Liste der Abkürzungen:

AD	Architectural Decision (Architekturentscheidung)
ADD	Architectural Design Decision (Architektur-Entwurfsentscheidung)
ADDEX	Architectural Design Decision Exploration
ADDSS	Architecture Design Decision Support System Tool
ADL	Architecture Description Languages
ADM	Architectural Decision Model (Architekturentscheidungsmodell)
ADMM	Architectural Decision Meta-Model (Architekturentscheidungsmeta-modell)
AD_{kwik}	Architectural Decision Knowledge Wiki
ADvISE	Architectural Design Decision Support Framework
AK	Architectural Knowledge (Architekturwissen)
API	Application Programming Interface
ArchPad	Architectural pattern- and decision-based design Method
ATAM	Architecture Tradeoffs Analysis Method
AREL	Architecture Rationale and Elements Linkage
C&C	Components and Connectors
CBAM	Cost Benefit Analysis Method
DD	Design Decision (Entwurfsentscheidung)
DDR	Design Decision Rationale (Entwurfsentscheidungbegründung)
DIE	Integrated Development Environment
DSL	Domain Specific Language
EJB	Enterprise JavaBean
EMF	Eclipse Modeling Framework
HTTP	Hypertext Transfer Protocol
J2EE	Java Platform, Enterprise Edition
KA	Knowledge Architect
MAD 2.0	Maps of Architectural Decisions 2.0
MVC	Model View Controller

NFR	Non-functional requirement (Nicht-funktionale Anforderungen)
OSGI	Open Service Gateway Initiative
PAKME	Process-based Architecture Knowledge Management Environment
QA	Quality Attribute (Qualitätsattribut)
QOC	Questions, Options and Criteria (Fragen, Optionen und Kriterien)
RADM	Reusable Architectural Decision Model (Wiederverwendbares Architekturentscheidungsmodell)
RAP	Rich Ajax Platform
REST	Representational State Transfer
RPC	Remote Procedure Call
RWT	RAP Widget Toolkit
SAAM	Software Architecture Analysis Method
SAHP	Applying Analytical Hierarchy Process to Software Architecture Decisions
SOA	Service-oriented architecture
SOAP	Simple Object Access Protocol
SRS	Software requirements specification
SSAR	Scenario-based Software Architecture Reengineering
STREAM-ADD	Strategy for Transition between Requirements and Architectural Models with Architectural Decisions Documentation
SWT	Standard Widget Toolkit
Universitäts-IS	Universitätsinformationssystem
VbMF	View-based Modeling Framework
WebADvISE	Web-based Architectural Design decIsion Support framEwork

1. Einleitung

1.1 Problemstellung und Motivation

Wie kann ein Softwarearchitekt die Softwarearchitektur effektiv, effizient und wiederverwendbar dokumentieren? Dies ist die erste Problemstellung in dieser Arbeit. Die vier traditionellen Hauptformen der Softwarearchitektur eines Systems sind, natürliche Sprachen, Bilder, formale Sprachen und *Architecture Description Languages* (ADLs). Um die Architektur präziser darzustellen, verwendet der Softwarearchitekt oft das „4+1“ Schichtenmodell [Kruchten, 1995], das die Architektur eines Systems in verschiedene Schichten aufteilt. Während der Entwicklung dieses Modells wird eine neue Entscheidungsschicht [Kruchten et al., 2009] in dieses Modell eingefügt. Diese neue Schicht kann als eine Kombination einer Reihe der *Architectural Design Decisions* (ADDs) angesehen werden. In dem Dokumentationstemplate der Architekturentscheidungen von Jeff Typree und Art Akerman [Tyree and Akerman, 2005] wird eine ADD in vierzehn Komponenten aufgeteilt. Das Template beschreibt jede Komponente und deren Beziehungen bei der Erstellung einer ADD.

Bei der Entwicklung einer Softwarearchitektur mit traditionellen Methoden gibt es die möglichen beiden Gefahren bzw. Nachteile architektonischer Drift (*architecture drift*) und architektonische Erosion (*architectural erosion*) [Perry and Wolf, 1992] und acht Hauptprobleme [vgl. Bosch, 2004, vgl. Jansen and Bosch, 2004, vgl. Jansen, 2007]. Um diese Gefahren und Probleme in der Praxis zu vermeiden, kann der Softwarearchitekt eine Reihe von ADDs anstatt der traditionellen Methoden verwenden. Der Vorteil der Verwendung der ADDs ist eine enge Beziehung zwischen diesen ADDs. Damit kann die Architektur eines Softwaresystems durch diese ADDs rekonstruiert werden, z.B. können andere Stakeholder davon profitieren. In der Arbeit wird weiter unten ein *Use-Case* Modell dazu angegeben. Dieses Modell beschreibt,

was die verbundenen Stakeholder in dem Architekturentscheidungsprozess machen sollen, um die oben erwähnten Gefahren und Probleme zu vermeiden.

Wie kann ein Softwarearchitekt ADDs genau dokumentieren? Dies ist die zweite Problemstellung in dieser Masterarbeit. Im Zuge dieser Arbeit werden einige Dokumentationsmethoden und vier Dokumentationsansätze empfohlen. Eine wichtige Methode ist *Questions, Options, and Criteria* (QOC) [MacLean et al., 1991]. Diese Methode konzentriert sich auf eine Frage/Antwort-Abfolge. *Questions* identifizieren die wichtigsten Kernprobleme eines Entwurfs, *Options* bieten die möglichen Antworten auf die Probleme an, und *Criteria* sind die Kriterien für die Beurteilung und den Vergleich der Optionen. Diese Vorgehensweise erlaubt einem Softwarearchitekt ADDs zu begründen und zu entscheiden.

Wie kann der Softwarearchitekt ADDs erfassen? Das ist die dritte Problemstellung in dieser Masterarbeit. Der Softwarearchitekt kann ADDs basierend auf einer Sammlung und Analyse der ursprünglichen funktionalen und nichtfunktionalen Anforderungen und der Qualitätsattribute der Software definieren. Weiters gibt es einige existierende und bewährte Methoden um ADDs zu erfassen und zu entscheiden. Zum Beispiel ATAM (*Architecture Tradeoffs Analysis Method*) [Kazman et al., 2000, Nord et al., 2003], SAAM (*Software Architecture Analysis Method*) [Kazman et al., 1994, Ionita et al., 2002], und ArchPad (*Architectural pattern- and decision-based design Method*) [Zimmermann et al., 2008]. Die meisten dieser Methoden bestehen aus vier Organisations- und Ablaufprozessen, einer Analyse der Design-Alternativen, einem Vergleich der Design-Alternativen, einer Identifikation der Qualitätsattribute und der Analyse der Trade-offs.

Einzelpersonen oder Organisationen haben verschiedene *Architectural Decision Model*s (ADMs) und dazu passende Tools entwickelt. Diese entwickelte Modelle und Tools können dem Softwarearchitekt helfen wiederverwendbare ADDs aus unterschiedlichen Blickwinkeln zu entscheiden. Zum Beispiel, das musterbasierte ADM [Harrison et al., 2007] bietet einen Mechanismus an, um die ADDs basierend

auf den Entwurfsmustern zu entscheiden. Diese Arbeit befasst sich mit Vor- und Nachteilen dieser ADMs und dabei zum Einsatz kommender Tools.

Basierend auf den zwei oben erwähnten Beispielen der QOC-Dokumentationsmethode für Architekturentscheidungen und dem musterbasierten ADM zur Erfassung der ADDs wird ein webbasiertes kollaboratives Tool WebADvISE zur Modellierung und Unterstützung von wiederverwendbaren ADDs als ein Teil dieser Arbeit entwickelt. Damit können mehrere Softwarearchitekten ein musterbasiertes ADM zeitgleich und synchron erstellen und editieren. Dann können sie ihre ADDs bzw. Architekturentscheidungen basierend auf den Antworten eines QOC Fragebogens zusammen entscheiden. WebADvISE wird basierend auf ADvISE [ADvISE, 2014] als Eclipse RAP Applikation entwickelt. Es basiert auf OSGI. Um eine hohe Skalierbarkeit und Modifizierbarkeit zu gewährleisten wird das MVC-Entwurfsmuster in WebADvISE verwendet.

Um dieses webbasierte Tool genauer und mittels einer praktischen Anwendung zu überprüfen, wird eine Fallstudie „Universitäts-IS“ durchgeführt. Dafür werden die fünf wiederverwendbaren musterbasierten ADMs Applikationsstruktur, Verteilte Komponenten, Taskverarbeitung, UI Design und Datenspeicherung definiert. Jedes dieser ADMs unterstützt ein oder mehrere Architekturmuster. Anschließend werden einige ADDs als Ergebnisse dieser Fallstudie behandelt.

1.2. Zielsetzung

Das Ziel der Arbeit ist die Verwendung von ADDs in der Softwarearchitektur zu diskutieren und ein webbasiertes Tool zur Modellierung und Unterstützung von wiederverwendbaren ADDs zu entwickeln. Durch die Analyse elf existierender Tools zur Erfassung von ADDs und deren Vergleich kann man darauf schließen, dass die meisten existierenden Tools außer AD_{kwik} und KA Tool-Suite keine kollaborative Arbeit an ADDs unterstützen. Mittels dieser Tools muss ein Softwarearchitekt entweder Export / Import-Mechanismen oder Versionsverwaltungsprogramme

verwenden, um die Daten eines ADDs Projekts mit anderen Softwarearchitekten austauschen zu können. Außerdem müssen plattformabhängige Tools (z.B. Archium Tool [Jansen and Bosch, 2005]) auf der jeweils unterstützten Plattform installiert werden. Aus diesen zwei Gründen wird das Web-basiertes Tool WebADvISE zur Modellierung und Unterstützung wiederverwendbarer ADDs entwickelt. Das wichtigste Entwicklungsziel von WebADvISE ist es, einen Kollaborationsmechanismus für mehrere Benutzer anzubieten. Mit diesem plattformunabhängigen Tool können alle Benutzer bzw. beteiligte Stakeholder eines Softwaresystems zeitgleich und synchron ein WebADvISE ADM-Projekt zusammen bearbeiten. Konkret ergeben sich daraus folgende Vorteile.

- Der universelle Zugriff auf WebADvISE über das Internet wird ermöglicht.
- WebADvISE steht allen Benutzern ohne Installation zur Verfügung.
- WebADvISE ist plattformunabhängig. Der Zugriff erfolgt über Web-Browser.
- WebADvISE bietet einen Kollaborationsmechanismus für mehrere Benutzer an. Dabei kann ein WebADvISE Projekt von verschiedenen Benutzern zeitgleich und synchron bearbeitet werden.
- Änderungen und Upgrades müssen bei WebADvISE nur auf der Server-Seite durchgeführt werden.

1.3 Aufbau der Arbeit

Der Aufbau dieser Arbeit gliedert sich wie folgt: Im Kapitel 2 werden die grundlegende Begriffe Dokumentation und Schichten der Softwarearchitektur aus dem Bereich ADDs erläutert. Außerdem werden die möglichen Probleme in der Softwarearchitektur diskutiert. Dieses Kapitel befasst sich auch mit der historischen Evolution der ADD, Dokumentationsmethoden und –ansätze der ADD und mit Methoden zum Auffinden von ADDs. Am Ende dieses Kapitels wird ein Lösungsansatz für die oben erwähnten Probleme vorgestellt und es werden die existierende Modelle und Tools analysiert und verglichen.

Kapitel 3 beschreibt die Grundlagen des webbasierten Tools WebADvISE, musterbasierter ADMs, des Tools ADvISE, der QOC Methode und deren Fragebogen. Durch die Verwendung der Dokumentationsmethode QOC und der architekturmusterbasierten Entwurfsmethode ArchPad werden die Architektur und deren Entscheidungsprozesse von WebADvISE rekonstruiert. Als Ergebnisse werden die verwendeten Technologien definiert. Dieses Kapitel befasst sich auch mit der Architektur der WebADvISE Frontend und Backend. Die wichtigen Funktionalitäten werden in diesem Kapitel vorgestellt. Und der Mechanismus „Atomisierung der Änderungszustände und Rollback“ zur Durchführung der Kollaboration und der Vermeidung von Ressourcenkonflikten wird beschrieben.

Kapitel 4 ist eine Fallstudie Universitätsinformationssystem (Universitäts-IS). Dafür werden die fünf wiederverwendbaren architekturmusterbasierten ADMs, Applikationsstruktur, Verteilte Komponenten, Taskverarbeitung, UI Design und Datenspeicherung von WebADvISE abgedeckt. Damit werden einige ADDs für Universitäts-IS entschieden.

Kapitel 5 befasst sich mit den Ergebnissen der Fallstudie Universitäts-IS und beschreibt mögliche zukünftige Erweiterungen.

2. Related Work

Da Softwareentwicklung immer komplexer wird, spielt Softwarearchitektur eine wichtigere Rolle als früher im Softwareentwicklungsprozess. Die Architektur eines Softwaresystems kann als Blaupause, Plan und Qualitätsprädiktor des Systems auf der höchsten Abstraktionsebene angesehen werden. Es beschreibt normalerweise die Hauptkomponenten und deren wichtigste Interaktionen. In den letzten Jahren wurden ADDs zur Erfassung und Nutzung des *design rationale* und des *architectural knowledge* (AK) verwendet. Softwarearchitektur wird immer mehr als eine Reihe von ADDs anstatt als *components and connectors* (C&C) angesehen. Aber was ist eine ADD? Anton Jansen und Jan Bosch geben folgende Definition:

> *A description of the set of architectural additions, subtractions and modifications to the software architecture, the rationale, and the design rules, design constraints and additional requirements that (partially) realize one or more requirements on a given architecture [Jansen and Bosch, 2005].*

Jansen und Bosch behaupten, dass in der aktuellen Sicht auf Softwarearchitektur der Begriff ADD fehlt, obwohl ADDs schon eine Schlüsselrolle bei dem Entwurf, der Entwicklung, der Evolution, der Wiederverwendbarkeit und der Integration einer Softwarearchitektur spielen. Softwarearchitektur besteht eigentlich aus eine Reihe von ADDs. Es kann den Wissensverlust (*knowledge vaporization*) [Bosch, 2004] von den Informationen der Entwurfsentscheidungen offensichtlich reduzieren, da ADDs bzw. Entwurfsentscheidungen ein expliziter Bestandteil der Architektur sind. Dies ist ein Grund, warum die ADDs mehr und mehr in der Praxis verwendet werden.

Im Kapitel 2 wird der Background für ADDs besprochen. Zuerst befasst sich Kapitel 2.1 mit den traditionellen Softwarearchitektur-Dokumentationsmethoden bzw. den Hauptformen zur Beschreibung einer Softwarearchitektur. Am Ende dieses Kapitels beschreiben wir die möglichen Probleme, die diese traditionellen Methoden mit sich

bringen können. Dann werden die Schichten der Softwarearchitektur besprochen, da ADDs in der Entscheidungsschicht angesiedelt sind. Kapitel 2.2 stellt die historische Evolution, die Dokumentationsmethoden und –ansätze, die Methoden zum Auffinden von ADDs und die architekturmusterbasierte Methode vor. Am Ende dieses Kapitels werden die Lösungsansätze für die Probleme in der Softwarearchitektur angeführt und es wird ein *Use-Case* Modell entwickelt. Das Kapitel 2.3 beschreibt die Modelle und vergleicht dabei zum Einsatz kommende Tools.

2.1 Background

2.1.1 Wiederverwendbare Softwarearchitekturdokumentation

Eine Kernfragestellung der Softwarearchitektur ist die Wiederverwendung einer ausgereiften Softwarearchitektur, um anderen Stakeholdern ein besseres Verständnis der Software zu ermöglichen. Man benötigt eine gute umfassende Dokumentation, mit der ein Softwaresystem dargestellt, beschrieben, spezifiziert, untergliedert werden kann und in einzelne Bestandteile sowie die Beziehungen dieser Teile zueinander zerlegt werden kann. Denn nur auf dieser Weise kann man sagen, dass die Architektur wiederverwendbar (im Sinne von *Reconstruction*) ist.

Die wiederverwendbare Architekturdokumentation ist der krönende Schritt in der Softwareentwicklung. Ohne eine entsprechend wiederverwendbare Dokumentation zur Beschreibung der Architektur wird ein Softwareprojekt sicher scheitern. Eine gute Architektur ist auch nutzlos, wenn niemand sie versteht oder die Stakeholder diese Architektur missverstehen. Falls man sich die Mühe macht, um eine robuste und ausgereifte Softwarearchitektur zu entwickeln, muss man sicherstellen, dass ausreichend detaillierte, unmissverständliche und organisierte Beschreibungen erstellt werden. Damit können andere Stakeholder schnell die benötigten Informationen finden. Andernfalls würde ihr Aufwand verschwendet werden, weil die Architektur unbrauchbar ist [vgl. Clements, 2002]. Um eine Softwarearchitektur deutlich zu beschreiben, verwenden die Softwarearchitekten verschiedene Methoden, die zum

Teil auch kombiniert werden.

Es gibt fünf Hauptformen, um die Architektur zu beschreiben. Erstens natürliche Sprachen mit der Softwarearchitekten am einfachsten die Architektur der Software beschreiben können. Zweitens Bilder, durch ein unkompliziertes und deutliches Bild lässt sich eine Softwarearchitektur intuitiver erfassen und besser verstehen. Zum Beispiel, die Abbildung 19 beschreibt die Betriebsumgebung von WebADvISE, damit man die Struktur von WebADvISE einfacher verstehen kann. Drittens formale Sprachen deren Hauptmerkmal die Abstraktheit ist. Sie sind keine konkrete Sprachen sondern mathematische Beschreibungen. Formale Sprachen nutzen oft mathematische präzise Eigenschaften im Umgang mit Zeichenketten zur Beschreibung einer Softwarearchitektur. Zum Beispiel, *Domain Specific Languages* (DSLs). Viertens *Architecture Description Languages* (ADLs), diese Sprachen werden zur Beschreibung der Software- oder Systemarchitektur entwickelt. Sie stehen sowohl in grafischer als auch in textueller Form zur Verfügung. Ihre Haupteigenschaft ist, dass eine technische Architektur durch diese Sprache eindeutig für die Stakeholder und Softwareentwickler beschrieben wird. Es gibt eine Reihe spezieller ADLs, z.B. Acme (entwickelt von CMU), Darwin (entwickelt von Imperial College London). Fünftens Diagramme, im Unterschied zu oben genannten einfachen Bildern sind Diagramme grafische Modellierungs-/Architektur-Sprachen. Zum Beispiel UML-Diagramme oder QOC Diagramme. Da QOC Diagramme ein wichtiger Bestandteil dieser Masterarbeit sind, werden sie zur Beschreibung der Architektur von WebADvISE im Kapitel 3.2.1 verwendet.

Diese vorgeschlagenen Methoden bzw. Formen ermöglichen es eine Softwarearchitektur gut und deutlich zu beschreiben, aber wir brauchen noch einen systematischeren und ausführlicheren Ansatz, indem wir unsere Softwarearchitektur in strukturierten Einheiten darstellen, den so genannten Softwarearchitekturschichten.

2.1.2 Softwarearchitekturschichten

Wie bereits beschrieben ist dieser Dokumentationsvorgang ein Zerlegungsprozess eines Systems. Um die richtige Übertragung und Beschreibung der Informationen dieses Dokumentationsvorgangs zu ermöglichen, muss man genau definieren, welchen Teil oder welche Teile der Softwarearchitektur, man jeweils bespricht. Dafür eignet sich die Unterteilung in Schichten. 1995 entwickelte Philippe Kruchten sein „4+1"-Schichten Softwarearchitekturmodell [Kruchten, 1995], um einen „*Blueprint*" des Softwaresystems aus verschiedenen Blickwinkeln, für die verschiedenen Stakeholder, sowie Endnutzer, Projektmanager, Entwickler und Softwaretester anzubieten. Das „4+1"-Schichtenmodell besteht aus fünf Teilen: *logical view*, *development view*, *process view* und *physical view*. *Use Cases view* bzw. *Szenarios view* kann als zusätzliche Schicht gesehen werden, um die Softwarearchitektur besser zu beschreiben, was das „+1" ausdrücken soll. Es wird verwendet, um die Beschreibung der Architektur eines Softwaresystems zu organisieren.

Danach schlugen Philippe Kruchten et al. vor, dass die Entwurfsentscheidungen, die entsprechende Einschränkungen oder Abhängigkeiten von anderen Entscheidungen bewertet werden müssen und die Architektur auf Schichten und Schichttypen basieren muss. Diese Schichten und Schichtentypen müssen die Bedeutung der Dokumentation der Entwurfsentscheidungen hervorheben. Allerdings gab es damals keine detaillierte Informationen darüber, wie dies zu tun und keine angemessene Prozesse für die Erfassung und Dokumentation dieser Entscheidungen. In den Jahren 1996 - 2006 erschienen noch viele andere Lösungsansätze, um dem Softwarearchitekt die Erfassung der ADDs in dem Entwicklungsprozess zu erleichtern. Zum Beispiel Philips' BAPO/CAFCR, Nokia's ASC, *Architecture Tradeoffs Analysis Method* (ATAM), *Cost Benefit Analysis Method* (CBAM) und *Software Architecture Analysis Method* (SAAM). Siehe dazu auch Kapitel 2.2.3 [vgl. Kruchten et al., 2009].

Von 2004 bis 2008 gab es zahlreiche Arbeiten [Tyree and Akerman, 2005, Kruchten et al., 2006, Capilla, 2007] zur Darstellung und Erfassung von ADDs, und viele Empfehlungen zur Unterstützung der Erstellung und der Nutzung des *Architectural knowledges* wurden definiert. Mehrere Ansätze beschreiben und stellen Entwurfsentscheidungen als *first-class-entities* dar. 2005 definieren Nick Rozanski und Eoin Woods [Rozanski and Woods, 2005, vgl. Dueñas and Capilla, 2005] sechs Sichtweisen (*viewpoints*), die die wichtigsten architektonischen Aspekte oder Elemente der Softwaresysteme besser klarstellen [vgl. Kruchten et al., 2009].

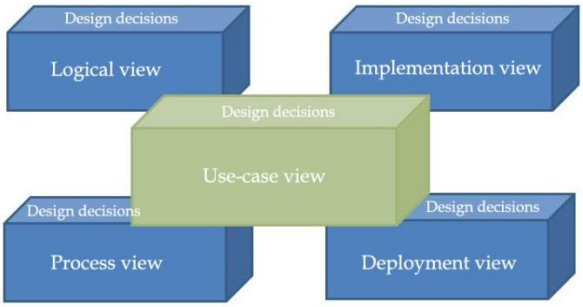

Abbildung 1: Entscheidungsschicht eingebettet in das „4+1“ Schichtenmodell

2009 veröffentlichte der Erfinder des „4+1“ Schichtenmodell Philippe Kruchten et al., wie sich die neue Schicht Entscheidungsschicht in das „4+1“ Schichtenmodell einfügen lässt. Traditionelle Architekturschichten dokumentieren die Entwurfsentscheidungen nicht, aber für Architekten ist es wichtig diese Entwurfsentscheidungen zu erfassen. Unter diesem Gesichtspunkt ist die Entscheidungsschicht sehr wichtig. Um die Entwurfsentscheidungen explizit zu erfassen und dokumentieren, stellt dieses *Architectural knowledge* eine neue Querschnittsansicht dar, die mit anderen Schichten überlappt wie auch in Abbildung 1 ersichtlich. Entwurfsentscheidungen verbessern die Rückverfolgbarkeit der produzierten Software-Engineering-Artefakte im Software-Lebenszyklus. Die Erfassung der Abhängigkeiten zwischen verschiedenen Entscheidungen unterstützt die Impactanalyse während des Hinzufügens, Änderns oder Entfernens der Entscheidungen. Die dokumentierten Entwurfsentscheidungen erleichtern nicht nur das allgemeine Verständnis eines Systems für die Softwarearchitekten insbesondere

bei Personalfluktuation, sondern auch den Wissenaustausch und Bewertungsverfahren, da unterschiedliche Stakeholder die Begründung von Entscheidungen der Vergangenheit leicht überprüfen können [vgl. Kruchten et al., 2009].

2.1.3 Probleme in der Softwarearchitektur

In den vorangegangenen Kapiteln wurden wichtige Begriffe, Eigenschaften, Dokumentationsmethoden und Schichten der Softwarearchitektur ganz kurz beschrieben. In diesem Kapitel besprechen wir einige häufige Probleme der traditionellen Softwarearchitektur, und es wird im folgenden Kapitel (siehe Kapitel 2.2.5) erklärt, wie man diese Probleme lösen kann.

Wie schon gezeigt wurde beschreiben die meisten Dokumentationsformen eine Softwarearchitektur in Schichten. In den letzten Jahren erstellen Softwarearchitekten diese Dokumentation als eine Menge von Entwurfsentscheidungen, die sie während der Gestaltung des Systems getroffen haben. Das ist eine Abkehr vom *component-and-connector* Ansatz. In diesem Dokumentationsansatz verstecken sich oft in der Praxis einige Gefahren. 1992 führten Dewayne Perry und Alexander Wolf [Perry and Wolf, 1992] zwei wichtige Gefahren bzw. Probleme an: Architektonischer Drift (*architecture drift*) und Architektonische Erosion (*architectural erosion*). Die Architektonische Drift entsteht wenn die Architektur aufgeweicht wird. Das führt häufig zu mangelnder Anpassungsfähigkeit und einem Mangel an Klarheit und Kohärenz. Zum Beispiel wenn während der Entwicklung das Softwaresystem größer und komplexer wird, ist ein großer Teil des neuen Codes nicht nur funktionalitäts- sondern auch strukturbezogen. Dieser strukturbezogene Code kann zwar eine positive zusätzliche Flexibilität in frühen Versionen eines neuen Softwaresystems anbieten, aber verursacht auch, dass diese neuen Versionen der Software schwerer zu verstehen sind. Dies kann zu architektonischer Drift führen. Stakeholder können den Aufbau der Architektur nicht vollständig verstehen. Architektonische Erosion ensteht bei

Verstößen gegen die Soll-Architektur. Diese Verletzungen führen häufig zur Zunahme der Probleme eines Systems und erhöhen die Zerbrechlichkeit des Systems. Zum Beispiel, alle Charakteristiken der endgültigen Version einer Software müssen auf Entwurfsentscheidungen früherer Versionen basieren. Allerdings wird die Konstruktion eines optimalen Systems in der Praxis bedeuten, dass Softwareentwickler auf einen Großteil des bestehenden Codes verzichten müssen. Aber diese Änderungen verstoßen oft gegen die früheren wichtigen Entwurfsentscheidungen. Das ist ein klassisches Beispiel für „Entfernen von tragenden Wänden führt häufig zu katastrophalen Ergebnissen“ [vgl. Perry and Wolf, 1992].

2004 fasste Jan Bosch die Ergebnisse der Forschung von Perry und Wolf zusammen und führt an, dass das Schlüsselproblem Wissensverlust (*knowledge vaporization*) [Bosch, 2004] ist. Der architektonische Wissensverlust wird oft durch den Verlust von kritischen architektonischen Entscheidungen verursacht, was zu einer Reihe von Problemen führt. Dieser Verlust ist hauptsächlich darauf zurückzuführen, dass die Softwarearchitekten ihre Entwurfsentscheidungen nicht laufend festhalten, so dass diese wichtigen Informationen über die Softwarearchitektur während des Entwicklungs- und Evolutionszyklus nicht verfügbar sind. Dazu gibt es ein bekanntes Sprichwort in der Softwarearchitektur: „Wenn etwas nicht aufgeschrieben wird, existiert es nicht.“ [vgl. Harrison et al., 2007].

Nur wenn die Softwarearchitekten ihre Entscheidungen laufend festhalten bieten diese Entscheidungen eine wichtige Informationsquelle. Da fast alle Kenntnisse und Informationen hinsichtlich der Ergebnisse der Domänenanalyse, Architekturschichten, Architekturstile, ausgewählte Entwurfsmuster und aller Entwurfsentscheidungen während der architektonischen Gestaltung des Systems in der resultierenden Softwarearchitektur eingebettet und somit nicht explizit dargestellt werden müssen, fehlt oft eine first-class Darstellung für die Softwarearchitektur. Dies hat eine zweifache mögliche Auswirkung: Erstens geht das Wissen über

Entwurfsentscheidungen in der Architektur schnell verloren. Zweitens können Änderungen der Softwarearchitektur leicht zu der Verletzung früherer Entwurfsentscheidungen, und zu erhöhter Architektur-Erosion während der Systementwicklung führen. Jansen und Bosch definiert acht häufige Probleme [vgl. Bosch, 2004, vgl. Jansen and Bosch, 2004, vgl. Jansen, 2007] in der Softwarearchitektur. Diese Probleme und deren Lösungsansätze werden in Kapitel 2.2.5 diskutiert.

2.2 Architectural Design Decisions

2.2.1 Historische Evolution der ADDs

Obwohl der Begriff „Architectural Design Decision" sehr früh genutzt wurde, z.B. in dem Buch „*Bass, Len; Clements, Paul; & Kazman, Rick. Software Architecture in Practice, Addison-Wesley, 1997.*" wurde diese Terminologie viele Male erwähnt, gab es lange Zeit keine klare Definition. Außderdem war in dieser Periode die Hauptmethode der Softwarearchitektur *components and connectors* (C&C). Erst nach ein paar Jahren begann man die Möglichkeit zu diskutieren, ob eine Menge der Entwurfsentscheidungen statt der traditionellen Methode der Softwarearchitektur als erste Wahl betrachtet werden kann. In dem oben erwähnten Buch schlagen die Autoren vor, dass die Softwarearchitekten eine Menge von Entwurfsentscheidungen beschreiben und andere Architekturartefakte erst später im Lebenszyklus der Softwarearchitektur erstellen. Dadurch ermöglichen die Softwarearchitekten ein Gleichgewicht zwischen Entwurfsregeln und Einschränkungen der technischen Umsetzung zu erhalten. Die Softwarearchitekten sollten unbedingt die Entscheidungen treffen, mit der die wichtigsten strukturellen Elemente eines Systems, die von außen sichtbaren Eigenschaften und deren Beziehungen, richtig identifiziert werden können.

2004 und 2005 konzentrierten sich einige Forschungstrends auf die Behandlung von ADDs als *first-class-entities* und ihre explizite Darstellung in der Dokumentation der

Softwarearchitektur. Jan Bosch [Bosch, 2004] und Jeff Tyree [Tyree and Akerman, 2005] führen an, dass ADDs als *first-class-entities* zur Dokumentation der Softwarearchitektur verwendet werden sollen. Die Softwarearchitektur muss eigentlich als eine Menge von ADDs betrachtet werden. Bald nach diesem Vorschlag versuchten Philippe Kruchten et al. das in die Praxis umzusetzen, und definiern eine Ontologie, die beschreibt wie Softwarearchitekten einige Attribute zur Entwurfsentscheidungsbeschreibung während der Softwareentwicklung erfassen müssen [Lee and Kruchten, 2007]. Dadurch wissen die Softwarearchitekten, wie sie diese Entscheidungen treffen sollen. P. Kruchten et al. identifizieren drei Arten der Entwurfsentscheidungen, Existenz-Entscheidungen, Eigenschafts-Entscheidungen, Ausführungsentscheidungen, und acht Attribute der Entwurfsentscheidungen, nämlich Abriss, Begründung, Geltungsbereich, Zustand, Geschichte, Kategorie, Kosten, Risiko [vgl. Kruchten, 2004].

2004 schlägt Jan Bosch [Bosch, 2004] vor, dass man von der traditionellen Architekturmethode, die auf *components and connectors* basiert, zu einer neuen Architekturmethode, die die Nutzung von ADDs vorsieht, umsteigen soll. Jan Bosch befürwortet die explizite Darstellung der Entwurfsentscheidungen als *first-class-entities*. Dies kann den Prozess der Architektur positiv leiten und beeinflussen. Gleichzeitig sollen die Begründungen und Einschränkungen (*constraints*) der Entscheidungen und die Designregeln in Echtzeit erfasst und aufgezeichnet werden. 2005 stellen Anton Jansen und Jan Bosch eine Lösung in dem Paper „*Software Architecture as a Set of Architectural Design Decisions*“ vor. In diesem Paper veröffentlichten sie gleichzeitig das erste ADD-Tool Archium, mit der Softwarearchitektur mittels ADDs beschrieben werden kann [vgl. Jansen and Bosch, 2005].

Um die Struktur der Softwarearchitektur besser zu beschreiben, sollten die Softwarearchitekten begründete Architekturentscheidungen treffen oder erfassen. Aber, wie kann man diese Architekturentscheidungen klar und deutlich

dokumentieren? Obwohl die Attribute und Bestandteile einer ADD in obigen Abschnitten schon vorgestellt werden kann aber damit ein Entscheidungsprozess noch nicht mit ADDs dokumentiert werden. Weil dies nur ein kurzer Blick darauf ist, welche Elemente in einer Architekturentscheidung enthalten sein müssen. Aber die Erstellung einer Architekturentscheidung und deren Beziehungen zwischen unterschiedliche voneinander abhängigen ADDs werden noch nicht berücksichtigt. Hierzu wurde ein bekanntes Architekturentscheidungen-Dokumentations-Template (siehe Tabelle 1) 2005 von Jeff Tyree und Art Akerman [Tyree and Akerman, 2005] vorgestellt. Dieses Template beschreibt die Hauptkomponenten der Architekturentscheidungen, und berücksichtigt insbesondere die Beeinflussung der Beziehungen zwischen unterschiedlichen alternativen Architekturentscheidungen während der Erstellung der Architekturentscheidungen. Zum Beispiel, wenn eine Architekturentscheidung getroffen wird, werden sich einige Implikationen wie neue Anforderungen oder andere Entscheidungen daraus ergeben.

Elemente	Beschreibung
Kernproblem	Das vorgestellte Kernproblem für den Architekturentwurf.
Entscheidung	Status oder ausgewählte Position in der Architektur-Richtung.
Status	Status der Entscheidung, z.B. unerledigt, entschieden oder genehmigt.
Gruppe	Ein einfaches Gruppieren, z.B. Integration, Präsentation, Daten usw. Es kann die Menge der Entscheidungen organisieren.
Hypothesen	Die zu Grunde gelegten Annahmen, mit der Architekten die Entwurfsentscheidungen machen können. Zum Beispiel Budget, Kosten, und Zeitplan.
Einschränkungen	Einschränkungen, die der Architekturentwurf für die Umgebung mit sich bringt.
Positionen	Die betrachteten Alternativen.
Argument	Der Grund, warum eine Position ausgewählt wird, z.B. Markteinführungszeit oder eine bestimmte funktionale Anforderung
Implikationen	Eine Entscheidung kommt mit vielen Auswirkungen bzw. Implikationen. Zum Beispiel neue Anforderungen oder zusätzliche Einschränkungen für die Umgebung des zu entwickelnden Systems.
Verbundene Entscheidungen	Entscheidungen, die die vorliegende *Architectural Design Decisions* beeinflussen können oder schon beeinflusst haben.
Verbundene Anforderungen	Die verbundene Anforderungen, die die Entscheidungen der Architektur zu den Zielen oder Anforderungen führen können.
Verbundene	Die Artefakte werden von *Architectural Design Decisions*

Artefakte	beeinflusst.
Verbundene Prinzipe	Eine Verbindung oder Beziehung zwischen der *Architectural Design Decision* und Unternehmensprinzipien. Dadurch wird sichergestellt, dass die Architekten Ausrichtungsentscheidungen entlang bestimmter Domänen oder Systeme erfassen können.
Notizen	Auflistung der Architectural Design Decisions

Tabelle 1: Dokumentation-Template der Architekturentscheidung von Tyree [vgl. Tyree and Akerman, 2005]

Um diesen Entscheidungsprozess für das Dokumentation-Template der Architekturentscheidungen von Tyree deutlicher darzustellen, wird der Prozess einer Architekturentscheidung als Workflow Diagramm mit dem Online-Workflow-Tool Signavio dargestellt (siehe Abbildung 2). Zuerst sollen die Softwarearchitekten die Kernprobleme vorstellen. Dann müssen diese Kernprobleme gemäß unterschiedlicher Kategorien oder Ansätze gruppiert werden. Dies kann hilfreich sein, um eine Menge von Entscheidungen zu organisieren. In dieser Phase können die Softwarearchitekten noch andere anspruchsvolle Architektur-Ontologien verwenden, z.B. John Kyaruzi und Jan van Katwijk [Kyaruzi and Katwijk, 1999] entwickelten mehrere Kategorien bzw. Methoden für die Gruppierung bzw. Klassifizierung von Entscheidungen. Demnach sollen die Softwarearchitekten alle verbundene Anforderungen, Artefakte, Prinzipien und anderen Entscheidungen sammeln und zusammenfassen. Diese Ressourcen, die Vorbedingungen für die endgültigen Entscheidungen sind, werden zukünftig verwendet. Mit diesen Bedingungen können die Hypothesen und Einschränkungen einfach festgelegt werden. Im nächsten Schritt sollen die Softwarearchitekten die geeigneten Positionen auswählen und die relevanten Argumenten genau beschreiben, und die endgültigen Entscheidungen treffen. Aus diesen Entscheidungen ergeben sich Implikationen für die Umwelt des Systems. Eine wichtigte Arbeit für die Softwarearchitekten ist die Sammlung und Gruppierung dieser Implikationen. Danach werden die getroffenen Entscheidungen noch einmal bewertet. Abschließend müssen die Softwarearchitekten den Status jeder Entscheidung und den ganzen Prozess der Architekturentscheidungen beschreiben.

Bis jetzt haben wir eine explizite Beschreibung des Erstellungsprozesses der ADDs.

Nun beschreiben wir kurz die fünf Vorteile der Nutzung von ADDs. Erstens ermöglichen ADDs dem Architekten und anderen Stakeholdern wichtige Änderungen der Architektur besser zu verstehen und die betroffenen Systeme und Schnittstellen eindeutig zu identifizieren. Zweitens sind die ADDs nicht nur eine Lösung für die bezogene Probleme und Risiken des Systems, sondern auch einen Klebstoff zwischen unterschiedlichen Ressourcen und Stakeholdern. Drittens ist der Entscheidungsprozess transparent. Jeder Stakeholder kann die Dokumentation der Entscheidungen lesen und verstehen. Viertens bieten die ADDs eine gute und einfache Rückverfolgbarkeit jeder Architekturentscheidung an. Abbildung 2 zeigt diese Möglichkeit. Fünftens stellen ADDs eine agile Dokumentation zu Verfügung. In einem agilen Entwicklungsprozess hat das Team keine Zeit, um auf die vollständige Entwicklung und Dokumentation der Architektur durch einen Softwarearchitekt zu warten. Aber eine ADD bietet einen expliziten Plan an, was die anderen Entwickler in der nächsten Phase machen sollen [vgl. Tyree and Akerman, 2005].

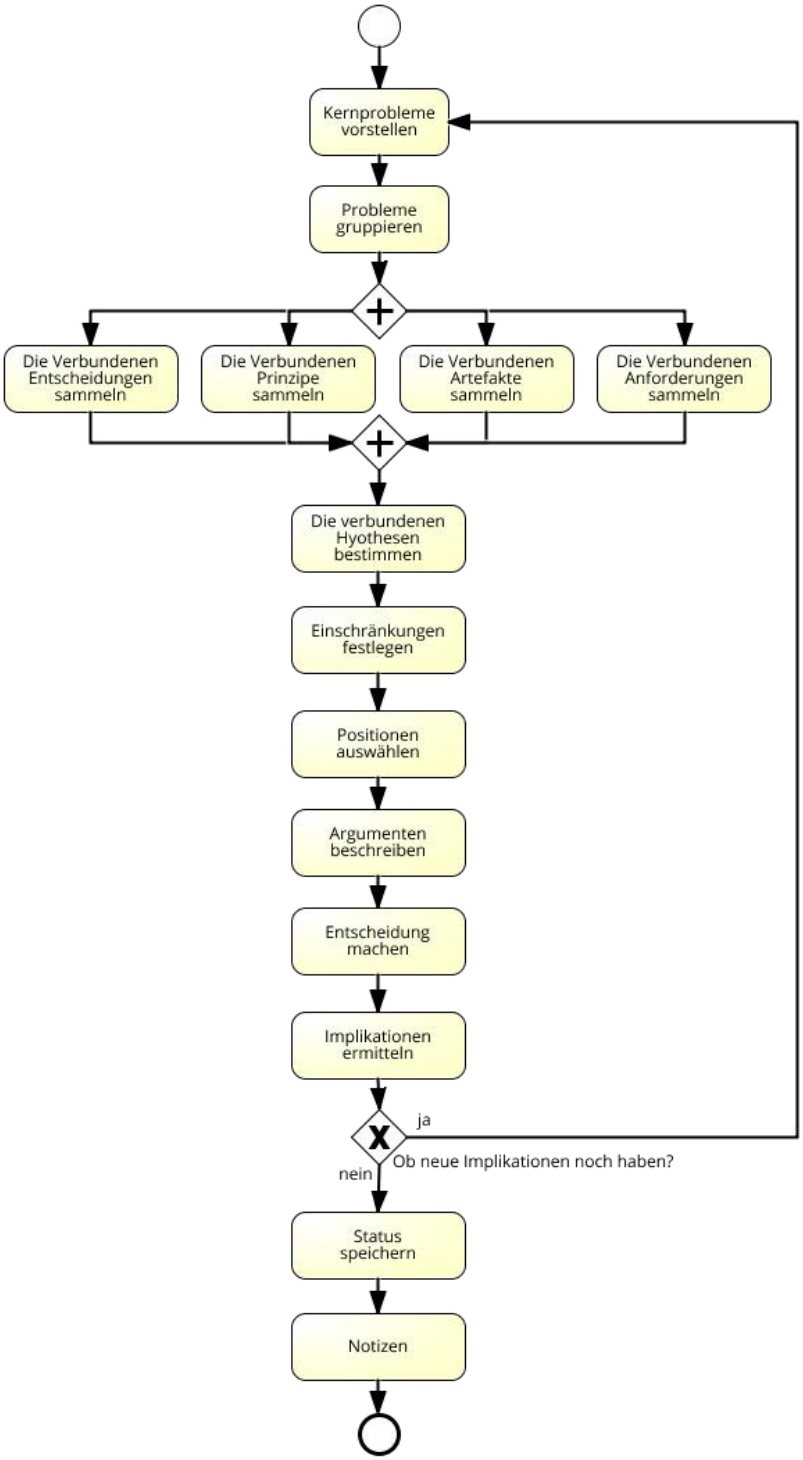

Abbildung 2: Workflow Diagramm einer Entscheidung für Dokumentation-Template der Architekturentscheidung von Tyree [vgl. Tyree and Akerman, 2005]

2.2.2 Dokumentationsmethoden und -ansätze der ADD

In diesem Kapitel werden wir weitergehend diskutieren, wie ein Softwarearchitekt

Architekturentscheidungen im Entwicklungsprozess der Softwarearchitektur effizienter erfassen bzw. treffen kann, und welche Methoden oder Ansätze in diesem Verfahren verwendet werden können. Am Ende des Kapitels diskutieren wir die Wiederverwendbarkeit von ADDs.

Wie bereits beschrieben ist Softwarearchitektur eine Beschreibung der internen Struktur eines Softwaresystems in Bezug auf die Komponenten und deren Beziehungen, Funktionalitäten und Charakteristika. Die Softwarearchitektur besteht aus dem Entwurf und eine Menge damit verbundener Entwurfsentscheidungen. Eine gute Entwurfsentscheidung vermeidet die Probleme im Entwicklungsprozess oder löst diese sogar. Die Korrektheit einer Architekturentwurfsentscheidung kann die Verfügbarkeit des Softwaresystems erheblich beeinflussen. Irrationale, unzweckmäßige oder willkürliche Entscheidungen führen oft zum Ausfall dieses Systems. Deswegen sind „richtige“ Dokumentationen dieser Entscheidungen sehr wichtig.

Vor der Beschreibung der Dokumentationsmethoden und der vier Dokumentationsansätze müssen wir die Voraussetzungen der Entwurfsentscheidungen und Anforderungen (*Requirements*) der Softwarearchitektur genau berücksichtigen, weil die Realisierung der Systemanforderungen die Schlüssel zum Erfolg ist. Man kann sich nicht vorstellen, dass eine erfolgreiche Softwarearchitektur die ursprünglichen Systemanforderungen nicht erfüllt. Diese Systemanforderungen bestehen aus funktionalen Anforderungen und nicht-funktionalen Anforderungen. Um die Erfüllung oder Auswertung dieser Systemanforderungen in einer späteren Phase der Softwareentwicklung auch zu ermöglichen, definiert der ISO 9126-Standard [ISO, 2000, ISO, 2003] eine Reihe von Qualitätsattributen der Software, und verfeinert sie in Teilcharakteristika. Um unsere Architekturentscheidungen immer abhängig von der ursprünglichen Anforderungen zu treffen und entsprechende Qualitätsattribute zu erzielen, haben wir mindestens drei Methoden, Modularisierung der Qualitätsattribute-Knowledge [Bachmann et al., 2005], Risiko-basiertes Reasoning

und Qualitätsattribute-basierte Methode [Xu et al., 2010]. Diese Methoden helfen dem Architekten die „richtigen“ Entwurfsentscheidungen zu treffen. Unglücklicherweise ist die Dokumentation der Architekturentscheidungen auch schwierig, weil die Kenntnisse der Architektur immer still und implizit [Polanyi, 1996] sind. Deswegen ist die Erfassung einer vernünftigen oder verfügbaren Architekturentscheidung oft eine Herausforderung. Die Softwarearchitekten versuchen immer bessere Methoden oder Ansätze, um eine Softwarearchitektur gut und vollständig beschreiben zu können. Derzeit gibt es viele Methoden, mit der die Softwarearchitektur gut dokumentiert werden kann. Viele Methoden lassen uns die ADDs beschreiben oder dokumentieren. Zum Beispiel, *Questions, Options,Criteria* (QOC).

Nun besprechen wir die vier Hauptansätze zum Dokumentation der Entwurfsentscheidungen. Der erste Ansatz ist die Erfassung der Entwurfsentscheidungen durch Entwurfsentscheidungbegründung [Lee and Lai, 1996] [Lee, 1997], der zweite Ansatz ist die formalisierte ontologische Darstellung [Graaf et al., 2012, Kruchten, 2004, Vliet, 2005, Jansen and Bosch, 2005] und deren Wechselbeziehungen bzw. Zusammenhänge. Der dritte Ansatz ist die *Flag-Filter-Form* [Lee and Kruchten, 2007] Methode. Der vierte Ansatz ist die Erfassung der Architekturentscheidungen basierend auf Architekturmustern [Harrison et al., 2007]. Zum letzten Ansatz gehört auch WebADvISE, es wird im Kapitel 2.2.4 im Detail diskutiert.

Die Wiederverwendbarkeit der ADDs ist für eine Dokumentationsmethode oder einen Dokumentationsansatz auch besonders wichtig, weil eine solche Methode oder Ansatz mehrfach in der Praxis verwendet werden muss. Es kann auch eine generellere Bedeutung oder Beschreibung für eine Domain anbieten, weil nur eine verdichtete Dokumentation der ADDs von verschiedenen Softwaresystemen wiederverwendet werden kann. Zum Beispiel Entwurfsmuster ermöglichen es, ein Design in der Produktlinienentwicklung (*Software product lin*e) mehrmals zu verwenden. UML-Diagramm bietet auch Möglichkeiten zur Wiederverwendung für Entwickler an. Wie

bereits besprochen, können wir viele solche Methoden oder Ansätze als wiederverwendbare Methoden oder Ansätze für Dokumentation der ADDs betrachten. Zu den Beispielen, in den wiederverwendbaren ADD Frameworks und den zwei ADDs Tools ADvISE und WebADvISE haben wir die QOC Methode und die musterbasierte Methode verwendet.

2.2.3 Methoden zum Auffinden von ADDs

Um die Gefahren und Probleme in der Softwarearchitektur zu vermeiden, scheint die begründete Architektur-Entscheidungsfindung besonders wichtig zu sein. In diesem Kapitel diskutieren wir verschiedene Methoden dafür. Eine rationale Entscheidungsfindung ist abhängig von vielen Bedingungen: den Entwicklungszeiten und –kosten; qualitative/quantitative Einschränkungen für verschiedene Funktions- und Qualitätsattribute, der Leistungsfähigkeit, Benutzerfreundlichkeit, Zuverlässigkeit des Softwaresystems usw. Und gemäß unterschiedlicher Anforderungen und Qualitätsattribute müssen die Softwarearchitekten zu unterschiedlichen Entscheidungen gelangen. Zum Beispiel müssen die Softwarearchitekten entscheiden, dass Echtzeit oder Wiederverwendbarkeit, Flexibilität oder Effizienz, Zuverlässigkeit oder Flexibilität eines entwickelten Softwaresystems vorrangig sein sollen. Gleichzeitig betreffen diese Anforderungen und Qualitätsattribute für eine hochwertige Softwarearchitektur nicht nur die Endkunden sondern auch jeden Stakeholder. Die architektonischen Endentscheidungen müssen die potenziellen Risiken vermeiden und lassen die ursprünglichen Anforderungen der Stakeholder mit minimalen Verlusten erreichen.

Um die oben erwähnten Bedingungen und Anforderungen zu erreichen, werden viele ausgereiften Methoden zur Architektur-Entscheidungsfindung entwickelt. Die repräsentativen Methoden sind: ATAM (*Architecture Tradeoffs Analysis Method*) [Kazman et al., 2000, Nord et al., 2003], CBAM (*Cost Benefit Analysis Method*) [Kazman et al., 2001, Stephanson et al., 2009], SAAM (*Software Architecture*

Analysis Method) [Kazman et al., 1994, Ionita et al., 2002], SSAR (*Scenario-based Software Architecture Reegineering*) [Bengtsson und Bosch, 1998], SAHP (*Applying Analytical Hierarchy Process to Software Architecture Decisions*) [Svahnberg et al., 2002], ArchDesigner [Al-Naeem et al., 2005] und ArchPad (*Architectural pattern- and decision-based design Method*) [Zimmermann et al., 2008]. Tabelle 2 beschreibt kurz jede Methode.

Methode	Beschreibung
ATAM	ATAM ist eine Methode, die Risikoidentifikation und –Bewertung der Entscheidungsfindung für das Verständnis der technischen Trade-offs und Risiken bietet.
CBAM	CBAM ist eine architektur-zentrierte Methode zur Analyse der Auswirkung auf Kosten, finanzielle Vorteile und des Zeitplans.
SAAM	SAAM stellt ein Verfahren zur Beschreibung und Analyse einer Softwarearchitektur dar. Der Zweck von SAAM ist es bestimmte Eigenschaften bzw. Qualitätsattribute der Softwares zu erfüllen.
SSAR	SSAR bietet vier verschiedene Ansätze an, um Qualitätsattribute der Softwarearchitektur-Ebene zu bewerten, nämlich szenario-basierte Auswertung, Simulation, Mathematische Modellierung und erfahrungsbasierte Argumentation.
SAHP	SAHP bietet einen strukturierten Lösungsansatz gemäß der Vorschläge und Präferenzen der Stakeholder für die gewünschte Qualitätsattribute und eine quantifizierte Bewertung der Vor- und Nachteile der verschiedenen Entscheidungskandidaten.
ArchDesigner	ArchDesigner stellt eine berechenbare Methode zur Entscheidungsfindung der Softwarearchitektur, in drei Schritten dar. Berechnung des Werts („Score“) möglicher alternativen Entwurfsentscheidungen, Normalisierung dieser berechneten Werte, Optimierung der Formulierung und Erfassung der optimalen Kombination von Entscheidungsfindungsalternativen.
ArchPad	ArchPad bietet eine Methode zur Architekturentscheidungsfindung durch die Kombination von Entwurfsmustern und Entwurfsentscheidungen an. Siehe dazu auch Kapitel 3.1.

Tabelle 2: Sieben Methoden zur architektonischen Entscheidungsfindung

Da Methoden zur Entscheidungsfindung kein Hauptbestandteil in dieser Arbeit sind, werden wir nicht jede Methode diskutieren. Stattdessen beschreiben wir eine typische Entscheidungsfindungs-Methode ATAM und deren konzeptionellen Ablauf und seine neun Prozessschritte als ein Muster der Methoden zur Entscheidungsfindung. Woraus folgt, dass fünf wichtige Eigenschaften, die gleichzeitig auch sehr wichtig für andere Methoden sind, als fünf Vergleichspunkte betrachtet werden. Dann vergleichen wir

zwischen oben erwähnten sieben Methoden mit fünf Vergleichspunkten. Die Methode ArchPad wird im Kapitel 3.2.1 im Detail beschrieben, da das musterbasierte Architekturentscheidungsmodell auf dieser Methode basiert, und es ein wichtiger Schwerpunkt in dieser Arbeit ist.

ATAM

Architecture Tradeoff Analysis Method (ATAM) ist eine Methode zur Bewertung in Bezug auf Qualitätsattribute der Softwarearchitektur. Es bewertet architektonische bestandsgefährdende Risiken, die möglicherweise die Erreichung der Geschäftsziele eines Unternehmens verhindern. Der Zweck der ATAM ist eine Beurteilung der Architekturentscheidungen mit Hinblick auf die Anforderungen der Qualitätsattribute. Es zeigt nicht nur, wie eine Architektur insbesondere Qualitätsziele und Qualitätsattribute rationaler erfüllt, sondern bietet auch einen Einblick an, wie diese Qualitätsziele und Qualitätsattribute miteinander interagieren müssen. Abbildung 3 beschreibt den klassischen konzeptionellen Ablauf der ATAM. Am Anfang eines Projekts werden die kritischen Geschäftsfaktoren und Softwarearchitekturen normalerweise von dem Projektentscheidungsträger vorgestellt. Diese Faktoren und Softwarearchitekturen werden durch Qualitätsattribute und Architekturansätze von unterschiedlichen Stakeholdern in Szenarien bzw. *Use Cases* und architektonische Entscheidungen verfeinert. Die Analyse dieser Szenarien und Entscheidungen führt zu Identifikation von Risiken, Nicht-Risiken, Sensitivitätspunkten und Trade-offs im Architekturentwurf. Dann werden die Risiken in eine Menge von Risikothemen synthetisiert, die zeigen, wie jeder Einzelne die kritischen Geschäftsfaktoren und die Softwarearchitektur beeinflusst. ATAM umfasst neun Schritte, siehe Tabelle 3 [vgl. Kazman et al., 2000].

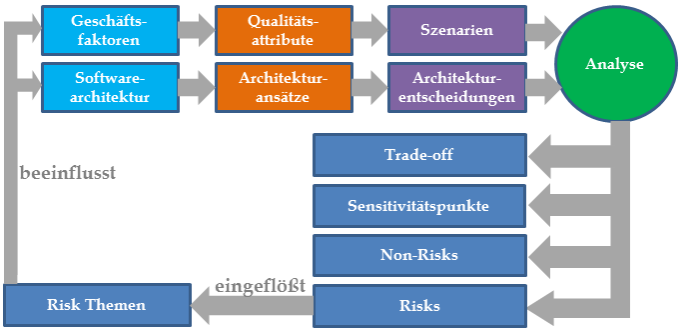

Abbildung 3: Ein klassischer konzeptioneller Ablauf des ATAM [vgl. Kazman et al., 2000]

Präsentation		
1	Präsentation der ATAM	Der Auswertungsverantwortliche beschreibt die Bewertungsmethode für die versammelten Teilnehmer und versucht ihre Erwartungen zu setzen.
2	Präsentation Kritischer Geschäftsfaktoren	Der Projektmanager beschreibt die Unternehmensziele und die primären Geschäftsfaktoren.
3	Präsentation der Architektur	Der Architekt beschreibt die vorgeschlagene Architektur, wobei der Schwerpunkt darauf liegt, wie die geschäftlichen Faktoren erfüllt werden.
Untersuchung und Analyse		
4	Identifizieren der Architekturansätze	Der Architekt identifiziert die Architekturansätze ohne diese zu analysieren.
5	Erstellen der Qualitätsattribute	Die Qualitätsattribute umfassen die „Nützlichkeit“ des Systems, z.B. Leistungsfähigkeit, Verfügbarkeit, Sicherheit, Modifizierbarkeit.
6	Analysieren der Architekturansätze	Auf der Grundlage der Attribute mit hoher Priorität aus Schritt 5 werden die Architekturansätze analysiert.
Testen		
7	Brainstorming und Priorisierung von Szenarien	Auf der Basis der Szenarien wird eine größere Menge von Szenarien von der Gruppe der Stakeholder ausgewählt.
8	Analysieren der Architekturansätze	Als Testfälle wird die Analyse der Architekturansätzen bestimmt. Wichtig ist ob die architektonischen Risiken, kritische Punkte und Trade-off Punkte mit diesen Testszenarien abgedeckt werden können.
Berichterstattung		
9	Präsentation der Ergebnisse	Basierend auf den in der ATAM gesammelten Informationen präsentiert das ATAM Team die Ergebnisse an die versammelten Stakeholder und verfasst einen Bericht.

Tabelle 3: Prozessschritte des ATAM [vgl. Kazman et al., 2000]

Gemäß der Beschreibung des konzeptionelles Ablaufs bzw. des Prozessschrittes einer typischen Methode zur Entscheidungsfindung ATAM und des Vergleichs mit anderer Methoden listen wir fünf wichtige Eigenschaften für die Methoden zur Entscheidungsfindungen: Analyse der Design-Alternativen, Vergleich der Design-

Alternativen, Identifikation der Qualitätsattribute (QA), Analyse der Trade-offs und Sensitivität. Es folgt die Beschreibung jeder Eigenschaft [vgl. Xu et al., 2007] und Tabelle 4 stellt einen Vergleich zwischen unterschiedlichen Methoden zur Entscheidungsfindung dar.

- **Analyse der Design-Alternativen:** Diese Analyse basiert auf der Meinung der Stakeholder, mathematischen Modellen oder anderen Analysetechniken, um die primären Qualitätsattribute zu erfüllen.
- **Vergleich der Design-Alternativen:** Die Qualitätsattribute werden gemäß der Anforderungen der Stakeholder, der verfügbaren Designzeit oder anhand von Entwurfsmustern identifiziert.
- **Identifikation der Qualitätsattribute(QA)**: Die Qualitätsattribute werden gemäß der Anforderungen der Stakeholders, der verfügbaren Designzeit oder anhand von Entwurfsmustern identifiziert.
- **Analyse der Trade-offs:** *Trade-offs*-Punkte haben vier Abstufungen: niedrig, mittel, hoch und nicht-unterstützt. Niedrige *Trade-offs*-Punkte können identifiziert werden. Mittlere *Trade-offs*-Punkte können identifiziert werden und mit relativen Ranking in Bezug auf Qualitätsattribute versehen werden. Hohe *Trade-offs*-Punkte können identifiziert werden und mit relativen Ranking in Bezug auf Qualitätsattribute versehen werden. Die Beziehungen zwischen Design-Alternativen sind auch *Trade-offs*.
- **Sensitivität:** Ob die Sensitivitätspunkte während des Prozesses der Entscheidungsfindung identifiziert werden können.

Methode	Analyse der Design-Alternativen	Vergleich der Design-Alternativen	Identifikation der QA	Analyse der Trade-offs	Sensitivität
ATAM	**Manuell**	**Qualitativ**	**Stakeholder-basiert**	**Niedrig**	**Sensitivitätspunkte**
CBAM	**Manuell**	**Quantitativ**	**Stakeholder-basiert**	**Mittel**	-
SAAM	**Manuell**	**Qualitativ**	**Stakeholder-basiert**	**Niedrig**	**Sensitivitätspunkte**
SSAR	**Maschine-basiert**	**Qualitativ,**	**Stakeholder-**	-	-

		Quantitativ	**basiert**		
SAHP	**Manuell**	**Quantitativ**	**Design-basiert**	**Mittel**	-
ArchDesigner	**Manuell**	**Quantitativ**	**Design-basiert**	**Hoch**	-
ArchPad	**Manuell**	**Qualitativ**	**Muster-basiert**	**Mittel**	-

- **Analyse der Design-Alternativen (Optionen):** Manuell, Maschinen-basiert
- **Vergleich der Design-Alternativen (Optionen)**: Qualitativ, Quantitativ
- **Identifikation der Qualitätsattribute (Optionen):** Stakeholder-basiert, Design-basiert, Muster-basiert
- **Analyse der Trade-offs (Optionen):** Niedrig, Mittel, Hoch, Nicht-unterstützt(-)
- **Sensitivität (Optionen):** Sensitivitätspunkte, Nicht-unterstützt(-)

Tabelle 4: Vergleich unterschiedlicher Methoden zur Entscheidungsfindung [vgl. Xu et al., 2007]

2.2.4 Erfassung der ADDs von Architekturmustern

Architekturmustern sind die durch wiederholte Nutzung in der Praxis verifizierten Lösungsansätze der allgemeinen Probleme in der Softwarearchitektur. Damit kann die Struktur einer Softwarearchitektur gut dokumentiert werden. Deswegen bieten sie eine effektive Möglichkeit an, einige der wichtigsten Entwurfsentscheidungen zu erfassen. Harrison, Avgeriou und Zdun [Harrison et al., 2007] beschrieben vier primäre Dokumentation-Herausforderungen, die von Architekturmustern behandelt werden können. Zum Beispiel umfassen Architekturmuster allgemeine Struktur- und Verhaltensinformationen, ermöglichen es, dass die Dokumentation der ADDs einfacher werden und schneller erstellt werden können. Architekturmuster folgen einer leicht verständlichen Form wobei sie gut zum beschriebenen Architekturentscheidungen-Dokumentations-Template (Siehe Kapitel 2.2.1) für Architekturentscheidungen passen. Avgeriou und Zdun [Avgeriou and Zdun, 2005] bieten einen umfassenden Überblick über die identifizierte Architekturmustern an. Für die acht unterschiedlichen Schichten haben wir insgesamt vierundzwanzig Architekturmuster, z.B. *Model-View-Controller* (MVC), ***Layers***, ***Peer-to-Peer***, ***Pipes-and-Filters*** usw. In dieser Arbeit wird das Architekturmuster MVC besonders empfohlen, da das beschriebene Web-basierte Tool WebADvISE mittels MVC als ein Teil dieser Arbeit entwickelt wird. In dem Papier „*Using Patterns to Capture Architectural Decisions*“ beschreiben die Autoren ihre Vorschläge für vier Hauptperspektiven für die Erfassung von Architekturmustern zu Architekturentscheidungen [vgl. Harrison et al., 2007].

- **Beziehung zwischen Muster und Entscheidung.** Architekturmuster und Architekturentscheidungen sind komplementäre Konzepte. Ein verwendetes Muster ist eigentlich ein alternativer Lösungsansatz, mit dem die Entscheidungen im spezifischen Kontext des Zielsystems zugeordnet werden.

- **Unterschiede zwischen Muster und Entscheidung.** Der wesentliche Unterschied zwischen Architekturmuster und Architekturentscheidungen liegt im Ausmaß der darin enthaltenen Informationen. Die Dokumentation der Architekturentscheidungen beschreibt eine individuelle Entscheidung über das Zielsystem. Im Gegensatz dazu beschreiben Muster Lösungsansätze, die sich mehrfach in der Praxis bewährt haben. Das heißt, Architekturentscheidungen sind spezifisch, aber vorläufig. Mustern sind bewährt, aber allgemein.

- **Vorteile der Erfassung von Architekturmuster-ADDs.** Der Name eines Musters bezieht sich auf eine generische Musterbeschreibung und bietet gleichzeitig zumindest eine minimale Dokumentation. Muster bezeichnen einige signifikante Architekturentscheidungen. Außerdem geben diese Muster Qualitätsattribute des Systems explizit an und können diese Qualitätsattribute dem Architekten helfen, seine Entscheidungen und deren Implikationen gut zu erkennen. Die Muster enthalten noch einige abhängige Muster, die Architekten auch helfen können, dass ein alternativer begründeter Lösungsansatz ausgewählt wird. Wenn die Softwarearchitekten nicht wissen, wie sie ihre Entscheidungen dokumentieren sollen, enthalten die Muster viele Hinweise dazu. Schließlich bieten Mustern auch eine Möglichkeit zur Wiederverwendung.

- **Grenzen dieses Ansatzes.** Architekturmustern entlasten natürlich nicht den Architekten von aller Verantwortung für die Dokumentation von Architekturentscheidungen. Das heißt, Architekten müssen anwendungsspezifische Entscheidungen noch selbst dokumentieren. Und nicht alle Entscheidungen haben entsprechende Muster. Einige Entscheidungen sind abhängig vom konkreten Projektbereich und der Domäne. Außerdem verwenden

die Architekten mehrere Muster gemeinsam. Aber, wenn es keine enge Beziehung bzw. Interaktion zwischen diesen Mustern gibt, wird diese musterbasierte Architektur undeutlich sein.

Architekturmuster und Architekturentscheidungen haben eine unterschiedliche Herkunft. Architekturentscheidungen lösen die konkreten Kernprobleme und fokussieren auf alternative Optionen. Architekturmuster beschreiben die Probleme in dem Problembereich und die Entscheidungen von den Lösungsansätzen. Sie haben unterschiedliche Dokumentationsformate. Harrison, Avgeriou und Zdun vergleichen die Dokumentation von Architekturmustern und Architekturentscheidungen und gaben die Ergebnisse in einer Tabelle [Harrison et al., 2007, Tabelle 1] wieder. So können Softwarearchitekten einfach wissen, welche Eigenschaften oder Attribute der Muster in dem Erfassungsprozess von Mustern zu Entscheidungen berücksichtigt werden müssen.

2.2.5 Lösungsansätze für die Probleme in der Softwarearchitektur

Das Ziel aller Forschungen ist die Lösung der konkreten Probleme. In diesem Kapitel fassen wir kurz die Lösungsansätze für die Probleme zusammen. Dann werden neun Lösungsansätze mittels eines *Use-Case* Modells vorgeschlagen, mit dem können Softwarearchitekten einfach wissen, welche *Use-Cases* und gleichzeitig welche Architekturentscheidungen sie in der industrielle Softwareentwicklung richtig verwenden können, um die acht in dem Kapitel 2.1.3 erwähnten Probleme (siehe Tabelle 5) zu vermeiden.

Um diese Probleme der Softwarearchitektur zu lösen bzw. den Wissensverlust zu vermeiden und um damit verbundene Probleme wie z.B. Mangel an Kommunikation der Stakeholder, teure Evolution des Systems, hohe Kosten für Änderung und begrenzte Wiederverwendbarkeit zu vermeiden, müssen Architekturentscheidungen als *first-class-entities* effizient aktualisieren oder umsteigen. 2005 stellen Anton Jansen und Jan Bosch ihren Vorschlag vor, um die Probleme der Architektur zu lösen.

Erstens laufende Überprüfung der konzeptionellen Integrität der Softwarearchitektur. Die Entwurfsentscheidungen beschreiben die Regeln und Einschränkungen, die eingehalten werden sollten. Zweitens hilft explizite Entwurfsraumbewertung Softwarearchitekten bei der Vermeidung von offensichtlichen Fehlern. Drittens steht die Analyse der Softwarearchitektur und des Designprozesses. Viertens die verbesserte Rückverfolgbarkeit der Entwurfsentscheidungen und ihrer Beziehungen. Gleichzeitig müssen auch einige Anforderungen erfüllt werden: Erstens sind *first-class* ADDs notwendig, um eine Softwarearchitektur als eine Menge von Entwurfsentscheidungen zu beschreiben. Zweitens bilden explizite Änderungen an der Architektur die Brücke zwischen *first-class-entities* und den ADDs. Drittens, Unterstützung für die Subtraktion und Addition der Änderungen und viertens, die Beziehungen zwischen der Softwarearchitektur und der Realisierung der Entwurfsentscheidungen müssen klar sein [vgl. Jansen and Bosch, 2005].

Mit dem Begriff *Knowledge Grid* entwickelten Jan S. van der Ven et al. [Ven et al., 2006] ein *Use-Cases* Modell und ein Domänenmodell in dem Projekt Griffin. Griffin studiert unterschiedliche Tools, Techniken und Methoden, die die verschiedenen Aufgaben erfüllen, um ein *Knowledge Grid* zu bilden. Die Definition von Knowledge Grid ist [Zhuge, 2004]: "*an intelligent and sustainable interconnection environment that enables people and machines to effectively capture, publish, share and manage knowledge resources*". *Knowledge Grid* unterstützt effektvolle Interaktion zwischen den Stakeholdern und einem Computersystem. Dies ist auch die Grundlage um Wissensverlust zu vermeiden. Für Architekturentscheidungen ist ein *Knowledge Grid* auch ein gutes System, um die effektive Zusammenarbeit von Teams bei Problemlösungen und Entwurfsentscheidungen zu unterstützen. Gleichzeitig verwendet es eine geeignete Ontologie, die die Komplexität der Architekturentscheidungen sowie ihre gegenseitigen Abhängigkeiten gut darstellen kann.

Das Domänenmodell des Projekts Griffins beschreibt die grundlegenden Konzepte für

die Speicherung, den Austausch und die Verwendung der Architekturentscheidungen und die Beziehungen zwischen diesen Konzepten. Das *Use-Case* Modell von Projekt Griffin beschreibt die erforderliche Nutzung des vorgestellten *Knowledge Grid*. Die *Use-Cases* des Modells sind in Bezug auf das Domänenmodell ausgedrückt, um die relevanten Architekturentscheidungen gemäß dieser *Use-Cases* richtig zu verwenden. In dem Modell gibt es insgesamt fünf Akteure, Architektur-Kritiker, Projektmanager, Architekt, Architektur-Betreuer und alle Stakeholder dazu noch siebenundzwanzig definierte *Use-Cases* [vgl. Ven et al. 2006]. Diese *Use-Cases* können die Softwarearchitektur in jeder Entwicklungsphase in der industriellen Praxis führen, und sie wurden auf der entsprechenden UML2.0-Spezifikation in dem Paper „*Using Architectural Decisions*“ [Ven et al. 2006] dargestellt. In diesem Artikel werden einundzwanzig wichtige *Use-Cases* von der siebenundzwanzig definierten *Use-Cases* zusammengefasst. Gleichzeitig werden die zugeordneten Akteure auch identifiziert (siehe Tabelle 5).

Akteure:

(A1) Architektur-Kritiker
(A2) Projektmanager
(A3) Architekt
(A4) Architektur-Betreuer
(A5) Alle Stakeholder

Use Cases

(UC1) Überprüfen der Implementierung der Architekturentscheidungen (abhängig von UC8)
(UC2) Identifizieren der Vorgesetzer von Stakeholder (abhängig von UC3)
(UC3) Identifizieren der wichtigsten Architekturentscheidungen für einen bestimmten Stakeholder (abhängig von UC1, UC9)
(UC4) Überprüfen der überflüssigen Architekturentscheidungen
(UC5) Richtigkeit Überprüfen (abhängig von UC8, UC9)
(UC6) Identifizieren der geänderten betroffenen Stakeholder (abhängig von UC3)
(UC7) Identifizieren der ungelösten *Concerns* für einen bestimmten Stakeholder (abhängig von UC9)
(UC8) *up-to-date* bleiben (abhängig von UC5)
(UC9) Informieren der betroffenen Stakeholder (abhängig von UC5)

(UC10) Abrufen einer Architekturentscheidung (abhängig von UC6)
(UC11) Überprüfen der Architekturentscheidungen während der Laufzeit auf Änderungen (abhängig von UC5)
(UC12) Hinzufügen einer Architekturentscheidung (abhängig von UC2)
(UC13) Entfernen eine Reihe der aufgehobenen Architekturentscheidungen (abhängig von UC8)
(UC14) Wiederverwendung der Architekturentscheidungen (abhängig von UC14)
(UC15) Beseitigen der überflüssigen Architekturentscheidungen
(UC16) Ausführen von inkrementellen Architektur-Reviews (abhängig von UC1, UC9)
(UC17) Durchführen der Architekturentscheidungen
(UC18) Bewerten der Auswirkung einer Architekturentscheidung
(UC19) Bewerten der Konsistenz der Architektur (abhängig von UC1)
(UC20) Unvollständigkeit identifizieren (abhängig von UC1)
(UC21) Durchführung von Risikoanalysen

Probleme	Use Cases	Akteure
1. Mangel an Kommunikation der Stakeholder	UC1,UC2,UC3,UC6,UC7, UC9,UC12	A1,A2,A3,A4,A5
2. Mangel an first-class Darstellung	UC5,UC8,UC19,UC20	A1,A3,A5
3. *Cross-Cutting* und enge Verflechtung zwischen verschiedenen Architekturentscheidungen	UC1,UC5,UC19	A1,A3
4. Verstoßen gegen Designregeln und Einschränkungen	UC5,UC10	A1,A3
5. Veraltete Entwurfsentscheidungen werden nicht entfernt	UC4,UC13,UC15,UC18	A3,A4
6. Teure Evolution des Systems	UC1,UC5,UC8,UC11,UC 16,UC19	A1,A3,A5
7. Hohe Kosten für Änderungen	UC5,UC8,UC11,UC21	A1,A3
8. Begrenzte Wiederverwendbarkeit	UC14,UC17	A3,A5

Tabelle 5: Vermeidung der Probleme in der Softwarearchitektur mittels Use-Cases Modells.

2.3 Modelle und Tools der ADDs

2.3.1 Modelle der ADDs

In dem Forschungsgebiet der Softwarearchitektur gibt es einen Paradigmenwechsel von der Beschreibung der Ergebnisse des Architekturprozesses mittels *component-and-connector* (***know-what***) zur Dokumentation der ADDs und deren Beziehungen (***know-how***). Dieser Paradigmenwechsel führt zu Entstehung der vielen verschiedenen Modelle. Im Folgenden zählen wir vierzehn existierende *Architectural*

Design Decisions Models ADMs auf und diskutieren kurz ihre Eigenschaften [vgl. Shahin et al., 2009b].

1. **Ontologie von Kruchten.** Kruchten führt eine Ontologie [Kruchten, 2004] der ADDs an, um komplexe und softwareintensive Systeme zu beschreiben. Diese Ontologie bietet drei Hauptklassen und acht Attribute für Entwurfsentscheidungen an, siehe Kapitel 2.2.2.

2. **Dokumentations-Template für Architekturentscheidung von Tyree.** Dieses Modell wurde von Tyree und Akerman [Tyree and Akerman, 2005] angeführt. Es bezeichnet sich die ADDs mittels Erfassung der Kernprobleme, Hypothesen und Einschränkungen der Softwaresystem, Argumentationen für Entwurfsentscheidungen und deren Beziehungen miteinander. Ein Dokumentations-Template (siehe Tabelle 3) der Architekturentscheidung enthält meistens vierzehn Elemente. Die wichtigsten Elemente sind Kernproblem, Entscheidung, Hypothesen, Einschränkung, Gründe, Implikation und verbundene Anforderungen usw.

3. **Archium Modell.** *Archium* ist ein Modell von Jansen und Bosch [Jansen and Bosch, 2005]. Architektur in *Archium* wird als eine Menge von Entwurfsentscheidungen beschrieben (eine Softwarearchitektur = dd1 + dd2 + dd3 +…+ ddn, ddx ist eine Entwurfsentscheidung). Der Kern dieses Entwurfsentscheidungmodells ist ein Konzept der Probleme. Dieses Konzept beschreibt das architektonische Problem mittels Motivations- und Ursache-Konzepte. Um die beschriebenen Probleme zu lösen, denkt man sich eine oder mehrere potentielle Lösungen aus und schlägt sie vor. Eine Lösung besteht aus Beschreibung, Designregeln, Konsequenzen, Vor- und Nachteilen. Abbildung 4 stellt den Prozessablauf des *Archium* Modells für ADDs dar.

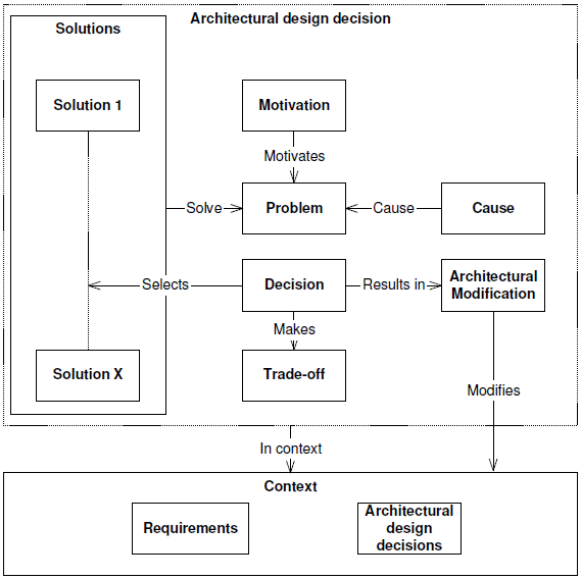

Abbildung 4: Archium Modell für Architectural Design Decisions [Quelle: Figur 1, Jansen and Bosch, 2005]

4. **Service-orientiertes Architektur-Entscheidungsmodell (SOA).** Zimmermann et al. [Zimmermann et al., 2006] bearbeiten das Thema der Verwendung von ADDs auf Basis von SOA. Sie schlagen die Architekturentscheidung als die zentrale Einheit dieses Modells vor und diese ADDs beschreiben und notieren nur die konkreten Entwurfsprobleme. Diese Einheit hat einige Charakteristiken z.B. Name, Geltungsbereich, Phase, Entscheidung-Driver, Problemstellung, Status und Abhängigkeitsbeziehung. Jede Entwurfsentscheidung hat eine oder mehrere ADD-Alternative Einheiten. Zimmermann et al. schlugen drei Stufen der Modellpflege für dieses Modell vor, Konzeptionelle (*High Level Platform Independent Model*, z.B. *Unified Modeling Language*, UML), Technologie (*Hybrid Platform Independent Model/Platform Specific Model*) und Vermögenswert (*Low Level Platform Specific Model*, z.B. *Service Component Architecture, SCA)*. Abbildung 5 stellt den Prozessablauf des SOA für ADDs dar [vgl. Zimmermann et al., 2006].

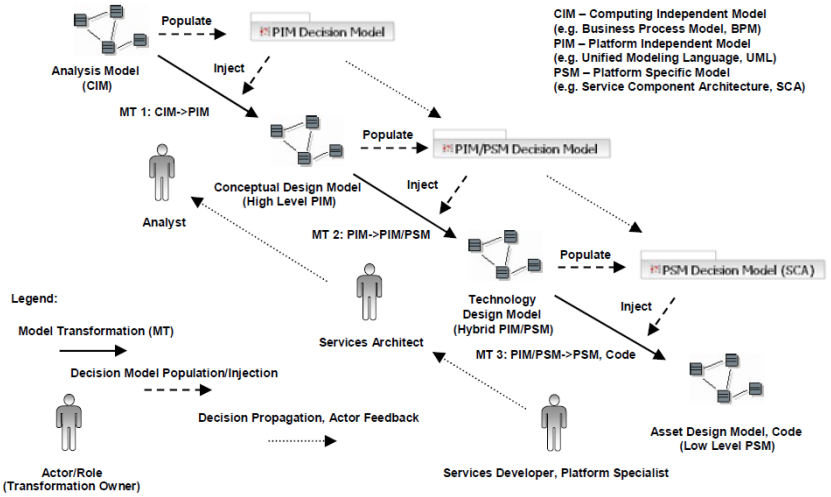

Abbildung 5: SOA Prozessablauf für Architectural Design Decisions [Quelle: Figur 1, Zimmermann et al., 2006]

5. **Integriertes Process-Driven SOA Modell.** Service-orientierte Architekturen (SOA) werden oft im Kontext von Geschäftsprozessen verwendet. Allerdings können die Modellierungsansätze für *Process-Driven* SOAs einige bis alle Arten der *Process-Driven* SOA Modelle nicht integrieren. Zum Beispiel vom Prozessmodell zum Software-Architekturmodell und dann zum Software-Designmodell. Deswegen schlagen Uwe Zdun et al. [Zdun and Dustdar, 2006] vor *Process-Driven* SOA Modelle mit einem Model-Driven Softwareentwicklungsansatz integrieren zu müssen, somit sollte dieser Ansatz gleichzeitig auf bewährten Software-Mustern basieren.

6. **DAMSAK.** Babar et al. [Babar et al., 2005, Babar et al., 2006] führen ein Datenmodell für die Softwarearchitektur an. Dieses Modell identifiziert und definiert das architektonische Grundprinzip der Konstruktionen und deren Beziehungen, um die Architektur-Prozessaktivitäten zu unterstützen. Dieses Datenmodell besteht aus zwölf ADD Elementen, einschließlich der Architekturentscheidungen, *Architectural-Significant-Requirement* (ASR) und Begründungen. Das Modell hat vier Phasen für den Prozessablauf, Planung der Wissenerfassung, Wissenerfassung, Wissensorganisation und –Validierung und Wissensspeicherung. Abbildung 6 stellt diesen Prozessablauf des DAMSAK für ADDs dar.

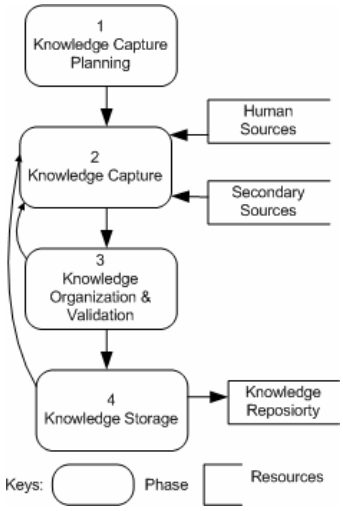

Abbildung 6: Prozessablauf des DAMSAK für Architectural Design Decisions [Quelle: Figur 3, Babar et al., 2005]

7. **AREL Modell.** AREL (*Architecture Rationale and Element Linkage*) [Tang et al., 2006] handelt sich um ein rational-basiertes Architektur-Designmodell, das sich auf die Argumentationsprozesse (*reasoning processes*) der ADDs konzentriert. Das AREL Modell ist eine Implementierung des konzeptionellen Modells mittels *Unified Modeling Language* (UML). Es erfasst die Entwurfsentscheidungen durch die Analyse der *Design Rationale* (bzw. Architektonische Rationale, AR), *Design Objekte* (bzw. Architektonische Elemente, AE) und ihre Beziehungen, siehe Abbildung 7. Es bietet eine Argumentation, um zu erklären, warum *Design Objekte* existieren und von welchen Annahmen und Einschränkungen sie abhängen. Der Prozess einer Entwurfsentscheidung für das AREL Modell wird in Abbildung 8 dargestellt.

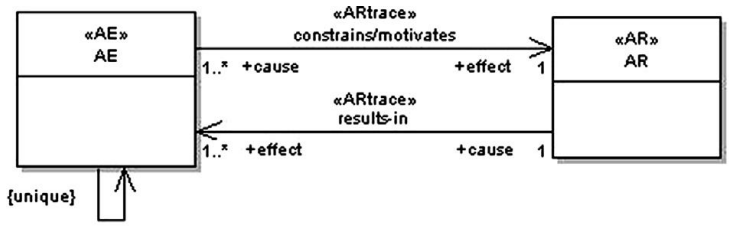

Abbildung 7: Beziehungen zwischen architektonischer Rationale und architektonischen Elementen [Quelle: Figur 2, Tang et al., 2006]

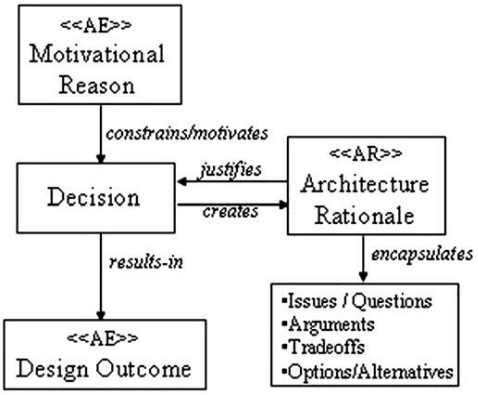

Abbildung 8: Der Prozess einer Entwurfsentscheidung für das AREL Modell [Quelle: Figur 1, Tang et al., 2006]

8. **Kernmodell.** R. C. de Boer et al. [Boer et al., 2007] führten ein Kernmodell an, um eine geeignete Architektur zu ermöglichen. Dieses Modell weist zwei grundsätzliche Charakteristiken auf, es ist minimalistisch und vollständig. Die erste Charakteristik bietet die einzigartige Möglichkeit, dass keine Mehrdeutigkeit produziert wird. Die zweite Charakteristik bedeutet, das Kernmodell sollte selbst eine Erweiterung schaffen, wenn einige wichtige Konzepte verpasst werden. Das Kernmodell umfasst zwölf Elemente.

9. **Muster-basiertes Modell.** Da die Softwarearchitekten ihre Entwurfsentscheidungen oft nicht ausreichend dokumentieren können (da sie nicht wissen, wie die Entscheidungen zu dokumentieren sind oder sie sogar die Entscheidungen nicht kennen) und weil das Entwurfsmuster bereits eine reife Technik bildet, versuchen einige IT-Wissenschaftler Entwurfsmuster zu ADDs zu erfassen, z. B. Neil B. Harrison et al. [Harrison et al., 2007]. Dieses Modell wurde im Kapitel 3.1.1 im Detail beschrieben.

10. **Wiederverwendbares Architekturentscheidungsmodell für die Enterprise-Anwendungsentwicklung.** Olaf Zimmermann et al. führten ein wiederverwendbares Architekturentscheidungs-Framework [Zimmermann et al., 2007] für die Enterprise-Anwendungsentwicklung an. Durch die Zusammenfassung vieler unterschiedlicher Techniken ermöglichten Zimmermann et al., die Architekturentscheidungen mittels eines Frameworks

wiederzuverwenden.

11. **Architecture-Design-Decision-Support-System Modell**. R. Capilla et al. [Capilla, 2007] führten ein Modell für die Architektur und Evolution der ADDs ein. Dieses Modell unterstützt das Architektur-Tool ADDSS. Dieses Modell wurde im Kapitel 2.3.2 zusammen mit dem ADDSS diskutiert.

12. **MAD 2.0 Modell**. MAD 2.0 (*Maps of Architectural Decisions 2.0*) wurde von Andrzej Zalewski et al. [Zalewski et al., 2011] angeführt. Es handelt sich um ein Notation-basiertes Modellierungsmodell. MAD 2.0 wird gefertigt, um die Entscheidungsfindung in der Softwarearchitektur für die Softwarearchitekten zu unterstützen, und zwar ohne einen effektiven Durchsetzungsmechanismus, einen bestimmten Architekturansatz oder eine Entscheidungsfindung-Ordnung. Es bietet einen grafischen Lösungsansatz für die Softwarearchitektur. Dieses Modell besteht aus zwei Diagrammen – *Architecture Decision Relationship Diagrams* (ADRD) und *Architecture Decision Problem Maps* (ADPM). Das ADRD Diagramm enthält zwei Grundelemente – das Entscheidungsproblem und den Verbindungsstecker. ADPM umfasst vier Grundelemente – Lösung, Anforderungen, Entscheidungshersteller und Vor- und Nachteile. Es verfügt über neun Syntaxen und Gültigkeitsregeln [vgl. Zalewski et al., 2011]. Abbildung 9 stellt die grafischen Elemente des ADPM Diagramms dar.

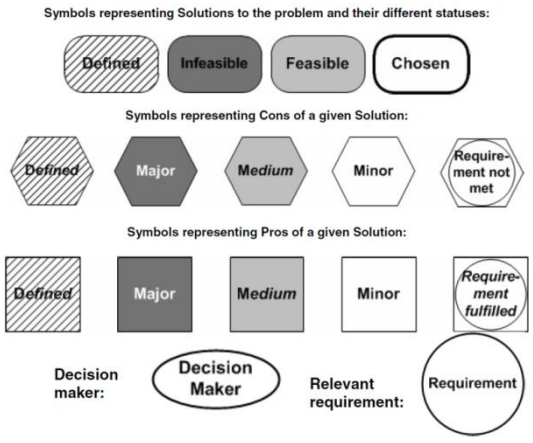

Abbildung 9: Die grafischen Elemente des ADPM Diagramms [Quelle: Figur 2, Zalewski et al., 2011]

13. **STREAM-ADD.** STREAM-ADD (***S****trategy for* ***T****ransition between* ***R****equirements and* ***A****rchitectural* ***M****odels with* ***A****rchitectural* ***D****ecisions* ***D****ocumentation*) wurden von Diego Dermeval et al. [Dermeval et al., 2012] angeführt. Dieses Modell zielt darauf ab, die Dokumentation der Architekturentscheidungen durch die Zeit zu systematisieren und die Verfeinerung der Architektur gemäß solchen Entscheidungen zu unterstützen. Der Prozessablauf des STREAM-ADD enthält drei Schritte – Anforderungen-*Refactoring*, Generierung des Architekturmodells und Verfeinerung dieses Architekturmodells mit den Architekturentscheidungen, siehe Abbildung 10.

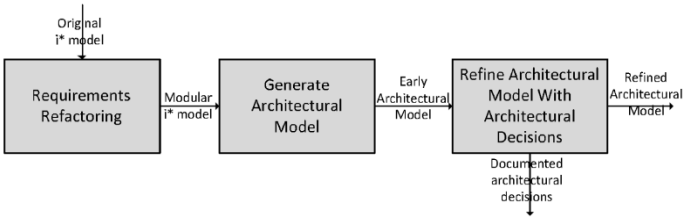

Abbildung 10: Überblick über den Prozess des STREAM-ADD [Quelle: Figur 2, Dermeval et al., 2012]

14. **FuzzDS.** FuzzDS wurde in dem Papier „*Supporting Architectural Decision Making for Systems-of-Systems Design under Uncertainty*" von Ioanna Lytra und Uwe Zdun angeführt. Das FuzzDS Modell bietet eine semi-automatische Entscheidungsunterstützung für wiederkehrende Architektur durch die Einführung eines *Fuzzy-Logik*-Expertensystems und spezifische ADDs sowie einen Ansatz zur Lösung inhärenter Unsicherheit eines Softwaresystems. Die *Fuzzy-Logik* kann den Softwarearchitekten helfen, die Ungenauigkeit und Vieldeutigkeit des Entscheidungsprozesses zu umgehen. Abbildung 11 stellt die Übersicht des FuzzsDS Modells dar, und zwar die beteiligten Tools und Rollen. Dieses Modell verfügt über zwei Stakeholder, d. h. Softwarearchitekten – den Experten und Benutzer. Die Softwarearchitekten (Experten) nutzen den Fuzzy-Entscheidungs-Modell-Editor (einen textuellen DSL-Editor), um die architektonischen Kenntnisse zu erfassen. Die Softwarearchitekten (Benutzer) nutzen den Anforderungs-Editor, um die gewünschten Anforderungen an einen

bestimmten Wert mittels eines Bewertungssystems für Fuzzy-Eingangsgrößen wie Leistung und Zuverlässigkeit zu evaluieren. Das *Fuzzy-Inferenz-System* liefert die entsprechenden Entwurfsalternativen und deren Ranking für die Anforderungen durch die Bewertung und die Kombination der Fuzzy-Regeln zurück. Schließlich werden die am besten geeigneten Designlösungen als eine Entscheidungshilfe für die Software-Architekten geliefert und die herangezogenen und bestätigten ADDs werden dokumentiert, siehe Abbildung 11 [vgl. Lytra and Zdun, 2013].

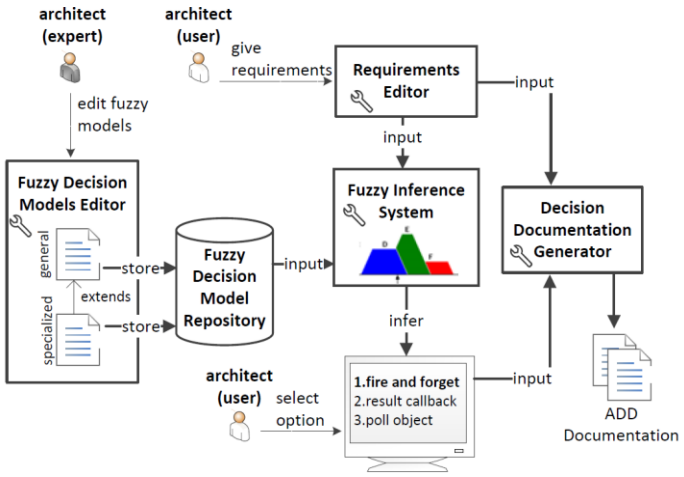

Abbildung 11: Übersicht des FuzzyDS [Quelle: Figur 3, Lytra and Zdun, 2013]

2.3.2 Existierende Tools für ADD

In dem Kapitel 2.3.1 diskutierten wir vierzehn ADMs, die meisten Modelle verfügen über ein eigenes Tool, das auf diesem Modell basiert. In diesem Kapitel stellen wir elf existierende Tools vor, die bei der Erfassung der ADDs Modelle unterstützen. Allerdings werden nicht alle diese ADD-Modelle mit einem eigenen Tool unterstützt, z. B. hat das Dokumentations-Template der Architekturentscheidung von Tyree (siehe Kapitel 2.2.1) kein entsprechendes Tool, sondern eine textbasierte Vorlage zur Erfassung der Entwurfsentscheidungen.

1. **ADDEX.** Lee implementierte ein Tool ADDEX (***A**rchitectural **D**esign **D**ecision **Ex**ploration*) [Lee, 2009], das die Ontologie von Kruchten (Siehe Kapitel 2.3.1, Modell 1) für seine Masterarbeit unterstützt. Im Gegensatz zu vielen anderen

Tools, die die ADDs erfassen, auflisten und ausführen, bietet dieses Tool die Visualisierungskomponenten für die Softwarearchitekten an, um bei der Exploration und Analyse der Entscheidungen zu helfen. Dieses Tool verfügt über vier wesentliche Aspekte: Es bietet erstens, die Auflistung der Entscheidungen und deren Beziehungen, zweitens, eine Entscheidungsstruktur-Visualisierungs-Schicht (*decision structure visualization view*), drittens, eine Entscheidung-Chronologie-Schicht (*decision chronology view*) sowie viertens, eine Entscheidung-Auswirkungen-Schicht (*decision impact view*). Diese vier Aspekte stellten ein effektives und leistungsfähiges Mittel zur Exploration und Analyse der Entscheidungen bereit [vgl. Lee, 2008].

2. **Archium Tool.** Im Jahr 2005 entwickelten Anton Janson und Jan Bosch dieses Tool (Siehe Kapitel 2.3.1, Modell 3). Bei diesem Tool handelt es sich um das früheste Tool zur Erfassung und Verwaltung der ADDs. Es basiert auf einem konzeptionellen ADM, das die Elemente der ADDs und deren Beziehungen detaillierter als das abstrakte Entwurfsentscheidungsmodell beschreibt. Das Archium Tool integriert eine Architekturbeschreibungssprache (*architectural description language, ADL*) mit Java. Diese Sprache ermöglicht dem Architekten, die Elemente aus einer *component-and-connector* Schicht zu beschreiben. Das Archium Tool bietet eine Visualisierungsmöglichkeit einer Architekturentscheidung durch ein *denpendency-graph* an, das einen Hinweis für eine Reihe von Entscheidungen ausgibt. Es kann als ein Eclipse Plug-In ausgeführt werden. Abbildung 12 zeigt die Konfigurationsstruktur des Archium Tools.

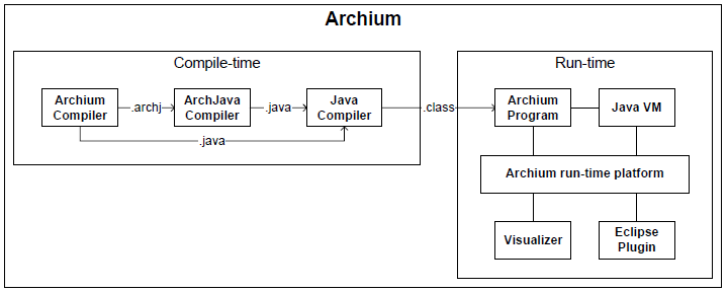

Abbildung 12: Konfigurationsstruktur des Archium Tools [Quelle: Figur 4, Jansen, 2007]

3. **AD_{kwik} .** ***A**rchitectural **D**ecision **K**nowledge **Wik**i* ist ein Web 2.0 Kollaborationssystem, das das service-orientierte Architektur-Entscheidungsmodell (SOA) (Siehe Kapitel 2.3.1, Modell 4) unterstützt. Dieses System wurde von einer Entwicklerin des AD_{kwik}, Nelly Schuster, in ihrer Diplomarbeit [Schuster, 2007] ausführlich beschrieben. Es geht um ein Web-basiertes Tool, dieses System muss nur einen Webbrowser zur Verfügung stellen. AD_{kwik} hat vier Funktionsbausteine: Kollaborationsfeatures (*Collaboration Features*), Entscheidungsverfahren (*Decision Workflow*), Content-Repository (*Content Repository*) und Abhängigkeitsverwaltung (*Dependency Management*). Ein Domänenmodell ist orthogonal zu diesen vier Bausteinen. Es unterstützt über fünfzig *Use-Cases*. Diese *Use-Cases* bieten eine Reihe an Operationen für die Entwurfsentscheidungen an. Kollaborationsfeatures und Entscheidungsverfahren realisieren die Erhaltung, Kopplung und Kooperation dieser *Use-Cases*. Das Content-Repository ermöglicht vier grundlegende Datenoperationen CRUDs, nämlich ***C**reate* (Daten anlegen), ***R**ead* oder ***R**etrieve* (Daten lesen), ***U**pdate* (Daten aktualisieren) für die Elemente des Domänenmodells. Die Abhängigkeitsverwaltung bietet eine visuelle Möglichkeit an, die die Erkenntnisse in einem Diagramm strukturiert. AD_{kwik} wurde in einer situativen Anwendung und Web 2.0-Mashup-Umgebung QEDWiki implementiert. QEDWiki erweitert die Zend PHP Framework und nutzt die XAMPP-Distribution von apachefrieds.org [vgl. Schuster et al., 2007].

4. **PAKME.** PAKME (***P**rocess-based **A**rchitecture **K**nowledge **M**anagement **E**nvironment*) ist ein Web-basiertes Tool, das das DAMSAK-Modell (Siehe Kapitel 2.3.1, Modell 6) unterstützt. PAKME besteht aus fünf Komponenten: 1. User Interface. Es wurde mittels *Java Server Pages* (JSP) und HTML Technologie implementiert, 2. *Knowledge-Management*. Diese Komponente bietet die Dienste zum Speichern, Abrufen und Aktualisieren der Artefakte an, die die Erkenntnisse der Architektur abbilden, 3. Sucher. Diese Komponente hilft Benutzern, die gewünschten Artefakte zu suchen, 4. Berichter. Diese Komponente

stellt die Dienste zur Darstellung der verschiedenen architektonischen Artefakte und deren positive oder negative Auswirkung aufeinander bereit. 5. *Repository-Management*. Diese Komponente erbringt alle Dienstleistungen einer persistenten Datenquelle, die mit PostgreSQL 8.0 implementiert wird, um die Daten zu speichern, zu verwalten oder abzurufen. PAKME bietet ein *Knowledge-Management* zur Unterstützung des Prozesses der Softwarearchitektur an. PAKME kann dem Entwickler helfen, die Architektur in Bezug auf die Entwurfsentscheidungen mit kontextbezogenen Informationen, Einschränkungen und Rationalen besser zu beschreiben. Im Gegensatz dazu können diese Kontextinformationen und Rationale dem Entwickler wieder helfen, die Kontexte und Argumentationen für jede ihrer ADDs sowie deren Einschränkungen und Designregeln besser zu verstehen [vgl. Babar and Gorton, 2007].

5. **AREL.** AREL (***A**rchitecture **R**ationale and **E**lements **L**inkage*) ist ein UML-basiertes Tool, das in der Erstellung und Dokumentation des Architektur-Designs mit dem Schwerpunkt auf Architekturentscheidungen und *Design Rationale* zielt (Siehe Kapitel 2.3.1, Modell 7). Es bietet eine bequeme Möglichkeit, damit die neue Vorlagen (*templates*) hinzugefügt werden, um die standardisierte UML zu erweitern, und verfolgt die ADDs von dem Problem zu dem Lösungsraum. Die Benutzer können den Ablauf von Entwurfsergebnissen zu Entwurfsentscheidungen und wieder von Entwurfsentscheidungen zu dem Design-Treiber zurückverfolgen. AREL kann die Auswirkungen der Änderung von *architectural knowledge* identifizieren und analysieren. Die Benutzer können alle ADDs und andere *architectural knowledge* Elemente, die direkt oder indirekt geändert werden, identifizieren und analysieren. AREL kann die Konflikte der unterschiedlichen ADDs entdecken [vgl. Tang et al., 2006].

6. **Knowledge Architect.** KA (***K**nowledge **A**rchitect*) ist eine Tool-Suite zu der Erstellung, Verwendung, Übersetzung, dem Austausch und der Verwaltung der Architektur-Kenntnisse. Diese Tool-Suite beinhaltet die Unterstützung für die Integration der verschiedenen Aktivitäten des Architekturprozesses und die

Förderung der Kollaboration zwischen den Stakeholdern. Diese Tool-Suite ist eines der Ergebnisse des GRIFFIN-Projekts und unterstützt das Kernmodell (Siehe Kapitel 2.3.1, Modell 8). Im Allgemeinen richtet sich die KA Tool-Suite nach vier erforderlichen Eigenschaften während des Herstellungsprozesses der Architekturentscheidungen: einer zentralen *Knowledge-Nabe*, Rückverfolgbarkeit der Architekturentscheidungen, Translation der *Knowledge* und automatisierten Prüfung der Architekturentscheidungen. Die KA Tool-Suite besteht aus sechs Subtools: Knowledge-Repository, Document-Knowledge Client, Excel Plug-in, Python Plug-in, Knowledge-Explorer und Knowledge-Translator. Die wichtigsten Funktionen der KA Tool-Suite sind: 1. Erfassen, Kommentieren, Bearbeiten und Visualisieren von AK-Entitäten und deren Beziehungen, 2. Absuchen und Abrufen der AK-Entitäten und deren Beziehungen in *Knowledge-Repository*, 3. Übersetzen der AK in verschiedene AK Domänenmodelle – von einem zum anderen – durch *Knowledge-Translator*, 4. Kooperation der AK-Entitäten mit anderen Benutzern und Stakeholdern [vgl. Liang et al., 2009].

7. **ADvISE.** ADvISE (***A**rchitectural **D**esign dec**I**sion **S**upport fram**E**work*) wurde als Muster-basiertes ADM (Siehe Kapitel 2.3.1, Modell 9) entwickelt. Die Autorin Ioanna Lytra und ihre Kollegen [Lytra et al., 2012] interviewten neun Plattform-Experten aus drei Unternehmen für die Überarbeitung der architektonischen *Knowledge* hinsichtlich der Erfassung der Architekturmuster und des Entscheidungsmodells. Dann verglichen sie diese Ergebnisse und Architekturmuster. Als endgültiges Ergebnis entwickelten sie ein Muster-basiertes Architekturentscheidungsmodell. Da WebADvISE sowie ADvISE, basierend auf diesem Modell, entwickelt werden, soll es in dieser Arbeit eingehender beschrieben werden (Siehe Kapitel 3.1.2).

8. **ADDSS.** ADDSS (***A**rchitecture **D**esign **D**ecision **S**upport **S**ystem Tool*) ist ein Web-basiertes Tool zur Erfassung und Dokumentation der ADDs (Siehe Kapitel 2.3.1, Modell 11). Dieses Tool basiert auf der Idee der Entscheidungssicht, die wir im Kapitel 2.2.1 bereits beschrieben haben. ADDSS stellt die typischen

Operationen für ADDs bereit, z. B. Erfassung, Verwendung, Verwaltung und Dokumentation der ADDs. Es kann noch andere Funktionen für die Architekten anbieten: 1. Erfassung der Design-Rationale, 2. Erfassung des *architectural knowledge*, 3. Erfassung der expliziten Beziehungen zu anderen Produkten des Software-Lebenszyklus, 4. Iterative Bauprozesse, 5. Explizit dokumentierte *Knowledge*. ADDSS wird mittels Client-Server Ansatz implementiert. Die Client-Seite verwendet AJAX und XML-Technologien, um die Visualisierung der Architekturen zu verbessern. Die Server-Seite setzt PHP mit der MySQL-Datenbank ein, um die Anforderungen, Architekturen und Entwurfsentscheidungen zu speichern [vgl. Capilla et al., 2008].

9. **ArchiTech.** Bei ArchiTech handelt es sich um ein Tool zur Unterstützung für nicht funktionsbezogene Anforderungen an eine (NFRs)-geführte architektonische Entscheidungsfindung in *Model-Driven Development* (MDD) [Ameller et al., 2010]. Es geht um eine Reihe von Qualitätsanforderungen und Einschränkungen einer *Software-Requirements-Specification* (SRS), die von den Softwarearchitekten stammen. Als Ergebnis der Entscheidungsprozesse werden eine Reihe von ADDs und eine Gesamtbewertung der Qualitätsattribute als Folge der Anwendung der daraus resultierenden Entwurfsentscheidungen angeboten. Die erzielten Ergebnisse erfordern ein Management und die Wartbarkeit der architektonischen Kenntnisse. Diese Methoden werden von ArchiTech-CRUD in Form von CRUD (Create Read Update Delete) angeboten. Abbildung 13 stellt eine Übersicht des ArchiTech dar [vgl. Ameller et al., 2012].

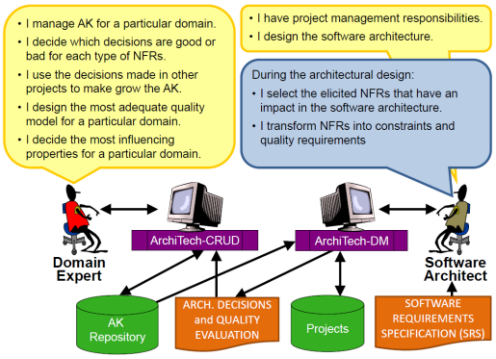

Abbildung 13: Übersicht des ArchiTech [Quelle: Figur 1, Ameller et al., 2012]

10. **ArchPad.** ArchPad (*Architectural pattern- and decision-based design method*) wurde als eine Muster-basierte Methode zur architektonischen Entwurfsentscheidung in dem Paper „*Combining Pattern Languages and Reusable Architectural Decision Models into a Comprehensive and Comprehensible Design Method*" von Olaf Zimmermann et al. [Zimmermann et al., 2008] angeführt. Damit können Architekten über die wiederwendbaren ADDs mittels Entwurfsmustern in der Praxis entscheiden. Wir verwenden diese Methode, um den Architekturprozess der ADDs von WebADvISE zu beschreiben. ArchPad hat vier Phasen. Die vier Phasen sollen keinen linearen, nicht-invertierbaren Wasserfall umfassen, sondern sind lediglich Informationsflüsse, siehe Abbildung 14. Sie werden im Kapitel 3.2.1 verwendet.

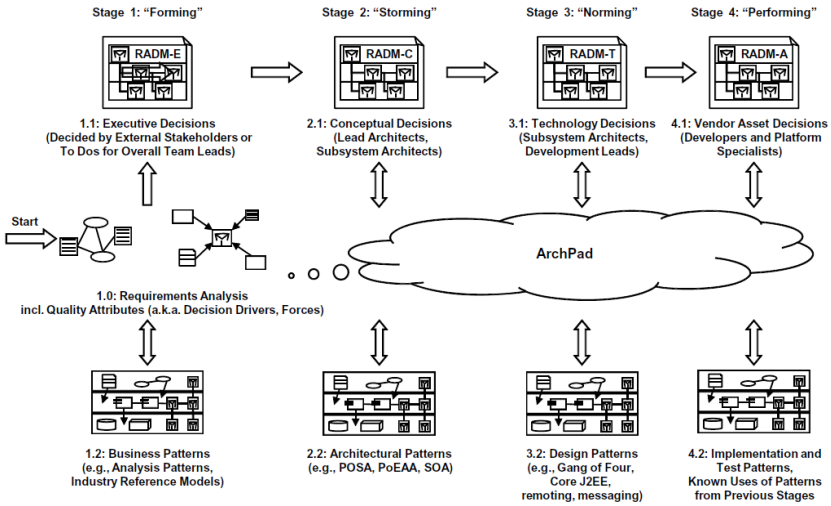

Abbildung 14: Ablaufsprozesse von ArchPad

11. **MAD 2.0 Tool.** MAD 2.0 (Maps of Architectural Decisions 2.0) Tool ist ein Notation-basiertes Tool zur Beschreibung der Architekturentscheidungen für eine Softwarearchitektur. Es unterstützt das MAD 2.0 Modell. Dieses Tool wird als ein Diagrammeditor und eine Erweiterung zu MS Word entwickelt. Die Architekten und Analysten können mittels MS Word die Dokumentation der Entwurfsentscheidungen schaffen. Es bietet auch ein Diagramm für das *Architecture Decisions Relationship Diagram* (ADRD) und *Architecture Decision Problem Map* (ADPM). Dieses Diagramm kann von dem MS Word Menü

aufgerufen werden [vgl. Zalewski et al., 2011].

2.3.3 Der Vergleich zwischen unterschiedlichen Tools

In dem Kapitel 2.2.3 diskutierten wir die historische Evolution der ADDs, und ein typisches Dokumentation-Template (siehe Tabelle 1) zur Beschreibung einer ADD wurde am Ende des Kapitels vorgeschlagen. Laut diesem Dokumentation-Template verglichen wir die acht unterschiedlichen ADD Modelle, die ein unterstützendes Tool aufweisen. Die Ergebnisse werden in Tabellen (siehe Tabelle 6 und 7) gegenübergestellt. M mit einer Nummer (z. B. M1) bedeutet ein Modell, das im Kapitel 2.2.1 definiert wird.

Dokumentation Template	**M1**	**M3**	**M4**	**M6**	**M7**	**M8**
Tools	**ADDEX**	**Archium**	**AD_{kwik}**	**PAKME**	**AREL**	**KA**
Kernproblem	-	▹ Probleme ▹ Motivation ▹ Ursache	▹ Probleme	-	▹ Entwurfsprobleme	▹ Entscheidungsthema
Entscheidung	▹ DD	▹ Entscheidung	▹ AK	▹ AD	▹ DD	▹ ADD
Status	▹ Zustand	-	▹ Zustand	-	-	-
Gruppe	▹ Kategorie	▹ AD Thema	▹ Kategorien der AK	-	-	-
Hypothesen	-	▹ Entwurfsregeln	▹ Architekturprozess	▹ Architektonische Signifikante	▹ Architektur-Treiber	▹ Entwurfs-Concerns
Einschränkungen	▹ Anforderung ▹ Defekt ▹ Risiko	▹ Entwurfs-Einschränkung	▹ Knowledge-Respositry	▹ Anforderungen ▹ QA ▹ Szenarios	-	-
Positionen	▹ Idee ▹ Probe	▹ Mögliche Lösung	▹ Mögliche AK	▹ Entwurfs-Optionen	▹ Alternativer Entwurf	▹ Alternative
Argument	▹ Begründung	▹ Begründung ▹ Vor- ▹ Nachteile	▹ AK	▹ Begründung	▹ Qualitative ▹ Qualitative	▹ Entscheidungsschleife
Implikationen	▹ DD ▹ Anforderungen	▹ Entwurfsregeln ▹ Entwurfs-Einschränkung	▹ AK	▹ Entwurfs-Optionen ▹ Anforderungen	▹ Alternativer Entwurf	▹ ADD
Verbundene Entscheidungen	▹ Beziehungen	-	▹ Beziehungen	▹ Beziehungen	-	▹ Entscheidungsschleife
Verbundene Anforderungen	▹ Anforderungen	-	-	▹ Anforderungen	-	-
Verbundene Artefakte	▹ Verfolgen	-	-	-	▹ Vorwärts- und Rückwärtsverfolgung	▹ Reflektieren
Verbundene Prinzipien	-	-		-		-
Notizen	▹ Text ▹ Graph	▹ Text ▹ Graph	▹ Text ▹ Graph	▹ Text	▹ Text ▹ Graph	▹ Text ▹ Graph

Tabelle 6: Der Vergleich zwischen unterschiedlichen Modellen, die acht bekannte ADD Tools unterstützen[vgl. Shahin et al., 2009a].

Dokumentation Template	**M9**	**M11**	**-**	**M9**	**M12**
Tools	**ADvISE**	**ADDSS**	**ArchiTech**	**ArchPad**	**MAD 2.0**
Kernproblem	▹ Entwurfsprobleme	▹ Anforderungen	▹ NFRs	▹ Entwurfsprobleme	▹ Entwurfsprobleme

Entscheidung	▹ ADD	▹ ADD	▹ AK	▹ ADD	▹ DD
Status	▹ Zustand	▹ Zustand	-	-	-
Gruppe	▹ Kategorie	▹ Kategorie der Entscheidungen	▹ Kategorie der AK	▹ Kategorie der Entwurfsmuster	▹ Entwurfs-diagramme
Hypothesen	▹ Architekturmuster	▹ Annahme	-	▹ Muster-basiert	▹ Notation-basiert
Einschränkungen	▹ Anforderungen	▹ Einschränkungen ▹ Qualitätsattribute	▹ NFRs ▹ Qualitätsattribute	▹ Anforderungen ▹ Qualitätsattribute	▹ Syntax ▹ Gültigkeitsregeln
Positionen	▹ Mustern ▹ DD	▹ Alternative Entscheidungen	▹ Mögliche AK	▹ Muster	▹ Alternative Lösungen
Argument	▹ Muster-basiert	▹ Begründung ▹ Vor- ▹ Nachteile	▹ AK	▹ Muster-basiert	▹ Notation-basiert
Implikationen	▹ Mustern ▹ ADD	▹ ADD ▹ Anforderungen	▹ AK	▹ Muster ▹ ADD	▹ Anforderungen
Verbundene Entscheidungen	▹ Beziehungen	▹ Abhängigkeiten	▹ Beziehungen	▹ Beziehungen	▹ Verbindungsstecker
Verbundene Anforderungen	▹ Anforderungen	▹ Anforderungen	▹ NFRs	▹ Anforderungen ▹ Bezogene Patterns	▹ Anforderungen
Verbundene Artefakte	-	-	▹ ArchiTech-CRUD	▹ Entwurfsmustern	-
Verbundene Prinzipien	▹ Architekturmustern	-	▹ SRS ▹ Architekturentscheidungen und Qualitätsbewertung	▹ RADMs	▹ Syntax ▹ Gültigkeitsregeln
Notizen	▹ Text	▹ Text ▹ Graph	▹ XML File	▹ Text	▹ Text ▹ Graph

Tabelle 7: Der Vergleich zwischen unterschiedlichen Modellen, die acht bekannte ADD Tools unterstützen[vgl. Shahin et al., 2009a].

In der Tabelle 8 vergleichen wir die Ähnlichkeiten und Unterschiede zwischen den acht verschiedenen ADD Modellen. Diese Unterschiede in der Benennung, Erfassung und Verwendung der Elemente von ADD führen zu einer erheblichen Differenz zwischen deren Tools, die auf der Basis verschiedener ADD Modelle implementiert werden. Folgend definieren wir zehn Kriterien, um diese Tools zu vergleichen. Diese Tools ermöglichen den Stakeholdern, miteinander zu kooperieren. Die Tabelle 8 vergleicht die acht ADD Tools mittels oben erwähnter zehn Kriterien, wobei „+“ „unterstützt“ bedeutet und „-“ „nicht unterstützt“ [vgl. Tang et al., 2010].

1. **Unterstützung für ADD.** Alle beschriebenen Tools stellen die Darstellung, Erfassung, sowie Verwaltung der ADDs zur Verfügung.

2. **Unterstützung für Schichten.** Wenn ein Architektur-Modell die Schichten unterstützen kann, bietet dieses Modell sicher einen deutlicheren Dokumentationsansatz. Aber unterschiedliche Modelle und deren Tools verwenden verschiedene Sichten, z. B. ADDSS setzt die Entscheidungssicht ein, AREL den *viewpoint*-basierten Ansatz, KA Tool-Suite verwendet die Topologie-Sicht und PAKME unterstützt die Architektur-Sichten.

3. **Unterstützung für Muster.** ADDSS, Archium, PAKME, ArchPad und ADvISE stellen die Unterstützung für Muster und Architekturstile bereit. Muster sind oft in PAKME, ADDSS, ArchPad und ADvISE sichtbar, nämlich die Architekturmuster wurden als die Designalternativen für eine Architekturentscheidung berücksichtigt.

4. **Prozessverfolgung für ADD.** Das KA Tool-Suite und ADDSS und ADvISE können den Zustand, die Herstellung und die Stakeholder für eine Entwurfsentscheidung verfolgen. AREL; PAKME, Archium, ArchiTech, ArchPad und MAD 2.0 bieten diese Prozessverfolgung nicht an.

5. **Klassifizierung/Kategorisierung der Entscheidungen.** ADDSS, PAKME und ADvISE stellen die Klassifizierung der Entwurfsentscheidungen bereit.

6. **Unterstützung für die quantitative Analyse.** AREL, PAKME und KA Tool-Suite, ArchiTech und MAD 2.0 unterstützen die quantitative Analyse.

7. **Unterstützung für Szenarien.** AREL, PAKME, ADvISE, ArchiTech, ArchPad, MAD 2.0 und Archium haben die Konzepte zur Unterstützung für architektonische Szenarien, wie in ATAM verwendet. Sowohl KA Tool-Suite als auch ADDSS unterstützen keine architektonischen Szenarien.

8. **Grafische Darstellung.** Nur PAKME, ADvISE, ArchiTech und ArchPad können die grafische Darstellung nicht unterstützen. Aber ADvISE kann mit VbMF (***V**iew-based **M**odeling **F**ramework*) zusammenarbeiten und eine Möglichkeit der grafischen Darstellung anbieten.

9. **Web-basiert.** Die Web-basierten ADD Tools sind AD_{kwik}, PAKME und ADDSS.

10. **Unterstützung für Kollaboration.** AD_{kwik} und KA Tool-Suite werden als Kollaborationssystem implementiert.

Tools	1. Unterstützung für ADD	2.Unterstützung für Schichten	3.Unterstützung für Muster	4.Prozessverfolgung für ADD	5.Klassifizierung/Kategorisierung	6.Unterstützung für quantitative Analyse	7.Unterstützung für Szenarien	8.Grafische Darstellung	9.Web-basiert	10.Unterstützung für Kollaboration
ADDEX	+	-	-	+	-	-	+	+	-	-
Archium	+	-	+	-	-	-	-	+	-	-
AD_{kwik}	+	-	-	-	-	-	-	+	+	+
PAKME	+	+	+	-	+	+	+	-	+	-
AREL	+	+	-	-	-	+	+	+	-	-
KA	+	+	-	+	-	+	+	+	-	+
ADvISE	+	-	+	+	+	+	+	-	+	-
ADDSS	+	+	+	+	+	+	-	+	+	-
ArchiTech	+	-	-	-	-	+	+	-	-	-
ArchPad	+	-	+	-	-	-	+	-	-	-
MAD 2.0	+	-	-	-	-	+	+	+	-	-

Tabelle 8: Der Vergleich zwischen unterschiedlichen ADD Tools[vgl. Tang et al., 2010]

3. Web-basiertes Tool WebADvISE

WebADvISE (***Web**-based **A**rchitectural **D**esign dec**I**sion **S**upport fram**E**work*) ist ein Web-basiertes wiederverwendbares ADDs-Tool und eine Web-Version von ADvISE (siehe Kapitel 3.1.2). Es handelt sich um einen Teil dieser Masterarbeit. Mit diesem Tool können die Stakeholder (meistens Softwarearchitekten) die wiederverwendbaren und zurückführbaren ADDs für ein Softwaresystem je nach dem zu einer Reihe architektonischer Fragen leicht finden, erfassen, editieren, verwalten und speichern. Es wird auf Basis der *Remote Application Platform* (kurz RAP, vormals: *Rich Ajax Platform*) [RAP, 2014] entwickelt. RAP arbeitet auf der Modulplattform *Open Service Gateway Initiative* (OSGI). WebADvISE erbt fast alle Funktionen von ADvISE und bietet eine Web-basierte Implementierung an, es bringt gleichzeitig eine Kollaborationsmöglichkeit für verschiedene Stakeholder bzw. Benutzer für die Softwarearchitektur mit, um eine effiziente Zusammenarbeit verschiedener Benutzer innerhalb eines gemeinsamen Projekts zu unterstützen. Es lässt sich die Historie (Logs) der Aktivitäten der verschiedenen Benutzer (Mitglieder einer Gruppe) automatisch erkennen, z. B. wird es erkannt, wann und welches ADDs-Projekt von welchem Mitglied in eine Gruppe geändert wird. Diese Informationen werden als ein „Change Log“ in WebADvISE gespeichert und sind für alle Mitglieder sichtbar.

Da WebADvISE auf der Architekturmuster-basierten ADM, QOC Methode, entwickelt wird, besprechen wir getrennt die Grundkenntnisse für WebADvISE als Schwerpunkte im Kapitel 3.1.1 und 3.1.3. ADvISE wird im Kapitel 3.1.2 besprochen. Das Kapitel 3.2 beschreibt die Implementierung und Architektur von WebADvISE. Wie ein Softwarearchitekt die Architektur und Architekturentscheidungen einer Softwareapplikation dokumentieren kann, stellt eine der Kernfragestellungen dieser Arbeit dar. Deswegen versuchen wir eine praxisnahe Art und Weise der Beschreibung bzw. eine Methode zur Beschreibung der Architektur und Entscheidungsprozesse von WebADvISE als Antwort auf diese Kernfragestellung darzustellen. Diese

Beschreibungsmethode besteht darin, dass wir die ArchPad und QOC Methoden zur Beschreibung der Architektur von WebADvISE verwenden. Die folgenden drei Gründe legen dar, warum wir diese Beschreibungsmethode in dieser Masterarbeit verwenden. Der erste Grund ist, um die Architektur von WebADvISE und deren Architekturentscheidungen deutlicher zu beschreiben. Als zweiter Grund sollen die zwei Grundkenntnisse bzw. Grundprinzipien der Arbeit von der WebADvISE Architekturmuster-basierten ADM und QOC Methode besser erläutert und präsentiert werden. Der dritte Grund liegt darin, die Durchführbarkeit dieser zwei Grundprinzipien in der Praxis nachzuprüfen. Als Ergebnisse dieser Forschung werden die Gebrauchstechnologien von WebADvISE im Kapitel 3.2.2 diskutiert. Das Kapitel 3.2.3 beschreibt die sieben Funktionalitätsmodule von WebADvISE als Plug-ins in der Eclipse RAP Plattform. Und die MVC Architekturmuster für UI Design werden im Kapitel 3.2.4 besprochen. Das Kapitel 3.3 dokumentiert alle Funktionalitäten von WebADvISE sowie ein Handbuch. Eine wichtige Funktionalität, die Kollaboration, wird im Kapitel 3.4 im Detail beschrieben.

3.1 Grundlagen

3.1.1 ADM für WebADvISE

Es wird dargelegt, dass in den letzten Jahren eine Softwarearchitektur einerseits immer weniger nur als *components-and-connectors* dargestellt und mehr und mehr als eine Reihe von ADDs betrachtet wird. Die wichtigsten Kontexte oder Inhalte zur Dokumentation einer Softwarearchitektur sind nicht nur *components-and-connectors,* sondern auch die *Design Rationale* für einen Architekturentwurf mit solchen ADM. Andererseits basiert die Softwarearchitektur heutzutage auf einer oder mehreren Software-Plattformen, zum Beispiel, SAP R/3 in *Enterprise Ressource Planning* (ERP), Facebook Plattform in sozialen Netzwerken, Amazon S3 in verteilter Datenhaltung, *Googles Android* und Apples *iOS* Plattformen in mobilen Anwendungen und *Mobicents* als wiederverwendbare *VoIP*-Infrastruktur. Diese

Software-Plattformen können eine Sammlung von Software-Subsystemen sowie Kommunikations-Middleware und Datenbanken sowie eine Schnittstelle, die gemeinsam mit einer wiederverwendbaren Infrastruktur für die Entwicklung einer Reihe von Software-Applikationen gebildet wird, darstellen [vgl. Lytra et al., 2012].

Aus diesem Grund entwickelten und führten Lytra, Soberning und Zdun ihr Muster-basiertes ADM [Lytra et al., 2012] an, das auf der ADDs für die Software-Plattformen basiert. Gleichzeitig behandelten Ioanna Lytra et al. eine systematische Literaturrecherche, und aus dieser leiteten sie eine Mustersprache (*pattern language*) für vierzig Plattform-integrierte Muster sowie ein Muster-basiertes ADM ab. Dann interviewten Ioanna Lytra et al. neun Plattform-Experten aus drei verschiedenen Unternehmen für die Überarbeitung der architektonischen Erkenntnisse hinsichtlich der Mustersprachen und des Entscheidungsmodells. Als Ergebnis führten sie ein qualitatives validiertes, Muster-basiertes ADM und ein generalisiertes Modell aus verschiedenen Ebenen und Phasen der ADDs für Service-basierte Plattformen an. Als ein Teil dieser Forschung entwickelte Ioanna Lytra eine ADD Tool ADvISE zur Unterstützung dieses Muster-basierten ADM. WebADvISE wird basierend auf diesen ADM und ADvISE entwickelt.

3.1.2 ADvISE

ADvISE (***A****rchitectural* ***D****esign dec****I****sion* ***S****upport fram****E****work*) ist einer der Beiträge der Software Architecture Group, Fakultät für Informatik, Universität Wien, für das EU FP7 Projekt INDENICA. Es wird für die Modellierung der wiederverwendbaren Architekturentscheidungen und Architekturentscheidungen unter Unbestimmtheit (unter Verwendung von *Fuzzy Logic*) auf verschiedenen Ebenen der Abstraktion, der High-Level sowie Technologie- und Domain-spezifischen Ebenen vorgeschlagen. Die Dokumentation dieser Architekturentscheidungen basiert auf QOC. Dieser Ansatz erlaubt es uns, den halbautomatisierten Entscheidungsprozess für wiederverwendbare Architekturentscheidungen zu erledigen und die Design-Rationale zu niedrigen

Kosten zu dokumentieren. Auf diese Weise können die Softwarearchitekten mehr Zeit für die anspruchsvollen architektonischen Probleme, die kreatives Denken erfordern, verwenden. Darüber hinaus sollen die Softwarearchitekten, um die Architekturentscheidungen zu halten und die konsistenten und nachvollziehbaren Designs anzubieten, die erzeugten wiederverwendbaren Architekturentscheidungen und architektonischen Entwürfe miteinander verknüpfen. Außerdem lässt ADvISE sich mit einer anderen Tool VbMF (***V**iew-**b**ased **M**odeling **F**ramework*) integrieren bzw. kann mit diesem zusammenarbeiten. Auf diese Art und Weise kann die Wartungsunterstützung für sich entwickelnde Architekturentscheidungen und Designs angeboten werden. Aus technischer Sicht wurde ADvISE als ein *Eclipse Plug-In* entwickelt. Dieses Tool bildet die Grundlage der Implementierung von WebADvISE.

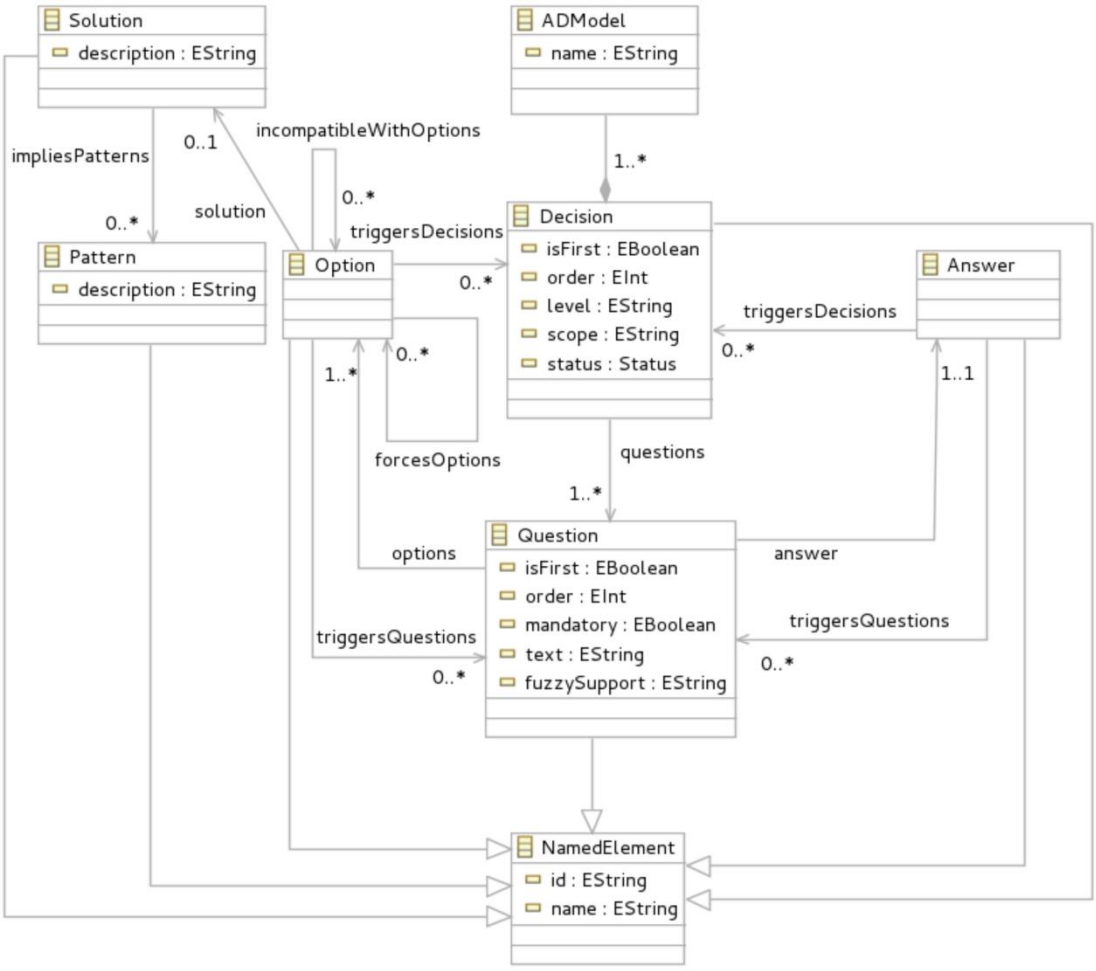

Abbildung 15: ADMM in ADvISE sowie in WebADvISE [Quelle:ADvISE, 2014]

Das *Architectural Decision Meta-Model* (ADMM) in WebADvISE wird vollständig

von ADvISE geerbt, siehe Abbildung 15. ADvISE und WebADvISE verwenden beide die Technologie *Eclipse Modeling Framework* (EMF), die es ermöglicht, die Java-Klassen automatisch zu generieren. Gleichzeitig definiert EMF die Struktur der ADMM. Die Abbildung 15 wird von EMF und in einem *Ecore-Diagram* Editor erstellt. Anhand dieser Abbildung können wir einfach erkennen, dass der ADMM in ADvISE und WebADvISE auf Basis der QOC Methode und der Muster-basierten Ideen entwickelt wird, wie im Kapitel 2.2.4 besprochen wurde. Wir definieren die Fragestellung (*Question*) und die entsprechende Antwort (*Answer*), Option (*Option*) sowie den entsprechenden Lösungsansatz (*Solution*). Dieser Lösungsansatz löst das entsprechende Muster (*Pattern*) oder die Muster aus. Schließlich lässt sich eine Entscheidung (*Decision*) aus diesem Entscheidungsprozess herstellen.

3.1.2 QOC Methode und Fragebogen

Bei QOC handelt es sich um einen Dokumentationsansatz der Softwarearchitektur zur Darstellung der *Design-Rationale*. *Questions* identifizieren die wichtigsten Kernprobleme eines Designs, *Options* bieten die möglichen Antworten auf die Probleme an, und *Criteria* sind die Kriterien für die Beurteilung und den Vergleich der Optionen. Diese Methode wurde von Allan MacLean et al. [MacLean et al., 1991] im Jahr 1991 veröffentlicht. Die Hauptcharakteristiken von QOC sind: 1. Fokussiert auf Design-Space, 2. Fokussiert auf *Criteria*, 3. Nebenprodukt des Designs, 4. Eingebettet in Design-Aktivität, 5. Semiformal, 6. Argument-basiert, 7. Erweiterbares Detail, 8. Zielgerichtet. Als Beispiele schlugen MacLean et al. zwei bekannte Beispiele, das Design-Konzept des Scroll-Bar von Windows und Workflow des ATM, vor.

Die Dokumentationsmethode QOC kann eine dauerhafte und wiederverwendbare Dokumentation der Entscheidungen für die Softwarearchitekten anbieten. Es ist wie ein „Fragebogen“ mit Fragen und Antworten zu betrachten. Der Softwarearchitekt kann die Architekturentscheidungen zu einer Reihe von architektonischen Fragen aus

einem Fragebogen für sein Softwaresystem machen. Diese gleiche Funktionalität wird auch in WebADvISE implementiert. Abbildung 31 zeigt einen Fragebogen-Editor in WebADvISE.

3.2 Implementierung

3.2.1 Architektur und Architekturentscheidungen von WebADvISE

Um eine deutliche und systematische Beschreibung der Architektur von WebADvISE zu erreichen, befolgen wir die vier Phasen der Architekturmuster-basierten Entwurfsmethode ArchPad und damit verwenden wir auch die ADD Dokumentierungsmethode QOC in diesen Phasen. Die drei Gründe, warum wir hier ArchPad und QOC zur Beschreibung der Architektur von WebADvISE verwenden, wurden am Anfang des Kapitels 3 bereits erklärt. Q (*Question*) bedeutet hier: Welche architektonische Frage müssen wir beantworten? O (*Option*) heißt hier: Welche Optionen haben wir? C (*Criteria)* bedeutet hier: Wieso wählen wir diese Option aus? Die rote Ablaufverfolgung jedes Diagramms beschreibt die Entscheidungsschritte der ADDs in jeder Phase.

Phase 1: Exekutive Entscheidungen, Anforderungsanalyse

Für erste Phase definieren wir zwei exekutive Entscheidungen bzw. architektonische Fragen, welche Programmiersprache auswählt und auf welcher Plattform diese Web-basierte Softwareapplikation entwickelt werden soll. Da WebADvISE basierend auf ADvISE entwickelt wird, sollen wir den Gebrauchstechnologien von ADvISE zu folgen versuchen, um die Wiederverwendbarkeit des Codes zu erhöhen. ADvISE ist ein Eclipse Plug-In, deshalb wählen wir die Programmiersprache Java als erste Architekturentscheidung. Für die Applikationsplattform haben wir nicht nur eine Auswahl, sondern können auch J2EE auswählen, weil es eine schwergewichte und leistungsfähige Framework darstellt. Aber ein Schwergewicht zu sein, ist auch ein Nachteil und andererseits haben wir eine bessere Auswahl bei der OSGI & Eclipse

RAP Plattform. Es ist nicht nur leichtgewichtig, sondern beinhaltet eine native Unterstützung für Eclipse Plug-In-Applikation.

Wie sollen wir ein WebADvISE Projekt für verschiedene Mitglieder in einer Forschungsgruppe verwalten – auch dies stellt eine sehr wichtige exekutive Architekturentscheidung dar. Es gibt normalerweise drei Arten der Projektverwaltung (*Project Management*). Die erste ist eine zentrale Verwaltung, sie bietet oft eine veröffentliche Verwaltung, z. B. traditionelle Newsgroups oder Foren. Die zweite betrifft die einzelne Verwaltung. Jeder Benutzer hat seine eigenen Projekte, z. B. das *Eclipse* RCP Projekt. Die dritte ist richtet sich auf die kollaborative Arbeitsverwaltung, mit ihr können alle identifizierten Benutzer eines Projekts zusammenarbeiten. Das bekannteste Beispiel bildet Wikipedia, hier können alle Benutzer einen Artikel kollaborativ bearbeiten, egal an welchem Ort auf der Welt. In den meisten existierenden ADDs Tools (siehe Kapitel 2.3.2) muss ein Softwarearchitekt entweder Export-/ Import-Mechanismen oder Versionsverwaltungsprogramme verwenden, um die Daten eines ADDs Projekts mit anderen Softwarearchitekten austauschen zu können. Dies ist keine ideale Wahl für die Zusammenarbeit zwischen verschiedenen Softwarearchitekten. Durch diese Anforderungsanalyse wollen wir versuchen, ein kollaboratives Web-basiertes ADDs Tool für mehrere Softwarearchitekten anzubieten. Abbildung 16 illustriert Phase 1.

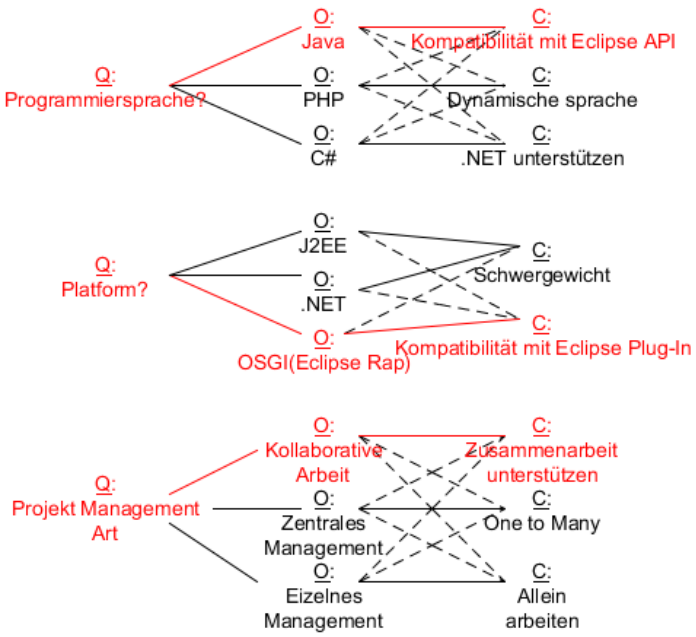

Abbildung 16: QOC Darstellung (Phase 1)

Phase 2: Konzeptionelle Entscheidungen einschließlich der Alternativen von Architekturmustern und Kerntechnologien

Für Message Exchange Pattern haben wir Request-Response ausgewählt, weil Eclipse RAP mit Servlet besser zusammenarbeiten kann. Und Java HTTP Servlets bietet eine native Unterstützung für die *Request-Response Pattern* an. Gleichzeitig ermöglicht diese Kombination, Eclipse RAP und Servlet APIs, einen guten Mechanismus des Thin Client für *Presentation Layer Binding* und den Mechanismus des *Client Session* für den Sitzungszustand. Abbildung 17 illustriert Phase 2.

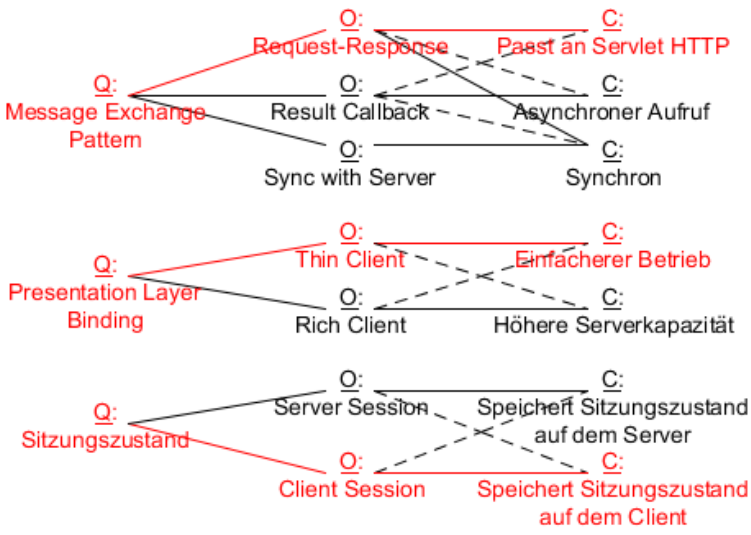

Abbildung 17: QOC Darstellung (Phase 2)

Phase 3: Detaillierte technische Entscheidungen, Entwurfsmuster als Alternativen der Architektur

Wir haben das Servlet *Model-View-Controller* (MVC) Entwurfsmuster ausgewählt. Ein Servlet MVC Muster kann nicht nur einen flexiblen Programmentwurf und hohe Wiederverwendbarkeit, Modifizierbarkeit sowie Testbarkeit anbieten, sondern auch eine ursprüngliche Unterstützung für Eclipse RAP Framework mit sich bringen. Dies wird im Kapitel 3.2.4 im Detail beschrieben. Die anderen zwei Muster, *Business Delegate* und *Dependency Injection,* müssen in schwergewichtigeren Frameworks verwendet werden, z. B. EJB Framework unterstützt *Business Delegate* und Spring Framework *Dependency Injection*. Die Netzwerkprotokolle SOAP und REST ergeben oft die typische Auswahl für *Message Exchange* Stil und Format, aber beide Protokolle stützen sich auf XML zur Übertragung der Nachrichten auf der Transport- und Anwendungsschicht. Dies reduziert die Skalierbarkeit eines Systems. Deswegen wählen wir ein reines RCP/IP Protokoll. Außerdem wird hier, da das Messaging Architekturmuster oft in einem Verteilten Softwaresystem verwendet wird, ein reines HTTP Protokoll als Message Exchange Stil und Format herangezogen. Als Service Provider Komponente Container Technologie ist J2EE eine gute Auswahl, aber ein leichtgewichtigeres Container *Jetty* (Servlet) wird in Eclipse RAP ursprünglich angeboten.

Aus obigen entwickelten Architekturentscheidungen kann entschieden werden, die Eclipse RAP für Service Provider Typen und APIs auszuwählen. Die Sitzungsverwaltung (*Session Management*) bildet eine wichtige Technik für eine Netzwerk-Applikation. Sie bietet einen semi-permanenten interaktiven Informationsaustausch, auch als Dialog bzw. Gespräch zwischen mehreren Rechnern über das Netzwerk. Es gibt normalerweise zwei Sitzungsverwaltungstypen – der erste ist die Browser-Sitzungsverwaltung oder *Cookie* in Client, der zweite die Web Server Sitzungsverwaltung. Davon wählen wir die Servlet HTTP- Sitzungsverwaltung aus. Diese Sitzungsverwaltung wird basierend auf *Cookies* oder *URL-Rewriting* enwickelt.

Damit bietet es eine Möglichkeit, eine Anfrage mit mehr als einer Seite von einem Benutzer zu identifizieren. Als eine normale Eclipse *Plug-In*-Applikation verwendet ADvISE das RWT Widgets als UI Toolkit, aber RWT kann leider nicht in der Web-Umgebung verwendet werden. Darum bietet Eclipse RAP einen eigenen Lösungsansatz, nämlich SWT statt RWT. Wir konnten ermitteln, dass SWT in dem *Jetty* (Servlet) Container auch gut arbeiten kann. Außerdem haben wir die *JFace* Technologie in der Praxis verwendet. Abbildung 18 illustriert Phase 3.

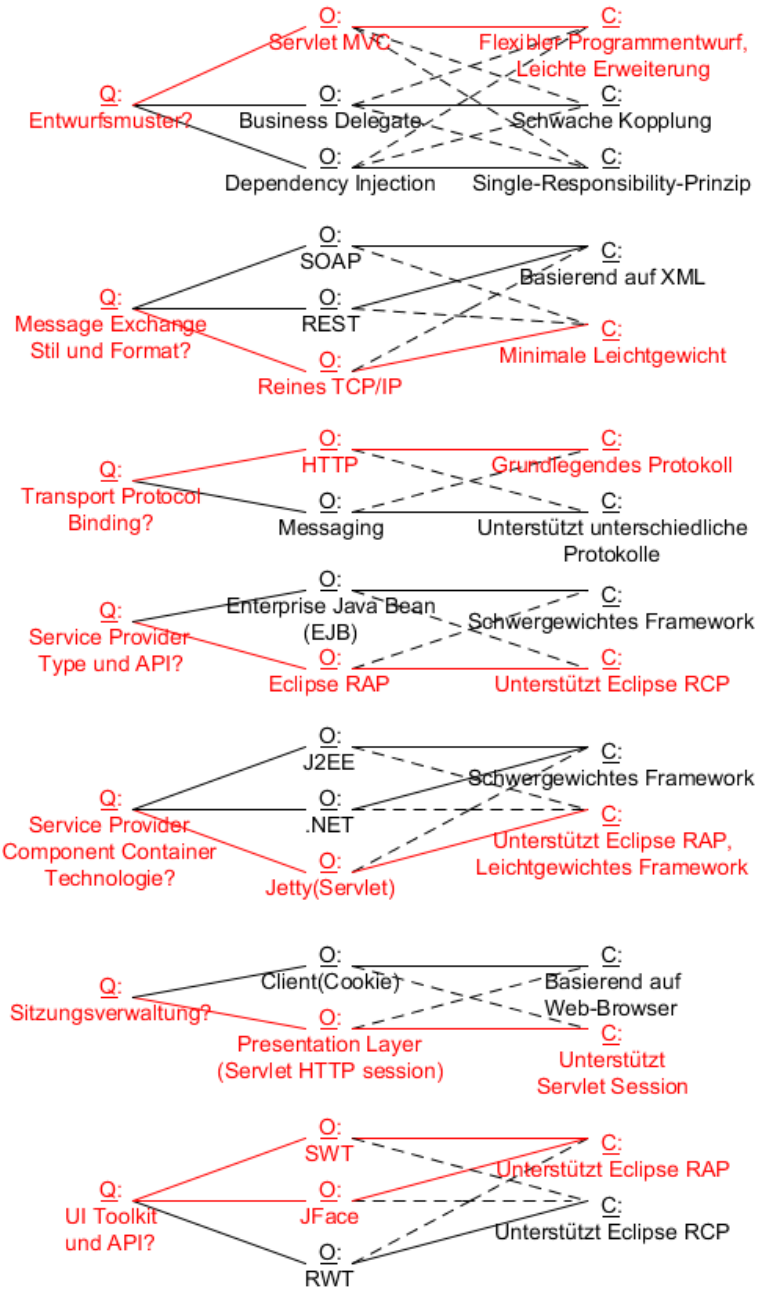

Abbildung 18: QOC Darstellung (Phase 3)

Phase 4: Entscheidungen auf der Vermögensebene des Verkäufers und Auswahl der Implementierung, Bereitstellung und Testmuster

In dem Projekt WebADvISE haben wir zwar keinen bestimmten Käufer und Verkäufer, aber es gibt ein grundsätzliches Prinzip während der Entwicklung des Projekts, um im Interesse (z. B. Zeitkosten, technische Reserven) eines „Verkäufers“ zu bleiben. Dieses Prinzip besteht darin, die Wiederverwendbarkeit des Codes zu erhöhen und gleichzeitig eine unnötige Doppelarbeit zu vermeiden. Aus diesem Grund haben wir die Java-basierte Modulplattform OSGI & Eclipse RAP als die erste wichtigste Architekturentscheidung herangezogen. Die OSGI & Eclipse RAP Plattform bietet lose gekoppelte Module, und zwar Bundles, an. Durch eine bestimmte Menge dieser Bundles können die Softwareapplikationen wiederverwendet werden. Außerdem ist Eclipse RAP ein Framework für die modulare Business-Applikation, auf die von verschiedenen Arten sowie Web-Browser, Rich Clients und mobilen Geräten zugegriffen werden kann. Eclipse RAP bietet eine leistungsstarke Multiplattform-Widget Toolkit mittels RWT-API und angewandte UI-Toolkit JFace an. Diese RWT-APIs erlaubt Softwareentwicklern, pure Java Applikationen und wiederverwendbare Codes auf verschiedenen Plattformen zu entwickeln, so dass diese Applikationen unabhängig von der Client-Plattform sein können. Die Eclipse RAP Applikationen können auch auf jedem Servlet-Container eingesetzt werden und über die HTTP-Protokolle zwischen Server und Clients kommuniziert werden. Die Eclipse RAP Plattform bietet eine OSGi-Integration an, damit eine hardwareunabhängige dynamische Softwareplattform oder Applikationen und deren Dienste per Bundle oder Service modularisiert, verwaltet und bereitgestellt werden können. Diese Begründungen der Verwendung der OSGI & Eclipse Rap Plattform und deren verbundenen Architekturmustern und Technologien sind offensichtlich rationale Architektur- oder Technologie-Entscheidungen für den „Verkäufer“ des WebADvISE.

3.2.2 Gebrauchstechnologien

Als Forschungsergebnisse des Kapitels 3.2.1 fassen wir die Gebrauchstechnologien von WebADvISE und die Architekturentscheidungen bzw. ADDs in dem Architekturprozess von WebADvISE zusammen. Da das Muster-basierte ADDs Tool ADvISE als ein Eclipse Plug-in entwickelt wird, werden die Programmiersprache Java und Software Plattform OSGI (Eclipse RAP) als erste Auswahl der Architekturentscheidungen für WebADvISE verwendet. Gleichzeitig wird der Jetty (Servlet) Container als *Service Provider Component Container Technologie* eingesetzt. Hier bietet uns Java und Servlet *Message Exchange Pattern* Request-Response und die Servlet HTTP-Sitzungsverwaltung an. Und die Kommunikationen zwischen Server und Client werden durch ein reines TCP/IP Protokoll in WebADvSIE transportiert. Wir wählen das Servlet MVC Entwurfsmuster als globalen Architekturentwurf aus. Für das UI Design werden JFace und SWT Widgets verwendet. Tabelle 9 zeigt alle kurze Beschreibungen der ADDs und die Gebrauchstechnologien von WebADvISE.

Kurze Beschreibung der ADD	**Technologie**
Programmiersprache	Java
Applikationsplattform	OSGI & Eclipse RAP
Message Exchange Muster	Request-Response
Presentation Layer Binding	Thin Client
Sitzungszustand	Server Session
Allgemeines Entwurfsmuster	Servlet MVC
Message Exchange Stil und Format	Reines TCP/IP
Transport Protocol Binding	HTTP
Service Provider Type und API	Eclipse RAP
Service Provider Component Container Technologie	Jetty (Servlet)
Sitzungsverwaltung	Servlet HTTP Session
UI Toolkit und API	SWT/JFace
Datenbank	MySQL
IDE	Eclipse 4.2
Modellierungsframework	EMF

Tabelle 9: Die Gebrauchstechnologien von WebADvISE

3.2.3 WebADvISE als Plug-Ins von Eclipse RAP

Mit Bezug auf die Komponentenstruktur der Eclipse können die

Funktionalitätsmodule von WebADvISE als einige Plug-Ins in der Eclipse RAP Plattform gesehen werden. Das rechtsseitige Teil der Abbildung 19 beschreibt die Komponentenstruktur der Eclipse RAP Plattform, und das linksseitige sieben wichtige Plug-Ins in WebADvISE. Die entsprechenden Arbeitsbereiche dieser Plug-Ins bzw. Funktionalitätsmodule, ausgenommen das ADD Model, können auf einer typischen Architekturentscheidungswebseite in WebADvISE gefunden werden, siehe Abbildung 20. Im Folgenden besprechen wir kurz die sieben Plug-Ins. Die auf diese Funktionalitätsmodule bezogenen Funktionalitäten besprechen wir im Kapitel 3.3.

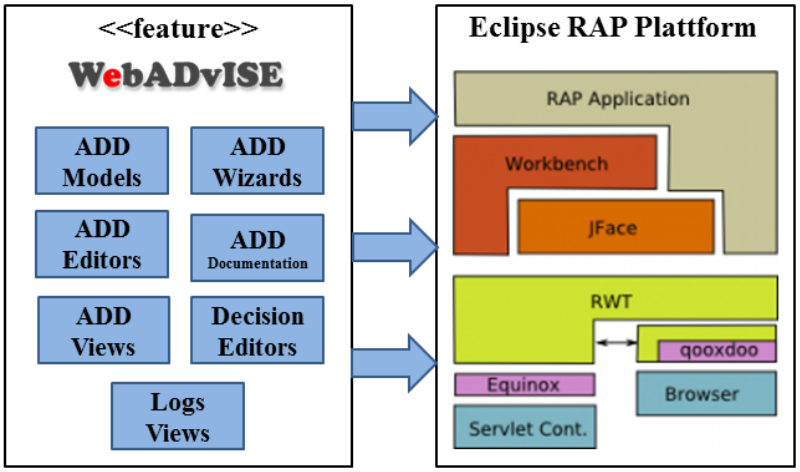

Abbildung 19: Betriebsumgebung der WebADvISE [vgl. RAP, 2014]

- **ADD Models:** Dieses Plug-In enthält die Definition der Meta-Modelle zur Modellierung wiederverwendbarer Architekturentscheidungen auf Basis der QOC Methode und die Fragebögen zur Unterstützung der Herstellung der Architekturentscheidungen. Es bietet die Erstellung, Änderung und Persistenz der ADMs an.

- **ADD Wizards:** Dieses Plug-In bietet die Web-Widgets zum Erstellen der WebADvISE Projekten und Modelle.

- **ADD Documentation:** Die Dokumentation der Architekturentscheidungen und deren Rationale können automatisch aus den beantworteten Fragebögen generiert werden. Schließlich werden diese Architekturentscheidungen gespeichert.

- **ADD Editors:** Dieses Plug-In stellt zwei Arten von Editoren zur Verfügung. Eine ist für die Bearbeitung bzw. Editierung der ADMs und die andere ist für die Bearbeitung bzw. Editierung der Fragebögen aus diesen generierten Modellen.

- **Decision Editors:** Eine entwickelte Architekturentscheidung kann in diesem Editor im Details bearbeitet und gespeichert werden.

- **ADD Views:** Es bietet zwei benutzerdefinierte ADD-Schichten. Eine ist für die Verwaltung der Ressourcen (ADM Projekten) und die andere für die Visualisierung der Entscheidungen zuständig, um die Entscheidungspunkte während der Herstellung der Architekturentscheidungen zu öffnen. Dieses Plug-in enthält auch einige Benutzermanipulationen (mit dem rechten Mausklick) zur Erstellung, zum Lösen oder zur Umbenennung der ADM Projekte.

- **Log Views:** Es bietet die Historie (Logs) der Aktivitäten eines Benutzers. Wenn ein Benutzer (nur Mitarbeiter in einem bestimmten Team) ein oder mehrere ADM Projekte ändert, wird diese Aktivität als ein Log automatisch in dieser Log-Schicht für alle anderen Mitarbeiter sichtbar sein.

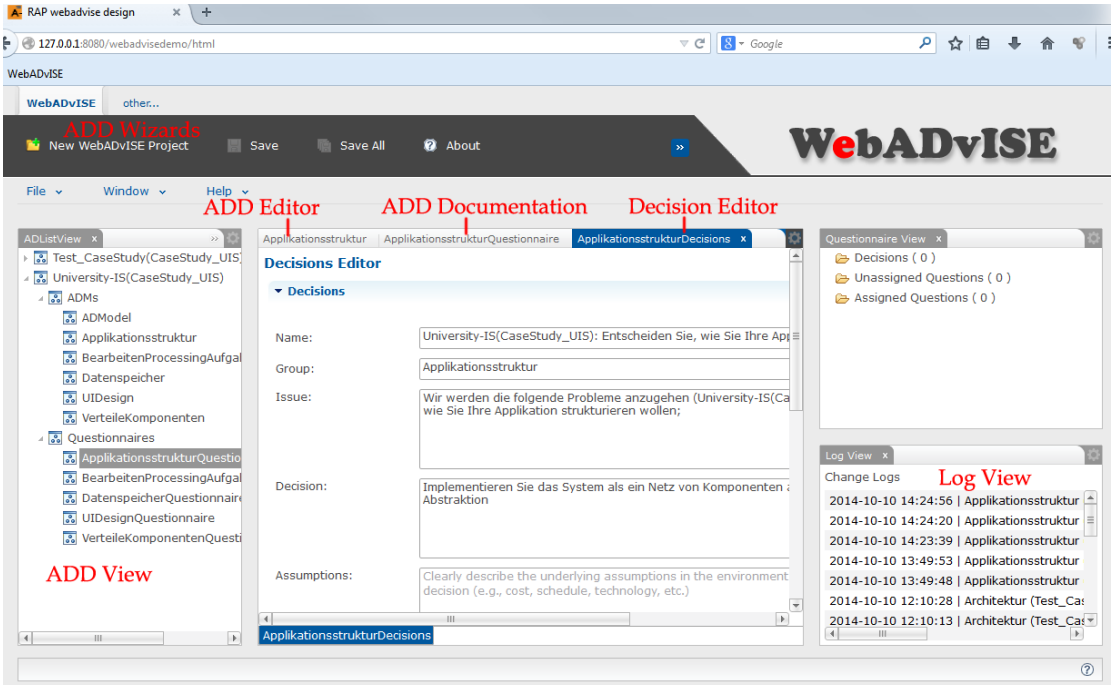

Abbildung 20: Ein typischer Editier-Bereich von WebADvISE

3.2.4 MVC Entwurfsmuster für UI Design

WebADvISE verwendet die MVC Entwurfsmuster für das UI Design. Wir definieren vier JavaBeans, *User*, *Colleague*, *Team* und *Editor,* als das *Model* Teil der MVC Struktur für WebADvISE. Diese JaveBeans Klassen können von bezogenen DAOs im *Controller* Teil von MVC kontrolliert werden. Und alle Informationen dieser JavaBeans können dadurch in der Datenbank gespeichert werden. In dem *Controller*-Teil gibt es noch einige Servlet-Klassen, die den verbundenen HTML-File in dem *View*-Teil von MVC gemäß der definierten Bedienlogik kontrollieren. Die Benutzer können die Website von WebADvISE durch einen Request-Response-Mechanismus besuchen. Abbildung 21 zeigt diese MVC-Struktur.

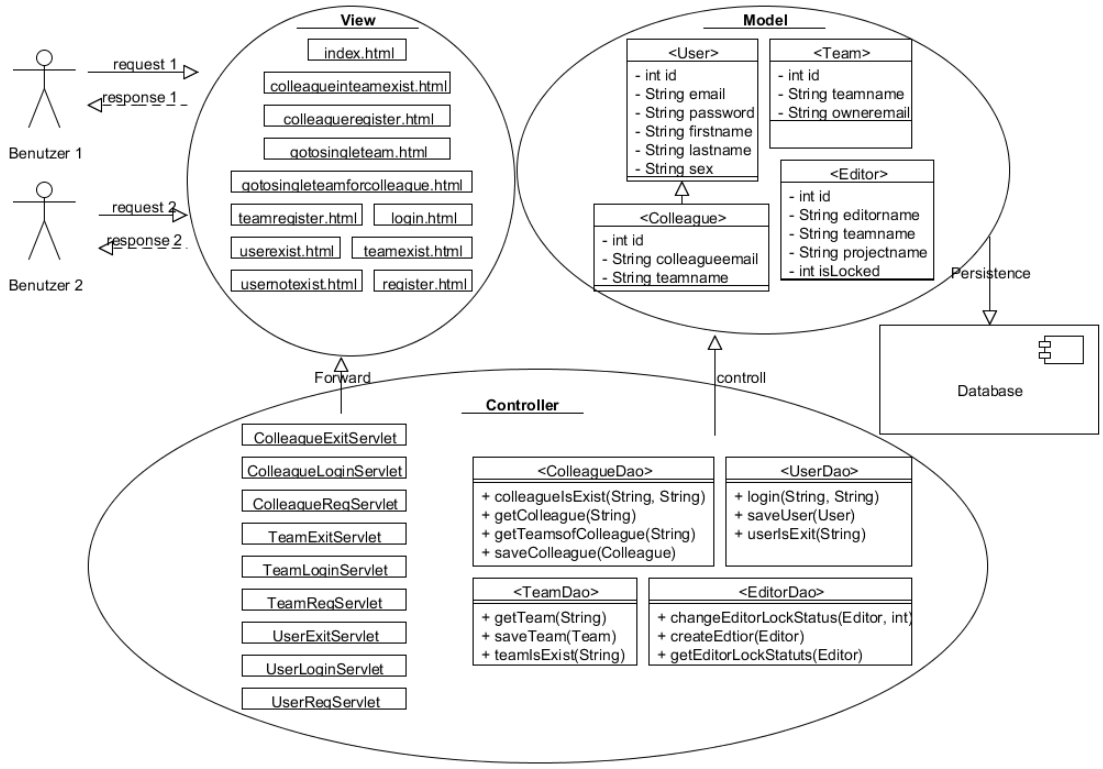

Abbildung 21: MVC Struktur für UI Design von WebADvISE

3.3 Funktionalitäten und Handbuch

Da die Zielgruppe von WebADvISE Softwarearchitekten oder Softwareentwickler sind, verwenden wir zuerst ein *Use-Case*-Diagramm, um eine professionellere und

intuitivere Darstellung aller wichtigen Aktivitäten der Benutzer von WebADvISE zur Verfügung zu stellen. Abbildung 22 stellt diese *Use-Cases* dar. Es gibt zwei Arten von Endbenutzern in WebADvISE, die normalerweise Softwarearchitekten sind. Einer von ihnen ist der Gruppenbesitzer (*Team owner*) und der andere ist Mitarbeiter (*Colleague*), der als ein Beteiligter in einer bestimmten Forschungsgruppe gelten kann. Natürlich kann sich ein Benutzer auch an keiner Gruppe beteiligen und sich mit eigenen ADM Projekten oder andren Projekten beschäftigen. Diese Aktivitäten aller Benutzer von WebADvISE können gemäß unterschiedlichen Funktionalitäten in vier Gruppen, grundsätzliche Funktionalitäten, Verwaltung eines ADMs, Bearbeitung eines ADMs und Erfassung einer Architekurentscheidung, eingeteilt werden. Und jede Gruppe dieser *Use-Cases* wird mit einer Farbe in Abbildung 22 gedruckt. In folgenden Kapiteln 3.3.1 bis 3.3.4 werden diese wichtige Aktivitäten der vier Gruppen und deren entsprechende Funktionalitäten, die von WebADvISE angeboten werden, im Detail besprochen. Das Kapitel 3.3.5 beschreibt andere Kernfunktionalitäten, Kollaboration und Log View von WebADvISE.

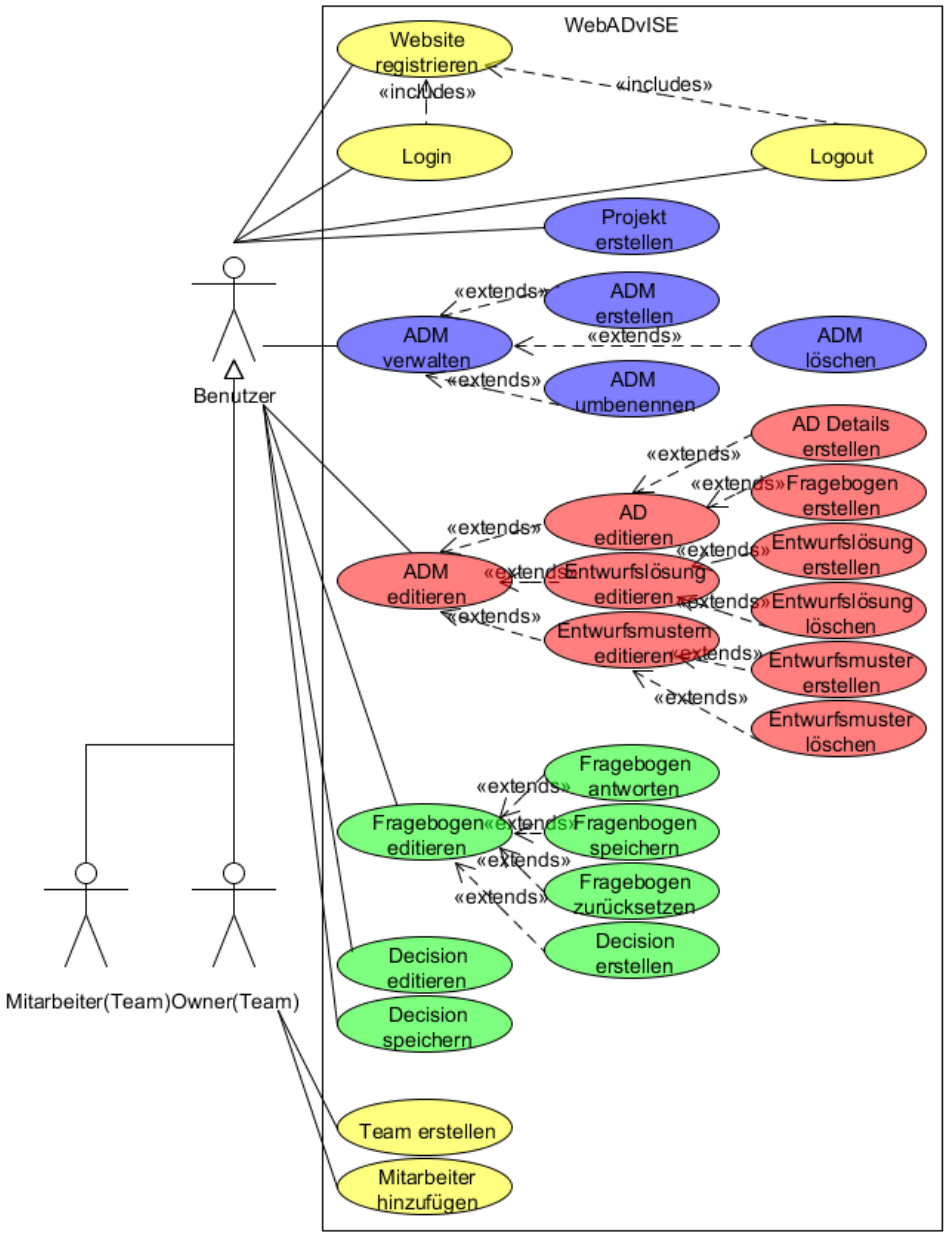

Abbildung 22: Use-Case Diagramm zur Beschreibung aller möglichen Use-Cases der Benutzer

3.3.1 Grundsätzliche Funktionalitäten

Die gelben *Use-Cases* der Abbildung 22 und deren gebundene Aktivitäten stellen die grundsätzlichen Funktionalitäten von WebADvISE dar. Wenn ein Benutzer noch kein Benutzerkonto hat, muss er sich zuerst in einem Formular registrieren. Hier muss er seine E-Mail-Adresse, das Password, den Vornamen, seinen Familiennamen sowie das Geschlecht angeben. Ein registrierter Benutzer legt dann seine registrierte E-Mail-Adresse als „*username*“ und sein registriertes Password als „*password*“ fest, um das

Login in WebADvISE erfolgreich zu ermöglichen. Jeder registrierte Benutzer kann sich eine oder mehrere Forschungsgruppen mit dem Gruppennamen als Gruppenbesitzer dieser Gruppen erstellen. Nur der Gruppenbesitzer einer Gruppe kann einen neuen Mitarbeiter in seine Gruppe hinzufügen, dafür soll er eine E-Mail-Adresse von diesem neuen Mitarbeiter als die identifizierende Information angeben. Nach dem erfolgreichen Login hat ein Benutzer eine Alternative. Erstens kann er zu einem eigenen Projekt gehen, das nur von diesem Benutzer bearbeitet wird. Zweitens kann er eine Gruppe aufsuchen, die diesen Benutzer enthält, unabhängig davon, ob er ein Teambesitzer oder Mitarbeiter ist.

3.3.2 Verwaltung eines ADMs

Die blauen Use-Cases der Abbildung 22 und deren gebundene Aktivitäten stellen dar, wie ein Benutzer von WebADvISE ein ADM verwalten kann. Diese Gruppe der Use-Cases bezieht sich auf das Funktionalitätsmodul ADD Wizards in WebADvISE, siehe Kapitel 3.2.3. Als erster Schritt kann ein eingeloggter Benutzer ein neues WebADvISE Projekt in folgenden Schritten erstellen: „New WebADvISE Project“ -> „Do you want to create a new AD Project?“ Ja oder Nein -> „Please enter your AD Project name:“ -> Finish. Dann wird dieses neue Projekt in ADListView sichtbar sein. Die möglichen Aktionen in ADListView sind:

- Erstellen eines neuen Architekturentscheidungsmodells
- Erstellen einer neuen Gruppe für die Organisation des Architekturentscheidungsmodells
- Umbenennen Gruppe/ Architekturentscheidungsmodell /Fragebogen
- Löschen Gruppe/ Architekturentscheidungsmodell /Fragebogen
- Aktualisieren aller Projekte
- Validieren Architekturentscheidungsmodell
- Generieren des Fragebogens von dem Architekturentscheidungsmodell

3.3.3 Bearbeitung eines ADMs

Die Herstellung eines wiederverwendbaren Architekturmuster-basierten ADMs bildet eine wichtigste Funktion von WebADvISE. Dieser Herstellungsprozess kann in drei Phasen aufgeteilt werden, und zwar in die Herstellung eines Entwurfsmusters, einer Entwurfslösung und eines ADs. Die rote *Use-Cases* der Abbildung 22 und deren gebundene Aktivitäten stellen diese drei Phasen dar. Die Gruppe dieser *Use-Cases* bezieht sich auf die Funktionalitätsmodule des ADD Editors in WebADvISE, siehe Kapitel 3.2.3. Im Folgenden besprechen wir die drei Phasen im Detail.

Herstellung des Entwurfsmusters

Gehen Sie auf den Tab „Design Patterns“ und drücken Sie auf „Add“. Ein Muster mit dem Namen „NewPattern“ wird automatisch erstellt. Sie lassen sich den gewünschten Namen und die Beschreibung dieses Patterns geben. Sie dürfen natürlich die Namen und Beschreibungen der alten Muster auf der linken Seite ändern und ein oder mehrere Solutions mit der Taste „Delete“ löschen, siehe Abbildung 23.

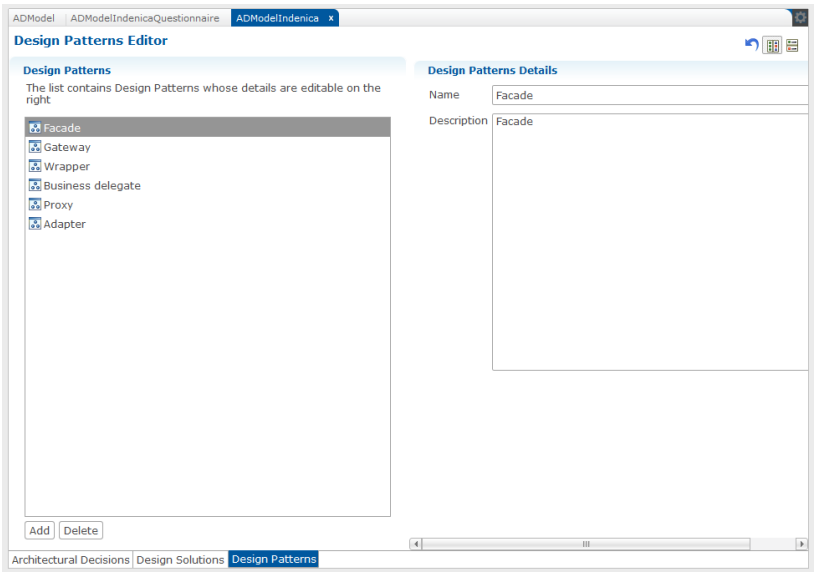

Abbildung 23: Design Patterns Editor

Herstellung einer Entwurfslösung

Gehen Sie auf den Tab „Design Solutions“ und drücken Sie auf „Add“. Eine Solution

mit dem Namen „NewSolution“ wird automatisch erstellt. Sie lassen sich den gewünschten Namen und die Beschreibung dieses Musters geben. Sie dürfen natürlich die Namen und Beschreibungen der alten Solutions auf der linken Seite ändern, und ein oder mehrere Muster mit der Taste „Delete“ löschen. Um die bezogenen Muster hinzufügen oder zu entfernen, können Sie die Taste „+“ und „-“ drücken, siehe Abbildung 24 und 25.

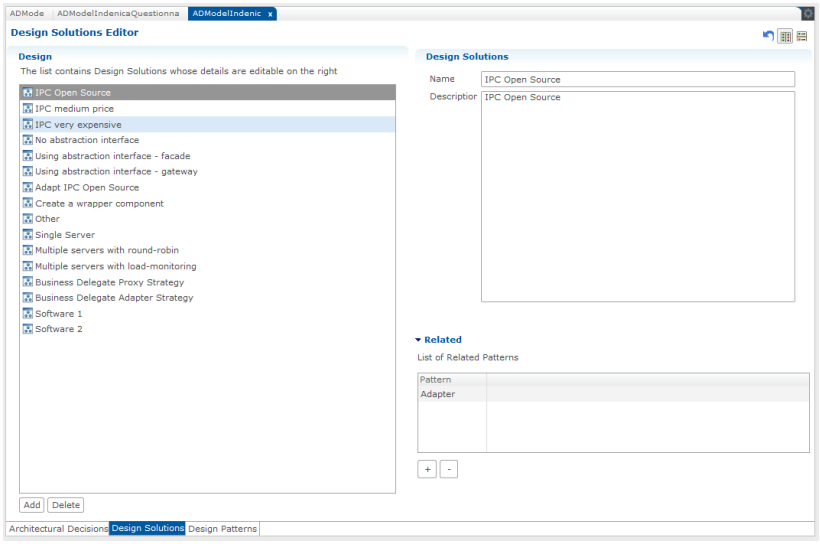

Abbildung 24: Design Solutions Editor

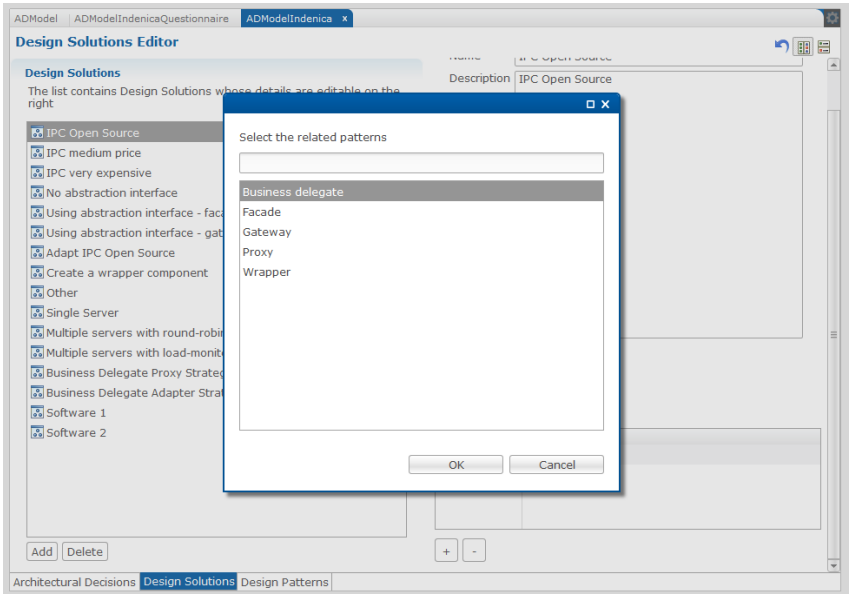

Abbildung 25: Die bezogenen Muster hinzufügen

Herstellung eines ADs

Gehen Sie auf den Tab „Architectural Decisions“ und drücken Sie auf „Add“. Eine

Entscheidung (*Decision*) wird mit dem Namen „NewAD“ automatisch erstellt. Sie lassen sich die folgenden Informationen für diese Entscheidung geben:

- Name: Name der Entscheidung
- Level: Design-Ebene für diese Entscheidung
- Scope: Beschreibung der Entscheidung
- Status: Status des Decision (In Progress, Finished, Not Reviewed)
- IsFirst: Die erste Entscheidung
- Order: Gibt die Reihenfolge der Entscheidung an, wie sie in der generierten Fragebogen erscheinen

Jede Entscheidung enthält eine Liste von Fragen. Um die bezogenen Fragen hinzuzufügen oder zu entfernen, können Sie die Taste „+“ und „-“ drücken. Sie lassen sich eine Frage mit einem Doppelklick editieren. Um eine Frage als erste Frage zu setzen, können Sie das Kontrollkästchen „First“ auswählen, siehe Abbildung 26.

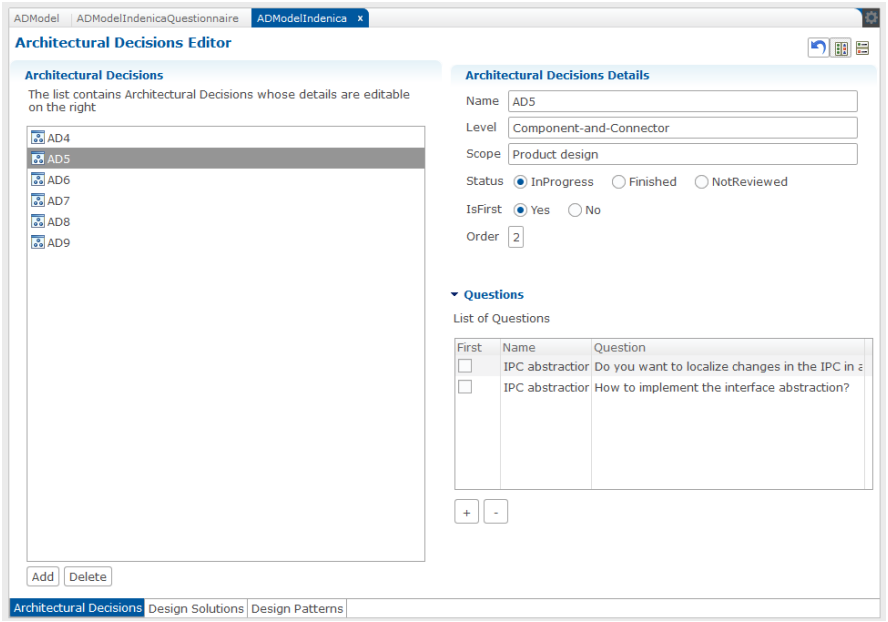

Abbildung 26: Architectural Decisions Editor

Sie können entweder eine Frage mit Antwort oder eine Frage mit Optionen hinzufügen, siehe Abbildung 27. Eine Frage mit einer erwarteten Freitext-Antwort enthält die folgende Informationen, siehe Abbildung 28.:

- Name: Symbolischer Name der Frage
- Question: Ist-Frage

- Mandatory-opritonal: Überprüfen, ob die Frage obligatorisch oder optional ist
- Next Decisions: Hinzufügen/Löschen der weiterführenden Entscheidungen
- Next Questions: Hinzufügen/Löschen der weiterführenden Fragen

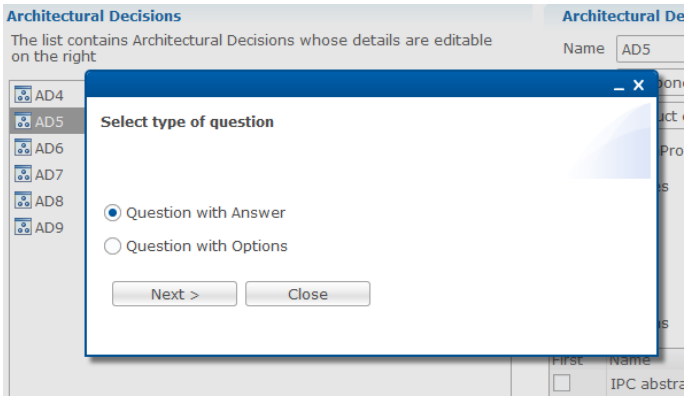

Abbildung 27: Editieren Fragen (Questions)

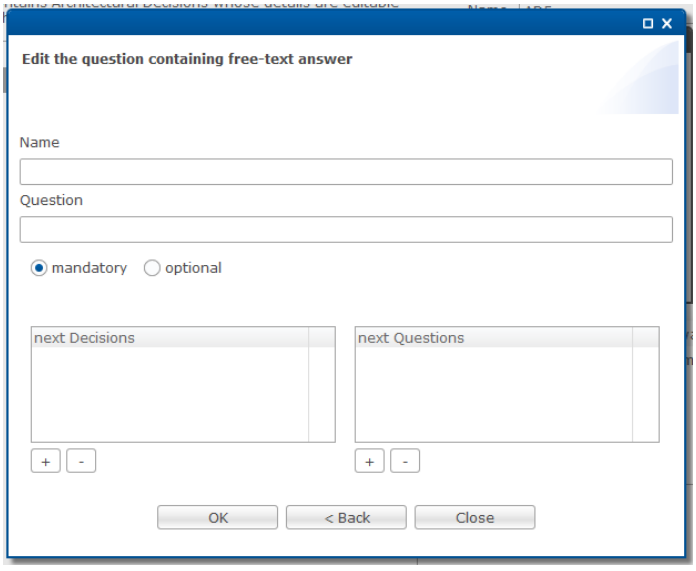

Abbildung 28: Eine Frage mit einer erwarteten Freitext-Antwort

Eine Frage mit den erwarteten alternativen Optionen enthält die folgenden Informationen, siehe Abbildung 29:

- Name: Symbolischer Name der Frage
- Question: Ist-Frage
- Mandatory/optional: Überprüfen, ob die Frage obligatorisch oder optional ist.
- Options: Drücken Sie die Taste „+“ oder „-“ zum Hinzufügen oder Löschen der Options

Nachdem eine Option ausgewählt wird, werden folgende Informationen aktiviert, siehe Abbildung 29:

- Solution: Wählen Sie eine Lösung, die der ausgewählten Option entspricht (falls vorhanden)
- Next Decisions: Hinzufügen/Löschen der weiterführenden Entscheidungen
- Next Questions: Hinzufügen/Löschen der weiterführenden Fragen
- Forces options: Hinzufügen/Löschen der Optionen, die durch die Auswahl dieser Option erzwungen sind
- Incompatible Options: Hinzufügen/Löschen der Optionen, die mit der Auswahl dieser Option inkompatibel sind

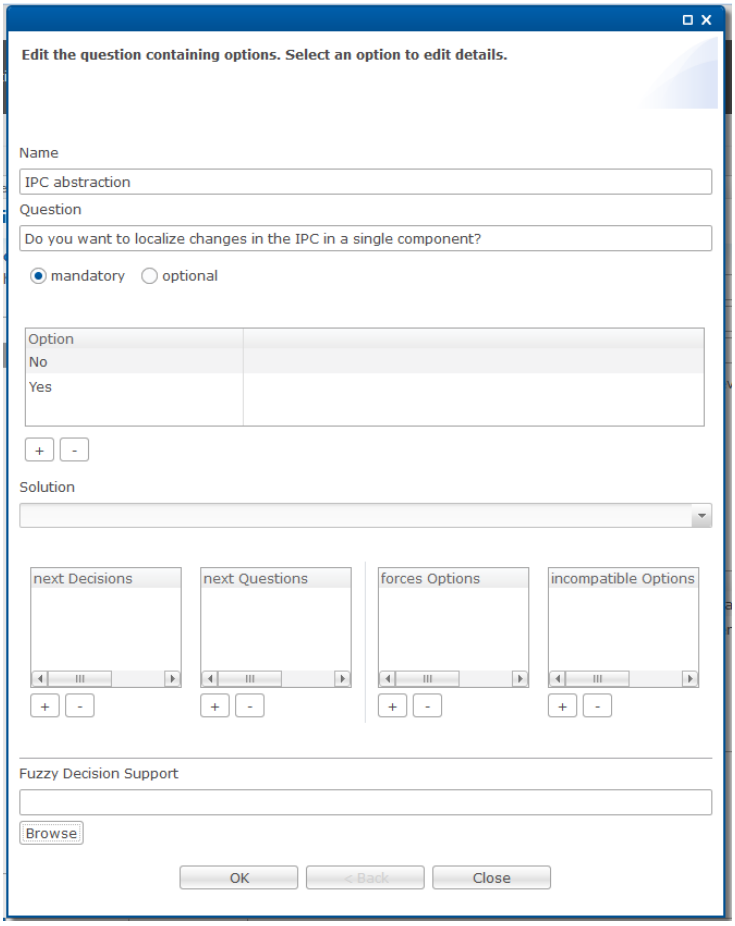

Abbildung 29: Eine Frage mit den erwarteten alternativen Optionen

3.3.4 Erfassung einer Architekturentscheidung

Die Herstellung und Persistenz eines wiederverwendbaren Architekturmuster-

basierten ADMs ist eine andere Funktion von WebADvISE. Die Herstellung eines ADs und die Bearbeitung dieses erstellten ADs sind die zwei Phasen für die Erfassung einer Architekturentscheidung. Die grünen *Use-Cases* der Abbildung 22 und deren gebundene Aktivitäten stellen diese zwei Phasen dar. Die Gruppe dieser *Use-Cases* bezieht sich auf die Funktionalitätsmodule des *Decision* Editors in WebADvISE, siehe Kapitel 3.2.3. Im Folgenden besprechen wir die zwei Phasen im Detail.

Herstellung eines ADs

Gehen Sie auf ADListView und machen Sie einen Rechtsklick auf einem ADModell, z. B. ist der Name dieses ADModells „Applikationsstruktur“, siehe linkes Bild von Abbildung 30. Drücken Sie auf „Generate Questionnaire“, der Fragebogen-File wird mit dem Namen „ApplikationsstrukturQuestionnaire“ in der Gruppe „questions“ automatisch generiert, siehe rechtes Bild von Abbildung 30.

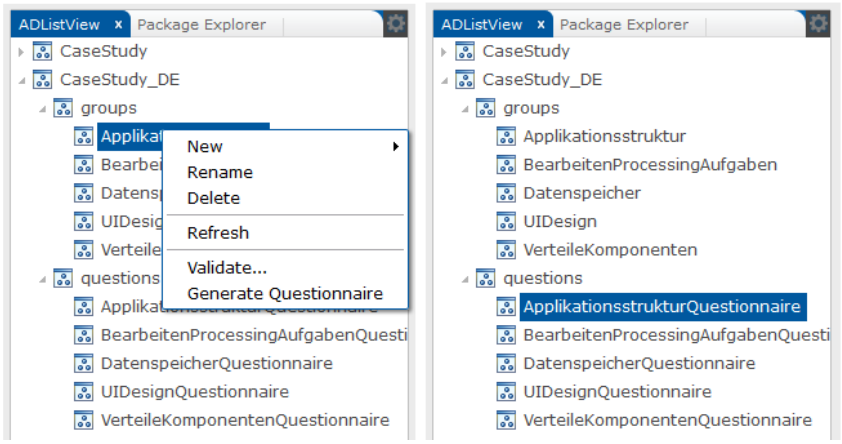

Abbildung 30: Generieren eines Fragebogen-Files

Doppelklicken Sie auf den Fragebogen-File „ApplikationsstrukturQuestionnaire“, so wird automatisch ein Fragebogen-Editor geöffnet, siehe Abbildung 31. Für die Beantwortung der Fragebogen finden sich einige mögliche Klicks auf den verfügbaren Optionen für die Fragen sowie für das Ausfüllen mit den erforderlichen Informationen. Wann auch immer Sie eine Option ausgewählt haben, werden Sie in die nächsten Fragen für die Folgeentscheidungen geführt, um Ihre erwarteten Entwurfsentscheidungen zu betrachten. Die grau hinterlegten Entscheidungen und Fragen können zu diesem Zeitpunkt nicht beantwortet werden. Nachdem Sie die

Optionen aller Fragen auswählen, klicken Sie den Button „Save“, hiermit werden die Entscheidungen für die ausgewählten Optionen und der Status der Fragen synchron in Questionnaire View festgelegt. Die Fragen werden gekennzeichnet als (M) oder (O), sie sind „mandatory“ oder „optional“. Die Änderung einer vorherigen Option kann zu ungültig erklärten nachfolgenden Fragen und Optionen führen. In diesem Fall werden die sich anschließenden Fragen und Optionen gemäß den neuen Einschränkungen von dem jeweiligen ADM neu berechnet. Die geänderten Fragen müssen noch einmal beantwortet werden.

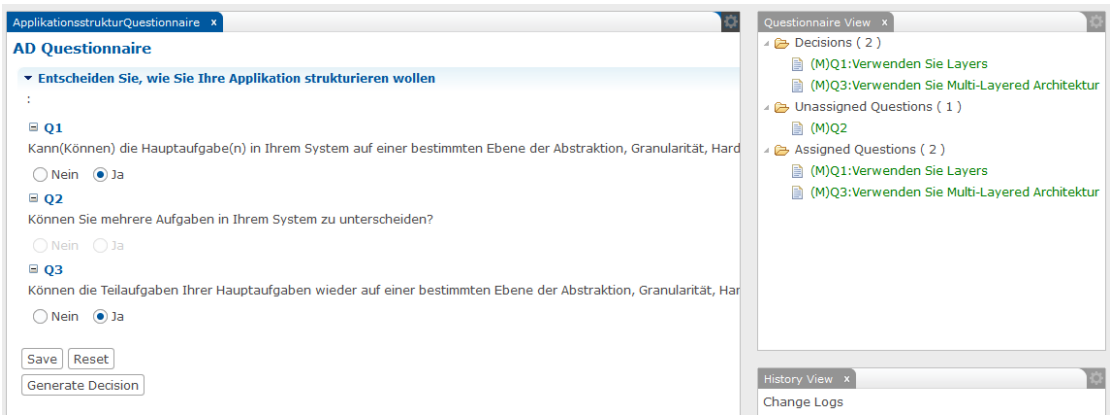

Abbildung 31: Fragebogen-Editor

Um einen Fragebogen zu speichern, klicken Sie den Button „Save“. Dann können Sie diesen Fragebogen mit den gespeicherten Ergebnissen wieder öffnen und neu editieren. Klicken Sie den Button „Reset“, um alle Antworten zu verwerfen und diesen Fragebogen neu zu starten.

Bearbeitung eines erstellten ADs

Nach der Bearbeitung der Fragebogen wird ein Architekturentscheidungseditor generiert, siehe Abbildung 47. Dort können Sie eine Architekturentscheidung editieren und speichern. Die in diesem Architekturentscheidungseditor vorhandenen Inhalte stellen eine Implementierung des Dokumentation-Template der Architekturentscheidung von Tyree dar, siehe Kapitel 2.3.1. Die Definitionen und Bedeutungen dieser Inhalte haben wir in dem Kapitel 2.2.3 bereits besprochen, siehe Tabelle 1. Nach der Bearbeitung einer Architekturentscheidung können die Ergebnisse

mit dem Button „Save“ online gespeichert werden.

3.4 Kollaborative Arbeit

Bei WebADvISE handelt es sich um ein kollaborativeres Tool. Es bietet einen Mechanismus der kollaborativen Arbeit für alle unterschiedlichen Mitglieder in einer bestimmten Forschungsgruppe an, um die jeweiligen ADMs oder Architekturentscheidungen mit den anderen Mitgliedern austauschen zu können. Dabei ermöglicht es, die ADMs oder Architekturentscheidungen verschiedener Mitglieder in verschiedenen Orten der Welt zusammen zu bearbeiten.

Wie im Kapitel 3.3.1 besprochen, besteht eine Forschungsgruppe aus einem Gruppenbesitzer und mehreren Mitarbeitern. Aber diese zwei Benutzertypen verfügen über unterschiedliche Berechtigungen, obwohl sich alle Mitglieder einer solchen Forschungsgruppe ein Projekt oder mehrere Projekte in dieser Gruppe erstellen, umbenennen, löschen und bearbeiten können. Aber nur der Gruppenbesitzer darf einen neuen Mitarbeiter in seine Gruppe hinzufügen. Jedes Mitglied einer Forschungsgruppe kann ein WebADvISE Projekt und einige ADMs und die gleiche Anzahl darauf bezogener Fragebogen erstellen, siehe Abbildung 20. Der linke Teil dieser Abbildung zeigt, dass zwei WebADvISE Projekte mit den Namen „CaseStudy“ und „CaseStudy_DE“ erstellt werden. Das Projekt „CaseStudy_DE“ hat fünf ADMs und fünf darauf bezogene Fragebogen. Alle Mitglieder dürfen gleichzeitig auf irgendeinen Editor dieser ADMs oder Fragebogen zugreifen und kollaborativ arbeiten. Dafür definiert WebADvISE sich seine eigenen Arbeitsregeln für die Kollaboration.

3.4.1 Konflikte der Ressourcen des Editors

Um diese Funktionalität für die Kollaboration von WebADvISE erfolgreich zu implementieren, sollten wir zuerst deutlich machen, worin die Schwierigkeit für diese Funktionalität liegt. Es handelt sich um das Auftreten von Konflikten, wenn mehrere

Nutzer zeitgleich (synchron) einen Lese- und Schreibzugriff auf die Ressource eines Editors von WebADvISE erhalten. Die Konflikte der Ressourcen können in allen Systemen auftreten, z. B. in einem Betriebssystem und Verteilten System. Es gibt normalerweise zwei verschiedene Arten an Konflikten der Ressourcen. Abbildung 32 zeigt die zwei Konflikte. Einer umfasst Lese-Schreib-Konflikte, wenn ein Mitglied (M1) lesenden Zugriff auf eine Ressource (Editor) erhält und ein anderes Mitglied (M2) zeitgleich einen schreibenden Zugriff auf genau diese Ressource (Editor) hat. Der andere beinhaltet Schreib-Schreib-Konflikte, wenn zwei Mitglieder (M1 und M2) zeitgleich einen schreibenden Zugriff auf die Ressource (Editor) bekommen [vgl. Silberschatz et al., 2009, S.263].

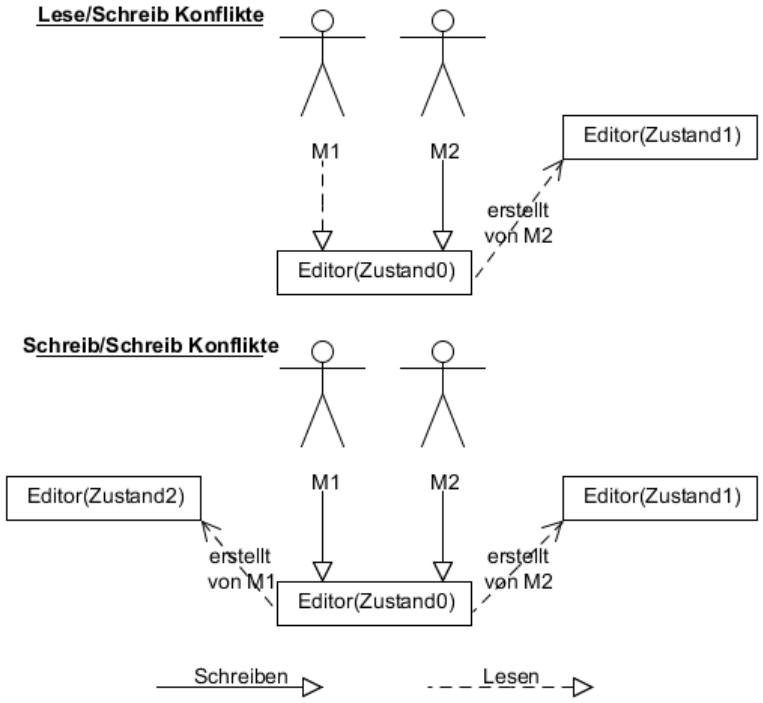

Abbildung 32: Lese-Schreib-Konflikte und Schreib-Schreib-Konflikte

Das heißt, während der ursprünglichen Architekturentscheidung von WebADvISE muss ein Lösungsansatz festgelegt werden, um die Konflikte der Ressourcen zu beseitigen, zu vermeiden oder direkt zu übergehen. Ein typischer Lösungsansatz der Konflitbeseitigung ist *Mutual Exclusion Lock* [vgl. Silberschatz et al., 2009, S. 228], damit kann ein Konflikt der Ressource beseitigt werden. Der User-Lösungsansatz stellt keine Methode zur Konfliktbeseitigung dar, sondern dient der

Konfliktvermeidung oder -übergehung. Dies wird im Kapitel 3.4.2 im Detail beschrieben.

3.4.2 Atomisierung der Änderungszustände und Rollback

WebADvISE implementiert einen Mechanismus, um die Konflikte der Ressourcen eines Editors zu übergehen. Anhand der Abbildung 32 lässt sich Folgendes erkennen: Unabhängig davon, ob Lese-Schreib-Konflikte oder Schreib-Schreib-Konflikte auftreten, erhalten wir zwei oder drei Änderungszustände von einer Ressource bzw. Versionen von einem Editor. Falls wir alle möglichen erstellten Änderungszustände in WebADvISE verzeichnen und noch einen Rollback-Mechanismus für verschiedene Mitglieder anbieten, können die Mitglieder selbst entscheiden, welchen Änderungszustand bzw. welche Version des Editors sie wirklich wieder benötigen. Dann können wir die Konflikte der Ressourcen bzw. den Verlust aus der Änderung eines Editors vermeiden oder übergehen.

Aus diesem Grund wird der Mechanismus „Atomisierung der Änderungszustände und Rollback“ zur Vermeidung der Konflikte für WebADvISE angeführt. Als Atomisierung wird die Zerkleinerung der Änderungszustände einer Ressource bzw. eines Editors in kleinstmögliche Teilchen, möglichst nahe den einzelnen „Atomen“, bezeichnet, um alle Änderungsdetails, die zu Konflikten führen können, zu verzeichnen. Dann lässt WebADvISE die Mitglieder selbst entscheiden, welchen Änderungszustand der Ressource bzw. welche Version des Editors er wieder verwenden möchte (Rollback).

3.4.3 Technische Implementierung und Log View

Um diese Funktionalität für die Kollaboration von WebADvISE aus technischer Sicht zu implementieren, entwickeln wir folgende drei Schritte:

- **1. Schritt:** Wir verwenden die Methoden *org.eclipse.rap.rwt.service.UISession.getHttpSession().getAttribute("team")* und *org.eclipse.rap.rwt.service.UISession.*

getHttpSession().getAttribute('user'), um die Informationen der aktuellen Forschungsgruppe und Benutzer aus HTTP Session zu erfassen und diese zu registrieren.

- **2. Schritt:** Wir verwenden *UI Thread (Threads in RAP)* und die Methoden *display.syncExec(Runnable)* sowie *display.asyncExec(Runnable)*, um zu implementieren, dass verschiedene Mitglieder einen synchronen und asynchronen lesenden oder schreibenden Zugriff auf eine Ressource bzw. einen Editor erhalten.

- **3. Schritt:** Wir speichern die Änderungszustände als Change Logs in Log View und verwenden die *addSelectionListener(new SelectionListener() {...})*. Das ist ein *Selection-Listener* für jedes Log hinzuzufügen. Sobald ein Log ausgewählt wird, wird die darauf bezogene alte Version von diesem ADM oder Fragebogen mit einer entsprechenden Uhrzeit automatisch im Editorbereich geöffnet, siehe Abbildung 49.

Als Rekordmarke einer kollaborativen Arbeit der Mitglieder wird ein Log View entwickelt. Ein „Change Log“ von Log View enthält das Datum, die Uhrzeit, die Namen der geänderten ADM oder Fragebogen, den Projektnamen, Benutzernamen und Benutzertyp der derzeitigen Mitglieder, siehe Abbildung 33.

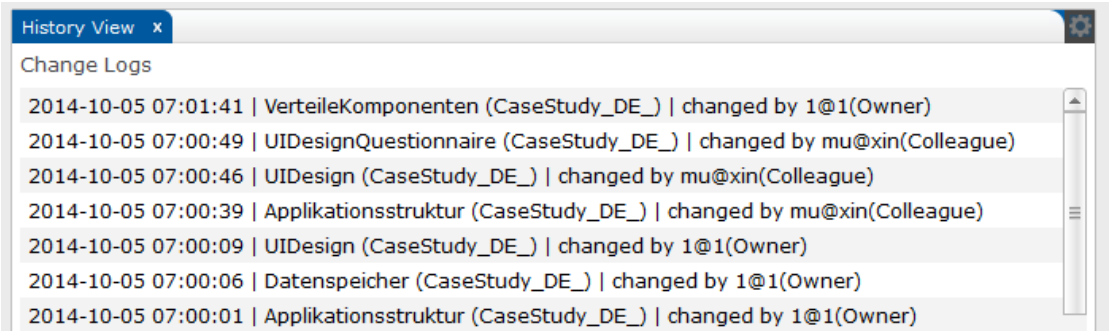

Abbildung 33: Change Logs in Log View

4. Fallstudie mittels WebADvISE

4.1 Grundlagen

In Kapitel 2 wurden die Definition, historische Evolution, Dokumentationsansätze, Dokumentationsmethoden, Modelle und Tools von ADDs besprochen. In Kapitel 3 empfahlen wir die Grundlagen, Implementierungstechnik und das Benutzerhandbuch der Web-basierten ADD Tool WebADvISE. In diesem Kapitel implementieren wir eine Fallstudie bezogen auf das Universitäts-Informationssystem (Universitäts-IS) [UIS, 2013] mittels WebADvISE. Die Dokumentation des Universitäts-IS kommt aus der *Software Architecture Group*, Fakultät für Informatik, Universität Wien. Die Beschreibung und Anforderungen der Universitäts-IS werden im Anhang aufgelistet. Diese Fallstudie enthält drei Schritte. Für den ersten Schritt entwickelten wir fünf wiederverwendbare Muster-basierte ADMs, eine Applikationsstruktur, Verteilte Komponenten, Taskverarbeitung, UI Design und Datenspeicherung, siehe Kapitel 4.2. In dem zweiten Schritt erstellten wir diese fünf entwickelten wiederverwendbaren Muster-basierten ADMs in WebADvISE. In dem dritten Schritt analysierten wir die Beschreibung und Anforderungen des Universitäts-IS und erstellten eine geeignete ADDs für das Universitäts-IS in WebADvISE mittels dieser ADMs, siehe Kapitel 4.4. In Kapitel 4.5 besprechen wir die Ergebnisse dieser Fallstudie und das Kapitel 4.6 stellt die kollaborative Arbeit in der Praxis dar.

4.2 Wiederverwendbare Muster-basierte ADMs

Wir entwickelten fünf wiederverwendbare Muster-basierte ADMs für eine generelle Webapplikation: Applikationsstruktur, Verteilte Komponenten, Taskverarbeitung, UI Design und Datenspeicherung. Diese fünf ADMs werden in zwei Schritten beschrieben: Erstens definieren und interpretieren wir der Architekturmuster, die in diesen Muster-basierten ADMs verwendet werden. Zweitens durch eine Reihe von Fragebögen wird dem Softwarearchitekten ermöglicht, eine geeignete

Architekturentscheidung zu treffen. Schließlich verwenden wir eine Baumstruktur zur Darstellung aller möglichen Entscheidungsprozesse.

4.2.1 Applikationsstruktur

Die erste wiederverwendbare Muster-basierte ADM-Applikationsstruktur bietet eine grundsätzliche Struktur für eine Softwarearchitektur an. Sie enthält eine Architekturentscheidung: Entscheiden Sie, wie Sie Ihre Applikation strukturieren wollen. Dieses Modell bietet das Architekturmuster *Layers* [Buschmann et al., 1996a]. *Layers* hilft, die Applikationen in eine Reihe von Teilaufgaben zu zerlegen. Jede Teilaufgabe befindet sich auf einer bestimmten Ebene der Abstraktion. In dem objekt- oder komponentenorientierten Softwareentwurf ist ein Layer eine Gruppe der Klassen, die in einer Ebene eingekapselt werden und durch bestimmte Schnittstellen miteinander zusammenarbeiten. Die Ebenen des *Layer*-Architekturmusters sind durch Schnittstellen miteinander verbunden, dies erhöht die Modifizierbarkeit, Portabilität, Nicht-Abhängigkeit und Wiederverwendbarkeit der Softwarearchitektur, da jede Ebene als eine geschlossene Einheit gesehen werden kann und deshalb die Änderung einer Ebene keinen Einfluss auf anderen Ebenen ausübt. Diese lose Kopplung bietet gleichzeitig die Skalierbarkeit, Fehlertoleranz und eine hohe Abstraktion für ein System.

Ein typisches Beispiel ist das OSI-Modell, ein Referenzmodell für Netzwerkprotokolle als Schichtenarchitektur. Es wird von der International *Organization for Standardization* (ISO) als Standard veröffentlicht, siehe Abbildung 34. Der Zweck dieses OSI-Modells liegt darin, die Kommunikation zwischen unterschiedlichen technischen Systemen bzw. Hardwares durch das Internet zu ermöglichen. Diese Modellgruppen der Kommunikationsfunktionen werden in sieben aufeinander folgende logische Schichten bzw. Ebenen mit eigenen begrenzten Aufgaben organisiert. Eine Ebene dient für die obere Ebene und wird auch von der darunter liegenden Ebene bedient. In der gleichen Ebene können die Datenpakete mit den auf sie bezogenen Netzwerkprotokollen durch klare Schnittstellen direkt

untereinander ausgetauscht werden.

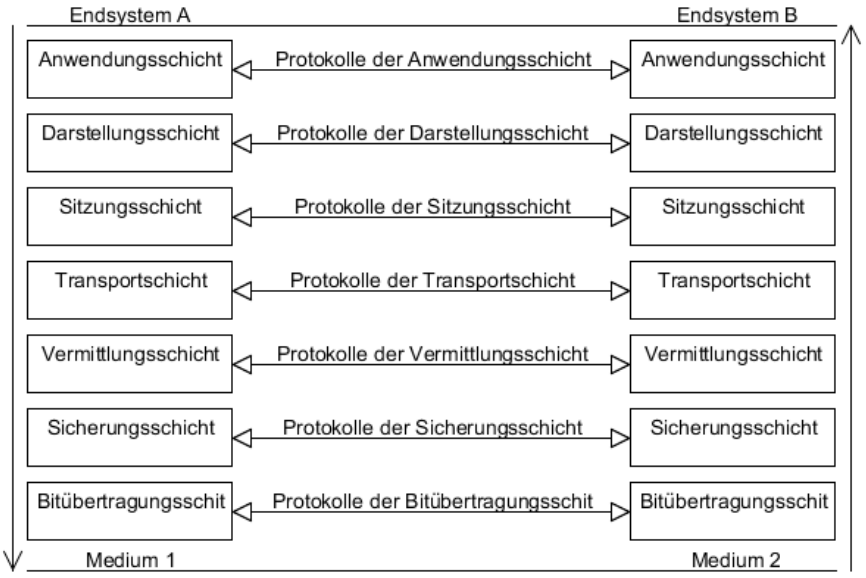

Abbildung 34: OSI-Modell

Im Hinblick auf diese Architekturmuster bietet die Applikationsstruktur ADM fünf Architekturentscheidungen – Layer-Architektur, Einfache Zwei-Layer-Architektur, Multi-Layered Architektur, monolithische Architektur und ein Netz von Komponenten auf dem gleichen Niveau der Abstraktion. Die Softwarearchitekten können dem Entscheidungsprozess mit einer Reihe an Fragen folgen, um eine ideale Architektur für die eigene Applikation zu finden. Tabelle 10 definiert darauf bezogene Architekturentscheidungen, Muster, Lösungen und Fragen.

AD1	Entscheiden Sie, wie Sie Ihre Applikation strukturieren wollen
P1	*Layers*
S1	Verwenden Sie *Layers* (unterstützt **P1**)
S2	Implementieren Sie das System als eine monolithische Architektur
S3	Implementieren Sie das System als ein Netz von Komponenten auf dem gleichen Niveau der Abstraktion
S4	Verwenden Sie eine einfache Zwei-Layers-Architektur, indem die Hauptaufgaben in der Top-Level-Ebene darin bestehen, die Teilaufgaben aufzurufen
S5	Verwenden Sie Multi-Layered-Architektur
Q1	Kann/Können die Hauptaufgabe/n ihres Softwaresystems in eine Reihe von Teilaufgaben bei einem bestimmten Niveau der Abstraktion, Granularität, des Hardware-Abstands oder bei anderen Partitionierungskriterien zerlegt werden?
Q2	Haben Sie multiple Aufgaben in Ihrem System?
Q3	Können die Teilaufgaben Ihrer Hauptaufgaben wieder in eine Reihe der Teilaufgaben bei einem bestimmten Niveau der Abstraktion, Granularität, des

	Hardware-Abstands oder bei anderen Partitionierungskriterien zerlegt werden?
AD(x) ist Xte Architectural Decision(Architekturentscheidung); P(x) ist Xte Pattern(Muster); S(x) ist Xte Solution(Lösung); Q(x) ist Xte Question(Frage)	

Tabelle 10: Architekturentscheidungen, Muster, Lösungen und Fragen von Applikationsstruktur

Abbildung 35 zeigt alle möglichen Entscheidungsprozesse mittels einer flachgelegten binären Baumstruktur für die Applikationsstruktur einer Softwarearchitektur. Im Folgenden besprechen wir diese Prozesse im Detail.

- Q1: Falls die Hauptaufgaben eines Softwaresystems in eine Reihe der Teilaufgaben bei einem bestimmten Niveau der Abstraktion, Granularität, des Hardware-Abstands oder bei anderen Partitionierungskriterien zerlegt werden müssen, soll das *Layers* (S1) Architekturmuster ausgewählt werden. Sonst soll eine einfache Architektur herangezogen werden.
- Q2: Falls multiple Aufgaben in diesem System vorliegen, wird ein einfaches Netz von Komponenten auf dem gleichen Niveau der Abstraktion (S3) verwendet. Sonst wird eine monolithische Architektur in ihrer Applikationsstruktur (S2) eingesetzt.
- Q3: Falls die Hauptaufgaben dieses Softwaresystems wieder in einer Reihe der Teilaufgaben bei einem bestimmten Niveau der Abstraktion, Granularität, des Hardware-Abstands oder bei anderen Partitionierungskriterien zerlegt werden müssen, ist eine Multi-Layered Architektur (S5) zu verwenden, und diese Architektur basiert auf dem Layer-Architekturmuster. Sonst wird eine einfache Zwei-Layer-Architektur (S4) eingesetzt, und diese Architektur basiert auf dem Layer-Architekturmuster.

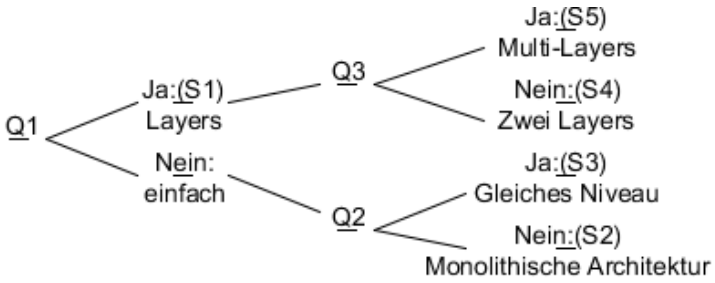

Abbildung 35: Die Baumstruktur für Applikationsstruktur

4.2.2 Verteilte Komponenten

Das wiederverwendbare Muster-basierte ADM Verteilte Komponenten definiert drei Architekturentscheidungen (siehe Tabelle 11) – Kommunikation mit Remote- oder Verteilte Komponenten (AD1), Verteilung (AD2) und Service-orientierte Architektur (AD3). Hierbei bietet die Architekturentscheidung AD1 vier fakultative Architekturmuster für die Architekten, und zwar *Streaming*, *Messaging*, *Remote Procedure Call* (RPC) und *Broker* [Buschmann et al., 1996c].

Das *Steaming* Architekturmuster fokussiert darauf, wie ein kontinuierlich fließender Datenstrom der Informationen ausgetauscht werden kann. Es bringt eine hohe Skalierbarkeit mit sich. Das Streaming einfach zu implementieren und zu warten. Es bietet einen Lösungsansatz an, um die webbasierte Audio- oder Videoübertragungen über ein Netzwerk zur Verfügung zu stellen. Das Messaging Architekturmuster stellt ein netzwerkorientiertes Architekturmuster dar, es fokussiert darauf, wie die verschiedenen Teile eines *Message-Passing-Systems* oder Verteilten Softwaresystems miteinander kommunizieren. Das Messaging wird verwendet, um Datenpakete häufig, unmittelbar, zuverlässig und asynchron über verschiedene Datenformate zu übertragen. Dadurch minimiert sich die Kopplung zwischen Server und Client, mit dieser Funktionalität werden verschiedene Softwarekomponenten durch den Kommunikationsmechanismus entkoppelt. Deshalb wird eine hohe Skalierbarkeit von Messaging aufgrund dieser losen Kopplung erbracht.

Im Gegensatz dazu bietet das RPC Architekturmuster eine enge Kopplung zwischen verschiedenen Softwarekomponenten an. Wenn der Softwarearchitekt dieses Muster auswählt, muss er berücksichtigen, dass unterschiedliche Kommunikationsfehler beim Einsatz von RPC wegen der engen Kopplung auftreten können. Somit könnte ein RPC-basiertes System mehr als unzuverlässig sein. Aber im Allgemeinen handelt es sich bei RPC um einen reifen und sicheren Interprozesskommunikation (IPC) Mechanismus. Und es ist einfach aufzubauen und zu warten und bietet eine effiziente Kommunikation. Das Broker Architekturmuster kann verwendet werden, um verteilte

Systeme mit entkoppelten Komponenten, die mit Remote-Service-Aufrufen interagieren, zu strukturieren. Broker kann alle kommunikationsbezogenen Codes in einer eigenen Ebene trennen, dadurch wird dieser Code isoliert, aber es werden nur dessen Funktionalitäten zur Verfügung gestellt. Außerdem bietet es eine einfache Kapselung komplexer Kommunikationslogik in den separaten Schichten an. Die Flexibilität ist auch ein Vorteil von Broker.

Die Architekturentscheidung hinsichtlich AD2 dieses ADM bringt drei Architekturmuster mit sich – *Client-Server*, *Publisher-Subscriber* [Buschmann et al., 1996e] und N-Schichten (N-Tier). *Client-Server* und *Publisher-Subcriber* können als Eins-zu-viele-Architektur gesehen werden. Aber das *Client-Server* Architekturmuster wird oft auf verteilten Systemen verwendet, die ein separates Client- und Server-System über ein Verbindungsnetzwerk einbinden, im Gegensatz zu dem *Publisher-Subscriber* Architekturmuster. Der *Client-Server* zeigt eine höhere Sicherheit, da alle Daten auf dem Server gespeichert werden. In der Regel stellt er eine bessere Kontrolle für die Sicherheit als Datenspeicherung auf dem Client-Rechner zur Verfügung. Ein weiterer Vorteil liegt in der Einfachheit der Wartung. Aber übermäßige Abhängigkeit von dem Server könnte negative Auswirkungen auf die Erweiterbarkeit, Skalierbarkeit und Zuverlässigkeit des Systems mit sich bringen. Einer der wichtigsten Vorteile von *Publisher-Subscriber* betrifft die lose Kopplung. Es bietet sich auch eine Möglichkeit für eine bessere Skalierbarkeit als bei herkömmlichen *Client-Server* Architekturmustern durch mehrere gleichzeitig parallel betriebene Operationen, Caching-Mechanismen der Nachrichten usw. Das N-Schichten Architekturmuster kann als eine weiterentwickelte Implementierung des Client-Server (zwei-Schichten) gesehen werden. Es bringt bessere Wartbarkeit, Flexibilität und Skalierbarkeit relativ zu dem Client-Server mit sich.

Für die Architekturentscheidung AD3 dieses ADM kann die Service-orientierte Architektur (SOA) ausgewählt werden. Diese Architektur basiert auf verschiedenen Teilen der Software, die die Anwendungsfunktionalität als Service für andere Anwendungen anbietet. Dies wird als Serviceorientierung benannt. Diese Services

sind auf die Bereitstellung einer schema- und nachrichtenbasierten Interaktion mit den Anwendungen über Schnittstellen konzentriert. Die Schnittstellen sind nicht komponenten- oder objektbasiert, sondern anwendungsbegrenzt. SOA bietet eine lose Kopplung für den Aufbau verteilter Softwaresysteme, da Services die Abhängigkeiten zwischen verschiedenen Komponenten minimieren. Außerdem wird die Geschäftslogik in geeignete Services aufgeteilt, um die Wiederverwendbarkeit des Codes zu fördern. Die Services bieten eine hohe Abstraktion durch das Verbergen der Geschäftslogik vor der Außenwelt.

AD1	Die Kommunikation mit Remote-/verteilten Komponenten
AD2	Verteilung (Distribution)
AD3	Service-orientierte Architektur
P1	Client-Server Architektur
P2	N-Schichten (N-Tier) Architektur
P3	Service-orientierte Architektur
P4	Broker
P5	Publisher-Subscriber
P6	*Remote Procedure Call* (RPC)
P7	Messaging
P8	Streaming
S1	Verteilen Sie Ihre Architektur als Client-Server (unterstützt **P1**)
S2	Verteilen Sie Ihre Architektur als N-Sichten (unterstützt **P2**)
S3	Implementieren Sie Ihre Architektur als SOA (unterstützt **P3**)
S4	Verwenden Sie einen Broker (unterstützt **P4**)
S5	Verwenden Sie einen Publisher-Subscriber (unterstützt **P5**)
S6	Verwenden Sie RPC (unterstützt **P6**)
S7	Verwenden Sie Messaging (unterstützt **P7**)
S8	Verwenden Sie Streaming (unterstützt **P8**)
S9	Verwenden Sie Punkt-zu-Punkt-Kommunikation
S10	Erstellen Sie eine benutzerdefinierte RPC-Implementierung
S11	Verwenden Sie einen beliebigen Kommunikationsstil wie Peer-to-Peer oder Client-Server (wenn Client-Server verwendet wird, ist es ratsam, die nächste Entscheidung "Serviceorientierte Architektur" zu beantworten)
Q1(AD1)	Müssen Sie Streams (sowie Video-/Audio-Streams) übertragen?
Q2(AD1)	Müssen Sie lose Kopplung in verteilten Komponenten und/oder eine hohe Zuverlässigkeit der Nachrichtenübertragung haben?
Q3(AD1)	Müssen Sie ein Middleware-Framework nutzen oder entwickeln, um alle kommunikationsbezogenen Aspekte in einer separaten Schicht oder Komponente zu kapseln?
Q1(AD2)	Werden Ihre Komponenten-Austausch-Events asynchron in einer Eins-zu-viele-Konfiguration eingesetzt (Viele Clients werden von eingehenden Events informiert)?
Q2(AD2)	Müssen Sie zwischen Client, Daten und Geschäftslogik in Ihren verteilten

	Komponenten differenzieren?
Q3(AD2)	Müssen Sie Ihre verteilten Komponenten auf mehr als zwei Maschinen bzw. Rechnern aus anderen Gründen (z. B. aus Gründen der Skalierbarkeit) einsetzen?
Q4(AD2)	Haben Sie die Dienstleistungen für mehrere Clients in verteilten Komponenten?
Q1(AD3)	Benötigen Sie standardisierte Service-Schnittstellen und/oder lose gekoppelte Services?
AD(x) ist Xte Architectural Decision(Architekturentscheidung); P(x) ist Xte Pattern(Muster); S(x) ist Xte Solution(Lösung); Q(x) ist Xte Question(Frage)	

Tabelle 11: Architekturentscheidungen, Muster, Lösungen und Fragen von den verteilten Komponenten

Abbildung 36 zeigt die Entscheidungsprozesse für die Erfassung der drei Architekturentscheidungen mittels des entwickelten ADM Verteilte Komponenten. Im Folgenden besprechen wir diese Prozesse im Detail.

- Q1 von AD1: Falls die Datenstreams (wie Video-/Audio-Streams) in der Softwareapplikation übertragen werden müssen, wird das Streaming Architekturmuster (S8) verwendet. Ansonsten wird kein Streaming Architekturmuster eingesetzt.
- Q2 von AD1: Falls die lose Kopplung in verteilten Komponenten vorliegen und/oder eine hohe Zuverlässigkeit der Nachrichtenübertragung aufweisen muss, wird das Messaging Architekturmuster (S7) verwendet, ansonsten das RPC Architekturmuster (S6).
- Q3 von AD1: Falls eine Middleware-Framework benötigt wird, um alle kommunikationsbezogenen Aspekte in einer separaten Schicht (Layer) oder Komponente zu kapseln, wird ein Broker Architekturmuster (S4) verwendet. Ansonsten wird eine benutzerdefinierte RPC-Implementierung (S10) erstellt.
- Q1 von AD2: Falls die Komponenten-Austausch-Events asynchron in einer Eins-zu-viele-Konfiguration eingesetzt werden, wird das Publisher-Subscriber Architekturmuster (S5) verwendet, ansonsten eine Punkt-zu-Punkt Kommunikation (S9).
- Q2 von AD2: Falls der Architekt zwischen Client, Daten und Geschäftslogik in der verteilten Komponenten differenzieren muss, wird das N-Schichten Architekturmuster (S2) verwendet, ansonsten kein Schichtenmuster.

- Q3 von AD2: Falls die verteilten Komponenten auf mehr als zwei Maschinen bzw. Rechnern aus anderen Gründen (z. B. aus Gründen der Skalierbarkeit) eingesetzt werden müssen, wird das N-Schichten Architekturmuster (S2) verwendet, ansonsten kein Schichtenmuster.
- Q4 von AD2: Falls die Dienstleitungen für mehrere Clients in verteilten Komponenten benötigt werden, ist das Client-Server Architekturmuster (S1) zu verwenden, sonst ein beliebiger Kommunikationsstil wie *Peer-to-Peer* (S11).
- Q1 von AD3: Falls die standardisierten Service-Schnittstellen und/oder lose gekoppelte Services benötigt werden, ist das SOA Architekturmuster zu verwenden, ansonsten kein SOA Architekturmuster (S3).

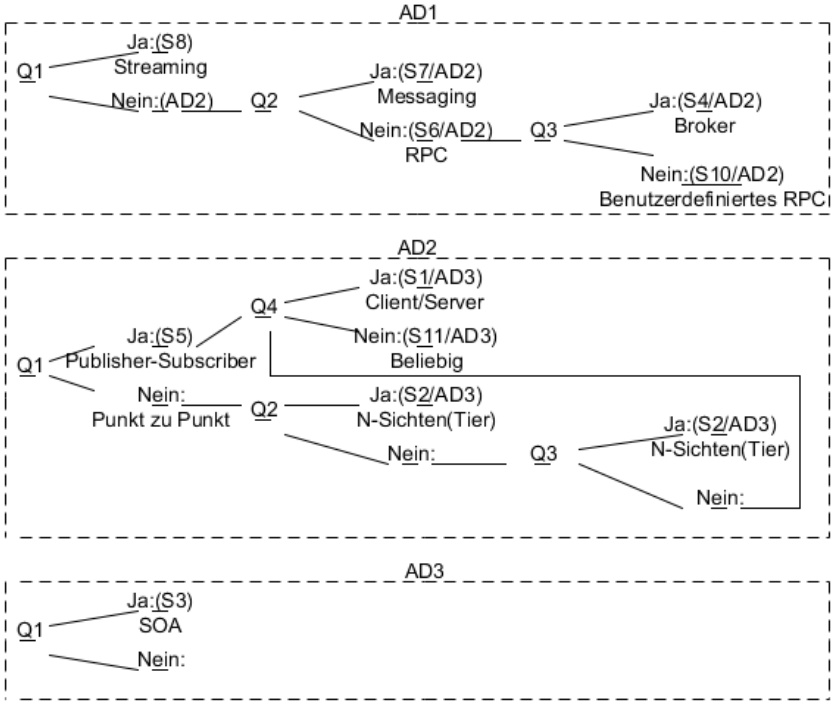

Abbildung 36: Baumstruktur für verteilte Komponenten

4.2.3 Taskverarbeitung

Wie können die Softwarearchitekten die komplexen Verarbeitungsaufgaben behandeln? Dies stellt ein häufig vorkommendes Problem während des Entwicklungsprozesses der Softwarearchitektur dar. Es bezieht sich auf die wichtigen Architekturentscheidungen. In dieser Muster-basierten ADM

„Taskverarbeitung“ bieten wir ein Architekturmuster *Pipes-and-Filters* [Buschmann et al., 1996b] an. Das *Pipes-and-Filters* Architekturmuster ermöglicht ein strukturiertes System. In ihm werden die Daten als Datenströme verarbeitet und durch die *Pipes* zwischen benachbarten *Filters* geleitet. Diese Daten kommen aus den Datenquellen durch die Transformation von einem oder mehreren Verarbeitungsschritt/en. Jeder Verarbeitungsschritt wird in einem Filter eingekapselt und diese Schritte können beliebig angeordnet oder getrennt voneinander entwickelt werden, siehe Abbildung 37.

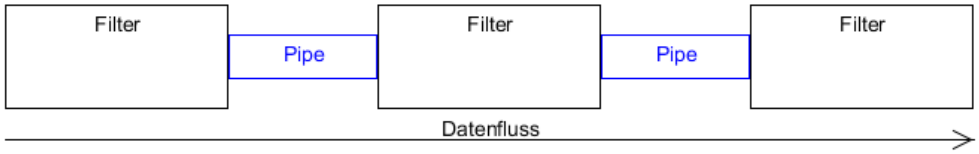

Abbildung 37: Pipes-and-Filters

Ein typisches Beispiel ist die Struktur des Compilers der Programmiersprachen, zum Beispiel, der Prozess einer Programmiersprache Java von Source Code bis zum ausführbaren Code, etwa wie Abbildung 38. Das Source Code von Java wird durch Preprocessor, Compiler, Objektcode Sammler und Linker in einen ausführbaren Code übersetzt. Der Datenfluss in dem Complier ist unidirektional, das heißt, der Source Code wird durch die verschiedenen Verarbeitungsstufen behandelt, und es gibt einen endgültigen Zielcode. Der Vorteil von *Pipes-and-Filters* liegt in Folgendem: Wenn die Schnittstelle zwischen den Phasen explizit definiert ist, können die Änderungen der Funktionen innen begrenzt werden. Ersetzen Sie den Filter so, dass das einzelne Filter die anderen Filter nicht beeinflusst.

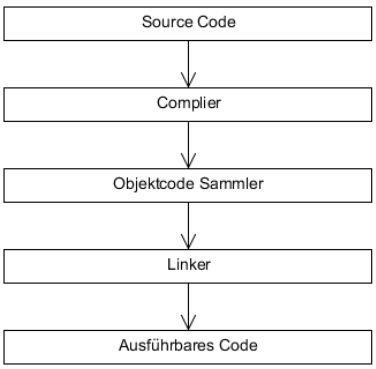

Abbildung 38: eine Struktur des Compilers für die Programmiersprache Java

Laut dem Architekturmuster *Pipes-and-Filters* entwarfen wir folgende Fragen und Lösungen, siehe Tabelle 12. Gleichzeitig bietet die Abbildung 39 eine binäre Baumstruktur zur Darstellung aller Alternativen an. Es folgt die Interpretation dieser Baumstruktur.

- Q1: Falls eine oder größere Verarbeitungsaufgabe/n in eine Reihe von kleineren, unabhängigen Verarbeitungsschritten aufgeteilt werden müssen, sind mehrere Funktionen in diesem Softwaresystem zu verwenden. Ansonsten wird eine einzelne Funktion eingesetzt, um jede komplexe Verarbeitungsaufgabe zu implementieren (S2).
- Q2: Falls eine Reihe der kleineren und unabhängigen Verarbeitungsschritte, die zur Implementierung der größeren Verarbeitungsaufgaben verwendet werden, flexibel verbunden werden müssen, ist das *Pipes-and-Filters* Architekturmuster (S1) zu verwenden. Ansonsten wird eine Reihe der kleineren und unabhängigen Funktionen (S3) herangezogen, um die komplexen Verarbeitungsaufgaben zu implementieren.
- Q3: Falls die aufgeteilten Verarbeitungsaufgaben, basierend auf dem *Pipes-and-Filter*, noch parallelisiert werden müssen, ist eine parallele Implementierung (S4) einzusetzen. Ansonsten wird eine sequentielle Implementierung (S5) verwendet.
- Q4 und Q5: Falls die Verarbeitungsaufgaben des Softwaresystems parallelisiert werden müssen, wird eine Reihe der kleinen und unabhängigen Funktionen und Multithreaded-Implementierungen für diese Verarbeitungsaufgaben (S6) eingesetzt. Ansonsten wird eine Reihe der kleinen und unabhängigen Funktionen und Single-Threaded-Implementierungen für diese Verarbeitungsaufgaben (S7) verwendet.

AD1	Entscheiden Sie, wie Sie die komplexen Verarbeitungsaufgaben behandeln.
P1	*Pipes-and-Filters*
S1	Verwenden Sie *Pipes-and-Filters* (Unterstützt P1)
S2	Verwenden Sie eine einzelne Funktion, um jede komplexe Verarbeitungsaufgabe zu implementieren
S3	Verwenden Sie eine Reihe der kleinen und unabhängigen Funktionen, um die komplexen Verarbeitungsaufgaben zu implementieren
S4	Parallele Implementierung von Pipes-and-Filters Komponenten

P5	Sequentielle Implementierung von Pipes-and-Filters Komponenten
S6	Multithreaded-Implementierung der Aufgaben
S7	Single-Threaded-Implementierung der Aufgaben
Q1	Müssen Sie eine oder mehrere größere Verarbeitungsaufgabe/n in eine Reihe kleinerer, unabhängiger Verarbeitungsschritte aufteilen?
Q2	Müssen Sie eine Reihe kleinerer und unabhängiger Verarbeitungsschritte, die zur Implementierung der größeren Verarbeitungsaufgaben verwendet werden, flexibel verbinden?
Q3	Wollen Sie die Verarbeitungsaufgaben parallelisieren?
Q4	Wollen Sie die Verarbeitungsaufgaben parallelisieren?
Q5	Wollen Sie die Verarbeitungsaufgaben parallelisieren?
AD(x) ist Xte Architectural Decision(Architekturentscheidung); P(x) ist Xte Pattern(Muster); S(x) ist Xte Solution(Lösung); Q(x) ist Xte Question(Frage)	

Tabelle 12: Architekturentscheidungen, Muster, Lösungen und Fragen hinsichtlich der Verarbeitungsaufgaben

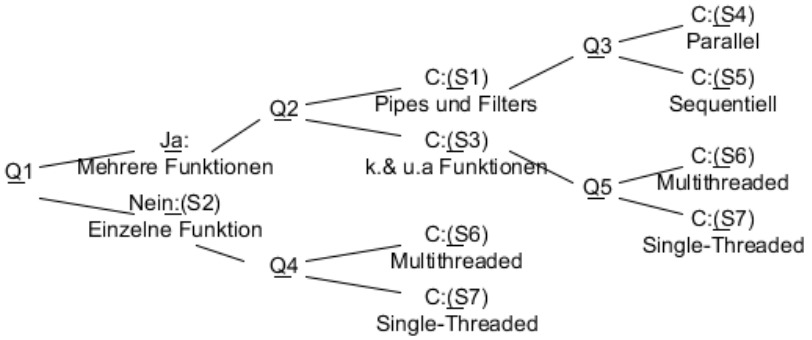

Abbildung 39: Baumstruktur für die Verarbeitungsaufgaben

4.2.4 UI Design

Das UI Design richtet sich oft direkt an die Endkunden. Wenn die Softwarearchitekten das UI Design rational entwerfen möchten, sind nicht nur die Anforderungen der Funktionen der Endkunden zu erfüllen, sondern auch die Bedienlogik einer komplexen Architektur ist gut anzupassen. Für eine generelle Softwareapplikation stellt das Architekturmuster *Model-View-Controller* (MVC) [Buschmann et al., 1996d] eine Möglichkeit dar. Es teilt eine Softwareapplikation in drei miteinander verbundene Bereiche, um die interne Darstellung von Informationen an Endkunden zu präsentieren und den Input von Endkunden zu empfangen. Ein Modell teilt den verbundenen Views und Controllers den Änderungsstatus mit. Das Modell enthält auch die Kernfunktionalitäten und Daten. Ein Controller kann die Befehle an ein Modell schicken, um den Zustand dieses Modells zu aktualisieren. Und es kann auch die Befehle an ein View schicken, um die darauf bezogenen Inhalte der Präsentation

zu ändern. Ein View fordert die Informationen aus dem Modell, das es verwendet, um eine Output-Darstellung für die Endkunden zu generieren. Der Hauptvorteil der MVC-Muster liegt in der Differenzierung der Ebenen eines Projekts in Modell, View und Controller für die Wiederverwendbarkeit und Wartung von Codes. Modifizierbarkeit und Testbarkeit ergeben also zwei wesentliche Vorteile dieses Musters. Das MVC-Muster wird in vielen bekannten Frameworks verwendet, z. B. .Net MVC, Spring, Struts 2.0, Zend 2.0 usw. Wie in Kapitel beschrieben, wurde das MVC Architekturmuster in der Entwicklungsphase von WebADvISE auch verwendet, siehe Kapitel 3.2.4.

AD1	Entscheiden Sie, wie man Ihr UI Design entwerfen kann
P1	Model-View-Controller (MVC)
S1	UI Design mit Model-View-Controller (MVC) (unterstützt **P1**)
S2	Implementieren Sie ein vereinfachtes kundenspezifisches UI Design
S3	Entkoppeltes UI Design des Anwendungslogik Codes
Q1	Müssen Sie verschiedene Präsentationen (Views) für verschiedene Benutzertypen anbieten?
Q2	Müssen Sie neue Präsentationen (Views) in der Zukunft hinzufügen, nachdem Sie ihr Softwaresystem implementieren?
Q3	Müssen die Modifizierbarkeit, unabhängige Prüfbarkeit oder lose Kopplung der Benutzeroberflächen und die Anwendungslogik gewährleitet werden?
AD(x) ist Xte Architectural Decision(Architekturentscheidung); P(x) ist Xte Pattern(Muster); S(x) ist Xte Solution(Lösung); Q(x) ist Xte Question(Frage)	

Tabelle 13: Architekturentscheidungen, Muster, Lösungen und Fragen bezogen auf das UI Design

Auf Basis der MVC-Muster entwickelten wir drei mögliche Architekturentscheidungen für UI Design ADM, MVC-basierte Architektur, vereinfachte kundenspezifische UI Architektur und Entkoppelte UI Architektur des Anwendungslogik Codes. Die Architekten können den Entscheidungen mit drei Fragen folgen, um eine geeignete Architektur für sich selbst zu finden. Tabelle 13 definiert diese Architekturentscheidungen, Muster, Lösungen und Fragen. Sämtliche möglichen Entscheidungsprozesse werden in Abbildung 40 dargestellt. Im Folgenden besprechen wir diese Prozesse im Detail.

- Q1: Falls die verschiedenen Präsentationen (Views) für verschiedene Benutzertypen angeboten werden müssen, ist das Modell Model-View-Controller (MVC) Architekturmuster für UI Design (S1) zu verwenden. Ansonsten muss der

Architekt ein anderes Architekturmuster oder gar kein Muster einsetzen.

- Q2: Falls der Architekt neue Präsentationen (Views) in der Zukunft einfügen will, nachdem er das Softwaresystem implementiert hat, wird das Model-View-Controller (MVC) Architekturmuster für UI Design (S1) verwendet. Ansonsten wird ein anderes Architekturmuster oder gar kein Muster eingesetzt.
- Q3: Falls die Modifizierbarkeit, unabhängige Prüfbarkeit oder lose Kopplung der Benutzeroberflächen und die Anwendungslogik gewährleistet werden müssen, ist ein entkoppeltes UI Design des Anwendungslogik Codes (S3) zu verwenden, ansonsten ein vereinfachtes kundenspezifisches UI Design (S2).

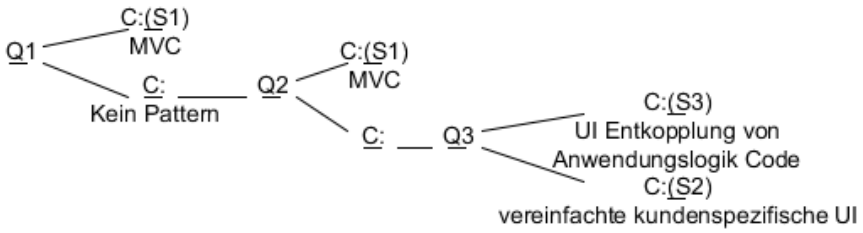

Abbildung 40: Baumstruktur für das UI Design

4.2.5 Datenspeicher

Die Datenpersistenz ist das Kernproblem einer Enterprise Software. Sicherheit und Skalierbarkeit der Kundendaten während der Softwareentwicklung sind sehr wichtig. In der Muster-basierten ADM-Datenspeicherung bieten wir ein *Shared Repository* Architekturmuster [Microsoft, 2014]. *Shared Repository* wird als ein zentraler Datenspeicher verwendet, um alle anderen unabhängigen Komponenten abzurufen. Es implementiert und zentralisiert eine Caching-Strategie für die Datenquelle. *Shared Repository* erhöht die Wartbarkeit und Lesbarkeit des Codes durch Trennung der Geschäftslogik von der Datenzugriffslogik und reduziert einen anteiligen redundanten Code. Es kann auch die Datenkonsistenz gewährleisten und das Problem der Ressourcenkonflikte behandeln.

AD1	Entscheiden Sie, ob Sie einen Datenspeicher benötigen
P1	*Shared Repository*
S1	Verwenden Sie ein Repository
S2	Verwenden Sie ein lokales Repository

S3	Verwenden Sie ein *Remote Repository* (unterstützt **P1**)
Q1	Werden einige Systemkomponenten benötigt, um Daten auszutauschen?
Q2	Müssen Sie große Datenmengen in Ihrer Softwareapplikation übertragen oder verarbeiten?
Q3	Sind verteilte Komponenten, die auf gemeinsam genutzte Daten zugreifen müssen, in Ihrem System vorhanden?
AD(x) ist Xte Architectural Decision(Architekturentscheidung); P(x) ist Xte Pattern(Muster); S(x) ist Xte Solution(Lösung); Q(x) ist Xte Question(Frage)	

Tabelle 14: Architekturentscheidungen, Muster, Lösungen und Fragen bezogen auf den Datenspeicher

Wie in die Tabelle 14 gezeigt, bietet dieses ADM drei Architekturentscheidungen – Repository, lokales *Repository* und *Shared Repository* bzw. Remote Repository. Tabelle 14 definiert auch die darauf bezogenen Architekturentscheidungen, Muster, Lösungen und Fragen. Abbildung 41 zeigt die Entscheidungsprozesse für die Erfassung der drei Architekturentscheidungen. Im Folgenden besprechen wir diese Prozesse im Detail.

- Q1: Falls einige Systemkomponenten benötigt werden, um Daten auszutauschen, wird ein Repository Architekturmuster für Datenspeicher (S1) verwendet, ansonsten kein Repository Architekturmuster.
- Q2: Falls die großen Datenmengen in der Softwareapplikation übertragen oder verarbeitet werden müssen, wird ein Repository Architekturmuster für den Datenspeicher (S1) verwendet. Ansonsten wird kein Repository Architekturmuster eingesetzt.
- Q3: Falls verteilte Komponenten in dem Softwaresystem enthalten sind, wird das *Shared Repository* Architekturmuster für Datenspeicher (S3) verwendet, ansonsten ein lokales Repository Architekturmuster (S2).

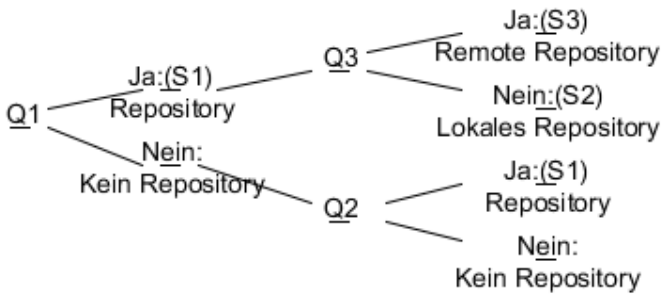

Abbildung 41: Baumstruktur für Datenspeichner

4.3 Herstellung der entwickelten ADMs in WebADvISE

In Kapitel 3.3 wurde bereits besprochen, wie ein ADM in WebADvISE erstellt werden kann. In diesem Kapitel werden diese Herstellungsschritte der fünf erstellten ADMs in WebADvISE dargelegt. Es gibt drei Herstellungsschritte für alle fünf ADMs. Da diese drei Herstellungsschritte bei den fünf ADMs ganz ähnlich sind, fassen wir die Herstellungsschritte hier kurz zusammen.

1. **Schritt:** Erstellen Sie die Architekturmuster (Patterns) für ein ADM im „Design Patterns“ Editor.
2. **Schritt:** Erstellen Sie einige Lösungen bzw. ADDs für dieses ADM auf Basis der geeigneten Muster im „Design Solutions“ Editor.
3. **Schritt:** Erstellen Sie eine Reihe von auf das bezogene ADM Entwurfsfragen im „Architectural Decisions“ Editor, um eine bestimmte und entschiedene Lösung bzw. ADD zu ermitteln.

4.4 Herstellung der ADDs für Universitäts-IS mittels entwickelter ADMs in WebADvISE

Kapitel 3.3.11 bis Kapitel 3.3.14 beschreiben, wie ein Softwarearchitekt die Fragebogen für die Architekturentscheidungen in WebADvISE editieren, speichern und zurücksetzen und einen Architekturentscheidungseditor editieren und online speichern kann. In diesem Kapitel besprechen wir die Herstellung der ADDs laut der Beschreibung und den Anforderungen (siehe Anhang) des Universitäts-IS mittels der fünf entwickelten ADMs (siehe Kapitel 4.2) in WebADvISE. Es gibt zwei Herstellungsschritte:

1. **Schritt:** Beantworten Sie die Entwurfsfragenbogen im „Questionnaire“ Editor und speichern Sie die Ergebnisse. Nach der Arbeit erstellen Sie einen „Decisions“ Editor.

2. Schritt: Editieren Sie die Architekturentscheidung (*Decision*) im „Decisions“ Editor.

Das Universitäts-Informationssystem (Universitäts-IS) stellt ein skalierbares, zuverlässiges und leistungsfähiges System dar. Es enthält viele typische Funktionalitäten für den Endbenutzer, z. B. Studentenservices, Services für Forschung und Administratives Personal, E-Learning-System, Bibliotheksservice, Stream Service und Forschungsnetzwerk usw. Im Folgenden werden in Tabelle 15 zwölf Anforderungen des Universitäts-IS im Detail zusammengefasst und aufgelistet. Gemäß diesen Anforderungen werden die Entscheidungsprozesse der ADDs für Universitäts-IS mittels entwickelter ADMs in Kapitel 4.4.1 bis Kapitel 4.4.5 beschrieben. Und jedes Kapitel bietet eine Diagrammdarstellung für diese Entscheidungsprozesse an. Diese Diagrammdarstellungen werden auf Basis der Abbildungen 35, 36, 39, 40 und 41 wiederverwendet und weiterentwickelt.

Anforderungen des UIS	
A1	Die verschiedenen Komponenten, die innerhalb des Universitäts-IS entwickelt werden, müssen auf demselben Abstraktionsniveau organisiert werden.
A2	Das Universitäts-IS wird das Aufnehmen und Streamen von Vorlesungen in den Hörsälen anbieten. Der Client kann die Umwandlung in andere Audio- und Video-Formate anfordern.
A3	Die Studenten werden sich für das Studium inskribieren, sich für Kurse und Prüfungen anmelden, Zeugnisse herunterladen und Information über die Kurse, für die sie angemeldet sind, sehen können. Das Universitäts-IS wird den Studenten einen Service zur Erzeugung von Zeugnissen anbieten.
A4	Die Streams werden von einem zentralen Server verwaltet und können auf Nachfrage heruntergeladen werden.
A5	Die Universität kollaboriert mit einem zentralen Forschungsnetzwerk für den Aufbau von Netzwerken zwischen Wissenschaftlern, Dissemination von Forschungsergebnissen, Zugang zu Publikationen, usw. Das Universitäts-IS muss Updates neuer Publikationen und Forschungsprojekte des Forschungspersonals verteilen.
A6	Durch das Universitäts-IS werden die Studenten Zugang zu unterschiedlichen E-Learning-Plattformen, die von den Universitätsinstituten angeboten werden, erhalten. Das Universitäts-IS muss automatisch durch die unterschiedlichen E-Learning-Plattformen aktualisiert werden.
A7	Das neue Universitäts-IS muss skalierbar, zuverlässig und leistungsfähig sein. Die Applikationskomponenten werden auf unterschiedliche Geräte deployed (die Daten werden auch in einen eigenen Server gespeichert werden), um eine hohe Skalierbarkeit zu erreichen.
A8	Standardisierte Service Interfaces werden sowohl für interne Komponenten als

	auch für externe Systeme angeboten.
A9	Das Forschungsnetzwerk verwendet nur das XML-RPC-Protokoll, um mit externen Systemen zu kommunizieren.
A10	Sie müssen eine Web-UI für alle Studenten implementieren, die leicht zu testen ist, sie muss veränderbar und entkoppelt von der Applikationsfunktionalität vorliegen.
A11	Die Universitätsassistenten, Lektoren, Professoren sowie das administrative Personal müssen unterschiedliche Interfaces haben, wo sie neue Kurse eintragen, Studenten benoten, angemeldete Studenten für bestimmte Kurse ansehen können usw. Sie müssen das berücksichtigen, wenn Sie die UI des Universitäts-IS entwerfen.
A12	Das Universitäts-IS bietet einen Bibliotheks-Katalog an, in dem sich Benutzer verfügbare Bücher, Journals usw. ansehen und für einen Tag reservieren können. Andere Komponenten, die Statistiken berechnen und die Bibliotheksdatenbank mit neuen Online-Publikationen aktualisieren, sollten auch durch einen Fernzugriff auf den Bibliothekskatalog vorliegen können.
A(x) bedeutet Xte Anforderungen	

Tabelle 15: Anforderungen des Universitäts-IS [vgl. UIS, 2013]

4.4.1 Applikationsstruktur

Die Anforderung A1 des Universitäts-IS von Tabelle 15 beschreibt, dass verschiedene Komponenten des Universitäts-IS auf demselben Abstraktionsniveau organisiert werden müssen. Das heißt, die Hauptaufgaben des Universitäts-IS müssen auf einer bestimmten Ebene der Abstraktion, Granularität, des Hardware-Abstands oder durch andere Partitionierungskriterien zerlegt werden und nicht gleichzeitig das Muster *Layers* verwenden. Aber das Universitäts-IS bietet multiple Aufgaben in seinem System. Abbildung 42 zeigt die Entscheidungsschritte der ADDs für das Universitäts-IS auf Basis einer entwickelten ADM-Applikationsstruktur in WebADvISE. Die rote Ablaufverfolgung beschreibt diese Entscheidungsschritte der ADDs.

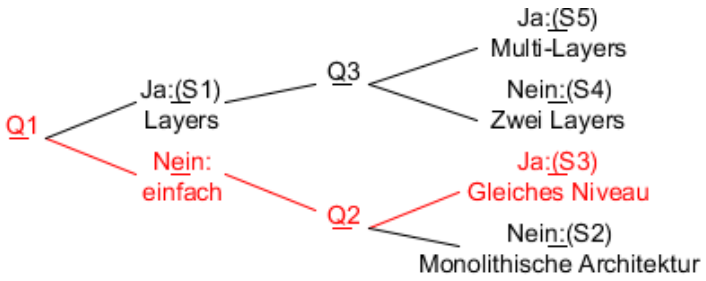

Abbildung 42: Die Entscheidungen in Baumstruktur für die Applikationsstruktur

4.4.2 Verteilte Komponenten

Wie in Kapitel 4.4.1, zeigt Abbildung 43 die Entscheidungsschritte der ADDs für das Universitäts-IS auf Basis der entwickelten ADM Verteilten Komponenten in WebADvISE. Diese Entscheidungsschritte werden in sechs Teilentscheidungsschritte der ADDs zerlegt, nämlich Stream Service, Universitäts-IS, Kommunikation in IS, Kommunikation mit Forschungskomponente, E-Learning Plattform und Verteilte Systeme des Universitäts-IS. Im Folgenden beschreiben wir diese kleineren Entscheidungsschritte im Detail, es finden sich alle folgenden Anforderungen aus Tabelle 15.

- **Stream Service:** Die Anforderung A2 des Universitäts-IS beschreibt, dass der Client die Umwandlung in Audio- und Video-Formate anfordern muss. Das heißt, wir müssen das Muster Messaging in diesem ADM verwenden. Die Begründungen diskutieren wir im Kapitel 4.2.2. Die rote Ablaufverfolgung im Bereich AD1 beschreibt diese Teilentscheidungsschritte der ADDs.

- **Universitäts-IS:** Die Anforderungen A3, A4, A8 beschreiben, dass das Universitäts-IS einen zentralen Service für die Studenten anbietet. Damit können viele Dienstleistungen angeboten werden, z. B. Herunterladen des Streams und Erzeugung von Zeugnissen. Das Universitäts-IS bietet auch standardisierte Service Interfaces an. Um diese Anforderungen zu erfüllen, wählen wir das Architekturmuster Serviceorientierte Architektur (SOA) für das Universitäts-IS aus, weil dieses Architekturmuster nicht nur zu den erwähnten Anforderungen gut passt, sondern auch eine lose miteinander verknüpfte Integration und bessere Verfügbarkeit sowie Stabilität für das Universitäts-IS anbieten kann. Die rote Ablaufverfolgung im Bereich AD3 beschreibt diese Teilentscheidungsschritte der ADDs.

- **Kommunikation in IS:** Unter Berücksichtigung der Anforderungen A2, A3, A4, A8 schlagen wir das Architekturmuster Messaging für die Kommunikation im IS vor, denn es unterstützt dahingehend, die Datenpakete häufig, sofort, zuverlässig

und asynchron zwischen Server und Client über unterschiedliche Datenformate zu übertragen. Messaging bietet eine lose Kopplung in verteilten Komponenten und eine hohe Zuverlässigkeit der Nachrichtenübertragung an. Und andererseits kann das Messaging SOA gut unterstützen. Die grüne Ablaufverfolgung im Bereich AD1 beschreibt diese Teilentscheidungsschritte der ADDs.

- **Kommunikation mit Forschungskomponente:** Für diese kleinen Entscheidungsschritte der ADDs wählen wir das Architekturmuster *Remote Procedure Call* (RPC) aus. Der erste Grund liegt darin, die Anforderung A5 zu erfüllen. Der zweite Grund ist, dass die Anforderung A8 vorschlägt, das XML-RPC-Protokoll zu verwenden, um mit externen Systemen zu kommunizieren. RPC bietet einen synchronen Mechanismus, den Kontrollfluss und Ergebnisdaten als Prozeduraufruf über das Internet zwischen Rechnern. Die blaue Ablaufverfolgung im Bereich AD1 beschreibt diese Teilentscheidungsschritte der ADDs.

- **E-Learning Plattform:** Die Anforderungen A3, A4, A5, A6 beschreiben eine typische Eigenschaft des Architekturmusters *Publisher-Subscriber*, nämlich die asynchrone Kommunikation des Komponenten-Austausch-Events in einer Eins-zu-viele-Konfiguration. Zum Beispiel, wird das Universitäts-IS den Studenten einen Service zur Erzeugung von Zeugnissen anbieten. Die Streams werden von einem zentralen Server verwaltet. Durch das Universitäts-IS werden die Studenten Zugang zu unterschiedlichen E-Learning Plattformen gewinnen, und das Universitäts-IS muss automatisch durch die unterschiedlichen E-Learning Plattformen aktualisiert werden usw. Der Publisher-Subscriber kann sicher nicht nur solche E-Learning Plattformen gut passen, sondern auch eine lose Kopplung und Skalierbarkeit für diese Plattformen anbieten. Die rote Ablaufverfolgung im Bereich AD2 beschreibt diese Teilentscheidungsschritte der ADDs.

- **Verteilte Systeme des Universitäts-IS:** Durch die Analyse der obigen fünf Teilentscheidungsschritte empfehlen wir, das Architekturmuster N-Schichten (N-

Tier) auf verteilten Systemen des Universitäts-IS zu verwenden. Insbesondere unter Berücksichtigung der Anforderung A7 müssen die Applikationskomponenten des Universitäts-IS auf unterschiedliche Geräte deployed werden, um eine hohe Skalierbarkeit zu erreichen. Die N-Schichten Architektur kann genau eine solche Skalierbarkeit für ein Softwaresystem anbieten, weil jede Schicht unabhängig von den anderen Schichten ist. Und Updates oder Änderungen dieser Schicht können das Softwaresystem als Ganzes nicht beeinflussen. Infolge dieser Unabhängigkeit aller Schichten bringt diese Architektur für Universitäts-IS Zuverlässigkeit, Leistungsfähigkeit und Flexibilität mit sich. Die blaue Ablaufverfolgung im Bereich AD2 beschreibt diese Teilentscheidungsschritte der ADDs.

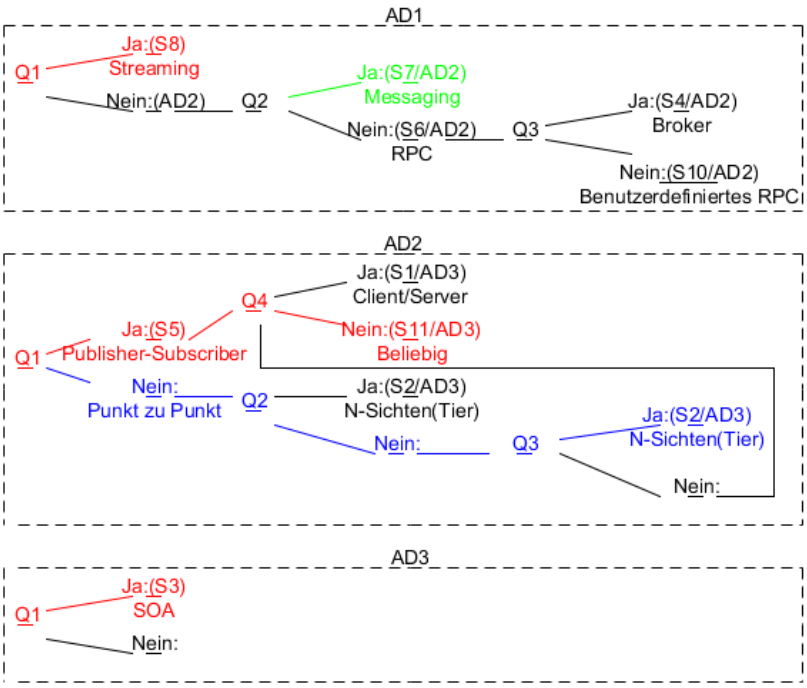

Abbildung 43: Die Entscheidungen in der Baumstruktur für Verteilte Komponenten

4.4.3 Taskverarbeitung

In der entwickelten ADM Taskverarbeitung definierten wir ein Architekturmuster *Pipes-and-Filters*, aber empfahlen, kein *Pipes-and-Filter* für das Universitäts-IS zu verwenden, weil sich keine Reihe der kleineren und unabhängigen Verarbeitungsschritte flexibel verbinden muss, um die größeren

Verarbeitungsaufgaben zu implementieren. Über dieses entwickelte ADM ist eine einzelne Funktion ausreichend für das Universitäts-IS, um jede komplexe Verarbeitungsaufgabe zu implementieren. Und diese Verarbeitungsaufgaben können nicht parallelisiert werden bzw. sie müssen als eine Single-Threaded-Implementierung vorliegen. Die rote Ablaufverfolgung aus Abbildung 44 beschreibt diese Entscheidungsschritte der ADDs.

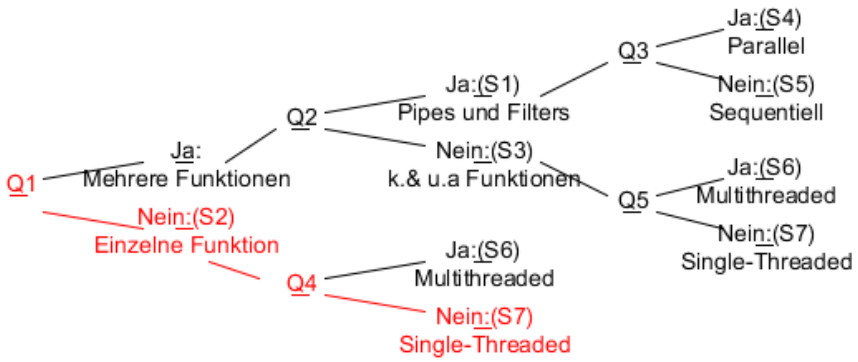

Abbildung 44: Die Entscheidungen in der Baumstruktur für die Taskverarbeitung

4.4.4 UI Design

Auf Basis zweier Gründe empfehlen wir, das Model-View-Controller (MVC) Architekturmuster in dem ADM UI Design für das Universitäts-IS zu verwenden. Der erste Grund ist, dass wir das N-Schichten Architekturmuster in den ADM Verteilten Komponenten für das Universitäts-IS auswählen. Um mit dieser Architekturentscheidung übereinzustimmen, verwenden wir die MVC Architektur im ADM UI Design. Der zweite Grund beruht auf der Beschreibung der Anforderungen A10 und A11 aus Tabelle 15 – wir haben verschiedene Benutzertypen in dem Universitäts-IS, z. B. Universitätsassistenten, Lektoren, Professoren, administratives Personal und Studenten usw. Jeder dieser Benutzertypen braucht seine selbständige und voneinander unabhängige Präsentation (View). Wie im Kapitel 4.2.4 beschrieben, kann die MVC Architektur offensichtlich diese zwei Gründe erfüllen. Sie bietet die Wiederverwendbarkeit, Wartung, Modifizierbarkeit und Testbarkeit für ein Softwaresystem an. Die rote Ablaufverfolgung aus Abbildung 45 beschreibt diese Entscheidungsschritte der ADDs.

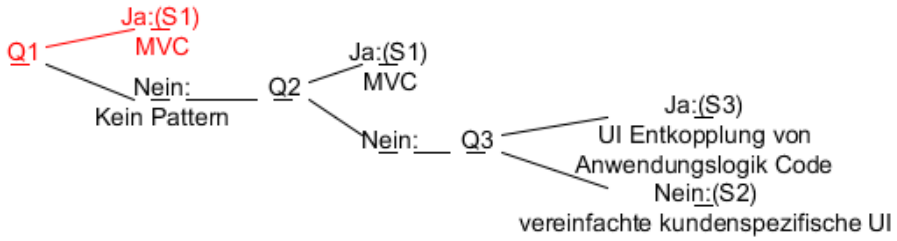

Abbildung 45: Die Entscheidungen in der Baumstruktur für das UI Design

4.4.5 Datenspeicherung

Im Kapitel 4.2.5 definierten wir ein *Shared Repository* Architekturmuster für die Datenspeicherung. Die Kernfragen zur Entscheidung hinsichtlich der Verwendung des *Shared Repository* sind, ob ein System einen zentralen Datenspeicher verwendet, um alle anderen unabhängigen bzw. verteilten Komponenten abzurufen, und ob diese Komponenten die gemeinsamen Daten benötigen. Offensichtlich werden laut Beschreibung der Anforderungen A7 und A12 aus Tabelle 15 die Daten des Universitäts-IS in einem eigenen Server gespeichert, und die verteilten Komponenten und verschiedenen Benutzer benötigen diese gemeinsamen Daten. Die rote Ablaufverfolgung aus Abbildung 46 beschreibt diese Entscheidungsschritte der ADDs.

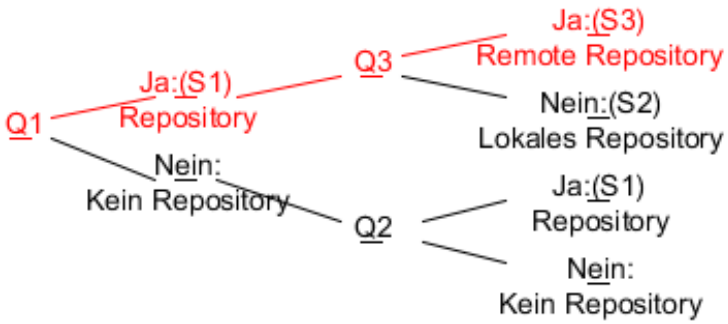

Abbildung 46: Die Entscheidungen in der Baumstruktur für die Datenspeicherung

4.5 Bearbeitung der entschiedenen ADDs

Am Ende dieser Fallstudie und gleichfalls am Ende eines WebADvISE Projekts soll der Softwarearchitekt einige Formulare der entwickelten Architekturentscheidungen als Endergebnisse in dem „Decisions Editor“ ausfüllen, siehe Abbildung 47. Diese

Abbildung zeigt die endliche Architekturentscheidung für die ADM Applikationsstruktur.

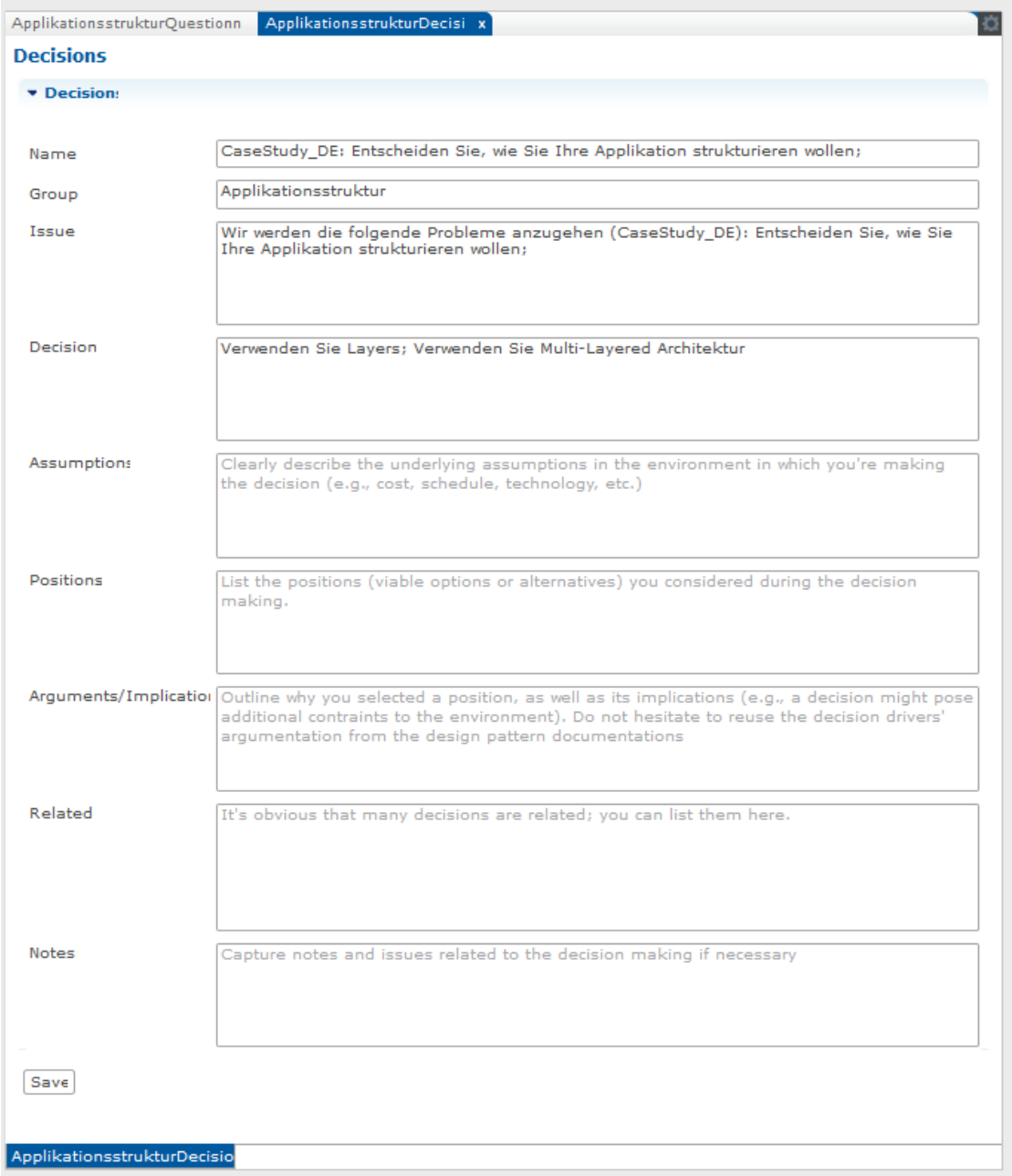

Abbildung 47: Inhalte eines Architekturentscheidungseditors

4.6 Kollaborative Arbeit

Das Universitäts-IS enthält fünf verschiedene ADMs und deren verbundene Fragebogen. Jeder dieser ADMs und Fragebogen kann einzeln mit den verschiedenen

Mitgliedern einer Gruppe zeitgleich bearbeitet werden. Im Allgemeinen können sich die Mitglieder im Zuge eines ganzen Projekts kollaborativ mit dem Universitäts-IS mittels WebADvISE beschäftigen. Jedes Mitglied kann mindestens mit einem ADM oder Fragebogen gleichzeitig arbeiten. In der gleichen Zeitperiode können andere Mitglieder auch einen oder mehrere andere ADMs oder Fragebogen bearbeiten.

Um die Prozesse einer kollaborativen Arbeit deutlich zu erläutern, definieren wir zwei Benutzer, „Han Dian“ (han@dian.com) und „Jack“ (jack@jack.com), die von der E-Mail-Adresse in WebADvISE identifiziert werden und in einer Forschungsgruppe CaseStudy_UIS tätig sind. Diese zwei Benuzter können zeitgleich einzelne ADM Applikationsstrukturen lesen oder auch ändern. Nach einer Änderung wird dieser Änderungszustand bzw. Änderungsstatus automatisch in Log View gespeichert, siehe Abbildung 48. Über die Informationen des Inhalts eines Change Logs siehe Kapitel 3.4.3.

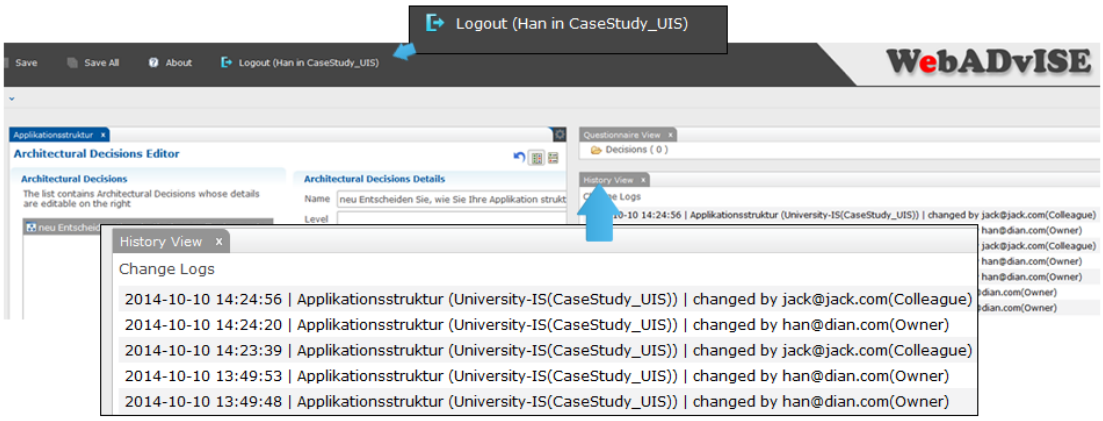

Abbildung 48: Kollaborative Arbeit der zwei Benutzer

Diese zwei Gruppenmitglieder können nicht nur zeitgleich einen Editor bearbeiten, sondern auch ihre Arbeiten durch die Change Logs von Log View beliebig austauschen, um die Konflikte (siehe 3.4.1) bzw. den Verlust aus der Änderung der Ressourcen zu vermeiden. Wird ein Change Log doppelgeklickt, bekommt er die bezogene alte Version von diesem ADM oder Fragebogen mit einer entsprechenden Uhrzeit, siehe Abbildung 49.

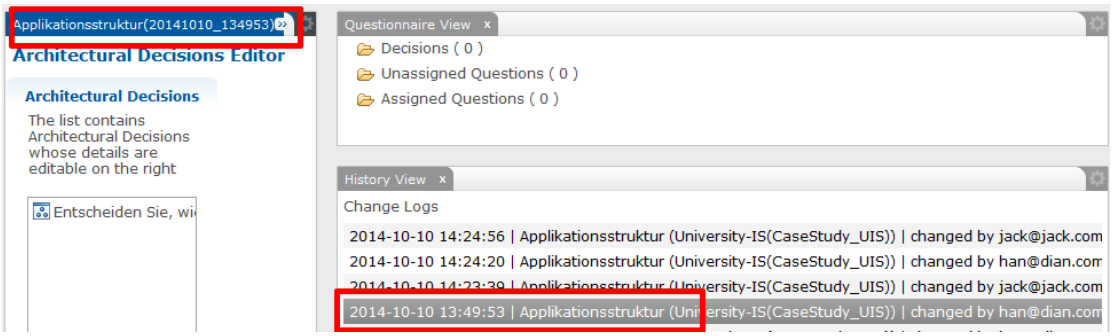

Abbildung 49: Rollback zur Vermeidung der Konflikte (Alte Version wieder editieren)

5. Zusammenfassung

Dieses Kapitel beleuchtet die Ergebnisse der Fallstudie der Universitäts-IS dieser Arbeit in konzeptueller Sicht und fasst sie zusammen. Abschließend werden die zukünftigen Erweiterungen und Entwicklungsmöglichkeiten von WebADvISE beschrieben.

5.1 Ergebnisbetrachtung

Durch die Analyse der Fallstudie Universitäts-IS wird klar erkennbar, wie ein Architekt ein wiederverwendbares Architekturmuster-basiertes ADM entwickeln, wie er dieses entwickelte ADM in WebADvISE erstellen und wie er ADDs für seine Softwarearchitektur (z. B. für Universitäts-IS) mittels des entwickelten ADMs in WebADvISE umsetzen kann. Die Ergebnisse dieser Fallstudie listen wir im Folgenden auf, siehe Tabelle 16. Diese Tabelle beschreibt die entwickelten ADMs und ihre Anforderungen, Lösungen und Architekturmuster für das Universitäts-IS.

ADMs	Anforderungen	Lösungen	Architekturmuster
Applikationsstruktur			
Applikationsstruktur	A1	S3	-
Verteile Komponenten			
Stream Service	A2	S8	*Streaming*
Universitäts-IS	A3,A4,A8	S3	SOA
Kommunikation in IS	A2,A3,A4,A8	S7	*Messing*
Kommunikation mit Forschungskomponenten	A5,A8	S6,S10	RPC
E-Learning Plattform	A3,A4,A5,A6	S5,S11	*Publisher-Subscriber*
Verteilte Systeme des Universitäts-IS	Allgemein	S2	*N-Tier*
Taskverarbeitung			
Verarbeitungsaufgaben	Allgemein	S7	-
UI Design			
UI Design	A10,A11	S1	MVC
Datenspeicherung			
Datenspeicherung	A7,A12	S1,S3	*Shared Repository*
A(x) kommt aus Tabelle 15, S(x) kommt aus Tabelle 10 bis 14			

Tabelle 16: Die Ergebnisse dieser Fallstudie

Am Ende der Fallstudie muss deklariert werden, dass einige entschiedene ADDs der

Ergebnisse nicht zwangsläufig sind, da ein Softwarearchitekt die Architektur seines Systems gemäß den tatsächlichen technischen Möglichkeiten oder technischen Qualifikationen der Softwareentwickler entscheidet. Zum Beispiel kann ein Softwarearchitekt ein anderes Architekturmuster für UI Design auswählen.

5.2 Zukünftige Erweiterungen

Wie bereits beschrieben, bietet WebADvISE einen Mechanismus zur Realisierung der Funktionalität der Kollaboration an, und zwar die Atomisierung der Änderungszustände und ein Rollback, siehe Kapitel 3.4.2. Das grundsätzliche Prinzip besteht darin, dass jede einzelne Änderung einer Ressource bzw. eines Editors zurückgerollt werden kann. Dies ist nur eine Möglichkeit der Kollaboration. Es kann natürlich ein anderer Mechanismus verwendet werden, um eine Kollaboration zu implementieren, weil die Kollaboration keine bestimmte Definition aufweist. Sie umfasst mehrere Bedeutungen.

Als Beispiel wird der *Locking*-Mechanismus betrachtet. Jeder Editor der fünf entwickelten ADMs wird nur von einem Mitglied in einer bestimmten Zeitperiode bearbeitet werden. Andere Mitglieder dürfen es in dieser Zeitperiode nicht bearbeiten, aber sie können andere ADMs dieses Projekts bearbeiten. Aus technischer Sicht kann der Softwareentwickler einen Integer-Wert *isLocked* als Status des Editors von diesem ADM ermitteln und legt diesen Integer-Wert in MySQL ab. Der ursprüngliche Wert von *isLocked* ist 0. Dann wird ein *DoubleClick Listener* für jedes ADM in ADD View erzeugt, um die Aktionen der Mitglieder zu beachten. Wenn das erste Mitglied einen ADM oder einen Fragebogen öffnet, wird der Integer-Wert *isLocked* für dieses ADM oder diesen Fragebogen auf 1 gesetzt. Dies bedeutet, ein Editor von dem ADM und Fragebogen wird von diesen Mitgliedern derzeitig bearbeitet und andere Mitglieder dürfen ihn nicht bearbeiten. Nur wenn dieser Editor abgeschlossen wird, muss *isLocked* wieder auf 0 gesetzt werden. Das heißt, jedes Mitglied kann ihn wieder öffnen. Dieser Locking-Mechanismus wurde in einer frühen Version von WebADvISE verwendet. In der Zukunft können geeignetere Mechanismen der Kollaboration in

WebADvISE verwendet werden.

Andererseits können in zukünftigen Versionen mehrere Funktionen der Kollaboration implementiert werden, zum Beispiel: Erstens, es wird ein Chat View eingefügt, in ihm können die Mitglieder zeitgleich miteinander über das Projekt diskutieren. Zweitens, Anzeigen des Login-Status anderer Mitglieder. Drittens, der Gruppenbesitzer kann eine Arbeitsliste für alle Mitglieder in eine Gruppe anfertigen. Viertens, jede einzelne Änderung wird mit einer anderen Farbe für verschiedene Mitglieder markiert. Fünftens, ein neuer Benutzertyp (Nur-Lese-Benutzer) wird erstellt. Dieser Benutzer kann in einem Editor nicht schreiben, sondern ihn nur lesen usw. Neben diesen Erweiterungen gibt es eine Vielzahl kleiner Änderungen, die WebADvISE verbessern lassen.

6. Anhang

Universitäts-Informationssystem [UIS, 2013]

- **Beschreibung (Description)**

In this exercise you will need to make high-level architectural decisions for a new University Information System (IS) you will have to design for your University that will replace an old legacy system which has been very slow and cannot support new requirements any more. The University IS will serve the central administrative needs of the university but it will also communicate with and provide interfaces to external applications. At this stage you will only need to make and document key architectural decisions, for instance, which architectural style(s) will be used, how you will perform data handling, how you will deal with communication issues, etc. In order to make these architectural decisions you will need to consider architectural design patterns that you learned this semester at the SWA lecture but also other solutions that you consider appropriate for your application. For this, we provide you some requirements that you need to consider for making your architectural decisions. Read carefully the following requirements before you start making and documenting your architectural decisions.

- **Anforderungen (Requirements)**

Requirements	Description
General Requirements	The new University IS has to be scalable, reliable, and provide high performance. The various components that will be developed inside the University IS have to be organized at the same level of abstraction. Standardized service interfaces will be provided for internal components and external systems as well. The application components will be deployed onto different machines (the data will be also stored in a separate server) in order to achieve high scalability.
Student Services	Students will be able to enroll for their studies electronically, register for courses and exams, download their student records, and view information about the courses they have registered to. The University IS will provide a service to the students for generating

	reports about their marks. In particular, it will provide a list of the accomplished (graded) courses, pending courses; it will also calculate the average, the points that are missing for the next best mark, and finally the percentage of the students that have had better marks than him/her in each course. Through the University IS the students will have access to various e-learning platforms that are provided by the University Institutes. The University IS should be able to get updates automatically by the various e-learning platforms (e.g., about the lectures' schedule). You will need to design a web UI common for all students that has to be easily testable, modifiable and decoupled from the application functionality.
Services for Research and Administrative Staff	The university assistants, lecturers, professors, as well as the administrative staff should have a different user interface where they will be able to create courses, grade the students, view students that are registered for specific courses, etc. You should have this in mind when you create the UI of the University IS.
Library Service	The University IS provides a library catalog where users can view available books, journals, etc. and reserve them for one day if they want to pick them up from the University Library. Other components that calculate some statistics and update the library database with new online publications should also have remote access to this library catalog.
Stream Service	The University IS shall allow recording and streaming of lectures in lecture halls; streams are kept on a central server and can be downloaded on demand. Before they are downloaded, the client can request the transformation in different audio and video formats.
Research Network	The University is collaborating with a central research system for establishing networks between researchers, dissemination of research results, access to publications, etc. The University IS should be able to send updates of new publications and research projects of its research staff. In this case reliability and performance are not very important. The Research Network uses only the XML-RPC protocol for communication with external systems.

7. Literaturverzeichnis

[MacLean et al., 1991] MacLean A., Young R., Bellotti V., and Moran T., *Questions Options, and Criteria: Elements of Design Space Analysis*, Human-Computer Interaction, Volume 6, 1991, S. 201-250.

[Perry and Wolf, 1992] Perry D. E., Wolf. A. L., *Foundations for the study of software architecture*, ACM SIGSOFT Software Engineering Notes, 17 no 4, 1992, S.40–52.

[Kazman et al., 1994] Kazman R., Bass L., Abowd G., Webb M., *SAAM: A Method for Analyzing the Properties of Software Architectures*, Proceedings of ICSE 16, Sorrento, Italy, May 1994, S. 81-90.

[Kruchten, 1995] Kruchten P., *The 4+1 View Model of Architecture*, IEEE Software, vol. 12, no. 6, 1995.

[Buschmann et al., 1996a] Buschmann F., Helm R., Johnson R., Vlissides H., *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1996, S. 31-51.

[Buschmann et al., 1996b] Buschmann F., Helm R., Johnson R., Vlissides H., *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1996, S. 53-70.

[Buschmann et al., 1996c] Buschmann F., Helm R., Johnson R., Vlissides H., *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1996, S. 99-122.

[Buschmann et al., 1996d] Buschmann F., Helm R., Johnson R., Vlissides H., *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1996, S. 125-143.

[Buschmann et al., 1996e] Buschmann F., Helm R., Johnson R., Vlissides H., *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1996, S. 339-343.

[Lee and Lai, 1996] Lee J., Lai K.-Y., *What's in design rationale? in Design rationale: concepts, techniques, and use*, Lawrence Erlbaum Associates, Inc., 1996, S.21-51.

[Polanyi, 1996] Polanyi M., *The Tacit Dimension*, Routledge & Kegan Paul, London, 1996.

[Lee, 1997] Lee J., *Design Rationale Systems: Understanding the Issues*, IEEE Expert, Vol. 12 no. 3, 1997, S.78-85.

[Bengtsson und Bosch, 1998] Bengtsson P., Bosch J., *Scenario Based Software Architecture Reengineering*, 5th International Conference of Software Reuse (ICSR), 1998.

[Kyaruzi and Katwijk, 1999] Kyaruzi J.K., van Katwijk J., *Beyond Components-Connections-Constraints: Dealing with Software Architecture Difficulties*, Proc. 14th IEEE Int'l Conf. Automated Software Eng., IEEE Press, 1999, S. 235–242.

[ISO, 2000] International Standard ISO 9126-1: *Information technology:*

software product evaluation–quality characteristics and guidelines for the their use, International Organization for Standardization, Geneva, 2000.

[Kazman et al., 2000] Kazman R., Klein M., Clements P., *ATAM: Method for Architecture Evaluation. Techical Report,* CMU/SEI-2000-TR-004, ESC-TR-2000-004, August, 2000.

[Kazman et al., 2001] Kazman R., Asundi J., Klein M., *Quantifying the Costs and Benefits of Architecture Decisions*, Proceedings of the 23rd International Conference on Software Engineering (ICSE 23), (Toronto, Canada), May, 2001, S. 297-306.

[Clements, 2002] Clements P., Bachmann F., Bass L., Garlan D., Ivers J., Little R., Nord R., Stafford J., *Documenting Software Architectures: Views and Beyond*, Pearson Education, February, 2002.

[Svahnberg et al., 2002] Svahnberg M., Wohlin C., Lundberg L., Mattsson M., *A Method for Understanding Quality Attributes in Software Architecture Structures*, SEKE'02, July, 2002, S. 15-19.

[Ionita et al., 2002] Ionita M.T., Hammer D.K., Obbink H., *Scenario-based software architecture evaluation methods: An overview*, ICSE/SARA, 2002.

[ISO, 2003] International Standard ISO 9126-2, 3: *Information technology: software product evaluation–external and internal metrics*, International Organization for

Standardization, Geneva, 2003.

[Nord et al., 2003] Nord R.L., Barbacci M.R., Clements P., Kazman R., Klein M., O'Brien L., Tomayko J.E., *Integrating the Architecture Tradeoff Analysis Method (ATAM) with the Cost Benefit Analysis Method (CBAM)*, Architecture Tradeoff Analysis Initiative, CMU/SEI-2003-TN-038, December 2003.

[Bosch, 2004] Bosch J., *Software Architecture: The Next Step*, EWSA, 2004, S. 194-199.

[Kruchten, 2004] Kruchten P., *An ontology of architectural design decisions in software intensive systems*. In 2nd Groningen Workshop on Software Variability, December, 2004, S. 54–61.

[Jansen and Bosch, 2004] Jansen A., Bosch J., *Evaluation of tool support for architectural evolution*, In Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE), IEEE, September 2004, S.375–378.

[Zhuge, 2004] Zhuge H., *The Knowledge Grid*. World Scientific Publishing Company, 2004.

[Al-Naeem et al., 2005] Al-Naeem T., Gorton I., Babar M.A., Rabhi F., Benatallah B., *A quality-driven systematic approach for architecting distributed software applications*, Proceedings of the 27th International Conference on Software Engineering (ICSE), St. Louis, USA, 2005.

[Avgeriou and Zdun, 2005] Avgeriou P., Zdun U., *Architectural Patterns Revisited—A Pattern Language*, Proc. 10th European Conf. Pattern Languages of Programs (EuroPLoP), UVK Konstanz, 2005, S. 431–470.

[Babar et al., 2005] Babar M.A., Gorton I., Jeffery R., *Toward a Framework for Capturing and Using Architecture Design Knowledge. School of Computer Science & Engineering*, University of New South Wales, Australia and National ICT Australia Ltd. UNSW-CSE-TR- 0513, June 2005.

[Bachmann et al., 2005] Bachmann F., Bass L., Klein M. and Shelton C., *Designing software architectures to achieve quality attribute requirements*, IEE Proc.-Software., Vol. 152 no. 4, 2005.

[Dueñas and Capilla, 2005] Dueñas, J.C., Capilla, R., *The Decision View of Software Architecture*, Proceedings of the 2nd European Workshop on Software Architecture (EWSA 2005), Springer-Verlag, LNCS 3047, 2005, S. 222-230.

[Jansen and Bosch, 2005] Jansen, A., Bosch, J. *Software Architecture as a Set of Architectural Design Decisions*, 5th IEEE/IFIP Working Conference on Software Architecture, 2005.

[Rozanski and Capilla, 2005] Rozanski N., Woods E., *Software Systems Architecture*, Addison-Wesley, 2005.

[Tyree and Akerman, 2005] Tyree J., Akerman J., *Architecture decisions: Demystifying architecture*. IEEE Software, 22 no 2, 2005,

S. 19–27.

[Vliet, 2005] van Vliet H., *The GRIFFIN project, A GRId For inFormatIoN about architectural knowledge*, Vrije Universiteit Amsterdam, April 16, 2005.

[Babar et al., 2006] Babar, M.A., Gorton, I., and Kitchenham, B., *A Framework for Supporting Architecture Knowledge and Rationale Management, In Rationale Management in Software Engineering*, A.H. Dutoit, et al., eds, Springer, 2006, S. 237-254.

[Kruchten et al., 2006] Kruchten P., Lago P., van Vliet H., *Building Up and Reasoning about Architectural Knowledge*, Proc. 2nd Int'l Conf. Quality of Software Architecture, (QOSA 06), Springer, 2006, S. 42–58.

[Tang et al., 2006] Tang, A., Jin, Y., and Han, J., „*A Rationale-based Architecture Model for Design Traceability and Reasoning, Journal of Systems and Software*", 80 no 6, 2006, S.918-934.

[Ven et al., 2006] van der Ven J. S., J. Jansen A. G., Avgeriou P., Hammer D. K., *Using architectural decisions. In Second International Conference on the Quality of Software Architecture*, K. Hammer University of Groningen, Department of Mathematics and Computing Science, PO Box 800, 9700AV Groningen, The Netherlands, 2006.

[Zdun and Dustdar, 2006] Zdun U., Dustdar S., *Model-Driven and Pattern-Based Integration of Process-Driven SOA Models*, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Distributed Systems Group, Information Systems Institute, Austria, 2006.

[Zimmermann et al., 2006] Zimmermann O., Koehler J., Leymann F., *The Role of Architectural Decisions in Model-Driven SOA Construction*, Portland, ACM, 2006.

[Babar and Gorton, 2007] Babar M. A., Gorton I., *A Tool for Managing Software Architecture Knowledge*, IEEE Computer Society Washington, DC, USA, 2007.

[Boer et al., 2007] de Boer, R.C., Farenhorst, R., Lago, P., Vliet , H. v., Clerc, V., and Jansen, A., *Architectural knowledge: Getting to the Core*, In the Quality of Software-Architectures (QoSA), Boston, USA, 2007, S. 197-214.

[Capilla, 2007] Capilla R., Nava F., Dueñas J.C., *Modeling and Documenting the Evolution of Architectural Design Decisions*, Proc. 2nd Workshop Sharing and Reusing Architectural Knowledge Architecture, Rationale, and Design Intent, IEEE CS Press, 2007, S. 9.

[Harrison et al., 2007] Harrison N., Avgeriou P., Zdun U., *Using patterns to capture architectural decisions*, IEEE Software, July/Aug. 2007, S. 38–45.

[Jansen, 2007] Jansen A. Ven J., Avegriou P., Hammer D. K., *Tool support for Architectural Decisions*, IEEE, 2007.

[Lee and Kruchten, 2007] Lee L., Kruchten P., *Capturing Software Architectural Design Decisions*, 2007.

[Schuster, 2007] Schuster, N., *ADkwik – a Collaborative System for Architectural Decision Modeling and Decision Process Support based on Web 2.0 Technologies*, Master thesis, Hochschule der Medien, 2007.

[Schuster et al., 2007] Schuster N., Zimmermann O., Pautasso C., *ADkwik: Web 2.0 Collaboration System for Architectural Decision Engineering, Nineteenth International Conference on Software Engineering and Knowledge Engineering*, (SEKE 2007), Boston, USA, 2007, S. 255-260.

[Xu et al., 2007] Xu L.H., Richardson D.J., Ziv H., *A Survey of Software Architecture Decision-Making Techniques*, Institute for Software Research, ISR Technical Report UCI-ISR-07-10, December, 2007.

[Zimmermann et al., 2007] Zimmermann O., Gschwind T., Küster J., Leymann F., Schuster N., *Reusable Architectural Decision Models for Enterprise Application Development*, 2007.

[Zimmermann et al., 2008] Zimmermann O., Zdun U., Gschwind T., Leymann F., *Combining Pattern Languages and Reusable Architectural Decision Models into a Comprehensive and*

Comprehensible Design Method, in Proc. 7th Work. IEEE/IFIP Conf. Softw. Architecture, IEEE, 2008, S. 157–166.

[Capilla et al., 2008] Capilla R., Nava F., Montes J., Carrillo C., *ADDSS: Architecture Design Decision Support System Tool, Automated Software Engineering*, ASE 2008. 23rd IEEE/ACM International Conference on, September 2008, S. 15-19, 487 – 488,

[Lee, 2008] Lee, L., and Kruchten, P., *A Tool to Visualize Architectural Design Decisions, Proceedings of the 4th International Conference on Quality of Software-Architectures: Models and Architectures (QoSA)*, Springer, 2008, S. 359-362.

[Kruchten et al., 2009] Kruhten P., Capilla R., Duenas J. C., *The Decision View's Role in Software Architecture Practice*, Software, IEEE, Volume 26 , Issue: 2, 2009.

[Lee, 2009] Lee L., *Tool-based capture and exploration of software architectural design decisions,* Master of Applied Science – MASc, 2009.

[Liang et al., 2009] Liang, P., Jansen, A., Avgeriou, P., *Knowledge Architect: A Tool Suite for Managing Software Architecture Knowledge*, Technical Report RUG-SEARCH-09-L01, University of Groningen, February 2009.

[Shahin et al., 2009a] Shahin Mo., Liang P., Khayyambashi M. R., *Architectural*

Design Decision: Existing Models and Tools, Software Architecture, 2009 & European Conference on Software Architecture, WICSA/ECSA 2009, Joint Working IEEE/IFIP, September 2009, S. 293 - 296.

[Shahin et al., 2009b] Shahin M., Liang P., Khayyambashi M.R., *A Survey of Architectural Design Decision Models and Tools*, Technical Report SBU-RUG-2009-SL-01, Sheikh Bahaei University & University of Groningen, 2009.

[Silberschatz et al., 2009] Silberschatz A., Galvin P.B., Gagne G., *Operating System Concepts*, 8th Edition. John Wiley & Sons, Inc. ISBN: 978-0-470-12872-5, 2009.

[Stephanson et al., 2009] Stephanson M.R., C.J.Brown, *Benefit-Cost Analysis: General Methods and Approach*, Puget Sound Regional Council (PSRC), 2009.

[Ameller et al., 2010] D. Ameller, Franch X., Cabot J., *Dealing with Non-Functional Requirements in Model-Driven Development*, in RE, Universitat Politècnica de Catalunya Barcelona, Spain, 2010.

[Tang et al., 2010] Tang A., Avgeriou P., Jansen A., Capilla R., Babar, M.A., *A comparative study of architecture knowledge management tools*. Journal of Systems and Software, 83 (3), ISSN 0164-1212, 2010, S. 352-370.

[Xu et al., 2010] Xu B.f., Huang Z.Q., Wei O., *Making Architectural*

Decisions Based on Requirements: Analysis and Combination of Risk-Based and Quality Attribute-Based Methods, IEEE, 2010.

[Zalewski et al., 2011] Zalewski A., Kijas S., Sokolowska D., *Capturing Architecture Evolution with Maps of Architectural Decisions 2.0*, Warsaw University of Technology, Institute of Automatic Control and Computational Engineering, 2011.

[Ameller et al., 2012] Ameller D., Collell O., Franch X., *ArchiTech: Tool Support for NFR-Guided Architectural Decision-Making*, Software Engineering for Information Systems research group (GESSI), IEEE, 2012.

[Dermeval et al., 2012] Dermeval D., Pimentel J., Silva1 C., Castro J., Santos E., Guedes G., Lucena M., Finkelstein A., *STREAM-ADD – Supporting the Documentation of Architectural Design Decisions in an Architecture Derivation Process*, 2012 IEEE 36th International Conference on Computer Software and Applications, 2012.

[Graaf et al., 2012] de Graaf K.A. et al., *Ontology-based Software Architecture Documentation*, In: 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, IEEE Computer Society, 2012, S. 121–130.

[Lytra et al., 2012] Lytra I., Sobernig S., Zdun U., *Architectural Decision*

Making for Service-Based Platform Integration: A Qualitative Multi-Method Study, Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP, 2012, S. 111-120.

[Lytra and Zdun, 2013] Lytra I., Zdun U., *Supporting Architectural Decision Making for Systems-of-Systems Design under Uncertainty*, Faculty of Computer Science University of Vienna, Austria, 2013.

[UIS, 2013] Software Architecture Group (SAG). Treffen und Dokumentieren von Architekturentscheidungen - Universitäts-Informationssystem, Faculty of Computer Science, University of Vienna, 2013.

[ADvISE, 2014] ADvISE, Architectural Design Decision Support Framework (ADvISE), https://swa.univie.ac.at/Architectural_Design_Decision_Support_Framework_(ADvISE)#ADvISE_2, [Zugriff am 23.10.2014].

[RAP, 2014] The Eclipse Foundation, https://eclipse.org/rap/, [Zugriff am 23.10.2014].

[Microsoft, 2014] Microsoft. The Repository Pattern. http://msdn.microsoft.com/en-us/library/ff649690.aspx, [Zugriff am 07.09.2014].