Fachhochschule Köln
University of Applied Sciences Cologne
Fakultät für
Informatik und Ingenieurwissenschaften

Master Thesis

Konzeption, prototypische Realisierung und szenariobasierte Validierung einer dienstorientierten Multimediaarchitektur

ausgearbeitet von

Dirk Breuer, 11038920

Medieninformatik Master

zur Erlangung des akademischen Grades

Master of Science

vorgelegt an der

Fachhochschule Köln

University of Applied Sciences Cologne

Fakultät für Informatik und

Ingenieurwissenschaften

im Studiengang

Medieninformatik (Master)

Erster Prüfer: Prof. Dr. Mario Winter
Fachhochschule Köln

Zweiter Prüfer: Prof. Dr. Kristian Fischer
Fachhochschule Köln

Gummersbach, im Januar 2009

Adressen: Dirk Breuer
Redwitzstr. 6
50937 Köln
dirk.breuer@gmail.com

Prof. Dr. Mario Winter
Fachhochschule Köln
Institut für Informatik
Steinmüllerallee 1
51643 Gummersbach
mario.winter@fh-koeln.de

Prof. Dr. Kristian Fischer
Fachhochschule Köln
Institut für Informatik
Steinmüllerallee 1
51643 Gummersbach
kristian.fischer@fh-koeln.de

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Abkürzungsverzeichnis

ACID	Atomicity, Consistency, Isolation, Durability
AOP	Aspektorientierte Programmierung
API	Application Programming Interface
ATAM	Architecture Tradeoff Analysis Method
BPEL	Business Process Execution Language
BPM	Business Process Management
COMM	Core Ontology for MultiMedia
CORBA	Common Object Request Broker Architecture
COSIMA	Cologne Service-Oriented Integrated Multimedia Architecture
CRUD	Create, Read, Update, Delete
CSCL	Computer Supported Collaborative Learning
CSCW	Computer Supported Cooperative Work
EAI	Enterprise Application Integration
ebXML	Electronic Business using eXtensible Markup Language
ESB	Enterprise Service Bus
HTTP	Hypertext Transfer Protocol
JMF	Java Media Framework
JMS	Java Message Service
JNDI	Java Naming and Directory Interface
LDU	Logical Data Units
MCU	Multipoint Control Unit
MEP	Message Exchange Pattern
MIAV	Modellierung in audio-visuellen Medien
MPEG-7	The Multimedia Content Description Interface
NFS	Network File System
OASIS	Organization for the Advancement of Structured Information Standards
OOP	Objektorientierte Programmierung
ORB	Object Request Broker
QoS	Quality of Service
REST	Representational State Transfer

SAAM ..... Software Architecture Analysis Method
SDP ..... Session Description Protocol
SEI ..... Software Engineering Institute
SOA ..... Service-Oriented Architecture
SOAP ..... Simple Object Access Protocol
UDDI ..... Universal Description Discovery and Integration
URI ..... Uniform Resource Identifier
XML ..... eXtensible Markup Language
YAML ..... YAML Ain't Markup Language

1 Einleitung

Im Titel der vorliegenden Arbeit wird bereits herausgestellt, dass eine Architektur von der Konzeption über die prototypische Implementierung bis hin zu einer ersten Validierung betrachtet wird. Dabei soll die Architektur dienstorientiert aufgebaut sein und sich für die Realisierung von Multimediaanwendungen eignen.

In dieser Arbeit wird dazu zunächst allgemein in die Thematik der Dienstorientierung eingeführt und die jeweiligen Besonderheiten im Zusammenhang mit Multimediaanwendungen dargelegt. Im weiteren Verlauf wird die Architektur prototypisch umgesetzt und auf Grund der Implementierung eines Anwendungsszenarios validiert.

1.1 Motivation

Die Motivation zu dieser Arbeit ist durch das COSIMA-Projekt begründet, welches im Rahmen des Wahlpflichtfaches *Modellierung in audio-visuellen Medien* an der Fachhochschule Köln entstanden ist. Das COSIMA-Projekt soll eine verteilte, dienstorientierte Architektur zur Realisierung von Multimediaanwendungen zum Ergebnis haben.

Da bis zu diesem Zeitpunkt ausschließlich konzeptionell an diesem Projekt gearbeitet wurde, bestand der dringende Bedarf einer ersten prototypischen Implementierung und eine anschließende Validierung des bis dato Konzipierten durchzuführen. Da COSIMA als langfristiges Projekt ausgelegt ist, ist es zu diesem Zeitpunkt notwendig, eine erste Überprüfung durchzuführen, bevor weitere konzeptionelle Arbeiten vorgenommen werden.

Das szenariobasierte Vorgehen, dass der Validierung zu Grunde liegt, wurde motiviert durch die Neuartigkeit des Projekts und der damit verbundenen Unerfahrenheit bei der Konstruktion entsprechender Anwendungen. Der Einsatz von Szenarien soll helfen, diese Unerfahrenheit zu einem gewissen Grad zu kompensieren.

1.2 Zielsetzung und Aufgabenstellung

Diese Arbeit verfolgt zwei wesentliche Ziele, von denen sich eines direkt aus dem Titel ableiten lässt, das andere ergibt sich aus dem COSIMA-Projekt selbst:

1. Es soll überprüft werden, inwiefern sich die bis zu diesem Zeitpunkt konzipierte Architektur von COSIMA für den Einsatz als dienstorientierte Architektur für Multimediaanwendungen eignet.
2. Das zweite Ziel ist, dass die Ergebnisse dieser Arbeit, vor allem die prototypische Implementierung, einen Rahmen für weitere Arbeiten geben sollen.

Diese Ziele werden operationalisiert und ergeben damit konkrete Aufgaben, die es im Rahmen dieser Arbeit zu erfüllen gilt. Die Erledigung der einzelnen Aufgaben hat dementsprechend die Zielerreichung zur Folge. Am Ende dieser Arbeit kann jedoch keine binäre Aussage über die Vollständigkeit oder Korrektheit der durchgeführten Implementierung getroffen werden, denn auch Teilergebnisse müssen dabei berücksichtigt bleiben.

Zunächst ist es notwendig die konzipierte Architektur und ihre wesentlichen Komponenten im Einzelnen vorzustellen und zu diskutieren. Daraus lassen sich die einzelnen Elemente ableiten, die später implementiert werden müssen.

Um überhaupt eine Aussage über die Eignung der Architektur für die Umsetzung von Multimediaanwendungen treffen zu können, muss eine entsprechende Anwendung realisiert werden. Zu diesem Zwecke wird zunächst ein Szenario entwickelt, in dem diese Anwendung eingebettet wird. Zur Realisierung dieser Anwendung muss demzufolge zunächst die Architektur selbst implementiert werden.

Am Ende wird somit ein Prototyp der Architektur entstanden und das Szenario entsprechend umgesetzt sein. An diesem Punkt lässt sich feststellen, ob sich die Architektur für den geplanten Einsatz eignet oder nicht.

Für die Erreichung des zweiten Ziels ist vor allem notwendig, dass die Implementierung so ausgeführt und dokumentiert wird, dass nachfolgende Arbeiten nach kurzer Einarbeitungszeit darauf aufsetzen können. Das Szenario sollte so breit ausgelegt sein, dass es ebenfalls für weitere Arbeiten als Grundlage dienen kann.

1.3 Abgrenzung

In der vorliegenden Arbeit soll das COSIMA-Projekt und dessen Architektur horizontal betrachtet und implementiert werden. Es fand keine detaillierte Einzelbetrachtung bestimmter Charakteristika, wie etwa *Synchronisation* oder *Streaming* statt. Aus diesem Grund ist das Ergebnis auch nicht eine vollständige Umsetzung der Gesamtarchitektur, wohl wurde aber eine solide Implementierung geschaffen, die es weiterführenden Projekten erlaubt, bestimmte Bereiche vertikal zu implementieren. Ebenso konnten nicht-funktionale Aspekte wie etwa *Skalierbarkeit* oder *Wartbarkeit* keine Berücksichtigung finden.

1.4 Aufbau der Arbeit

Zunächst wird in das COSIMA-Projekt eingeführt und die Konzeption der zugrunde liegenden Architektur vorgestellt (→ Kapitel 2). In diesem Zuge werden ebenfalls die zentralen Begriffe *Architektur*, *Dienstorientierung* und *Multimedia* definiert.

Im Anschluss daran wird eine Einführung in szenarienbasiertes Vorgehen im Allgemeinen gegeben und im Speziellen welche Methode in dieser Arbeit verwendet wurden. Des Weiteren wird das Szenario vorgestellt, gegen das die prototypische Realisierung validiert werden soll (→ Kapitel 3).

Im nachfolgenden Kapitel werden sowohl die prototypische Implementierung der Architektur von COSIMA selbst, als auch die Umsetzung des zuvor beschriebenen Szenarios auf Basis dieser Architektur (→ Kapitel 4) diskutiert.

Die Ergebnisse und Erkenntnisse, die sich aus der Validierung der beiden Implementierungen ergeben haben, finden sich in Kapitel 5. Hier findet zudem eine Einordnung des Begriffs *Validierung* statt.

Die Arbeit schließt mit einer Zusammenfassung und kritischen Bewertung der wesentlichen Aspekte. Ebenso wird an dieser Stelle ein Ausblick gegeben, welche nächsten Schritte als sinnvoll zu erachten sind, um das COSIMA-Projekt voranzutreiben (→ Kapitel 6).

2 COSIMA - Eine dienstorientierte Multimediaarchitektur

Im Abschnitt 1.1 wurde bereits kurz darauf eingegangen, dass die vorliegende Arbeit im Rahmen des COSIMA-Projekts entstanden ist. In diesem Kapitel soll dieses Projekt so weit vorgestellt werden, so dass für den weiteren Verlauf der Arbeit ein grundlegendes Verständnis über die Ziele, Alleinstellungsmerkmale und Herausforderungen existiert.

2.1 Motivation

Das COSIMA-Projekt ist aus dem Wahlpflichtfach *Modellierung in audio-visuellen Medien* (MIAV) an der Fachhochschule Köln im Masterstudiengang der Medieninformatik hervorgegangen. Im Rahmen einer Projektarbeit wurde die Projektidee weiter ausgearbeitet und konzipiert. Die Ergebnisse dieser Arbeit wurden als Institutsbericht an der Fachhochschule Köln bereitsgestellt und sind dort im Detail einsehbar [Breuer u. a. 2008].

Das folgende Kapitel wird daher nur auf die wesentliche Punkte des COSIMA-Projekts eingehen und ihre Relevanz für diese Arbeit herausstellen. Die ursprüngliche Idee hinter COSIMA bestand darin ein Rahmenwerk zu entwickeln, dass die Entwicklung von Multimediaanwendungen vereinfacht. Im Gegensatz zu anderen Medienframeworks, wie etwa dem *Java Media Framework*[1] (JMF), wurde bei dem COSIMA-Projekt dabei aber ein ganzheitlicher Ansatz verfolgt[2].

Im Institutsbericht wird darauf hingewiesen, „dass die Entwicklung von Multimediaanwendungen derzeit verhältnismässig aufwendig ist“ [Breuer u. a. 2008, S. 2]. Eine Ursache

[1] `http://java.sun.com/javase/technologies/desktop/media/jmf/`

[2] Laut der offiziellen FAQ von Sun spezifiziert das JMF eine „einfache, vereinheitlichte Architektur zur Synchronisation und Kontrolle von Audio, Video oder anderen zeitbasierten Daten innerhalb von Java Applikationen oder Applets.“ Durch die JMF 2.0 API wird Verhalten zum Abspielen, Aufnehmen, Übertragen und Transkodieren von Daten spezifiziert [Sun Microsystems].

dieser Problematik liegt nach Aussage der Autoren darin begründet, dass sich die zur Zeit verfügbaren Rahmenwerke im Bereich der Multimediaverarbeitung auf einen sehr engen Einsatzbereich[3] beschränken. Neben JMF sind hier zusätzlich noch *QuickTime*[4] und *ImageJ*[5] zu nennen. Andere Aspekte von Multimediaanwendungen, wie etwa die Integration von Metadaten, müssten von dem Anwendungsentwickler erst manuell mit diesen Rahmenwerken integriert werden. „Ein Meta-Framework, welches die bestehenden Ansätze verbinden und integrieren könnte, würde die Wiederverwendbarkeit und generelle Entwicklungsarbeit positiv beeinflussen, beziehungsweise vereinfachen“ [Breuer u. a. 2008, S. 3]. Daher wird von den Autoren des Berichts die Entwicklung eines solchen Rahmenwerks als Bestreben hinter dem COSIMA-Projekt angeführt.

Neben der Notwendigkeit ein *Meta-Framework*[6] zu schaffen, führen die Autoren als weiteren Beweggrund das Fehlen einer Architektur für Multimediaanwendungen an. Innerhalb dieser Architektur könnten sich Anwendungsentwickler wesentlich effektiver bewegen und müssten nicht erst eine eigene Architektur von Grund auf entwerfen.

Da sich mit bestehenden Multimedia-Rahmenwerken keine verteilten Anwendungen realisieren lassen, lag auch dieser Aspekt von Beginn an im Fokus der Konzeptionierung. Als Grundlage eine geeignete Architektur zu konzipieren, die es ermöglicht, verteilte Anwendungen zu realisieren, diente das Konzept der *Service-oriented Architecture* (SOA) oder *dienstorientierten Architektur*.

Die hier aufgeführten Punkte haben initial die Entwicklung eines Rahmenwerkes motiviert, das später im COSIMA-Projekt aufgehen sollte. Die im Verlauf der Projektarbeit entwickelten Ziele von COSIMA sind im nächsten Abschnitt zusammengefasst.

2.2 Ziele

Das *Mission Statement* des COSIMA-Projekts fasst bereits alle Ziele des Projekts in einer Kernaussage zusammen:

[3] JMF beispielsweise bleibt auf der Protokollebene und stellt keine Integration von Metadaten bereit.
[4] `http://www.apple.com/quicktime/`
[5] `http://rsbweb.nih.gov/ij/`
[6] Das ursprüngliche Ziel war tatsächlich ein reines Framework zu schaffen. Im Verlauf dieser Arbeit hat sich diese Wahrnehmung jedoch verschoben, was in Abschnitt 2.5.1 noch weiter diskutiert wird.

> *"[COSIMA] ist ein integratives, komponentenbasiertes Meta-Framework mit gezielter Ausrichtung auf Multimediaverarbeitung. Es vereinfacht die Entwicklung von verteilten Multimedia-Applikationen durch eine flexible, dienstorientierte Architektur. Die Wiederverwendbarkeit von Komponenten und bestehenden Frameworks wird dadurch begünstigt."* (aus [Breuer u. a. 2008, S. 2])

Neben den zentralen Aspekten *dienstorientierte Architektur*, *Integration* und *Meta-Framework*, die im Abschnitt zuvor bereits dargestellt wurden, nennen die Autoren hier zusätzlich noch die Aspekte der *komponentenbasierten Architektur*, *Wiederverwendbarkeit* und natürlich der *Medienverarbeitung*.

Neben den hier genannten Zielen, die das COSIMA-Projekt zu erreichen versucht, zeichnet sich das Projekt durch seine spezifischen Charakteristika in Bezug auf andere Multimedia-Rahmenwerke aus. Diese Alleinstellungsmerkmale werden im nächsten Abschnitt genauer betrachtet.

2.3 Alleinstellungsmerkmale

Aus den in Abschnitt 2.2 dargestellten Zielen des COSIMA-Projekts lassen sich die folgenden Merkmale extrahieren, die COSIMA im Bereich der Multimedia-Rahmenwerke und -Anwendungen einmalig machen [Breuer u. a. 2008, S. 3f]:

Verteiltheit COSIMA ist als verteiltes System konzipiert.

Dienstorientierung Angelehnt an die *Service-Oriented Architecture* (SOA), sind die Bausteine in COSIMA als Dienste modelliert.

Integration Bestehende Frameworks können in Form von Diensten integriert werden und so lässt sich ihre Funktionalität integrieren.

Erweiterbarkeit Die Dienstorientierung erlaubt die Einbindung eigener Komponenten.

Skalierbarkeit In einer verteilten, dezentralisierten Umgebung können einzelne Funktionalitäten als Dienste völlig unabhängig voneinander betrieben werden. Was die vollständige Flexibilität in Bezug auf die Skalierbarkeit des gesamten Systems zur Folge hat [Papazoglou 2008, S. 294].

Medienobjekt-Modellierung Modellierung von Medien in ganzheitlicher Betrachtungsweise von Rohdaten und Metadaten in einem Objekt oder Container.

Meta-Ebene COSIMA fokussiert nicht auf Datensicht oder Metadatensicht sondern abstrahiert auf höheren Ebenen.

Medienverarbeitung Ganzheitliche Sicht auf Medienverarbeitung: Produktion, Verarbeitung, Transformation, Anreicherung, Wiedergabe, Ausgabe von Daten und Metadaten.

Architektur COSIMA stellt eine Architektur für Multimediaanwendungen zur Verfügung.

Basierend auf den vorgestellten Zielen und Alleinstellungsmerkmalen wurde die Architektur entworfen, die in dieser Arbeit validiert und prototypisch realisiert wird. Im Folgenden Abschnitt wird diese Architektur im Detail vorgestellt.

2.4 Konzeption einer Architektur

Die Architektur des COSIMA-Projekts ist iterativ nach einem dedizierten Vorgehensmodell[7] bis zu dem Punkt entwickelt worden, der als Ausgangspunkt für die Betrachtungen in dieser Arbeit dient. Bevor dieser aktuelle Stand jedoch in den folgenden Abschnitten im Detail vorgestellt wird, sollen zunächst einmal die Begriffe *Architektur*, *Dienstorientierung* und *Dienst* definiert und eingeordnet werden.

2.4.1 Software Architekturen

Der Begriff der Architektur im Kontext der Softwaretechnik und -entwicklung lässt sich sehr breit fassen. Es existieren unzählige Bücher zu diesem Thema und das Software Engineering Institute (SEI) der Carnegie Mellon Universität hat auf seiner Webseite bisher über 80 Definitionen von Architektur zusammen getragen[8]. Daher soll an dieser Stelle eine Einordnung des Begriffs der *Software Architektur* vorgestellt werden, wie er in dieser Arbeit Verwendung findet.

[7] Dieses Modell wird in Kapitel 3 kurz vorgestellt und detailliert im Institutsbericht [Breuer u. a. 2008, S. 7ff] beschrieben und diskutiert.

[8] `http://www.sei.cmu.edu/architecture/definitions.html`, zuletzt abgerufen am 03. November 2008

2.4.1.1 Definitionen

Das IEEE definiert in ihrem Glossar zur Softwaretechnik den Begriff Architektur wie folgt:

Definition 1 (Architektur (IEEE))**.** *„The organizational structure of a system or component.“* [IEEE 1990].

Diese sehr einfache Definition von Architektur weist schon auf die Eigenschaft der Strukturierung hin. Die Auswahl der folgenden Definitionen stellen die Eigenschaften von Architektur aber noch einmal deutlicher heraus.

Definition 2 (Architektur (Crispen))**.** *„An architecture [. . .] consists of (a) a partitioning strategy and (b) a coordination strategy. The partitioning strategy leads to dividing the entire system into discrete, non-overlapping parts or components. The coordination strategy leads to explicitly defined interfaces between those parts.“* [Crispen u. Lynn D. Stuckey 1994, S. 272]

Definition 3 (Architektur (Reussner et al.))**.** *„Die Software-Architektur ist die grundlegende Organisation eines Systems, dargestellt durch dessen Komponenten, deren Beziehungen zueinander und zur Umgebung, sowie die Prinzipien, die den Entwurf und die Evolution des Systems bestimmen.“* [Reussner u. Hasselbring 2006, S. 1]

Definition 4 (Architektur (Bass et al.))**.** *„The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationship among them.“* [Bass u. a. 2003, S. 21]

2.4.1.2 Software- und Systemarchitekturen

Darüber hinaus ist noch die Unterscheidung von *Software*- und *System*-Architekturen zu treffen [Clements u. a. 2002, S. xix]. Eine System-Architektur berücksichtigt dabei deutlich mehr Komponenten, wie etwa Hardware und Umgebung auf der die Software installiert werden soll. Diese Trennung wird auch in den entsprechenden Definitionen des SEI deutlich:

Definition 5 (Software-Architektur (SEI))**.** *„The structure or structures of a system, which comprise the software elements, the externally visible properties of those elements, and the relationships among them.“* [Mellon]

Definition 6 (System-Architektur). *„A means for describing the elements and interactions of a complete system including its hardware elements and its software elements.“* [Mellon]

Bei der Architektur innerhalb des COSIMA-Projekts handelt es sich also um eine Software-Architektur, da ausschließlich die Softwarekomponenten ohne Rücksicht auf mögliche Hardware betrachtet werden. Genauer kann auch gesagt werden, dass es sich um eine Referenzarchitektur nach Reussner et al. handelt:

Definition 7 (Referenzarchitektur). *„Eine Referenzarchitektur ist eine abstrakte Software-Architektur, sie definiert Strukturen und Typen von Software-Elementen sowie deren erlaubte Interaktionen und ihre Verantwortlichkeiten speziell für einen Anwendungsbereich. Die Strukturen sind jeweils für alle Systeme innerhalb einer Domäne anwendbar.“* [Reussner u. Hasselbring 2006, S. 358]

Da mit dem COSIMA-Projekt Multimediaanwendungen realisiert werden sollen, wird an dieser Stelle ergänzend noch der Terminus der *Multimedia-Architektur* definiert:

Definition 8 (Multimedia-Architektur). *„Eine Multimedia-Architektur ist eine Software-Architektur, die der Erzeugung, der Speicherung, der Transformation, der Präsentation und / oder dem Transport von multimedialen Daten dient.“* [Reussner u. Hasselbring 2006, S. 423]

2.4.1.3 Zusammenfassung

Für den weiteren Verlauf dieser Arbeit soll der Begriff der Architektur wie folgt verstanden werden: Die Architektur einer Software ist die strukturelle Aufteilung dieser Software in einzelne, voneinander unabhängige Komponenten und deren Eigenschaften sowie Beziehungen untereinander. Im Vordergrund stehen dabei multimediale Daten, die erzeugt, gespeichert, transformiert, präsentiert sowie transportiert werden müssen.

2.4.2 Dienst und Dienstorientierung

Mit dem Begriff der *dienstorientierten Architekturen* oder auch *SOA* beschäftigt sich die Fachliteratur thematisch bereits seit über 10 Jahren[9]. Während der letzten Jahre hat er immer mehr Prominenz, auch für die Verwendung im industriellen Umfeld erlangen können. Es finden sich unzählige Informationen zu diesem Thema, wie es einen echten Mehrwert für Unternehmen und ihre IT-Landschaft bieten kann. Viele dieser Aussagen sind nicht zuletzt mehr durch Marketing-Abteilungen geprägt [Liebhart 2007], als durch einen technischen Hintergrund. Die Gartner Group hat in einer Pressemitteilung von 2007 zum Thema SOA folgende Prognose abgegeben:

> *„Service-oriented architecture (SOA) will be used in more than 50 percent of new mission-critical operational applications and business processes designed in 2007 and in more than 80 percent by 2010.“*[10]

Es ist somit anzunehmen, dass SOA nicht nur einen kurzweiligen, technologischen Trend beschreibt, sondern vielmehr einen Paradigmenwechsel bei der Konstruktion von komplexen und verteilten Applikationen [Papazoglou 2003, S. 1]. Gerade dann ist es jedoch notwendig, die relevanten Begriffe der dienstorientierten Architektur im Rahmen dieser Arbeit zu definieren. Da es jedoch keine einheitliche und allgemein anerkannte Definition dazu gibt [Liebhart 2007, S. 6], werden im Folgenden Definitionen unterschiedlicher Herkunft zur Eingrenzung herangezogen.

2.4.2.1 Dienst

Die Begriffe *Dienstorientierung* und *dienstorientierte Architektur* beinhalten beide als konstituierendes Element den Begriff *Dienst*, daher soll dieser Begriff vor den anderen beiden definiert werden.

Eine sehr allgemeine Definition, was ein Dienst ist, liefert Masak. Der Wert dieser Definition liegt vor allem darin, dass durch sie „alles“ als Dienst aufgefasst werden kann:

Definition 9 (Dienst (allgemein))**.** *„Alles, was aus- oder durchgeführt werden kann ist ein Service!“* [Masak 2007, S. 16][11]

[9] Die Gartner Group hat sich das erste Mal 1996 mit diesem Begriff beschäftigt: SSA Research Note SPA-401-068, 12 April 1996, „Service-Oriented Architectures, Part 1“ und SSA Research Note SPA-401-069, 12 April 1996, „Service-Oriented Architectures, Part 2“ [Natis 2003]

[10] `http://www.gartner.com/it/page.jsp?id=503864`, zuletzt abgerufen am 02. Dezember 2008

[11] Die Begriffe „Dienst“ und „Service“ können synonym verwendet werden.

Im Rahmen dieser Arbeit und für das allgemeine Verständnis ist es jedoch essentiell, welche Eigenschaften einen Dienst am Ende auszeichnen. Die Definition der OASIS nennt hier die Möglichkeit des Zugriff auf bestimmte Funktionalitäten über eine wohl-definierte und formal beschriebene Schnittstelle:

Definition 10 (Dienst (OASIS))**.** *„A service is a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description.“* [MacKenzie u. a. 2006, S. 12]

Die Definition von Papazoglou nennt darüber hinaus drei essentielle Eigenschaften, die ein Dienst erfüllen muss, um als Dienst gelten zu können:

Definition 11 (Dienst (Papazoglou))**.** *„A service in SOA is an exposed piece of functionality with three essential properties. An SOA-based service is a self-contained [...] and platform-independet [...] service that can be dynamically located and invoked.“* [Papazoglou 2008, S. 258]

Für den weiteren Verlauf dieser Arbeit können also folgenden Eigenschaften festgehalten werden, die eine Dienst auszeichnen:

- Eine wohl-definierte und formale Schnittstelle
- In sich abgeschlossen
- Plattformunabhängigkeit
- Dynamische Lokalisierung und Ausführung

2.4.2.2 Dienstorientierung

Bevor auf die dienstorientierte Architektur eingegangen werden kann, soll noch der Begriff der Dienstorientierung für sich definiert werden:

Definition 12 (Dienstorientierung)**.** *„Service-orientation is a design paradigm intended for the creation of solution logic units that are individually shaped so that they can be collectively and repeatedly utilized in support of the realization of a specific set of strategic goals and benefits associated with SOA and service-oriented computing.“* [Erl 2008]

Es handelt sich bei der Dienstorientierung demnach um das Entwurfsparadigma, dass hinter einer dienstorientierten Architektur steht. Erl nennt in diesem Zusammenhang die folgenden acht Entwurfsprinzipien, die wiederum der Dienstorientierung selbst zu Grunde liegen [Erl 2008].

- Standardisierte Dienstverträge
- Lose Kopplung der Dienste
- Abstraktion der Dienste
- Wiederverwendbarkeit der Dienste
- Autonomie der Diente
- Dienste sind zustandslos
- Dienste sind auffindbar
- Dienste lassen sich komponieren

Teile dieser Prinzipien finden sich auch in den Eigenschaften von Diensten selbst wieder, wie im vorherigen Abschnitt deutlich wurde. Erl weisst darüber hinaus darauf hin, dass die Dienstorientierung ihre Wurzeln in vielen unterschiedliche Disziplinen der modernen Softwareentwicklung hat, wie Abbildung 2.1 zeigt.

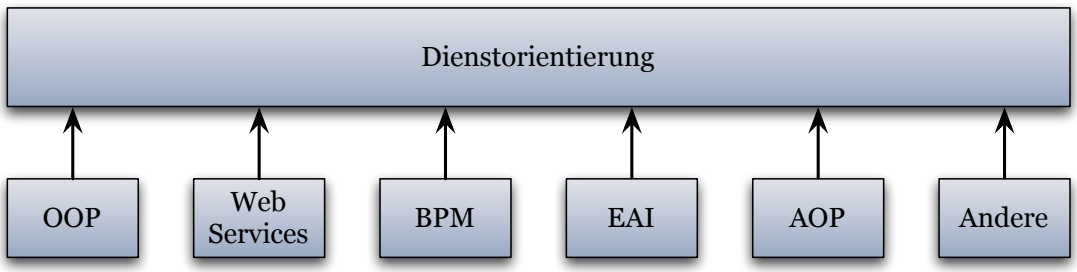

Abbildung 2.1: Einflüsse auf das Paradigma der Dienstorientierung nach [Erl 2008]

2.4.2.3 Dienstorientierte Architektur

Bei der Definition von Natis wird zunächst die Schnittstelle zwischen den Diensten in den Vordergrund gerückt:

Definition 13 (SOA (Natis)). *„Essentially, SOA is a software architecture that starts with an interface definition and builds the entire application topology as a topology of interfaces, interface implementations and interface calls.“* [Natis 2003, S. 2]

Obwohl die Konstruktion einer Architektur, die rund um definierte Schnittstellen aufgebaut wird, als Teilaspekt auch innerhalb einer SOA von Bedeutung ist, macht das allein noch keine dienstorientierte Architektur aus[12]. Die Definition der OASIS fokussiert im Gegensatz dazu stärker auf den Aspekt der Verteiltheit und das einzelne Einheiten unterschiedlichen Domänen und Besitzern zugeordnet sein können:

Definition 14 (SOA (OASIS Reference Model)). *„Service-Oriented Architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains.“* [MacKenzie u. a. 2006, S. 8]

Nach Josuttis ist der Hintergrund der Softwareentwicklung die Abstraktion. Entscheidend ist lediglich nur aus welcher Perspektive die Abstraktion durchgeführt wird. Im Falle der dienstorientierten Architekturen findet diese Abstraktion aus Sicht der Geschäftsaspekte statt [Josuttis 2007, S. 16]. Darüber hinaus erweitert er die Definition der OASIS noch um die Größe der IT-Landschaft und stellt die Heterogenität der beteiligten Systeme heraus:

Definition 15 (SOA (Josuttis)). *„SOA is an architectural paradigm for dealing with business processes distributed over a large landscape of existing and new heterogeneous systems that are under the control of different owners.“* [Josuttis 2007, S. 24]

Ein weiteres wesentliches Merkmal, vor allem auch für das COSIMA-Projekt, ist nach Papazoglou die technologie-agnostische Charakteristik der einzelnen Dienste. Zusätzlich beschreibt er die SOA als eine Meta-Architektur und betont die lose Kopplung der Dienste untereinander:

Definition 16 (SOA (Papazoglou)). *„[. . .] SOA is a meta-architectural style that supports loosely coupled services to enable business flexibility in an interoperable, technology-agnostic manner.“* [Papazoglou 2008, S. 257]

[12] Auch in der objektorientierten Programmierung gilt das Prinzip, dass gegen Interfaces und nicht gegen Implementierungen programmiert werden soll [Gamma u. a. 1995, S. 18].

Papazoglou unterscheidet weiter zwischen zwei Arten dienstorientierter Architekturen, die sich jeweils im Grad ihrer Komplexität unterscheiden. Die einfachste Form einer SOA ist in Abbildung 2.2 dargestellt.

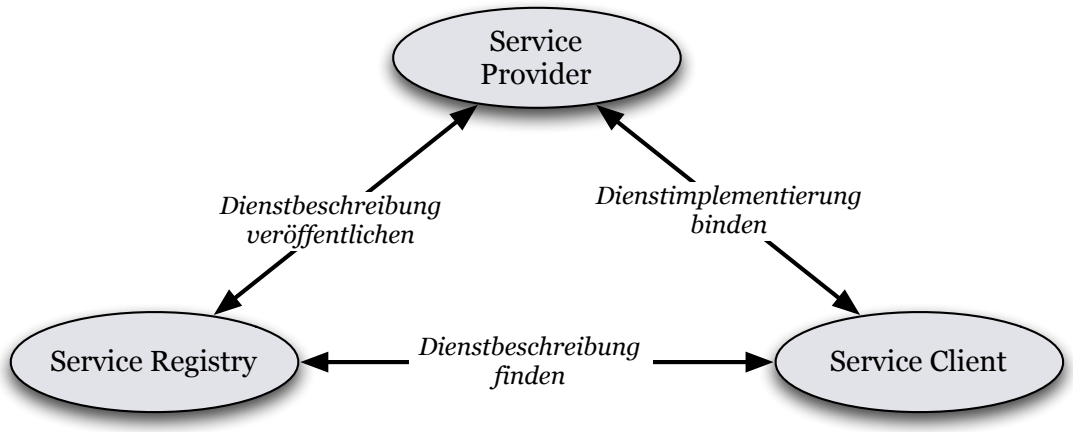

Abbildung 2.2: Einfachste Form einer dienstorientierten Architektur nach [Papazoglou 2003]

In diesem Modell sind zwar einige Eigenschaften einer SOA, wie sie in den vorangegangen Definitionen herausgestellt wurden, erfasst, jedoch bei weitem nicht alle. Lediglich die lose Kopplung und das sich Dienste auffinden lassen ist explizit modelliert. Alle anderen Eigenschaften, wie etwa Management, Dienstkomposition, Transaktionen zwischen und Koordination von Diensten oder Sicherheitsaspekte lassen sich wenn nur implizit wieder finden [Papazoglou 2003, S. 8]. Aus diesem Grunde stellt Papazoglou eine *erweiterte* dienstorientierte Architektur (siehe Abbildung 2.3) vor, in der diese Punkte explizit modelliert werden.

Von besonderem Interesse im Kontext von COSIMA ist die Ebene der *komponierten Dienste*. Auf dieser Ebene wird die Ausführung der einzelnen Dienste, sowie die Datenübertragung koordiniert [Papazoglou 2003, S. 8]. Auf dieser Ebene lassen demnach die Mechanismen zur Synchronisation (→ Abschnitt 2.5.3) und Medienvermittlung (→ Abschnitt 2.4.8.4) einordnen.

In diesem, sehr viel umfangreicheren Modell, finden sich fast alle bisher vorgestellten Eigenschaften einer SOA wieder. Vor allem der Aspekt, dass sich einzelne Dienste zu komplexeren Diensten zusammen fassen, also komponieren lassen, ist in diesem Modell eine

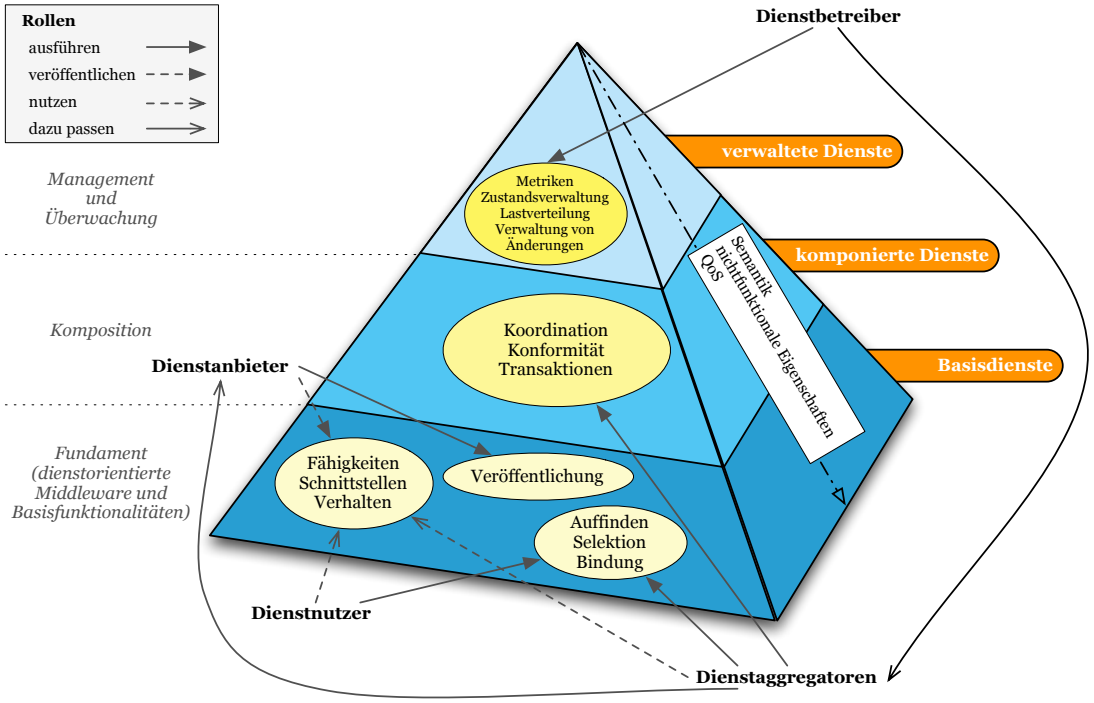

Abbildung 2.3: Erweiterte dienstorientierte Architektur nach [Papazoglou u. a. 2007]

markante Eigenschaft[13]. Lediglich ein Bezug zu (Geschäfts-)Prozessen wird nicht explizit modelliert[14]. Die nachfolgende Liste fasst die wesentlichen Eigenschaften einer dienstorientierten Architektur noch einmal zusammen:

- Lose Kopplung
- Interoperabilität
- Dienste aus unterschiedlichen Domänen
- Bereitstellung einer Infrastruktur
- Architektur
- Komposition von Diensten zu Prozessen
- Betrachtung von nicht-funktionalen Anforderungen

Nach dieser Einordnung der wesentlichen Begriffe, werden im Folgenden nun die einzelnen Komponenten der COSIMA Architektur im Detail vorgestellt.

[13] Auf diesen Aspekt wird in Abschnitt 2.4.6 noch genauer eingegangen.
[14] Eine begriffliche Einordnung von „(Geschäfts-)Prozessen“ findet sich an entsprechenden Stelle in Abschnitt 4.1.2. An dieser Stelle soll nicht weiter darauf eingegangen werden.

2.4.3 Einführung in COSIMA

In [Starke 2008, S. 83] werden vier unterschiedliche Sichten beschrieben, die sich in Bezug auf eine Software-Architektur einnehmen lassen: *Kontextsichten*, *Bausteinsichten*, *Laufzeitsichten* und *Verteilungssichten*. Diese Sichten adressieren dabei den Bedarf, die „Vielschichtigkeit und Komplexität“ einer Architektur ausdrücken zu können [Starke 2008, S. 81]. Bei der Konzeption und Entwicklung der COSIMA-Architektur wurden daher ebenfalls Vertreter einiger dieser Sichten erstellt.

Für das COSIMA-Projekt wurden bisher Darstellungen aus zwei dieser Kategorien angelegt: eine der Kontextsicht und drei der Bausteinsicht. Die Kontextsichten legen laut Starke den „Fokus auf den Zusammenhang oder das Umfeld des Systems“ [Starke 2008, S. 87], was auch auf die in Abbildung 2.4 dargestellte Kontextsicht von COSIMA zutrifft. Hier wird lediglich ein abstrakter und vor allem nicht formaler Überblick über die Architektur und beteiligte Systeme gegeben. Die Darstellungen, die sich den Bausteinsichten zuordnen lassen, sind namentlich das *Komponentendiagramm*, das *Kompositionsstrukturdiagramm* und der *Grobentwurf* der Architektur. Diese finden sich jeweils in [Breuer u. a. 2008] und sollen an dieser Stelle nicht weiter betrachtet werden. Für die weitere Vorstellung der Architektur soll an dieser Stelle die Kontextsicht genügen.

2.4.4 Medienverarbeitende Komponenten

Den Kern von COSIMA bilden die medienverarbeitenden Komponenten, die vom Anwendungsentwickler in eine entsprechende Abfolge gebracht werden und damit die eigentliche Multimediaanwendung darstellen: *Producer*, *Transformer* und *Consumer*. Dem Entwurf der Komponenten liegt das *Quelle-Komponente-Senke* Prinzip zu Grunde. Die innerhalb des COSIMA-Projekts verwendete Bezeichnung ist nach [Gibbs 1995; de Mey u. Gibbs 1993] ebenso valide wie die geläufigere Bezeichnung „Quelle-Komponente-Senke“. Da innerhalb von COSIMA jedoch unterschiedliche Komponenten enthalten sind, kamen Benennungsschwierigkeiten auf, die die Verwendung einer alternativen Begrifflichkeit notwendig machten.

Diese medienverarbeitenden Komponenten sind dabei als Dienste, wie sie in der SOA verstanden werden (→ Abschnitt 2.4.2.1), ausgeprägt. Zur Entwicklung einer Multimediaanwendung ist demnach eine Komposition dieser Dienste durch die in Abschnitt 2.4.6

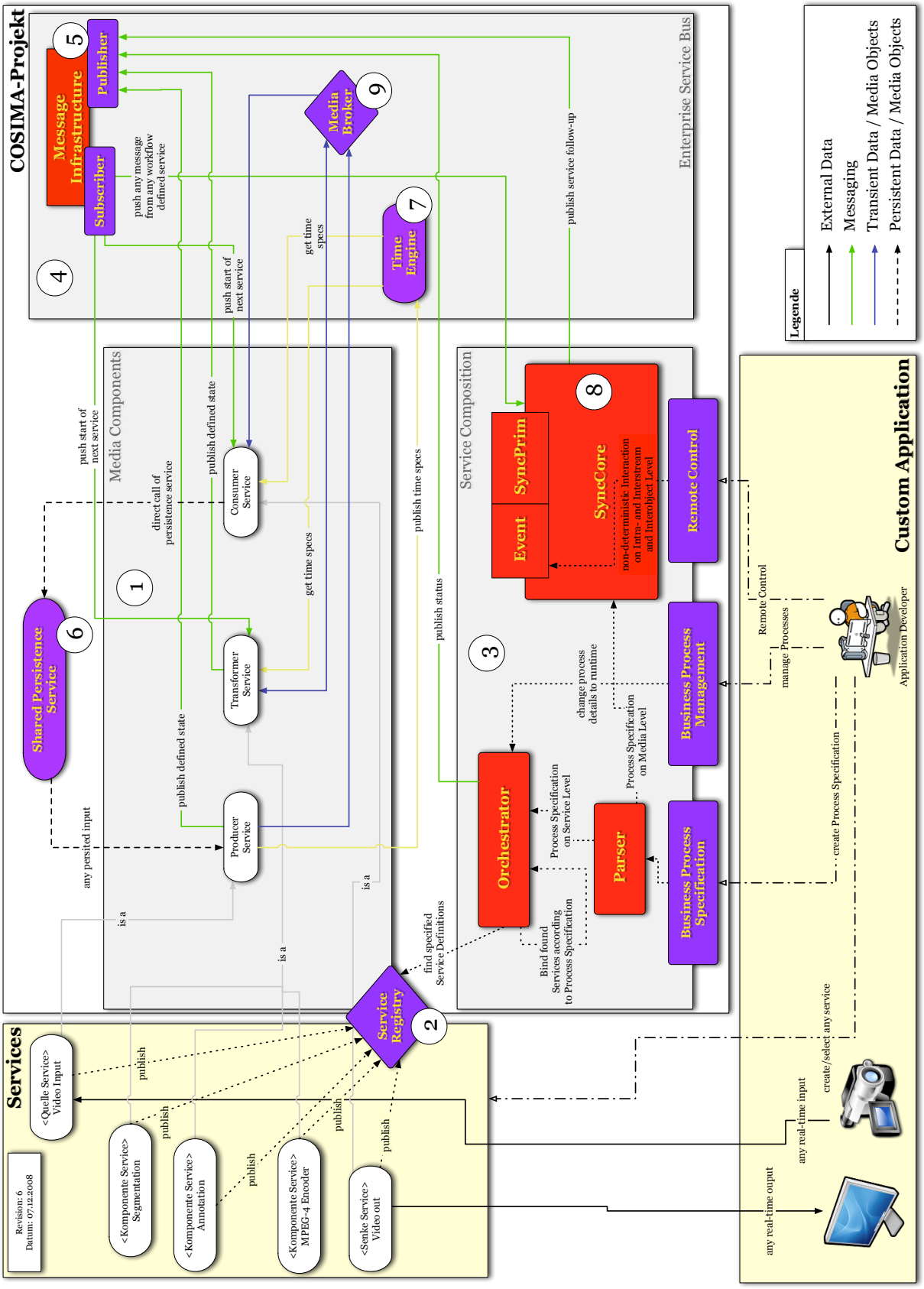

Abbildung 2.4: Kontextsicht der Architektur des COSIMA-Projekts

vorgestellten Ansätze notwendig. Eine formale Beschreibung der Dienste selbst wird in der *Service Registry* vorgehalten, welche in Abschnitt 2.4.5 näher erläutert wird.

Der Austausch der zu verarbeitenden Medien zwischen den einzelnen Diensten geschieht auschließlich über den *Media Broker*, der im Abschnitt 2.4.8.4 weiter ausgeführt wird. Ein genereller Nachrichtenaustausch zwischen den Diensten und mit den restlichen Teilen der Architektur wird über das Nachrichtensystem (→ Abschnitt 2.4.7.1) realisiert. Persistenz von Daten und die Synchronisation werden von einem *Persistence Data Service* (→ Abschnitt 2.4.7.2) beziehungsweise dem *SyncCore* und der *Timing Engine* übernommen (→ Abschnitt 2.4.8.2). Beide Komponenten werden in Abschnitt 2.4.7 näher beschrieben. Teile dieser Architekturelemente sollen später in einem Service Bus aufgehen [Breuer u. a. 2008, S. 18].

2.4.5 Service Registry

Wie bereits in Abschnitt 2.4.2.3 gezeigt wurde, ist die Fähigkeit Dienste entdecken zu können eines der konstituierenden Elemente innerhalb einer SOA. Nach [Papazoglou 2003] ist diese selbst wieder als Dienst zu realisieren. Über die Service Registry lassen sich anhand von abstrakten (formalen) Dienstbeschreibungen konkrete Dienste identifizieren und eine Lokalisierung durchführen. Sie unterstützt damit die Forderung nach loser Kopplung innerhalb einer SOA. Eine Realisierung einer solchen Service Registry wäre beispielsweise auf Basis von UDDI[15] oder ebXML[16] möglich.

2.4.6 Servicekomposition

Als eine weitere wichtige Eigenschaft einer dienstorientierten Architektur ist die Fähigkeit zur Komposition von Diensten genannt worden. Ähnlich der Service Registry dient auch sie der Entkopplung der einzelnen Dienste in einer SOA. Wesentlich entscheidender ist jedoch ihr Beitrag Dienste in *Prozessen*[17] zu aggregieren und damit neue Applikationen innerhalb einer dienstorientierten Architektur entwickeln zu können. Die Servicekomposition selbst setzt dabei auf die grundlegenden Elemente einer SOA auf [Milanovic u. Malek 2004, S. 51], wie sie in Abschnitt 2.4.2.3 vorgestellt wurden.

[15] `http://www.oasis-open.org/committees/uddi-spec/`

[16] `http://www.ebxml.org/`

[17] Eine zusätzliche begriffliche Einordnung dieses Begriffs findet sich während der Implementierung in Abschnitt 4.1.2

Die Servicekomposition selbst kann dabei aus zwei unterschiedlichen Perspektiven betrachtet werden [Masak 2007, S. 104]: der *Geschäftsprozesskomposition* und der *Servicelevel-Komposition*. Bei der Geschäftsprozesskomposition werden „völlig neue Geschäftsprozesse aus bestehenden Teilprozessen oder Services“ ([Masak 2007, S. 104]) komponiert. Eine Einbettung in eine Organisation steht hier klar im Vordergrund. Bei der Servicelevelkomposition steht vielmehr „die Interoperabilität und technische Machbarkeit im Vordergrund“ ([Masak 2007, S. 105]) und es wird keine Rücksicht auf eine mögliche Organisationstruktur genommen.

2.4.6.1 Kompositionsarten

Bei der Komposition von Diensten werden in der Literatur verschiedene Vorgehen unterschieden. Die beiden grundsätzlichen, für COSIMA relevanten, werden in [Papazoglou u. a. 2007, S. 41] genannt: *Orchestrierung* und *Choreographie*. Im Folgenden sollen diese beiden Ansätze gegeneinander abgegrenzt werden.

Die Orchestrierung wird von Papazoglou wie folgt definiert:

Definition 17 (Orchestrierung)**.** *„Orchestration describes how services interact at the message level, including the business logic and execution order of interactions under control of a single end point. It is an executable business process that can result in a long-lived, transactional, multistep process model.“* [Papazoglou u. a. 2007, S. 41]

Wesentlich sind also die Interaktionen verschiedener Dienste, die von einer zentralen Stelle (dem *single end point*) über einen Nachrichtenaustausch koordiniert und kontrolliert werden. Des Weiteren ist das Ergebnis ein ausführbarer Geschäftsprozess.

Eine geeignete Definition für die Choreographie findet sich bei Peltz:

Definition 18 (Choreographie)**.** *„[. . .] choreography, which is more collaborative and allows each involved party to describe its part in the interaction. Choreography tracks the message sequences among multiple parties and sources – typically the public message exchanges that occur between Web services – rather than a specific business process that a single party executes.“* [Peltz 2003, S. 46]

Der Unterschied zwischen beiden Ansätzen wird durch die Definitionen sehr deutlich: Choreographie verfolgt eher einen dezentralen Ansatz; Es gibt nicht eine Instanz, die den

Nachrichtenfluss kontrolliert, vielmehr tritt der Nachrichtenfluss zwischen den einzelnen Diensten in den Vordergrund. Es liegt dadurch auch kein ausführbarer Prozess bei der Choreographie vor [Peltz 2003, S. 46].

Für die konkrete Umsetzung dieser beiden Varianten zur Komposition existieren zahlreiche Technologien und Standards. Dabei fäll jedoch auf, dass eine strikte Trennung beider Ansätze auf der technischen Ebene eher künstlich erscheinen und daher in einem technologischen Ansatz vereint werden sollten [Papazoglou u. a. 2007, S. 42].

Eine detaillierte Auseinandersetzung mit der Thematik der Geschäftsprozesskomposition und -modellierung im Rahmen des COSIMA-Projekts findet sich in der Master Thesis von Matthias Richter [Richter 2008].

2.4.7 Infrastruktur und Enterprise Service Bus

Innerhalb einer dienstorientierten Architektur besteht der Bedarf nach einer Infrastruktur, die in der Lage ist, die einzelnen Dienste zu verwalten und zu integrieren [Papazoglou 2008, S. 270]. Neben der bereits separat vorgestellten Servicekomposition gehören auch die in diesem Abschnitt vorgestellten Elemente dieser, in der Literatur als *Enterprise Service Bus* (ESB) bezeichneten Infrastruktur an.

Ein Enterprise Service Bus lässt sich dabei wie folgt definieren:

Definition 19 (ESB)**.** *„The Enterprise Service Bus is an open standards-based message backbone designed to enable the implementation, deployment, and management of SOA-based solutions with a focus on assembling, deploying, and managing distributed service-oriented architectures.“* [Papazoglou 2008, S. 270]

Nach dieser Definition wird durch den Einsatz eines ESB überhaupt erst die Realisierung und der Betrieb einer dienstorientierten Architektur ermöglicht. Darüber hinaus lässt sich festhalten, dass auch das „Assembling“, also das Zusammenführen, der einzelnen Teile innerhalb der Architektur in den Aufgabenbereich des ESB fallen. Demnach kann die Servicekomposition in einer SOA auch als eine Teilmenge des ESB angesehen werden [Chappell 2004, S. 3].

Neben der Servicekomposition werden von dem ESB noch eine Vielzahl weiterer Aufgaben übernommen, die jeweils die Punkte *Implementierung*, *Deployment*[18] und *Management* unterstützen sollen. Die Aufgaben, die im ESB dabei mindestens implementiert sein müssen, sind in der folgenden Liste nach [Liebhart 2007, S. 137] und [Masak 2007, S. 146] dargestellt:

- Routing von Nachrichten
- Kommunikationsbus als Integrationsgrundlage
- Datentransformation und -zuordnung
- Prozess- und Regelausführung
- Überwachung der einzelnen Komponenten
- Adaptoren für Applikationen
- Bereitstellen von standardisierten Schnittstellen

Historisch gesehen ist ein ESB die Weiterentwicklung von EAI Brokern, die bereits ähnliche oder gleiche Funktionalitäten bereitstellen konnten [Masak 2007, S. 146]. Der Enterprise Service Bus verzichtet dabei aber auf den zentralistischen Integrationsansatz von EAI Brokern und etabliert anseinerstatt eine verteilte Integration [Chappell 2004, S. 4]. Die einzelnen Funktionalitäten werden, um diese Verteiltheit zu erreichen, selbst wieder als Dienste realisiert [Chappell 2004; Masak 2007; Papazoglou u. a. 2007]. Durch die Aufteilung in einzelne Dienste und deren Verteilung innerhalb des Service Bus, kann dann auch von einer virtuellen Infrastruktur gesprochen werden [Liebhart 2007, S. 136].

Das COSIMA-Projekt sieht bis zu diesem Zeitpunkt nur einen sehr rudimentären ESB vor[19]. Dennoch übernimmt auch der Enterprise Service Bus in COSIMA die gleichen Verantwortlichkeiten, einige davon sind jedoch für den Einsatz in Multimediaanwendungen adaptiert worden. Im Folgenden werden die für das COSIMA-Projekt relevanten Komponenten des ESBs näher beschrieben.

[18] Das englische Wort „Deployment“ kann in diesem Kontext am ehesten mit „Inbetriebsetzung“ übersetzt werden. Da sich „Deployment“ jedoch in der Fachsprache weitestgehend eingeprägt hat, wird es sinngemäß nach der Beschreibung in der Wikipedia verwendet: *„Software deployment is all of the activities that make a software system available for use.“* (aus Wikipedia: `http://en.wikipedia.org/wiki/Software_deployment`, zuletzt abgerufen am 22.10.2008)

[19] Gemessen an dem Umfang, wie er in der Literatur beschrieben wird und in kommerziellen Systemen vorkommt.

2.4.7.1 Nachrichtensystem

Im vorangegangen Kapitel wurde als eine Aufgabe des Enterprise Service Bus, die Vermittlung und Übertragung von Nachrichten zwischen den einzelnen Teilnehmern genannt. Das *Messaging*, also die Nachrichtenvermittlung ist immer das Herzstück eines ESB [Chappell 2004, S. 77] und daher auch in COSIMA. Die Nachrichtenvermittlung realisiert dabei eine sehr performante, asynchrone und verlässliche Kommunikation zwischen Applikationen. Eine Nachricht selbst ist dabei ein wohl-definiertes, datengetriebenes Textformat [Papazoglou 2008, S. 60f].

Das in COSIMA integrierte Nachrichtensystem ist nach dem *Publish/Subscribe*-Pattern [Hohpe u. a. 2004, S. 106] aufgebaut. Es ist eine von vielen Formen, um asynchrone und verlässliche Kommunikation zu realisieren, die dabei etwas besser skaliert als andere Lösungen [Papazoglou 2008, S. 69]. Zusammengefasst lässt sich die Funktionsweise wie folgt beschreiben:

- Ein *Publisher* veröffentlicht eine Nachricht zu einem bestimmten Thema.
- Alle *Subscriber*, die dieses Thema abonniert haben, erhalten genau eine Kopie dieser Nachricht.

Bei [Liebhart 2007, S. 127] wird darüber hinaus gesagt, dass es sich bei dieser Form der Nachrichtenvermittlung, um eine *nicht-gerichtete* Kommunikation handelt.

In COSIMA selbst werden über das Nachrichtensystem vor allem Kontroll- und Synchronisationsdaten vermittelt. Die Kontrolldaten werden dabei von der Komponente der Servicekomposition versendet, und beinhalten Informationen, wo und wie die medienverarbeitenden Komponenten ihre Daten beziehen können. Das Nachrichtensystem soll nicht die Medien selbst vermitteln. Für diese Aufgabe wurde eine dedizierte Komponente eingeführt, die in Abschnitt 2.4.8 näher beschrieben wird. Auf die Synchronisationsdaten wird in Abschnitt 2.4.8.2 näher eingegangen.

2.4.7.2 Persistenz

Um Daten dauerhaft ablegen zu können, ist innerhalb von COSIMA ein weiterer Dienst vorgesehen, der *Shared Persistence Service*. Allerdings kann nur eine *Consumer*-Komponente über diesen Dienst Daten ablegen lassen. Umgekehrt kann nur eine *Producer*-Komponente Daten über diesen Dienst wieder einlesen. Eine *Transformer*-Komponente, kann

nach dem Quelle-Komponente-Senke Prinzip keinen Zugriff auf dauerhaft abgelegte Daten erhalten.

Innerhalb von COSIMA ist dieser Dienst noch sehr rudimentär modelliert. In der Literatur zu SOA findet sich keine dedizierte Komponente, um Daten persistent zu speichern; Diese Funktionalität kann durch jeden beliebigen Dienst bereitgestellt werden. Auf Grund dieser Tatsache und durch die Etablierung eines dedizierten Medienbrokers (→ Abschnitt 2.4.8.3), der die Vermittlung der Medienobjekte realisiert, ist die weitere Existenzberechtigung dieser Komponente jedoch eher zweifelhaft. Eine weitere Betrachtung dieser Zusammenhänge findet in Abschnitt 2.4.8 statt.

Die bis hierher vorgestellten Komponenten der Architektur innerhalb des COSIMA-Projekts sind weitestgehend übertragbar auf Komponenten in anderen dienstorientierten Architekturen. Zur Umsetzung von Multimediaanwendungen sind aber noch weitere, medienspezifische Komponenten erforderlich. Auch wenn sie sich grundsätzlich dem Enterprise Service Bus zuordnen lassen, sollen sie dennoch dediziert im nächsten Abschnitt diskutiert werden.

2.4.8 Integration von Medien

Die in diesem Abschnitt vorgestellten Komponenten tragen maßgeblich dazu bei, dass sich mit COSIMA Multimediaanwendungen realisieren lassen. Um jedoch die Relevanz dieser Komponenten im Rahmen von Multimedia begreifen zu können, soll im ersten Schritt eine Definition des Begriffs Multimedia gegeben werden, aus dem sich dann später die Anforderungen an die hier vorgestellten Komponenten ableiten lassen.

2.4.8.1 Multimedia

Verbaliter lässt sich Multimedia durch die Bedeutung der beiden Wortteile „Multi-“ und „Media“ oder „Medium“ erschließen:

Definition 20 (Multi-)**.** *„vielfach, mehrer. . . , viel. . . /Viel. . . “*[20]

Definition 21 (Medium / Medien)**.** *„**Medium** (lat. »Mitte«) **1.** bildungssprachlich für: vermittelndes Element [. . .] **3.** Kommunikationswissenschaften: Vermittlungsinstanz von*

[20] aus: Duden - Deutsches Universal Wörterbuch A-Z, 3. Aufl., 1996

*Informationen [...]. **Medien** (von engl. »media«), Sg. Medium, Sammel-Bez., für alle techn. Mittel zur Verarbeitung von Information, d.h. für Kommunikationsmittel (z.B. Zeitung, Zeitschrift, Buch, Plakat, Hörfunk, Fernsehen, Internet) [...]"*[21]

Der Begriff *Multimedia* ließe sich daher so definieren, dass es sich um viele vermittelnde Instanzen von Informationen handelt. Diese Definition wäre für den konkreten Kontext jedoch zu unspezifisch und daher ist eine weitere Einordnung notwendig.

Bei Steinmetz wird die genannte Definition dahingehend erweitert, dass ein Medium „ein Mittel zur Verbreitung und Darstellung von Informationen" [Steinmetz 2000, S. 7] ist. Sie wird also explizit um die *Darstellung* von Informationen ergänzt. Der Begriff *Medium* lässt sich selbst dabei nach den folgenden Kriterien noch weiter differenzieren: Perzeptions-, Repräsentations-, Präsentations-, Speicher-, Übertragungs-, und Informationsaustauschmedien. Dabei entsprechen die Perzeptionsmedien am nächsten dem, was in einer Multimediaanwendungen unter Medien verstanden wird: Die Informationen, die der Mensch mit seinen fünf Sinnen wahrnehmen kann [Steinmetz 2000, S. 9][22]. Jedes Medium definiert dabei einen *Darstellungsraum* mit entsprechenden *Darstellungsdimensionen*. Neben den räumlichen Dimensionen dient vor allem die zeitlich Dimension dazu, eine weitere Einordnung von Medien vornehmen zu können [Steinmetz 2000, S. 10]:

Diskrete Medien Medien, deren Informationen zeitunabhängig auftreten, etwa Text oder Grafik. Ihre Verarbeitung ist *zeitunkritisch*.

Kontinuierliche Medien Medien, deren Informationen sich über die Zeit hinweg ändern, etwa Ton oder Video. Ihre Verarbeitung ist *zeitkritisch*.

Mit dieser Erkenntnis wäre es nicht angemessen, den Begriff Multimedia rein quantitativ zu beschreiben, sondern viel mehr ist eine qualitative Definition adäquat:

Definition 22 (Multimedia)**.** *„Als Multimedia wird der Einsatz von sowohl mindestens einem diskreten und mindestens einem kontinuierlichem Medium verstanden."* [Steinmetz 2000, S. 14]

Aufbauend auf dieser Definition von Multimedia kann abschließend auch der Begriff *Multimediaanwendungen* respektive *Multimediasystem* definiert werden:

[21] aus: Brockhaus - die Enzyklopädie, 21. Auflage, 2006
[22] Dies bedeutet im Umkehrschluss jedoch nicht, dass die anderen Arten von Medien keine Rolle spielen.

Definition 23 (Multimediasystem)**.** *„Ein Multimediasystem ist durch die rechnergesteuerte, integrierte Erzeugung, Manipulation, Darstellung, Speicherung und Kommunikation von unabhängigen Informationen gekennzeichnet, die in mindestens einem kontinuierlichen (zeitabhängigen) und einem diskreten (zeitunabhängigen) Medium kodiert sind.“* [Steinmetz 2000, S. 13]

Obwohl die Informationen selbst unabhängig voneinander sind, so muss in der Regel dennoch ein Bezug zwischen den einzelnen Informationen hergestellt werden können. Dieser Bezug wird auch *Synchronisation* genannt und im nächsten Abschnitt näher beschrieben.

2.4.8.2 Synchronisation

An sich unabhängige Medien müssen in der Regel in einer Multimediaanwendungen in einen Zusammenhang gebracht werden; Beispielsweise muss ein Video in einen Zusammenhang mit Untertiteln gebracht werden. Die Etablierung dieses Zusammenhangs wird als *Synchronisation* bezeichnet. Allgemein kann sie als das „Herstellen des Gleichlaufs von Vorgängen, Maschinen u. a.“[23] bezeichnet werden. Wie bereits bei der allgemeinen Definition von Multimedia, ist auch diese zu unspezifisch für den gegebenen Einsatz und auch hier muss eine weitere Klassifikation stattfinden. Bereits aus dem Eingangs genannten Beispiel lassen sich separate Eigenschaften extrahieren: Nicht nur müssen die Untertitel zur rechten Zeit angezeigt werden, sondern sie haben auch im Videobild selbst eine räumliche Positionierung. Es existiert also eine *zeitliche* wie *örtliche* Beziehung. Neben diesen beiden Arten von Beziehungen existiert noch eine Weitere, die *Inhaltliche*:

Inhaltliche Beziehung Definiert die Abhängigkeit einzelner Medien von bestimmten Daten.

Örtliche Beziehung Ist auch bekannt als *Layout* Beziehung und definiert die Beziehung von unterschiedlichen Medien im Raum, bezogen auf ein Ausgabegerät zu einem bestimmten Zeitpunkt.

Zeitliche Beziehung Definiert die zeitlichen Abhängigkeiten von Medien und ist immer dann von Bedeutung, wenn mit kontinuierlichen Medien umgegangen wird.

[23] aus: Der Brockhaus in einem Band. 10. Auflage, 2005

Alle drei Arten müssen in einer Multimediaanwendungen betrachtet und daher auch im COSIMA-Projekt berücksichtigt werden. Synchronisation wird im eigentlichen Sinne jedoch auf die zeitliche Beziehung von unterschiedlichen Medien bezogen [Steinmetz u. Nahrstedt 1995, S. 572]. Diese besondere Bedeutung der zeitlichen Komponente bei der Betrachtung von Multimediadaten wurde auch von Bertino und Ferrari festgestellt:

> *„One of the inherent characteristics of multimedia data is of being heavily time-dependent in that they are usually related by temporal relations which have to be maintained during their playout.“* ([Bertino u. Ferrari 1998, S. 612])

Auf Grund der besonderen Relevanz der Zeitdomäne soll im Rahmen dieser Arbeit die folgende Definition von Synchronisation Geltung besitzen:

Definition 24 (Synchronisation)**.** *„Synchronization in the context of multimedia refers to the mechanisms used by processes (also specific to multimedia) to coordinate their ordering in the time domain.“* [Steinmetz 1990, S. 401]

Nach dem eine Definition für Synchronisation gefunden wurde, sollen im weiteren Verlauf unterschiedliche Ansätze zur Modellierung von zeitlicher Synchronisation in Multimediaanwendungen vorgestellt werden.

Modelle zur Beschreibung zeitlicher Synchronisation

Um zeitliche Synchronisation zu beschreiben und in einer Multimediaanwendung abbilden zu können, existieren unterschiedliche Modelle und Klassifikationen, von denen eine für diesen Kontext relevante Auswahl hier näher beschrieben werden soll.

Natürliche und Synthetische Synchronisation Eine erste Klassifikation kann über eine Unterscheidung in *natürliche* (oder *live*) und *synthetische* Synchronisation [Little u. Ghafoor 1991; Little u. a. 1991; Steinmetz u. Meyer 1992] erstellt werden. Bei der natürlichen Synchronisation wird die zeitliche Beziehung zwischen den einzelnen Medien implizit während der Aufnahme erfasst. In der Regel tritt eine natürliche Synchronisation nur bei kontinuierlichen Medien auf. Im Gegensatz dazu definiert die synthetische Synchronisation die zeitlichen Beziehung explizit über eine Spezifikation. Bei der natürlichen Synchronisation besteht das Ziel darin, dass die zeitlichen Beziehung, die während der Aufnahme der Medien implizit mit erfasst wurden, bei der Präsentation immer noch Geltung besitzen. Durch die synthetische Synchronisation gibt die Möglichkeit, möglichst flexibel zeitliche

Abhängigkeiten zu modellieren, die bei der Präsentation Geltung haben müssen [Bertino u. Ferrari 1998, S. 613]. Im COSIMA-Projekt müssen in jedem Fall beide Varianten Berücksichtigung finden.

Granularität von Synchronisation Neben der Klassifikation, ob eine Synchronisation implizit oder explizit erfasst wurde, lassen sich ebenfalls unterschiedliche Granularitäten von zeitlichen Beziehungen unterscheiden, die in Abbildung 2.5 exemplarisch dargestellt und im Anschluss beschrieben sind.

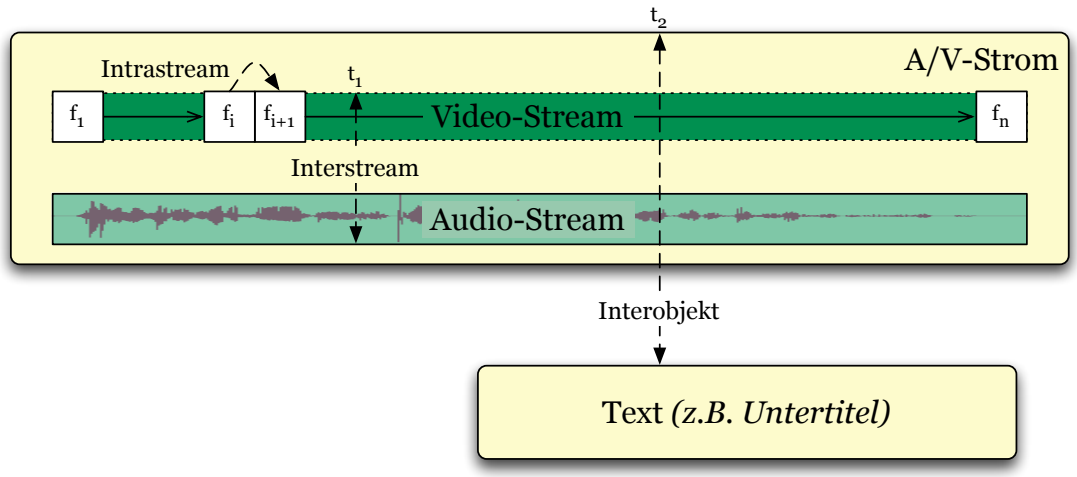

Abbildung 2.5: Granularitätsebenen der zeitlichen Synchronisation nach [Antons 2009]

Intrastream Die Intrastream-Synchronisation bezieht sich auf die einzelnen (Präsentations-) Einheiten innerhalb eines Objekts. Diese Einheiten werden auch als *Logische Dateneinheiten* (*Logical Data Units* (LDU)) bezeichnet [Steinmetz 1990].

Interstream Die Interstream-Synchronisation bezieht sich auf verschiedene Ströme innerhalb einer Stromgruppe [Steinmetz 2000], etwa die zeitliche Beziehung von einer Ton- und Videospur in einem A/V-Strom[24].

Interobjekt Im Zusammenhang mit der Referenzarchitektur nach Meyer und Steinmetz (→ Abschnitt 2.4.8.2) ist es in Betracht zu ziehen, diese zusätzliche Granularitätsebene einzuführen [Wu u. a. 2001, S. 264], die zwischen verschiedenen Medienobjekten zeitliche Beziehungen herstellt. In der Literatur lässt sich dazu jedoch keine

[24]Im MET++-Rahmenwerk werden für Intrastream und Interstream die Begriffe *Intramedia (low-level)* und *Intermedia (high-level)* verwendet [Ackermann u. Eichelberg 1996, S. 73].

einheitliche Aussage finden. In Bezug auf eine verteilte Architektur kann diese Unterscheidung allerdings von Vorteil sein. Eine abschließende Betrachtung dieser Ebene kann im Rahmen dieser Arbeit jedoch nicht erfolgen.

Im Folgenden Abschnitt werden diese Ebenen in ein Referenzmodell zur Multimedia-Synchronisation eingeordnet.

Referenzmodelle Um ein besseres Verständnis über die Anforderungen an eine Multimedia-Synchronisation bilden zu können und verschiedene Multimedia-Synchronisationssysteme miteinander zu vergleichen, ist ein Referenzmodell notwendig [Steinmetz 2000, S. 601]. In der Fachliteratur werden eine Reihe von potentiell geeigneten Modellen genannt [Steinmetz 2000, S. 601]. Eines davon ist das Referenzmodell nach Little und Ghafoor [Little u. a. 1991]. Es unterscheidet zwischen der *physikalischen Ebene*, *Systemebene* sowie *menschlichen Ebene*, lässt dabei aber eine detaillierte Beschreibung von Klassifikationskriterien vermissen [Steinmetz 2000, S. 601]. Auch andere Modelle verfolgen nach Steinmetz einen eher orthogonalen Ansatz [Steinmetz 2000, S. 601]. Wesentlich besser, auch für die Zwecke von COSIMA lässt sich das Referenzmodell nach Meyer et al. [Meyer u. a. 1993] verwenden, dass in seiner durch Steinmetz weiterentwickelten Form [Steinmetz u. Nahrstedt 1995] in Abbildung 2.6 dargestellt ist. Zu seinem Vorgänger unterscheidet es sich dabei durch die Einführung in der zusätzlichen Spezifikationsschicht[25].

Die Eigenschaften und Aufgaben, die die einzelnen Schichten dabei erfüllen, sind im Folgenden kurz zusammengefasst:

Medienschicht Sie stellt eine geräteunabhängige Schnittstelle auf einzelne kontinuierliche Medienströme bereit. Dieser Medienstrom wird dabei als eine sequentielle Abfolge von LDUs behandelt [Steinmetz 2000, S. 603].

Stromschicht In dieser Schicht sind alle Medienströme aus der Medienschicht verfügbar und die LDUs sind dabei bereits nicht mehr sichtbar. Gleichzeitig wird eine Intrastream-Synchronisation garantiert und eine Interstream-Synchronisation zwischen allen Medienströmen realisiert. Die Stromschicht verarbeitet die einzelnen Ströme dabei in einer Echtzeitumgebung [Steinmetz 2000, S. 604].

[25] In diesem Referenzmodell bietet jede höhere Ebene neben einer stärkeren Abstraktion in der Programmierung zusätzlich auch eine stärkere Abstraktion bei der Behandlung der *Dienstgüte* (*Quality of Service* (QoS)) [Steinmetz u. Nahrstedt 1995, S. 601].

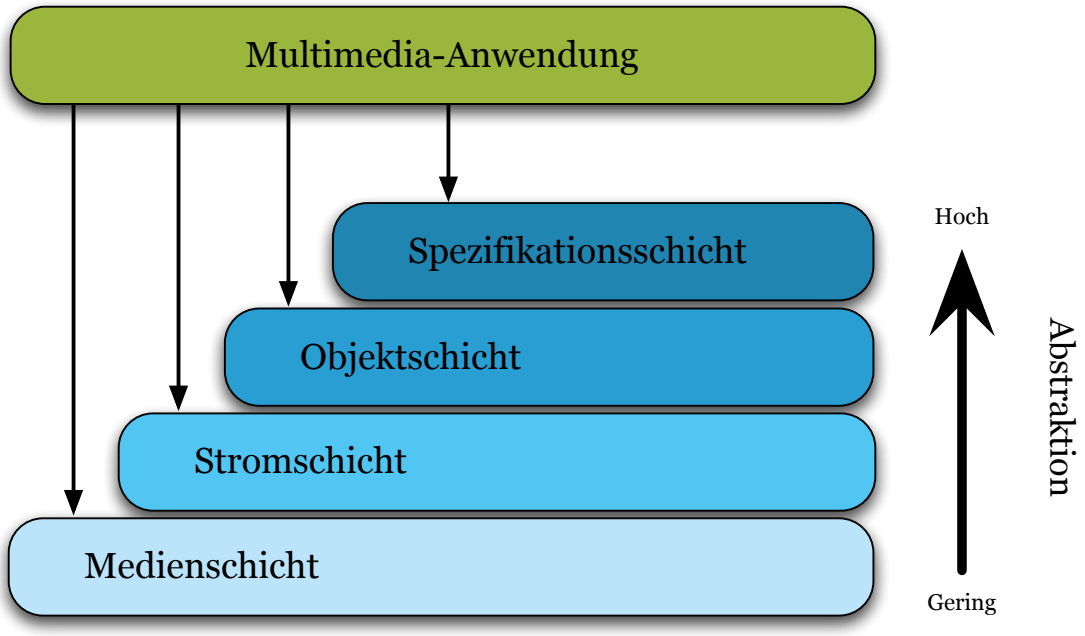

Abbildung 2.6: Erweiterte Ebenen der Synchronisation nach [Steinmetz u. Nahrstedt 1995, S. 601]

Objektschicht Diese Schicht kapselt die Unterschiede zwischen diskreten und kontinuierlichen Medien, wodurch sich *playout sequences* exakt spezifizieren lassen [Meyer u. a. 1993, S. 99]. Darüber hinaus realisiert sie die konkrete Ausführung einer Spezifikation auf der Stromschicht [Steinmetz 2000, S. 605].

Spezifikationsschicht Hier wird eine offene Schnittstelle bereitgestellt, um deklarative Synchronisations-Spezifikationen für komplexe multistrom Multimediapräsentationen zu erstellen [Blakowski u. Steinmetz 1996, S. 13]. Diese können dann von der Objektschicht ausgeführt werden. Zu diesem Zweck stellt diese Schicht auch Editoren und andere Werkzeuge bereit. Des Weiteren werden in dieser Schicht Anforderungen an die Dienstgüte aus Sicht des Benutzers auf die Objektschicht abgebildet [Steinmetz 2000, S. 607].

Sowohl die unterschiedlichen Klassifikationsansätze, als auch die einzelnen Referenzmodelle haben jeweils ihre Vor- und Nachteile für die Umsetzung von zeitlicher Synchronisation in einer Multimediaanwendungen. Bereits in [Breuer u. a. 2008, S. 28ff] wurde festgestellt, dass zu diesem Zeitpunkt kein Ansatz vollständig ausgeschlossen werden kann. Daher ist es notwendig, dass bei der Integration von Funktionalität zur Synchronisation von Medien im COSIMA-Projekt weiterhin alle Ansätze betrachtet werden. Dabei müssen auch Aspekte

zur inhaltlichen und örtlichen Synchronisation mit einbezogen werden. Wie eine mögliche Integration in COSIMA aussehen könnte, wird im nächste Abschnitt ausgeführt.

Umsetzung in COSIMA

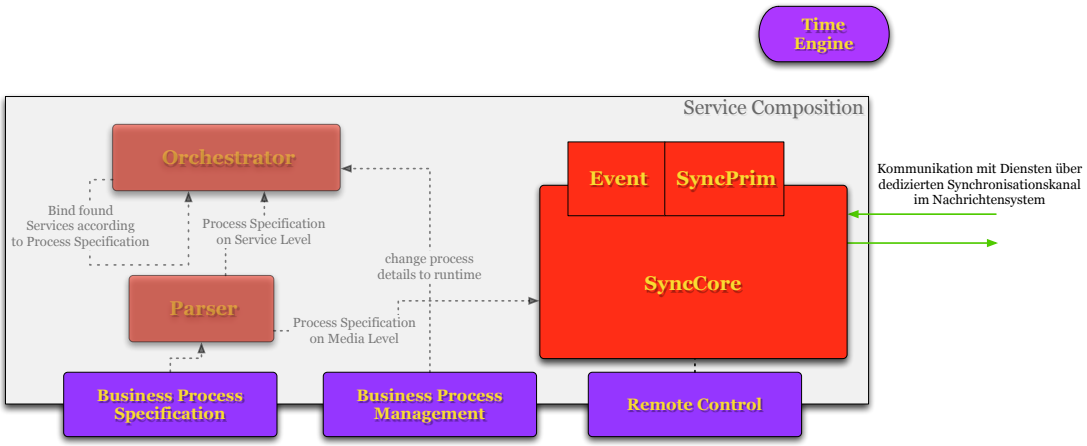

Abbildung 2.7: Komponenten zur Synchronisation im COSIMA-Projekt

In Abbildung 2.7 sind noch einmal jene Komponenten aus der Kontextsicht dargestellt, die die Funktionalitäten zur Synchronisation bereitstellen sollen. Sie lassen sich dabei jedoch nicht ohne weiteres auf eins der im vorherigen Abschnitt vorgestellten Referenzmodelle übertragen, da diese vor allem für lokale Umgebung konzeptioniert sind. Um eine Synchronisation in einer verteilten Umgebung, wie COSIMA zu realisieren, sind zusätzliche Mechanismen und Komponenten notwendig [Steinmetz 2000, S. 608ff].

Besonderheiten bei verteilten Umgebungen Das Problem, dass es in verteilten Umgebungen zu lösen gilt, besteht darin, dass Medienobjekte und Synchronisationsinformationen an unterschiedlichen Stellen vorgehalten werden [Steinmetz 2000, S. 607]. Eine Möglichkeit dieses Problem zu lösen, ist die Etablierung eines separaten Kanals[26], über den die Synchronisationsinformationen übertragen werden können [Steinmetz 2000, S. 608]. Die Synchronisation unterschiedlicher Medienobjekte wird in diesem Fall von einer Consumer-Komponente übernommen [Steinmetz 2000, S. 608]. Eine Alternative dazu wäre die Synchronisation in der Producer- (oder Transformer-) Komponente. Dazu werden

[26] Dieser Kanal ist in der erweitere SOA nach Papazoglou auf Ebene der komponierten Dienste einzuordnen (→ Abschnitt 2.4.2.3).

die zu synchronisierenden Medien in einem *Multiplex*-Datenstrom zusammengefasst und erst anschließend übertragen [Steinmetz 2000, S. 609].

Des Weiteren ist es notwendig einen globalen Zeitgeber einzuführen, um eine zeitliche Koordination der unterschiedlichen Komponenten zu erreichen [Steinmetz 2000, S. 610]. Diese Aufgabe wird im COSIMA-Projekt von der *Time-Engine* Komponente realisiert.

Spezifikation von Synchronisation Unabhängig dieser Besonderheiten, lässt sich das Synchronisationsmodell innerhalb von COSIMA am ehesten dem Referenzmodell von Meyer und Steinmetz zu ordnen. Dabei entspricht die Spezifikationsschicht in COSIMA der Komponente der Servicekomposition mit deren Subkomponente des *SyncCore*. Die Spezifikation der Synchronisationssanforderungen kann grundsätzlich über unterschiedliche Modelle erfolgen, die in [Bertino u. Ferrari 1998, S. 617] wie folgt zusammengefasst werden:

- Graphen-basierte Modelle,
- Petri-Netz basierte Modelle,
- Objektorientierte Modelle sowie
- sprach-basierte Modelle.

Grundsätzlich wurde bereits in [Breuer u. a. 2008, S. 34] darauf hingewiesen, dass unterschiedliche Modelle unterstützt werden müssen. Die Modellierung des Medienobjekt (→ Abschnitt 2.4.8.3) basiert konkret jedoch bereits auf einem Objektorientierten Modell. Zu diesem Zeitpunkt kann dadurch die Verwendung eines der anderen Modelle jedoch nicht ausgeschlossen werden.

Die konkrete Ausführung dieser Spezifikation kann auf zwei Arten geschehen. Zum einen Ereignis-gesteuert, also nicht-deterministisch[27], wie sie unter anderem bei [Little u. a. 1991] und [Bertino u. Ferrari 1998] beschrieben wird[28]. Zum anderen ist noch eine Synchronisation über sogenannte *Synchronisationsprimitive*, wie sie unter anderem bei [Gaggi u. Celentano 2005] beschrieben wird, vorgesehen.

[27] Relevant etwa bei Benutzer-Interaktionen oder Ausnahmebehandlungen

[28] Diese Sychronisationsbehandlung lässt sich sehr gut in COSIMA integrieren, da auch innerhalb einer dienstorientierten Architektur eine starke Ereignisorientierung anzustreben ist, um ein Maximum an Flexibilität zu erhalten [Masak 2007, S. 96].

Die Objektschicht entspricht innerhalb von COSIMA dem Medienobjekt (→ Abschnitt 2.4.8.3) und dem Media Broker (→ Abschnitt 2.4.8.4), die im nächsten Abschnitt eingehender beschrieben werden.

Die Stromschicht und Medienschicht sind bisher nicht explizit in COSIMA modelliert und sollten auch vollständig über das Medienobjekt für die Architektur abstrahiert werden. Eine Auseinandersetzung mit diesen Schichten sollte ausschließlich innerhalb der medienverarbeitenden Komponenten (→ Abschnitt 2.4.4) geschehen. Dadurch kann die Multimediaanwendungen auch nicht länger direkt auf alle Schichten zugreifen, wie es noch in dem Referenzmodell vorgesehen war [Blakowski u. Steinmetz 1996, S. 13]: Aus Sicht der Servicekomposition kann nur auf die Spezifikationsschicht zugegriffen werden. Aus Sicht der medienverarbeitenden Komponenten ist nur die Objektschicht sichtbar, die gekapselt die anderen Schichten bereitstellt.

Die gesamte Thematik der Synchronisation ist sehr komplex und eine Vielzahl an Aspekten muss betrachtet werden [Breuer u. a. 2008, S. 27ff]. In dieser Arbeit soll das grundsätzliche Konzept der Architektur im COSIMA-Projekt validiert werden, daher wird im Rahmen des Szenarios eine sehr einfache Betrachtung dieser Thematik durchgeführt und später entsprechend implementiert. Wesentlich umfangreicher wird dieses Thema bei [Antons 2009] behandelt. Dort wird eine nachrichtenbasierte Synchronisation im Zusammenhang von Streaming angestrebt, die auf Basis von *Message Exchange Pattern* (MEP) realisiert werden soll. Auch der Frage, ob drei Granularitätsstufen bei der Betrachtung von Synchronisation von Vorteil sind, wird dort eingehender gestellt.

2.4.8.3 Medienobjekt

Im Zusammenhang mit Synchronisation findet sich in der Literatur immer wieder der Begriff des *Medienobjekts*. Auch in dem komponentenbasierten Multimediarahmenwerk MET++ [Ackermann 1994] wird ein Medienobjekt modelliert. Unter anderem aus diesem Grund wurde auch im COSIMA-Projekt ein dediziertes Medienobjekt modelliert [Breuer u. a. 2008]. Ergebnis dieser Modellierung waren das *Medienobjekt* und der *Media Broker*, die in diesem Abschnitt näher beschrieben werden.

Es wurde bereits in Abschnitt 2.4.7.1 erläutert, dass in einer dienstorientierten Architektur der Datenfluss durch die Nachrichtenvermittlung realisiert wird[29]. In den vorherigen Ab-

[29] Obwohl der Kontrollfluss auch durch die Nachrichtenvermittlung realisiert werden kann, stehen in diesem Zusammenhang die Daten im Vordergrund.

schnitten wurde jedoch deutlich, dass Multimediadaten mitunter spezielle Anforderungen an ihre Vermittlung stellen. Um überhaupt eine Vermittlung von Medien zu realisieren, wie sie im nächsten Abschnitt beschrieben wird, muss jedoch zuvor eine Repräsentation von Medien entwickelt werden, die sich für den Einsatz in dienstorientierten Architekturen eignet. Vor diesem Hintergrund wurde das *Medienobjekt* in COSIMA mit den folgenden Anforderungen modelliert [Breuer u. a. 2008, S. 34]:

Erweiterbarkeit Das COSIMA-Projekt soll Verwendung zur Entwicklung von Multimediaanwendungen in sehr vielen Domänen finden. Daher muss sich das Medienobjekt in der jeweiligen Domänen leicht adaptieren lassen. Dies kann durch eine abstrakte Darstellung der Eigenschaften von Medien erreicht werden.

Metadaten Dort wo Medien verwendet werden, findet sich in den meisten Fällen auch eine Beschreibung dieser Medien. Diese Metadaten sind daher ebenso als *first-class citizen* zu betrachten und inhärent im Medienobjekt verankert.

Streaming Beim Umgang mit Medien, vor allem mit kontinuierlichen Medien spielen Datenströme[30] eine wesentliche Rolle [Steinmetz 2000, S. 14ff]. Aus diesem Grund muss das Medienobjekt Streaming explizit vorsehen. Eng verbunden mit dem Streaming ist die *Dienstgüte* [Steinmetz 2000] und muss daher ebenfalls berücksichtigt werden.

Verteilung In einer dienstorientierten und damit potentiell verteilten Architektur müssen auch die Medien verteilt zu prozessieren sein. Innerhalb des Medienobjektes müssen also dementsprechende Voraussetzungen geschaffen werden.

Synchronisation Wie bereits in Abschnitt 2.4.8.2 erläutert, ist es für eine Multimediaarchitektur inhärent notwenig, die Synchronisation der Medien abzubilden. Neben den bereits beschriebenen Komponenten muss auch das Medienobjekt entsprechend dieser Anforderung modelliert werden.

Das Klassendiagramm in Abbildung 2.8 zeigt das Medienobjekte in COSIMA und seine relevanten assoziierten Klassen. Im weiteren Verlauf werden die wesentlichen Merkmale des Medienobjekts im Detail beschrieben.

[30] Das englische Wort *Stream* wird in dieser Arbeit synonym verwendet.

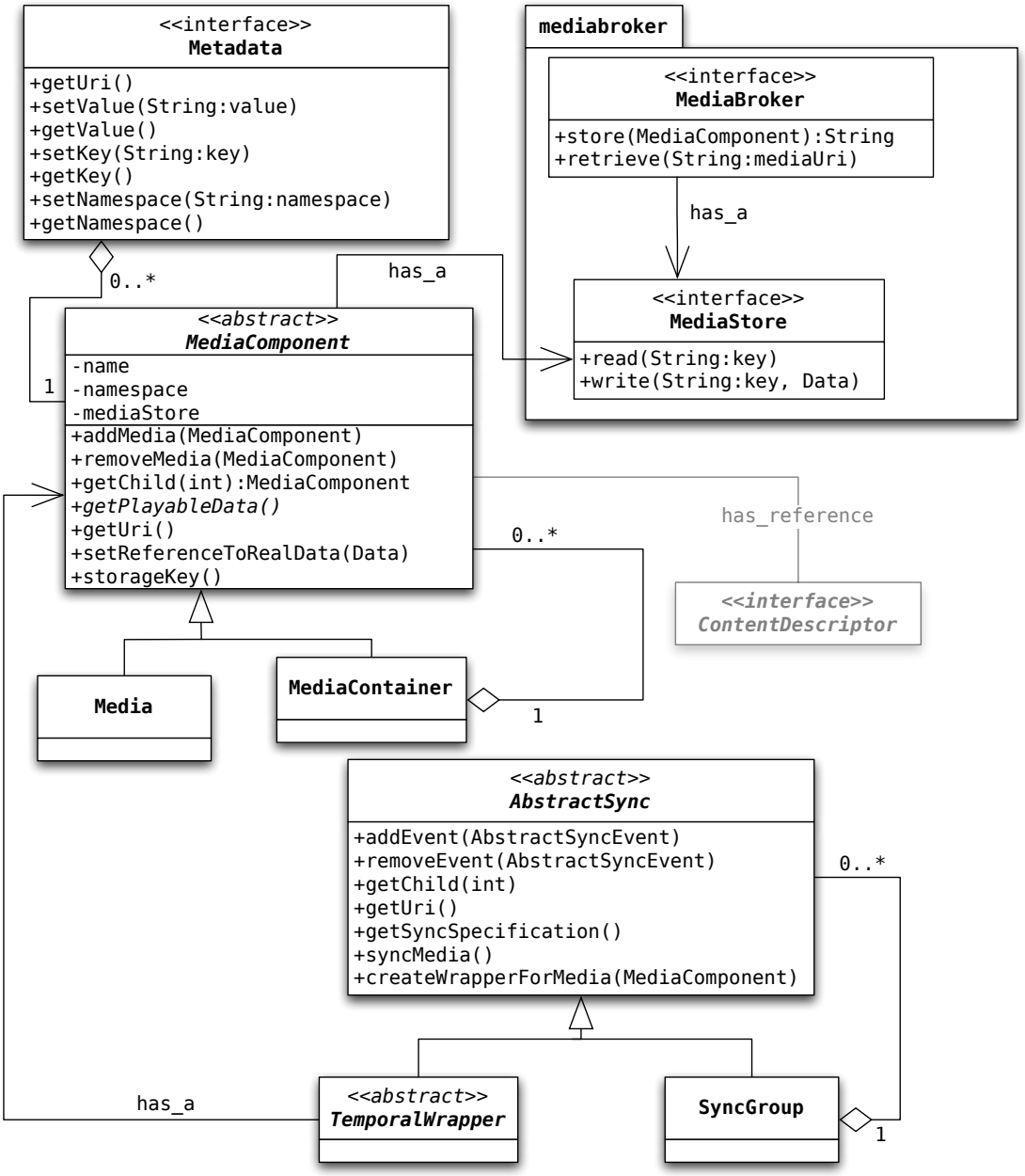

Abbildung 2.8: Klassendiagramm des Medienobjektes

Eigenschaften des Medienobjekt

Ähnlich der LDUs, wie sie bei Meyer und Steinmetz beschrieben sind (→ Abschnitt 2.4.8.2) kennt auch das Medienobjekt unterschiedliche Granularitätsstufen von Medien[31]. Diese Funktionalität ist dabei über das *Composite*-Entwurfsmuster [Gamma u. a. 1995, S. 163] realisiert. Diese Hierarchie muss dabei jedoch in eine zeitliche Reihenfolge gebracht wer-

[31] Im Falle eines Videos wäre folgende Hierarchie denkbar: *Video* ⇒ *Szene* ⇒ *Einstellung* ⇒ *Einzelbild*

den. Diese Aufgabe übernehmen dedizierte Synchronisationsobjekte, die selbst wieder als *Composite* modelliert sind. Eine generelle Anordnung in der Zeitdomäne geschieht dabei über unterschiedliche `SyncGroup`-Instanzen. Die konkrete Umsetzung der Synchronisation übernehmen dann entsprechende `TemporalWrapper`-Instanzen. Sie halten dabei Referenzen auf einzelne Medienobjekte (vgl. dazu auch [Ackermann u. Eichelberg 1996, S. 84]).

Diese Modellierung ist dabei stark an die Modellierung des MET++-Rahmenwerk angelehnt [Ackermann u. Eichelberg 1996, S. 75], im Gegensatz dazu werden medienspezifische Operationen wie *Start*, *Stop* oder *Pause* in COSIMA aber nicht im Medienobjekt realisiert, sondern in den verantwortlichen Diensten. Dadurch werden die Medienobjekte wesentlich leichtgewichtiger und die Prozessierungslogik kann auf die Dienste ausgelagert werden.

Zur Integration von beliebigen Metadaten, kann die sehr einfache `Metadata`-Schnittstelle implementiert werden. Obgleich auch als Metadaten zu interpretieren, wird die Möglichkeit zur Beschreibung des Inhalts der Medien (etwa durch MPEG-7[32]) von einer separaten Komponente realisiert, dem `ContentDescriptor`-Interface. Neben MPEG-7 sollte für COSIMA ebenfalls die Multimedia Ontolgie COMM [Arndt u. a. 2007] zur Beschreibung von Inhalten in Betracht gezogen werden. Sie setzt auf MPEG-7 auf und bietet eine wesentlich bessere Integration in webbasierten Umgebungen[33] als MPEG-7 [Arndt u. a. 2007, S. 31].

Eine weitere wesentliche Eigenschaft des Medienobjekts ist, dass es nicht die eigentlichen Mediendaten enthält, sondern über die `MediaStore`-Schnittstelle die extern gespeicherten Daten abzurufen und an die Anwendung durchzureichen. Jedes Medienobjekt lässt sich dafür über eine eindeutige URI in der Multimediaanwendungen referenzieren und ist somit lediglich eine reichhaltige Referenz auf die eigentlichen Daten. Diese liegen dabei in Form entsprechend geeigneter Formate vor, wodurch die beschriebene Granularität der *Composites* jeweils abhängig von der bereitgestellten Granularität des verwiesenen Datenformats ist.

Bei einer Einordnung in das Referenzmodell von Meyer und Steinmetz, lässt sich erkennen, dass die Medienschicht nach Meyer und Steinmetz zwar zum Teil gekapselt wird, über einzelne Composites erhalten die nutzenden Dienste dennoch Zugriff auf die jeweils kleinsten Einheiten. Des Weiteren ist die Stromschicht in COSIMA nicht länger in der

[32] `http://www.chiariglione.org/mpeg/standards/mpeg-7/mpeg-7.htm`

[33] Gemeint sind damit Systeme die auf Basis von Technologien aufgebaut sind, die vornehmlich im Internet zum Einsatz kommen. Dazu lassen sich auch Web Services zählen.

Applikation oder dem Rahmenwerk selbst realisiert, sondern wird durch externe Formate oder Protokolle bereitgestellt.

Wie diese Medienobjekte innerhalb einer Multimediaanwendungen vermittelt und übertragen werden, stellt der folgende Abschnitt vor.

2.4.8.4 Medienbroker

Definition 25 (Broker)**.** *„**1.** [engl. broker, eigtl. = Weinhändler]: Effektenhändler an der englischen und amerikanischen Börse.“*[34] *„**2.** One employed as a middleman to transact business or negotiate bargains between different merchants or individuals.“*[35] *„**3.** A middleman, intermediary, or agent generally; an interpreter, messanger, comissioner.“*[36]

Als einen Broker kann man demnach einen Makler oder Vermittler zwischen zwei Parteien bezeichnen, die Informationen oder Waren austauschen wollen.

Diese Bedeutung lässt sich auch im Umfeld von verteilten Systemen weitestgehend übernehmen[37] und im Kontext der dienstorientierten Architekturen lässt sie sich auf den *Message Broker* anwenden. Dieser stellt statt einer Punkt-zu-Punkt Verbindung zwischen Diensten, eine zentrale Vermittlungsstelle zur Verfügung [Alonso 2004, S. 71]. Dabei muss es sich aber nicht immer um ein zentrales System handeln und wird in modernen Umgebungen zumeist als dezentrale Komponente realisiert [Chappell 2004]. Grundsätzlich kann auch gesagt werden, dass ein Broker dem *Mediator* Pattern [Gamma u. a. 1995, S. 273] in einer verteilten Umgebung entspricht [Hohpe u. a. 2004, S. 83].

Der Medienbroker bei COSIMA soll ähnliche Funktionen bei der Vermittlung von Medien zwischen einzelnen Diensten übernehmen. Zur Zeit ist der Medienbroker so konzeptioniert, dass er die im vorherigen Abschnitt beschriebenen Medienobjekte zwischen den verschiedenen medienverarbeitenden Komponenten vermitteln kann.

Da die Medienobjekte selbst Referenzen auf die eigentlichen Daten darstellen, würde der Medienbroker somit nur diese Referenzen vermitteln. Wie sich Abbildung 2.8 jedoch entnehmen lässt besitzt eine `MediaBroker`-Instanz eine Referenz auf ein `MediaStore`-Objekt. Über diese Schnittstelle realisiert der Medienbroker die Persistenz der eigentlichen Daten.

[34] aus: Duden - Deutsches Universal Wörterbuch A-Z, 3. Aufl., 1996
[35] aus: The Oxford English Dictionary, 3. Aufl., 1970
[36] aus: The Oxford English Dictionary, 3. Aufl., 1970
[37] Hier ist beispielhaft der *Object Request Broker* (ORB) bei CORBA zu nennen [Balzert 1999; Coulouris u. a. 2001].

Die propagierte Trennung der Medienobjekte als Referenz und ihrer eigentlichen Mediendaten findet demnach auch bei der Speicherung statt.

Zu beachten ist allerdings, dass für den Einsatz in einer verteilte Umgebung die Implementierung der `MediaStore`-Schnittstelle einen entsprechend verteilten Datenzugriff überhaupt erst erlauben muss. Eine sehr rudimentäre Realisierung könnte beispielsweise auf der Verwendung eines NFS basieren. Eine deutlich mächtigere Lösung stellt hingegen der *Multi-Monster*-Medienserver[38] der Universität Erlangen dar. Er kann Medien in einer Echtzeitumgebung bereitstellen und dabei gleichzeitig Anforderungen an unterschiedliche Dienstgüten gerecht werden. Zusätzlich ist er in der Lage, Mediendaten in unterschiedliche Formate zu konvertieren, ist plattformunabhängig und lässt sich leicht erweitern [Suchomski u. a. 2004][39].

Der Medienbroker lässt sich in die erweiterte SOA nach Papazoglou (siehe Abbildung 2.3 auf Seite 15) als einen weiteren Kanal, neben dem Daten- und Kontrollkanal sowie Synchronisationskanal auf der Ebene der komponierten Dienste einordnen.

2.5 Offene Fragen

In diesem Kapitel ist der Status Quo des COSIMA-Projekts vorgestellt worden. Die bereits in [Breuer u. a. 2008] konzipierte Architektur wurde im Rahmen dieser Arbeit, vor allem bei der Verwendung der Begriffe aus der SOA-Domäne, weiter gefestigt. Das hier vorgestellte im weiteren Verlauf als Grundlage für die prototypische Realisierung und szenariobasierte Validierung dienen.

COSIMA ist ein junges Projekt und daher gilt es noch viele Punkte zu klären. Im Folgenden sollen die wichtigsten Punkte kurz erläutert werden. Nur einige der Fragen, die dadurch aufgeworfen werden, können aber in dieser Arbeit beantwortet werden.

2.5.1 Framework oder Architektur

In der Zieldefinition und dem Mission Statement des Projekts, wie sie zu Beginn dieses Kapitels und in [Breuer u. a. 2008] genannt wurden, wird von der Erstellung eines *Fra-*

[38] `http://www6.informatik.uni-erlangen.de/research/projects/retavic/multimonster/index.html`

[39] Das M.e.T.t. Projekt hat sich bereits im Rahmen des COSIMA-Projekts prototypisch mit einer möglichen Integration beschäftigt: `https://mims01.gm.fh-koeln.de/twiki/bin/view/MIMaster/MeT`

meworks gesprochen. Scherp und Boll nennen die folgenden drei Punkte als wesentliche Eigenschaften von Frameworks [Scherp u. Boll 2006, S. 396f]:

- Umkehrung des Kontrollflusses
- Vorgabe einer konkreten Anwendungsarchitektur
- Anpassbarkeit durch Variationspunkte

Ohne im Detail auf diese Eigenschaften einzugehen, fällt schnell auf, dass der Punkt „Umkehrung des Kontrollflusses“ sich nur schlecht mit den Konzepten einer dienstorientierten Architektur vereinen lässt: Der Servicekomposition liegt nach [Papazoglou 2008, S. 320] ein *Flow-Model* zu Grunde, dass, wie bereits in Abschnitt 2.4.6 detailliert beschrieben, den Ablauf einer Applikation steuert. Es unterliegt dabei dem direkten Einfluss des Anwendungsentwicklers, wie in Abbildung 2.4 deutlich zu erkennen ist. Aus Sicht des Anwendungsentwicklers handelt es sich demnach nicht um ein Rahmenwerk.

Wird die Perspektive jedoch zu einem Dienstanbieter gewechselt, erfüllt COSIMA durchaus alle Kriterien, um als Rahmenwerk bezeichnet zu werden: Ihm wird eine konkrete Anwendungsarchitektur vorgegeben, er kann einige Eigenschaften gezielt beeinflussen und hat keinen Einfluss auf den Kontrollfluss.

Diese Divergenz wird noch verstärkt, wenn die Spezifikation der Dienstkomposition genauer betrachtet wird. Die Spezifikation wird lediglich deklarativ vorgenommen, ihre konkrete Ausführung selbst obliegt einer Komponente, auf die der Anwendungsentwickler nur wenig Einfluss nehmen kann. Ähnlich dem Optimieren von Anweisungen bei Compilern oder Datenbanken, kann auch in diesem Fall die eigentlich Ausführung deutlich anders aussehen, als sie von dem Anwendungsentwickler spezifiziert wurde. Demnach gibt tatsächlich auch der Anwendungsentwickler einen Teil des Kontrollflusses wieder an COSIMA ab.

Es lässt sich demnach nicht eindeutig feststellen, ob es sich bei COSIMA um ein Rahmenwerk oder eine reine Architektur handelt. Abhängig von der Sichtweise sind beide Charakterisierungen valide und können demnach so verwand werden. In weiteren Arbeiten sollte dieser Januskopf aber weiterhin bedacht werden.

2.5.2 Servicekomposition

Bei einer Weiterentwicklung muss in jedem Fall die Komponente der Servicekomposition weiter ausdifferenziert werden. Es muss eingehend die Eignung bestehender Sprachen

zur Komposition von Diensten im Kontext von Multimediaanwendungen geprüft werden. Auch die Frage, ob eine Orchestrierung oder eine Choreographie vorzuziehen ist, oder eine Mischung aus beiden Ansätzen, wie sie auch bei [Papazoglou u. a. 2007] gefordert wird, muss geklärt werden.

Ein weiterer Aspekt, der in den Verantwortungsbereich der Servicekomposition fällt, ist die Behandlung nicht-funktionaler Anforderungen einzelner Dienste [Papazoglou u. a. 2007, S. 42]. Dieser Aspekt wurde bisher bewusst bei der Entwicklung des COSIMA-Projekts völlig ausgeblendet und sollte in zukünftigen Arbeiten prominenter behandelt werden.

Nach [Papazoglou 2003, S. 8] stehen im Kontext des *Service-Oriented Computing* (SOC)[40] komponierte Dienste für weitere Kompositionen zur Verfügung. Diese Möglichkeit ist zur Zeit nicht explizit in COSIMA modelliert worden. Auch für die Entwicklung von Multimediaanwendungen kann diese Fähigkeit von Vorteil sein. Welche Konsequenzen eine solche verschachtelte Komposition haben könnte, kann hier nicht abschließend geklärt werden und sollte daher Teil weiterer Forschung sein.

Ebenfalls sollte evaluiert werden, inwiefern sich die Ergebnisse aus [Richter 2008] in Bezug auf die Verwendung von BPEL und der dort vorgestellten Erweiterung von BPEL, *BPEL for Multimedia* (BPEL-MM) in die jetzige Dienstkomposition integrieren lassen.

2.5.3 Synchronisation

Es wurde gezeigt, dass Synchronisation ein elementares Element für jede Multimediaanwendung darstellt. Die Tiefe, in der dieses Thema im Rahmen der vorliegenden Arbeit behandelt werden kann, wird dieser Relevanz jedoch nicht vollständig gerecht. So bleibt unter anderem die Frage offen, wie die Realisierung von Echtzeit-Anforderungen aussehen kann. Bei Steinmetz ist die Stromschicht in eine Echtzeitumgebung eingebettet [Steinmetz 2000, S. 604], die Schichten darüber jedoch nicht mehr. In der jetzigen Modellierung wird die Stromschicht von einer externen Komponente realisiert. Bei Verwendung des MultiMonster-Medienservers ist dieser zwar echtzeitfähig, eine echte Überprüfung für die Realisierung von Echtzeitanforderungen steht aber noch aus.

Bei [Antons 2009] wird eine Realisierung von Synchronisation über die Verwendung von Message Exchange Pattern und einem separaten Synchronisationskanal angestrebt. Ob

[40] Bei SOC handelt es sich um ein Paradigma, in das sich eine SOA begrifflich einbetten lässt [Liebhart 2007, S. 8].

dieser Ansatz zu einem optimalen Ergebnis führt, bleibt abzuwarten. In jedem Fall sollten andere Möglichkeiten evaluiert werden, verteilte Synchronisation umzusetzen.

2.5.4 ESB

Bei der Realisierung eines Enterprise Service Bus muss zunächst die fundamentale Entscheidung getroffen werden, ob er API- oder protokollgetrieben implementiert werden soll [Josuttis 2007, S. 59]. Ohne diese beiden Ansätze an dieser Stelle weiter zu vertiefen, muss die Weiterentwicklung des Projekts sich für einen der beiden entscheiden.

Es existieren eine Vielzahl von kommerziellen und offenen ESB Lösungen. Deren Eignung für die Verwendung innerhalb des COSIMA-Projekts muss im Einzelnen geprüft werden. Bei dieser Integration ist jedoch zu beachten, dass COSIMA selbst bereits als ESB charakterisiert werden kann. Viele der genannten Eigenschaften lassen sich in der Architektur von COSIMA wieder finden. Es könnten demnach entweder Teile eines bestehenden ESB in COSIMA integriert werden, oder ein bestehender ESB um die Multimedia Charakteristika von COSIMA erweitert werden.

3 Szenario

Wie bereits im vorangegangen Kapitel erwähnt, liegt dem COSIMA-Projekt ein dediziertes Vorgehensmodell zu Grunde, das speziell für dieses Projekt entwickelt wurde [Breuer u. a. 2008, S. 7]. Die schematische Darstellung dieses Modell findet sich in Abbildung 3.1. Im Folgenden werden die Kernpunkte dieses Vorgehens noch einmal zusammengefasst:

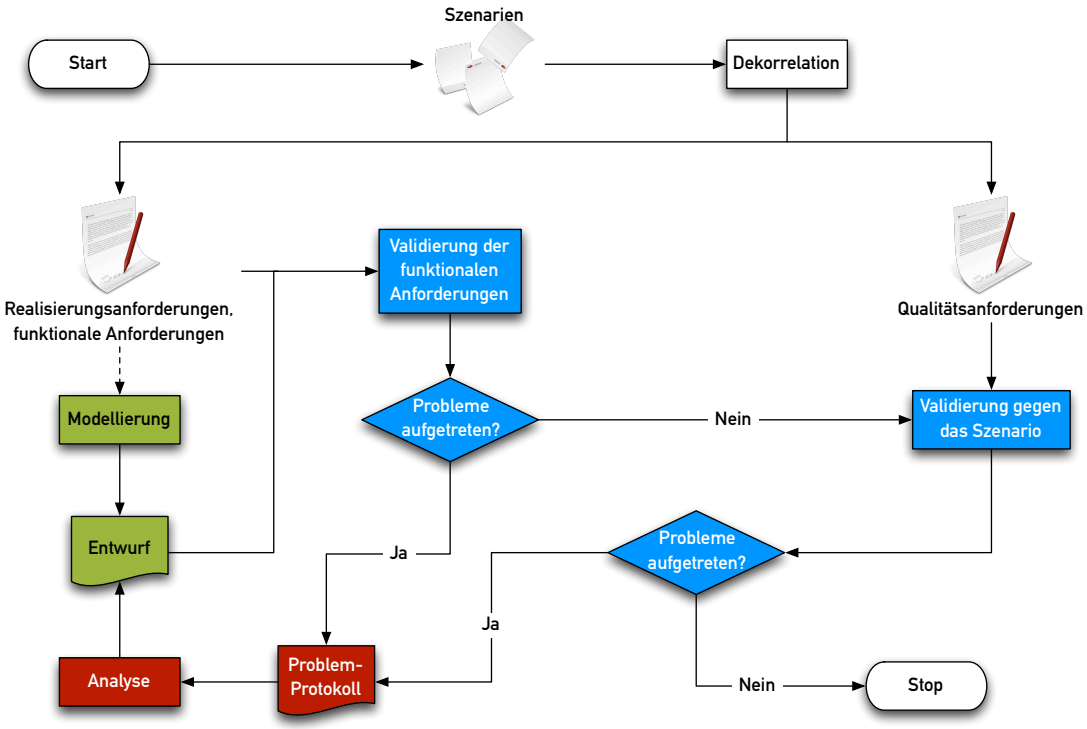

Abbildung 3.1: Schematische Darstellung des Vorgehensmodell hinter dem COSIMA-Projekt nach [Breuer u. a. 2008]

Iterativ Ein Iteratives Vorgehen bei der Entwicklung von Software ist ein etabliertes Verfahren, um der Unvollständigkeit der Anforderungsermittlung gerecht zu werden [Basili u. Turner 2005; Boehm 1986; Brooks 1987; Kruchten 2004]. Da auch im

COSIMA-Projekt nicht zu erwarten war, dass alle relevanten Anforderungen von Anfang an offensichtlich sind, wurde in diesem Kontext ein iterativer Prozess gewählt.

Szenario-basiert Die Verwendung von Szenarien ist in vielen Bereichen der Softwareentwicklung[1] ein anerkanntes Verfahren um Anforderungen und Qualitätsmerkmale von Software konkret zu formulieren.

Top-Down Es werden zuerst abstrakte Szenarien definiert, die zum einen sehr informell, zum anderen sowohl funktionale, wie auch nicht-funktionale Anforderungen in sich vereinen[2]. Im Zuge der Iterationen werden diese Szenarien immer detaillierter und formeller. Die Architektur entwickelt sich entsprechend ebenso weiter. Daher kann man hier von einem *top-down* Vorgehen sprechen.

Ein elementarer Bestandteil dieses Vorgehenmodells ist also die Verwendung von Szenarien. Da diese Arbeit erstens in das Umfeld des COSIMA-Projekts einzuordnen ist und Szenarien eine anerkannte Methodik darstellen, werden sie auch in dieser Arbeit dazu verwendet, die bisherige Architektur zu validieren. Im Folgenden Abschnitt wird der Begriff des Szenario genauer definiert und eingeordnet. Bevor dann das eigentliche Szenario dargestellt wird, findet noch eine Abgrenzung zu Szenarien statt, wie sie den, in der Literatur beschriebenen szenariobasierten Methodiken Verwendung finden.

3.1 Definitionen

Im Duden findet sich zu „Szenario“ der folgende Eintrag:

Definition 26 (Szenario (allg.))**.** *„(in der öffentlichen u. industriellen Planung) hypothetische Aufeinanderfolge von Ereignissen, die zur Beachtung kausaler Zusammenhänge konstruiert wird.“*[3]

Eine Beschreibung, die aus der Domäne der Softwareentwicklung stammt, liefern Kazman et al.:

[1] Neben der Architekturentwicklung [Bass u. a. 2003; Ionita u. a.] und der Anforderungsermittlung [Weidenhaupt u. a. 1998a], etwa im Bereich der Mensch-Computer-Interaktion [Carroll 2000].

[2] Nach [Weidenhaupt u. a. 1998b, S. 42f] ist dieses Vorgehen auch bei anderen szenariobasierten Modellen durchaus üblich.

[3] aus: Duden - Deutsches Universal Wörterbuch A-Z, 3. Aufl., 1996

Definition 27 (Szenario (Kazman et al.)). *„Scenarios are brief narratives of expected or anticipated use of a system from both development and end-user viewpoints.“* [Kazman u. a. 1996, S. 2]

Auch in dieser Definition wird von einer Narration und damit eher etwas Hypothetischem gesprochen. Die Folgende, etwas später formulierte Definition stellt noch einen weiteren wichtigen Punkt heraus:

Definition 28 (Szenario (Clements et al.)). *„A scenario is a short statement describing an interaction of one of the stakeholders with the system.“* [Clements u. a. 2002, S. 33]

Betrachtet man zu dieser Definition, die aus der reinen Softwareentwicklung stammt, eine weitere, die ihren Ursprung in der Mensch-Computer-Interaktion hat, so ergibt sich ein, für diese Arbeit verwendbares, Gesamtbild:

Definition 29 (Szenario (Carroll & Rosson)). *„A [...] scenario is a story about people and their activities. [They] have a plot; they include sequences of actions and events, things that actors do, things that happen to them, changes in the setting and so forth.“* [Rosson u. Carroll 2002, S. 16/18]

Im Rahmen dieser Arbeit soll ein Szenario daher als eine hypothetische „Geschichte“ verstanden werden, die unterschiedliche *Stakeholder*[4], ihre Aktivitäten und Interaktionen mit einem Softwaresystem in einem Nutzungskontext beschreibt.

Als Stakeholder werden dabei alle Gruppen oder Personen aufgefasst, die irgendwann mit dem System in Kontakt treten können. So kennt die *ATAM-Methode* von Softwarearchitekten über den Softwareentwickler und Tester bis hin zum Projektmanager und Endanwender, eine Vielzahl von Stakeholdern [Clements u. a. 2002, S. 63ff]. Im Rahmen dieser Arbeit ist vor allem der Softwareentwickler von Interesse, da geprüft werden soll, in wie weit er in der Lage ist, mit der gegebenen Architektur eine Zielanwendung zu implementieren.

Im folgenden Abschnitt werden nun einige anerkannte szenariobasierte Methoden skizziert und diskutiert, welche Methode dieser Arbeit zu Grunde liegen wird.

[4] Ein *Stakeholder* ist ein Aktuer oder Mitglied einer Interessengruppe. Da der Begriff auch in der deutschsprachigen Fachliteratur Verwendung findet, wird er auch in dieser Arbeit weiterhin so angewandt.

3.2 Szenariobasierte Methoden

Es existieren eine Vielzahl von szenariobasierten Methoden zur Evaluation von Softwarearchitekturen. Die meisten gehen dabei auf die Methoden SAAM und ATAM zurück [Ionita u. a., S. 1], die initial am Carnegie Mellon Institut entwickelt worden sind. All diese Methoden haben dabei das Ziel, die Qualitätsmerkmale, also die nicht-funktionalen Anforderungen einer Architektur zu evaluieren. In diesem Fall „operationalisieren [Szenarien die] Qualitätsmerkmale [einer Architektur] und machen sie messbar“ [Starke 2008, S. 61]. Wie aber bereits erwähnt (→ Abschnitt 2.5.2), wurden bei der Entwicklung von COSIMA nicht-funktionale Anforderungen ausgeblendet. Daher eignen sich diese Methoden nur bedingt zur Verwendung im Rahmen dieser Arbeit. Dennoch ist es aber durchaus als sinnvoll zu erachten, die Architektur in einem so frühen Stadium bereits zu überprüfen:

> *„Evaluation need not wait until an architecture is fully specified.“* ([Clements u. a. 2002, S. 24])

Gleichzeitig ist es nicht zu empfehlen, die Evaluation in diesem frühen Stadium in vollem Umfang durchzuführen:

> *„[. . .] in practice, the expense and logistical burden of convening a full-blown evaluation is seldom undertaken when unwarranted by the state of the architecture.“* ([Clements u. a. 2002, S. 24])

Aus diesen beiden Gründen, *Ausblendung der nicht-funktionalen Anforderungen* und *Unvollständigkeit der Architektur*, wird im Rahmen dieser Arbeit darauf verzichtet, eine der etablierten szenariobasierten Methoden anzuwenden[5].

Dennoch ist es weiterhin empfehlenswert, Szenarien zu verwenden, da sie sich gut zur Abbildung von Interaktionen zwischen Stakeholdern und dem System eignen. Daher soll das Szenario dieser Arbeit die Grundstrukturen etablierter Vorgehen adaptieren. Im Folgenden werden diese Strukturen kurz skizziert.

[5] Hinzu kommt, dass der Umfang einer Master Thesis es nicht erlaubt eine Evaluation durchzuführen, an der methodisch mehrere Personen arbeiten müssen, wie es etwa bei der ATAM oder SAAM Methode der Fall ist.

3.2.1 Arten von Szenarien

Es existieren unterschiedliche Arten oder Typen von Szenarien, abhängig von der Domäne in der sie Verwendung finden. Da für die Domäne der Softwarearchitekturen die ATAM-Methode der Carnegie Mellon University [Kazman u. a. 2000] ein etablierter Vertreter ist, wird hier die Einteilung nach dieser Methode als Grundlage genommen:

Anwendungsszenarien Anwendungsszenarien beschreiben die intendierte Interaktion zwischen dem System und einem Benutzer. Dabei ist das System als vollständig und lauffähig zu erachten [Kazman u. a. 2000, S. 14].

Änderungsszenarien Änderungsszenarien repräsentieren die Durchführung von typischen, erwarteten Änderungen an dem System [Kazman u. a. 2000, S. 14f].

Stressszenarien In Stressszenarien wird das System an seine definierten Grenzen gebracht, um zu sehen, wie es in Extremsituationen reagiert [Kazman u. a. 2000, S. 15].

Das Szenario in dieser Arbeit ist den Anwendungsszenarien zuzuordnen. In diesem speziellen Fall ist der „Anwender“ der Entwickler einer Applikation, die durch die Situationsbeschreibung in 3.3 dargestellt ist.

Die Szenarien selbst folgen einem bestimmten formalen Aufbau, der im folgenden Abschnitt dargestellt ist.

3.2.2 Aufbau von Szenarien

Nach [Bass u. a. 2003, S. 75] konstituieren folgenden Elemente ein Szenario[6]:

Quelle des Auslösers Der Stakeholder, der den Stimulus generiert. Die Quelle kann dabei sowohl ein Mensch, als auch ein anderes System sein.

Auslöser (*Stimulus*) Stellt die auslösende Interaktion zwischen System und Stakeholder dar.

Umgebung (*Environment*) Beschreibt den Zustand in dem sich das System zum Zeitpunkt des Eintritts des Stimulus befindet.

[6] Übersetzungen nach [Starke 2008, S. 63]

Systembestandteil (*Artifact*) Beschreibt welche Teile des Systems durch den Auslöser betroffen sind. Es kann dabei auch das ganze System betroffen sein.

Antwort (*Response*) Das Verhalten, das eintritt, nach dem der Auslöser auf das System gewirkt hat.

Antwortmetrik Beschreibt wie das Verhalten erfasst und gemessen werden kann.

Die ATAM-Methode im speziellen konzentriert sich auf das Tripple von *stimulus-environment-response*. Dabei sind idealerweise alle Szenarien in dieser Form zu formulieren. In der praktischen Durchführung lässt sich dieser Idealzustand jedoch nur selten in seiner Vollständigkeit erreichen [Clements u. a. 2002, S. 53]. Da der Fokus dieser Methode auf der Evaluierung von nicht-funktionalen Anforderungen liegt, ist eine vollständige Übertragbarkeit auf die Bedürfnisse in diesem Rahmen ohnehin nicht gegeben. Dennoch soll versucht werden, eine Gliederung des Szenario in ähnlicher Art und Weise vorzunehmen.

Im Folgenden wird nun das Szenario für die später Validierung vorgestellt, die in Kapitel 5 durchgeführt wird.

3.3 Eine Verteilte Lehrveranstaltung

Das hier beschriebene Szenario bildet die Grundlage für das in Abschnitt 3.4, das in Abschnitt 4.2 prototypisch implementiert werden soll.

3.3.1 Zur Fachdomäne des Szenarios

Grundlegend für den Erfolg der Durchführung einer szenariobasierten Methode ist die Qualität der Szenarien selbst. Diese wiederum werden aus Workshops oder Interviews mit den unterschiedlichen Stakeholdern gebildet. Das Endergebnis ist also im hohen Maße abhängig von der richtigen Auswahl der Stakeholder [Clements u. a. 2002, S. 187]. Da für das Szenario für die spätere Validierung der Architektur jedoch nur der Autor selbst als Stakeholder zur Verfügung stand, war es notwendig eine Domäne für das Szenario zu wählen, mit der der Autor hinreichend vertraut ist. Ein Szenario angesiedelt in der Hochschullehre lag daher nahe. Da es sich um eine verteilte Anwendung handeln muss, wird ein Szenario gewählt, dass sich grob dem Umfeld von CSCW beziehungsweise CSCL zuordnen

lässt. Darüber hinaus wurde bereits erkannt, dass sich der Einsatz einer dienstorientierten Architektur für den Einsatz einer Lernplattform durchaus als sinnvoll zu erachten ist [Hüvelmeyer u. a. 2004].

Im Vordergrund dieser Arbeit steht aber nicht die Anwendung selbst, sondern die Validierung der Architektur des COSIMA-Projekts. Daher wird auf eine genauere Vorstellung der Domäne an dieser Stelle verzichtet. Im Folgenden wird nun zuerst die Situation hinter dem eigentlichen Anwendungsszenario beschrieben.

3.3.2 Beschreibung der Situation

Die Fachhochschule Köln unterhält mehrere Standorte innerhalb und rund um Köln, deren angeschlossene Institute jeweils thematische Überschneidungen aufweisen. Dadurch aber, dass immer mehr interdisziplinäre Studiengänge angeboten werden, hemmen diese geographischen Grenzen mitunter die Entfaltung des vollen Potentials dieser interdisziplinären Studiengänge. Abhilfe soll hier die Integration unterschiedlicher Studiengänge und Fachbereiche auf Ebene einzelner Veranstaltungen schaffen. Eine Zusammenlegung der Standorte ist dabei aber als unrealistisch einzustufen und so haben sich die Verantwortlichen für eine *virtuelle* Lösung des Problems entschieden: Durch den Einsatz eines verteilten, virtuellen Klassenraums sollen Dozenten und Studenten unterschiedlicher Fachbereiche und Standorte an einer gemeinsamen Lehrveranstaltung teilnehmen und diese aktiv mitgestalten können.

Als Pilotprojekt wurde die Veranstaltung „Medienrezeption“ im Studiengang „Medieninformatik Master“, am Standort Gummersbach ausgewählt. Diese soll zusammen mit den Studenten des Studiengangs „Medientechnik“, am Standort Deutz durchgeführt werden. Auf Basis des COSIMA-Projekts wurde ein System entwickelt, das diesen verteilten Klassenraum bereitstellen soll. Eine narrative Beschreibung dieser Situation findet sich in dem Kasten „Narrative Beschreibung der Situation“.

Narrative Beschreibung der Situation

Es ist ein sonniger Mittwoch Vormittag und die Studenten des Masterstudiengangs der Medieninformatik finden sich gegen 09:10 Uhr vor dem Raum 3324 an der Fachhochschule Gummersbach ein. Heute steht „Medienrezeption“ auf dem

Plan. Prof. Dr. Hugo Klebb ist bereits anwesend, da er noch einige technische Vorbereitungen treffen musste. Die Veranstaltung findet zusammen mit Studenten des Studiengangs Medientechnik aus Deutz stattfinden. Das Besondere dabei ist, dass die Studenten nicht vor Ort sein werden, sondern durch ein neues System virtuell an der Veranstaltung teilnehmen.

Kurz nachdem sich alle gesetzt haben, steht auch schon die Videoverbindung nach Deutz. Über die zwei im Raum befindlichen Beamer sehen die Gummersbacher Studenten neben den Folien von Professor Klebb auch die Folien von Prof. Dr. Julius Largo aus Deutz. Des Weiteren ist sowohl ein Bild vom Professor Largo selbst, sowie ein Bild der versammelten Deutzer Studenten zu sehen.

Nachdem nun alle Studenten anwesend sind und die Daten- und Videoverbindungen zwischen den Standorten Gummersbach und Deutz steht, beginnt die Veranstaltung. Aufgabe für heute war die Vorbereitung eines wissenschaftlichen Artikels zum Thema „Agenda-Setting“. Jeder Student sollte eine Textanalyse dieses Artikels durchführen. Die Ergebnisse wurden im Vorfeld, über das interne Wiki-System der Medieninformatik, online den anderen Studenten zur Verfügung gestellt, so dass jeder bereits über das Material des anderen verfügt. Die Veranstaltung ist als seminaristischer Unterricht ausgelegt, es findet also kein klassischer Frontaluntericht statt, sondern eine Diskussion zwischen Dozenten und Studenten auf Augenhöhe. Ziel der Veranstaltung ist es, bei der gemeinsamen Diskussion über den Artikel ein tieferes Verständnis der Thematik zu erhalten.

Die Studenten, die Laptops dabei haben, melden sich zusätzlich noch im internen Netz der Fachhochschule an. Jeder kann dadurch einem speziellen Chat-Raum beitreten, der nur für diese Veranstaltung gedacht ist. Sie können so parallel zu dem Geschehen in den beiden Klassenräumen, weitere Punkte diskutieren, dokumentieren oder anmerken. Außerdem steht ihnen über eine spezielle Applikation auf dem eigenen Laptop die Möglichkeit zur Verfügung, die aktuelle sowie vergangene Veranstaltungen Revue passieren zu lassen. Von dieser Möglichkeit macht denn auch gleich Vanessa gebrauch, die etwa 15 Minuten zu spät erscheint. Sie stand noch im Stau auf dem Weg nach Gummersbach. Sie meldet sich nachträglich am System an und scannt über das bisherige Chat Log. Sie kann so die bisher wesentlichen Punkte leicht erfassen und kommt schnell wieder in die Veranstaltung rein.

Während der Veranstaltung haben die Studenten den Text, über den gesprochen wird, ebenfalls auf ihren lokalen Systemen im PDF-Format verfügbar. Sie können dabei das PDF annotieren und diese Annotationen den anderen Studenten direkt wieder zur Verfügung stellen. Auch die Textanalysen der anderen Studenten stehen ihnen über den selben Kanal zur Verfügung. Im Laufe der Veranstaltung reichern alle Studenten und Dozenten über ihre textuellen Annotationen und das Gesprochene den Text immer weiter mit Informationen an. Das System stellt dabei selbstständig Querverweise zwischen den Informationen und dem Text her. Der Autor jeder Information wird dabei ebenfalls gespeichert. Zusätzlich zur reinen Speicherung der Informationen und ihrer Beziehungen untereinander, bewertet das System auch die Relevanz der Informationen im Kontext der Veranstaltung.

Am Ende ist so eine stark angereicherte Version des ursprünglichen Textes entstanden, der mit unterschiedlichen Medien in Bezug gesetzt wurde, die allesamt dokumentieren, wie sich das Verständnis über den Text im Verlauf der Veranstaltung verändert hat. Im Nachhinein steht diese Version an einer zentralen Stelle zur Verfügung und kann zum Beispiel später zur Prüfungsvorbereitung genutzt werden.

3.3.3 Systemkomponenten

Die im vorherigen Abschnitt beschriebene Situation beschreibt sehr visionär, welche Applikationen das COSIMA-Projekt einmal ermöglichen soll. Auch wenn im Rahmen dieser Arbeit sicher nur ein kleiner Teil dieser Vision prototypisch umgesetzt werden kann, sollen hier dennoch die wesentlichen Systemkomponenten beschrieben werden, die zur Realisierung dieser Anwendung nötig sind. In Abbildung 3.2 sind diese und ihre jeweiligen Kommunikationskanäle schematisch dargestellt[7]. Im Folgenden sind die Komponenten im Einzelnen beschrieben.

- An jedem Standort sind zwei hochauflösende Kameras installiert. Eine davon filmt den Dozenten, die andere das Auditorium.

[7] Die lokalen Rechner mit der COSIMA-basierten Applikation entsprechen in etwa einer MCU in einem Telekonferenzsystem. Wobei neben dem Controller auch ein Prozessor vorhanden ist, da verschiedene Medientypen gemuxt werden müssen [Schwabe u. a. 2001, S. 224].

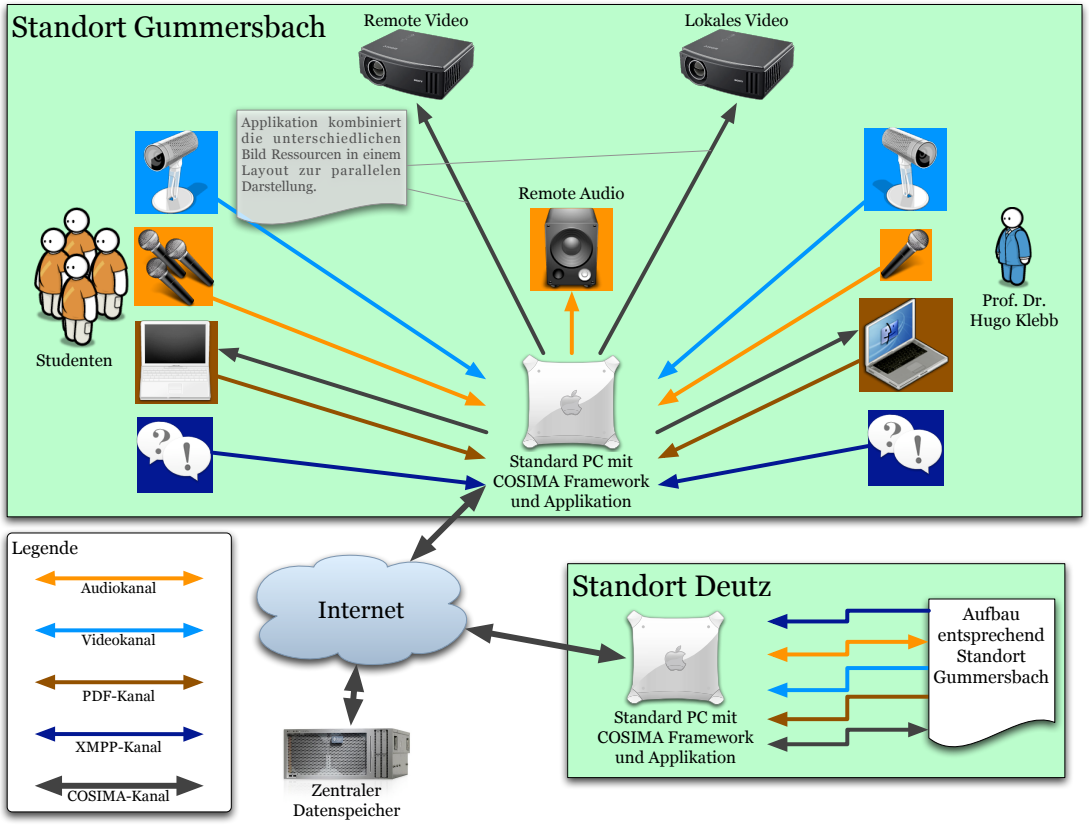

Abbildung 3.2: Schematische Darstellung der Komponenten und ihrer Kommunikationskanäle

- An jedem Standort sind zwei Projektoren installiert. Einer zeigt beide Videobilder der Gegenseite, der andere Folien oder andere Informationen des jeweils nichtentfernten Standortes.
- Jeder Standort verfügt über einen PC, an dem alle Ein- und Ausgabegeräte angeschlossen sind, und der die Kommunikation zwischen beiden Standorten verwaltet.
- Mobile Computersysteme der Studenten oder Dozenten können bei Bedarf über den Kommunikations-PC in das System integriert werden.
- Über ein Dozentenmikrofon sowie zusätzliche Raummikrofone werden die Gespräche erfasst und an die Gegenseite übermittelt.
- Jeder Standort verfügt zur Wiedergabe der Audiodaten über ein Lautsprechersystem.

3.3.4 Zusätzliche Informationen

Die Veranstaltung „Medienrezeption“ findet erfahrungsgemäß im kleinen Kreise statt, da es sich um ein Fach im Masterstudiengang handelt. Etwa 10-15 Studenten der Medieninformatik besuchen sie. Bei den Medientechnikern wird es nicht als Pflichtfach, sondern als Wahlpflichtfach angeboten, daher nehmen in der Pilotphase nur 5 Studenten aus Deutz teil.

Es wird für dieses Setup keine außergewöhnliche Technik benötigt, was die Umsetzung deutlich vereinfacht: Bei allen Komponenten handelt es sich um handelsübliche Standard Hardware. Die gesamte Steuerung und Kommunikation wird durch die Applikation auf den beiden Kommunikationsrechnern geregelt, die dabei auf dem COSIMA-Projekt basiert. Bei den Rechnern handelt es sich ebenfalls um Standard PCs.

Die Applikation übernimmt unterschiedliche Aufgaben bei diesem Anwendungsfall, so wird von ihr zum einen das Bild beider Kameras übertragen zum anderen auch das Signal der jeweils zusätzlich angeschlossenen Endgeräte, etwa die Laptops der Studenten. Zusätzlich wird das Audiosignal der Mikrofone übertragen. Die Darstellung des Computersignals erfolgt dabei synchron zu der Darstellung des Videosignals. Die Lautstärke der Mikrofone wird von der Applikation automatisch so angepasst, dass der aktuelle Sprecher optimal zu hören ist und nur wenig Umgebungsgeräusche übertragen werden.

3.4 Das Anwendungsszenario

Wie bereits zu Beginn dieses Kapitels erwähnt, ist für die Validierung der Architektur im Rahmen dieser Arbeit der Anwendungsentwickler als Stakeholder von Interesse. Es gilt zu validieren, ob sich die Architektur grundsätzlich dazu eignet, eine verteilte Multimediaapplikation zu implementieren. Grundlage für die Validierung soll ein *Anwendungsszenario* sein, das in diesem Abschnitt definiert wird. Die in den vorherigen Abschnitten vorgestellte Situation einer verteilten Lehrveranstaltung bietet den visionären Rahmen für dieses Szenario.

Als eine zentrale Komponente für die Realisierung der zuvor beschriebenen Situation einer verteilten Lehrveranstaltung kann die Erfassung von Video- und Audiosignalen sowie deren Transport über ein Netzwerk identifiziert werden. Der Anwendungsentwickler soll in diesem Szenario daher genau diese Funktionalität implementieren. In Abbildung 3.3 ist dieses

Szenario einmal schematisch dargestellt. Im Folgenden wird genauer auf die einzelnen Elemente eingegangen.

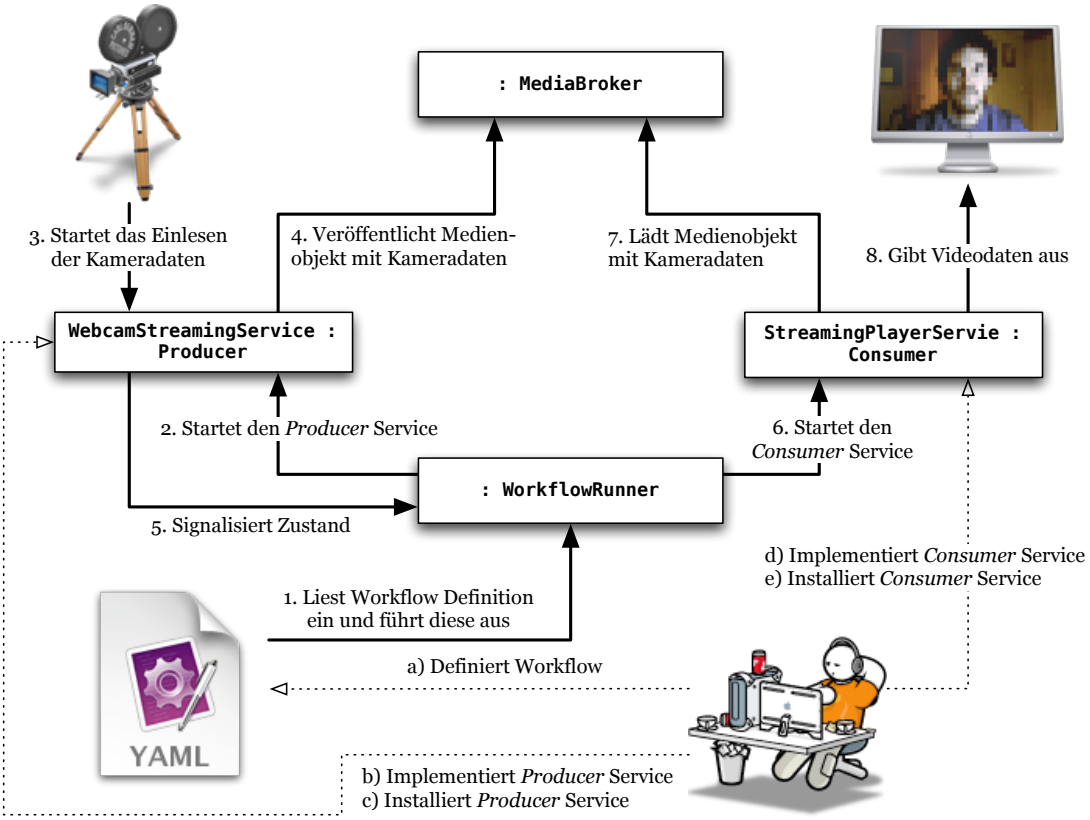

Abbildung 3.3: Schematische Darstellung des Anwendungsszenario

In diesem Szenario sind von dem Anwendungsentwickler eine Reihe von Schritten notwendig, um die Funktionalität am Ende liefern zu können. Der erste Schritt ist dabei, die Definition des Ablaufs der Anwendung. Dies geschieht in Form einer Beschreibungsdatei (Schritt *a)* in der Abbildung). Anschließend implementiert er die, zur Realisierung der Anwendung notwendigen Dienste. In diesem Falle also den `WebcamStreamingService` für die Erfassung der Videodaten und den `StreamingPlayerService` für die Darstellung der Videodaten (Schritte *b)* und *d)* in der Abbildung). Die Vermittlung der Daten wird dabei von dem `MediaBroker` übernommen. Abschließend müssen die beiden Dienste nur noch auf den entsprechenden Systemen an den jeweiligen Standorten installiert werden (Schritte *c)* und *d)* in der Abbildung).

Das Ergebnis der Implementierung ist, dass der `WebcamStreamingService`, nach Start der Anwendung beginnt, Daten von der angeschlossenen Videokamera abzugreifen und diese

Daten über den `MediaBroker` bekannt zu geben. Auf der Gegenseite kann der `StreamingPlayerService` anschließend wieder über den `MediaBroker` auf diese Daten zugreifen und sie darstellen. Eine *unidirektionale* Videoverbindung zwischen zwei Endpunkten ist hergestellt.

4 Prototypische Realisierung

Nachdem in das COSIMA-Projekt eingeführt und ein Szenario beschrieben wurde, das zur Validierung der bisherigen Architektur dienen soll, wird in dem vorliegenden Kapitel die prototypische Realisierung der Architektur und des beschriebenen Szenarios vorgestellt.

In [Reussner u. Hasselbring 2006] wird beschrieben, dass bei der Entwicklung von Rahmenwerken in der Regel zuvor eine Reihe von ähnlichen Anwendungen entstehen, aus denen dann die Gemeinsamkeiten in ein Rahmenwerk extrahiert werden. Da es sich bei COSIMA zum Teil auch um ein Rahmenwerk handelt (→ Abschnitt 2.5.1) hätte bei der Entwicklung entsprechend verfahren werden können. Auf Grund der beschriebenen Neuartigkeit des Projekts wurde jedoch auf das in Kapitel 3 beschriebene szenariobasierte Vorgehen zurückgegriffen.

Da bis zu diesem Zeitpunkt die Architektur nur konzeptionell existierte, boten sich bei der Realisierung und Validierung zwei Vorgehensweisen:

a) Die Implementierung einer prototypischen Anwendung für das in Abschnitt 3.4 beschriebene Szenario, bei gleichzeitiger Umsetzung der Architektur *oder*

b) Die prototypische Realisierung der Architektur anhand eines von dem Szenario unabhängigen Anwendungsfall und anschließende Implementierung der eigentlichen Anwendung auf Basis dieser so entstandenen Architektur.

Da die zu entwickelnde Anwendung und die zugrunde liegende Architektur beziehungsweise das verwendete Rahmenwerk relativ unabhängig voneinander sind, ist Alternative *a)* nur bedingt empfehlenswert. Eine Dekorrelation beider Aspekte ist, wie bereits zuvor festgestellt, notwendig und zudem gängige Praxis. Aus diesem Grund wurde sich zu Gunsten der Alternative *b)* entschieden. Demzufolge ist eine Implementierung[1] in zwei, voneinander unabhängigen Stufen vorgenommen worden:

[1] Die Implementierung sowohl der Architektur als auch des Szenario sind in der Programmiersprache Java (→ `http://java.sun.com/j2se/1.5.0/`) in Version 5.0 durchgeführt worden.

1. Prototypische Realisierung der wesentlichen Architekturmerkmale anhand eines einfachen Anwendungsfalls *und*
2. Implementierung des in 3.4 beschriebenen Szenario auf Basis dieser Architektur.

Im ersten Abschnitt dieses Kapitel wird daher zunächst detailliert auf die Umsetzung der Architektur eingegangen. Im Anschluss daran findet sich eine Erläuterung zur Implementierung des Szenario. Die Validierung und deren Ergebnisse werden dann im Anschluss in Kapitel 5 dargelegt.

4.1 Realisierung der Architektur (Santiago)

Bei dem gewählten *bottom-up* Ansatz zur Entwicklung des Architekturprototypen musste zunächst ein geeigneter Anwendungsfall gefunden werden, der sich trotz einem Minimum an Komplexität dazu eignen musste, alle wesentlichen Architekturmerkmale extrahieren zu können. Zur besseren Kommunikation erhielt diese Anwendung den Codenamen *Santiago* und ist formlos in Abbildung 4.1 strukturell dargestellt.

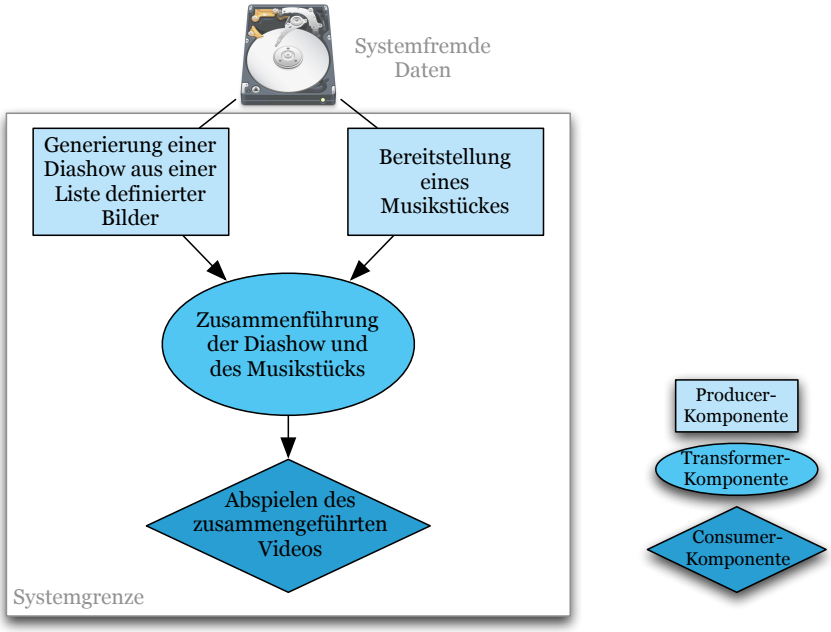

Abbildung 4.1: Anwendungsfall für die Realisierung der Architektur

Anhand lieser lassen sich leicht die drei wesentlichen Merkmale der Santiago Anwendung feststellen:

- Verarbeitung von unterschiedlichen Medien
- Umsetzung der einzelnen Verarbeitungsschritte in dedizierten Komponenten
- Anordnung dieser Komponenten nach dem Quelle-Komponente-Senke Prinzip

Trotz ihrer offensichtlichen Einfachheit konnte iterativ um diese Anwendung eine Architektur entwickelt werden, die alle notwendigen Charakteristika von COSIMA aufweist. Wie diese Entwicklung in den einzelnen Schritten im Detail aussah, beschreiben die folgenden Abschnitte.

4.1.1 Erste Schritte

In der ersten Iteration wurde lediglich ein einfaches Programm geschrieben, dass sequentiell jede der einzelnen Operationen auf den Medien ausführt und in Listing 4.1 dargestellt ist. Die einzelnen Operationen wurden dabei bereits in dedizierten Klassen gekapselt und über statische Methoden zugänglich gemacht. Der Kontrollfluss ist in dieser Implementierung implizit über Objektaufrufe realisiert und durch die Ausführungsreihenfolge konkret festgelegt. Der Datenfluss entsteht durch die Übergabe beziehungsweise Rückgabe von Objektinstanzen.

Listing 4.1: `SantiagoPlain`-Klasse zur einfachen Ausführung des Anwendungsfalls

```
11 package de.fhkoeln.santiago.codesamples;
12
13 public class SantiagoPlain {
14   public static void main(String[] args) {
15     String imagePath, musicPath;
16     String slideshow, slideshowWithMusic;
17
18     if (args.length == 2) {
19       imagePath = args[0];
20       musicPath = args[1];
21
22       slideshow = SlideshowGenerator.generateSlideshowForImagesInPath(imagePath);
23       slideshowWithMusic = MusicOMat.addMusicTrackToSlideshow(musicPath, slideshow);
24       VideoPlayer.playMovieFile(slideshowWithMusic);
25     } else {
26       System.err.println("Paths to the images and the music track are needed!");
27       System.exit(-1);
28     }
29   }
30 }
```

Die markierten Klassen implementieren dabei die jeweiligen Operationen in geeigneter Weise[2]. Den Zugriff auf diese Operationen über Klassenmethoden zu realisieren eignet sich jedoch nicht für den Einsatz in verteilten Umgebungen: Eine parallele Verwendung ein und der selben Komponente wäre so nur schwer umsetzbar. In der nächsten Iteration war es daher notwendig, die Ausführung der Operationen auf Objektebene zu implementieren. Gleichzeitig sollte der entsprechende Aufruf dabei polymorphisch erfolgen können. Die Umsetzung dieser beiden Anforderungen erfolgte über die Etablierung der abstrakten Oberklasse `AbstractComponent` (siehe Listing 4.2), da alle Komponenten zwei wesentliche Gemeinsamkeiten aufweisen, *a)* sie Übergabe von Eingabeparametern und *b)* die Ausführung der Operation unter Berücksichtigung dieser Parameter. Die Oberklasse definiert dabei drei Methoden: Die Methode `setInput(String [] inputs)`, um die notwendigen Parameter zu setzen; Die Methode `execute()` als nach außen sichtbare Schnittstelle, um die Operation zu starten und die Methode `_execute()` in der die eigentliche Operation intern implementiert ist.

Listing 4.2: Einfache `AbstractComponent`-Klasse

```
8  package de.fhkoeln.cosima.codesamples;
9
10 /**
11  * This class is just for didactic reasons and is only useful for the Santiago
12  * application codesamples. The real AbstractComponent class can be found here
13  * {@link de.fhkoeln.cosima.components.AbstractComponent}.
14  *
15  * @author Dirk Breuer
16  * @version 1.0 Jan 19, 2009
17  */
18 public abstract class AbstractComponent {
19
20   private String[] inputs;
21
22   public final String execute() {
23     return _execute();
24   }
25
26   protected abstract String _execute();
27
28   public void setInput(String[] inputs) {
29     this.inputs = inputs;
30   }
31
32   protected String[] getInput() {
33     return this.inputs;
34   }
35 }
```

[2]Ein Einblick in die Implementierung der einzelnen Klassen ist an dieser Stelle nicht weiter relevant. Der vollständige Quellcode findet sich auf der Begleit-CD zu dieser Arbeit.

Das ausführende Programm aus Listing 4.1 kann nun die generalisierten Komponenten mit ihren homogenen Schnittstellen verwenden um die einzelnen Operationen anzustoßen. Die entsprechenden Änderungen der Anwendung sind dabei in Listing 4.3 dargestellt. Als eine erste Abstraktion der Parameter dient in diesem Fall noch ein primitives String-Array, im weiteren Verlauf wird daraus ein vollständiges *Value-Object* [Fowler 2003, S. 486] entstehen. Die Implementierung der `execute()`-Methode ist nach dem *Template Method* Entwurfsmuster vorgenommen worden [Gamma u. a. 1995, 325][3].

Listing 4.3: Erweitertes Santiago Programm mit generalisierten Komponenten

```
23      // Generate the slideshow from the images in imagePath
24      AbstractComponent slideshowGenerator = new SlideshowGenerator();
25      slideshowGenerator.setInput(new String[] {imagePath});
26      slideshow = slideshowGenerator.execute();
27
28      //Add Music to the slideshow
29      AbstractComponent musicOMat = new MusicOMat();
30      musicOMat.setInput(new String[] {musicPath, slideshow});
31      slideshowWithMusic = musicOMat.execute();
32
33      // Play the slideshow
34      AbstractComponent videoPlayer = new VideoPlayer();
35      videoPlayer.setInput(new String[] {slideshowWithMusic});
36      videoPlayer.execute();
```

Für die nächste Iteration war es notwendig, wie bereits in Abbildung 3.3 dargestellt, dass eine Producer-Komponente das Musikstück erst innerhalb der Systemgrenzen *bekannt machen* muss. Der Grund dafür ist, dass innerhalb einer COSIMA-Anwendung eine Transformer-Komponente nur Mediendaten verarbeiten kann, die bereits im System vorhanden sind, also über den Medienbroker abgerufen werden können. Im Kontext von COSIMA hat das die Bedeutung, dass ein Medienobjekt erzeugt und über den Medienbroker bereitgestellt wird. Für die Santiago Anwendung bedeutet es momentan lediglich, dass das String-Objekt `musicPath` durch eine Instanz der `MusicProvider`-Klasse (siehe Listing 4.4) dem String-Objekt `music` zugewiesen wird.

Listing 4.4: Integration einer dedizierter Komponente zur Bereitstellung der Musik

```
23      // Make the music file known within the system borders
24      AbstractComponent musicProvider = new MusicProvider();
25      musicProvider.setInput(new String[] { musicPath });
26      String music = musicProvider.execute();
```

[3] In späteren Iterationen lassen sich so leicht Funktionalitäten hinzufügen, die vor oder nach der eigentlichen Operation durchgeführt werden müssen.

Dabei ist natürlich zu beachten, dass die `MusicOMat`-Instanz nicht länger die Referenz auf das `musicPath`-Objekt, sondern auf das `music`-Objekt übergeben wird.

Nachdem die notwendigen Komponenten zur Medienverarbeitung implementiert wurden, musste in der nächsten Iteration der Kontrollfluss über eine deklarative Beschreibung anzugeben sein. Des Weiteren musste diese Beschreibung durch eine dedizierte Komponente interpretiert und ausgeführt werden. Die Entwicklung dieser Ablaufbeschreibung und ihrer ausführenden Instanz wird im folgenden Abschnitt näher beschrieben.

4.1.2 Einführung einer deklarativen Ablaufbeschreibung

Nachdem die einzelnen Komponenten über eine einheitliche Schnittstelle ausführbar gemacht worden sind, war der nächste Schritt, eine Komposition der einzelnen Komponenten zu realisieren. Im Rahmen der prototypischen Implementierung wurde sich für eine Orchestrierung (→ Abschnitt 17) entschieden, wodurch eine zentral ausführende Einheit umgesetzt werden musste, die eine deklarative Ablaufbeschreibung interpretiert und ausführt. Um im Rahmen der prototypischen Umsetzung ein möglichst gutes Verständnis dieser Komponente entwickeln zu können, wurde bewusst darauf verzichtet, bestehende Lösungen wie etwa BPEL einzusetzen. Ein weiterer Grund dafür ist, dass nicht geklärt ist inwieweit sich BPEL überhaupt für den Einsatz in einem Multimediakontext eignet, wie auch schon bei [Richter 2008] ausgeführt. Dort wird der Einsatz von BPEL für Multimediaanwendungen erst nach einer entsprechenden Erweiterung empfohlen.

Daher wurde eine einfache properitäre Kompositionseinheit selbst implementiert. Auf Grund der so entstandenen Minimierung von Abhängigkeiten komplexer Bibliotheken Dritter, konnte auch die Gesamtkomplexität minimal gehalten und dadurch dem Ziel, die Architektur an sich zu validieren, mehr Rechnung getragen werden.

Der Begriff „Workflow“

Bevor sich der Implementierung der Servicekomposition zugewandt wird, soll noch eine Abgrenzung der Begriffe *Workflow* und *Prozess* erfolgen. Im Rahmen der Implementierung ist die Verwendung dieser beiden Begriffe nicht im selben

Kontext von Geschäftsprozessen zu verstehen, wie sie bei [Richter 2008] ebenfalls im gegebenen Zusammenhang mit dem COSIMA-Projekt behandelt werden. Im Kontext dieser Arbeit hat daher die folgende Definition von Workflow Gültigkeit:

Definition 30 (Workflow)**.** *„The computerised facilitation or automation of a process, in whole or part.“* [Hollingsworth 1995, S. 54]

Wesentlich dabei ist die computergestützte Automatisierung eines Prozesses, es fehlen demnach also manuelle Aktivitäten. Ein Prozess ist in diesem Kontext wie folgt definiert:

Definition 31 (Prozess)**.** *„A co-ordinated (parallel and/or serial) set of process activity(s) that are connected in order to achieve a common goal. Such activities may consist of manual activity(s) and/or workflow activity(s).“* [Hollingsworth 1995, S. 52]

Der Prozess beinhaltet demnach eine Reihe von definierten Aktivitäten, die der Erreichung eines bestimmten Ziels dienen. Diese Aktivitäten können dabei entweder manueller oder automatisierter Natur sein. Die Definition der Aktivitäten geschieht dabei in Form einer *Prozessdefinition*. Da innerhalb der Santiago-Anwendung als auch in dem Anwendungsszenario aus Abschnitt 3.4 keine manuellen Aktivitäten enthalten sind, kann im weiteren Verlauf daher ausschließlich von Workflow gesprochen werden. Demnach ist auch nur der Teil der Prozessdefinition relevant, der die automatisierten Aspekte des Prozesses beinhaltet. Diesen Teil bezeichnet man als *Workflowdefinition* [Hollingsworth 1995, S. 52].

Diese sehr vereinfachte begriffliche Verwendung ist für eine weitere Bearbeitung im Rahmen des COSIMA-Projekts in jedem Fall zu erweitern. Die Erweiterungen müssen sich dabei auch in der Implementierung der Architektur wieder finden lassen. Als Ausgangspunkt für die weiteren Betrachtungen sollte die Arbeit von [Richter 2008] herangezogen werden.

Die Implementierung der Servicekomposition innerhalb der Santiago-Anwendung besteht im wesentlichen aus vier Komponenten:

1. Eine Schnittstelle, die die Ablaufbeschreibung in der Anwendung darstellt;
2. Eine deklarative Ablaufbeschreibung in einem gegebenen Format;
3. Eine Klasse, die die einzelnen Elemente in dieser Beschreibung repräsentiert;
4. Eine ausführende Komponente.

Das `WorkflowDefinition`-Interface stellt die Schnittstelle für die Abbildung und den Zugriff auf die Ablaufbeschreibung dar und ist in Listing 4.5 zu sehen[4].

Listing 4.5: Das `WorkflowDefinition`-Interface

```
12 package de.fhkoeln.cosima.workflow;
13
14 import java.util.Iterator;
15 import java.util.Set;
16
17 /**
18  * The Interface for accessing a WorkflowDefinition instance from a client.
19  *
20  * @author Dirk Breuer
21  * @version 1.0 Jul 3, 2008
22  */
23 public interface WorkflowDefinition {
24
25   /**
26    * @return The amount of containing Workflow Elements.
27    */
28   public int size();
29
30   /**
31    * Every workflow consists of a certain number of elements, which must be
32    * processed in a certain order. In addition to this on every step in that
33    * processing chain there could be elements which do not depend on each other
34    * and so are not required to be run in a certain order, but must be run
35    * within that very step. This method will return an iterator of the type
36    * {@link WorkflowDefinitionIterator} to provide the described functionality
37    * to the using client.
38    *
39    * @return The Iterator for all the containing elements.
40    */
41   public Iterator<Set<WorkflowElement>> elementsIterator();
42 }
```

Diese einfache Schnittstelle bietet dem Client zum einen an, über die `size()`-Methode die Größe aller auszuführenden Elemente nachzufragen und über die Implementierung des *Iterator*-Pattern [Gamma u. a. 1995, S. 257] alle Elemente in ihrer jeweils definierten Sequenz.

[4] Eine Erläuterung des Begriffs „Workflow“ findet sich in dem gleichnamigen Kasten.

Die entsprechende Realisierung des `WorkflowDefinitionIterator` ist in Listing 4.6 zu sehen. Die `next()`-Methode liefert dabei immer ein `java.util.Set` von `WorkflowElement`-Objekten zurück, die als nächstes ausgeführt werden müssen, allerdings ohne die Notwendigkeit sie in einer definierten Reihenfolge auszuführen[5].

Listing 4.6: Implementierung der `WorkflowDefinitionIterator`-Klasse

```
12 package de.fhkoeln.cosima.workflow;
31 public class WorkflowDefinitionIterator implements
32     Iterator<Set<WorkflowElement>> {
33
34   private Map<String, WorkflowElement> elements;
35   private Set<WorkflowElement> currentElements;
36
37   /**
38    * @param A Map with the {@link WorkflowElement} objects to iterate through.
39    */
40   public WorkflowDefinitionIterator(Map<String, WorkflowElement> elements) {
41     this.elements = elements;
42   }
43
44   public boolean hasNext() {
45     if (currentElements == null)
46       return true;
47     if (currentElements.isEmpty())
48       return false;
49
50     for (Iterator<WorkflowElement> iterator = currentElements.iterator(); iterator
51         .hasNext();) {
52       WorkflowElement element = iterator.next();
53       if (element.hasSuccessors())
54         return true;
55     }
56     return false;
57   }
58
59   public Set<WorkflowElement> next() {
60     if (currentElements == null) {
61       currentElements = new HashSet<WorkflowElement>();
62       for (WorkflowElement element : elements.values()) {
63         if (element.getPredecessors() == null)
64           currentElements.add(element);
65       }
66     } else {
67       // This works only because this is a Set, and due to this all
68       // elements at this point with the same successor will store
69       // only once that successor.
70       Set<WorkflowElement> nextElements = new HashSet<WorkflowElement>();
71       for (Iterator<WorkflowElement> iterator = currentElements.iterator(); iterator
72           .hasNext();) {
73         WorkflowElement nextElement = iterator.next();
74         if (nextElement.getSuccessors() != null)
```

[5] Für die erfolgreiche Ausführung der Santiago-Anwendung ist es irrelevant, ob zuerst die Diashow erstellt wird und dann das Musikstück bereitgestellt wird oder vice versa.

```
75          for (Iterator<Successor> iter =
76              nextElement.getSuccessors().iterator(); iter.hasNext();) {
77            WorkflowElement workflowElement =
78                elements.get(iter.next().getUri());
79            nextElements.add(workflowElement);
80          }
81      }
82      currentElements = nextElements;
83      if (currentElements.isEmpty())
84        throw new NoSuchElementException("There are no more Elements");
85    }
86    return currentElements;
87  }
88
89  public void remove() {
90    throw new UnsupportedOperationException();
91  }
92 }
```

Die Ablaufbeschreibung selbst wurde für Santiago im YAML-Format angebeben[6]. Es wurde sich bewusst für ein datenzentiertes Format und nicht für ein dokumentenzentriertes Format wie XML entschieden, um die bereits erwähnte Abgrenzung zu BPEL und Geschäftsprozessen[7] auch in der Implementierung herauszustellen. Die Ablaufbeschreibung, die den Anwendungsfall in Santiago darstellt, ist in Listing 4.7 zu sehen.

Listing 4.7: Einfache deklarative Ablaufbeschreibung für Santiago im YAML-Format

```
1  ---
2  type: "producer"
3  description: "Producer:SlideshowGenerator"
4  uri: "de.fhkoeln.santiago.codesamples.SlideshowGenerator"
5  input:
6    - type: "external"
7      uri: "/Users/dbreuer/santiago/slideshow_images/"
8  output:
9    - uri: "de.fhkoeln.santiago.codesamples.SlideshowGenerator:output"
10 predecessors: null
11 successors:
12   - uri: "de.fhkoeln.santiago.codesamples.MusicOMat"
13
14 ---
15 type: "producer"
16 description: "Producer:MusicProvider"
17 uri: "de.fhkoeln.santiago.codesamples.MusicProvider"
18 input:
19   - type: "external"
20     uri: "file:///Users/dbreuer/santiago/music_track.mp3"
21 output:
```

[6] YAML ist ein datenzentierter Serialisierungsstandard für alle Programmiersprachen, der sehr menschenlesbar ist (siehe `http://www.yaml.org`).

[7] Bei der Beschreibung von Geschäftsprozessen lässt sich sehr viel eher von einem Dokument sprechen, dass entsprechend in einem System modelliert werden sollte.

```yaml
  - uri: "de.fhkoeln.santiago.codesamples.MusicProvider:output"
predecessors: null
successors:
  - uri: "de.fhkoeln.santiago.codesamples.MusicOMat"

---
type: "transformer"
description: "Transformer:MergeMedia"
uri: "de.fhkoeln.santiago.codesamples.MusicOMat"
input:
  - type: "internal"
    uri: "de.fhkoeln.santiago.codesamples.SlideshowGenerator:output"
  - type: "internal"
    uri: "de.fhkoeln.santiago.codesamples.MusicProvider:output"
output:
  - uri: "de.fhkoeln.santiago.codesamples.MusicOMat:output"
predecessors:
  - uri: "de.fhkoeln.santiago.codesamples.SlideshowGenerator"
  - uri: "de.fhkoeln.santiago.codesamples.MusicProvider"
successors:
  - uri: "de.fhkoeln.santiago.codesamples.VideoPlayer"

---
type: "consumer"
description: "Consumer:VideoPlayer"
uri: "de.fhkoeln.santiago.codesamples.VideoPlayer"
input:
  - type: "internal"
    uri: "de.fhkoeln.santiago.codesamples.MusicOMat:output"
output: null
predecessors:
  - uri: "de.fhkoeln.santiago.codesamples.MusicOMat"
successors: null
```

Wie im Listing zu sehen ist, besteht die Ablaufbeschreibung aus vier Teilbereichen, die jeweils durch die Zeichenkette `---` getrennt sind. Jeder Bereich steht dabei für ein Element, dass später im Workflow ausgeführt werden muss und beschreibt dementsprechend eine der vier medienverarbeitenden Komponenten innerhalb der Anwendung. Innerhalb jedem dieser Elemente sind die zur Ausführung des Workflows notwendigen Attribute einer Komponente definiert. Ein Attribut wird dabei durch ein `key: value`-Paar markiert, wobei der Doppelpunkt (`:`) den `key` von dem `value` trennt. Attribute können dabei verschachtelt werden, was durch eine Einrückung dargestellt wird[8]. Im Folgenden sind die wesentliche Attribute eines Elements zusammengefasst:

[8] Eine genaue Spezifikation zu der aktuellen YAML Version 1.2 findet sich unter `http://yaml.org/spec/1.2`.

Beschreibung (`description`) Eine informelle Beschreibung der jeweiligen Komponente.

URI (`uri`) Die URI anhand derer später die entsprechende Komponente lokalisiert und ausgeführt werden kann.

Eingänge (`input`) Eine Liste von Eingabedaten. Eingabedaten können entweder *extern* (in diesem Beispiel also die Musikdatei und die Bilder der Diashow) oder *intern* sein (die Diashow und die Diashow mit Musik). Die einzelnen Eingaben wird über eine URI referenziert.

Ausgänge (`output`) Eine Liste von Ausgabedaten. Eine hier formulierte Ausgabe ist zur Zeit immer systemintern. Auch die Ausgaben werden über eine URI referenziert[9].

Vorgänger (`predecessors`) Eine Liste von Komponenten, die *vor* dieser Komponente ausgeführt werden müssen. Die Komponenten werden dazu durch ihre URI referenziert. Elemente ohne Vorgänger werden als erstes ausgeführt.

Nachfolger (`successors`) Eine Liste von Komponenten, die *nach* dieser Komponente ausgeführt werden müssen. Die Komponenten werden dazu durch ihre URI referenziert. Elemente ohne Nachfolger markieren das Ende des Workflows.

Neben der eigentlichen Ablaufbeschreibung im YAML-Format ist darüber hinaus noch eine Klasse notwendig, die das `WorkflowDefinition`-Interface implementiert und die in der Lage ist, die angegebene Ablaufbeschreibung einzulesen. Die einzelnen Elemente der abstrakten Ablaufbeschreibung werden auf Programmebene dabei durch Instanzen der Klasse `WorkflowElement` repräsentiert. Die Klasse implementiert dabei die die einzelnen `key: value`-Paar entsprechend als Attribute, die über *Getter*-Methoden nach außen verfügbar gemacht werden. Die `YamlWorkflowDefinition`-Klasse setzt dafür auf jeder Instanz eines `WorkflowElement` die entsprechenden Werte über die gleichnamigen *Setter*-Methoden automatisch. Die `YamlWorkflowDefinition`-Implementierung findet sich in Listing 4.8 und das vollständige Listing A.1 zu der `WorkflowElement`-Klasse findet sich im Anhang.

Listing 4.8: Implementierung des `WorkflowDefinition`-Interface auf YAML-Basis

```
12 package de.fhkoeln.cosima.workflow;
35 public class YamlWorkflowDefinition implements WorkflowDefinition {
36
37   private Map<String, WorkflowElement> elements;
38
39   /**
```

[9] Zu beachten ist, dass es sich bei diesen Referenzen nicht zwangsläufig um Referenzen auf die eigentlichen Medien handeln muss: Sie werden zur Verknüpfung der Ein- und Ausgänge der einzelnen Komponenten während der Ausführung des Workflows verwendet.

```
40     * Build a new YamlWorkflowDefinition object which generates
41     * WorkflowElements from a definition file in the YAML format.
42     *
43     * @param pathToYamlDefinition
44     *          The path to the YAML definition file which holds all
45     *          information about the workflow elements.
46     * @throws IOException
47     *           If the given pathname doesn't exists or any other error
48     *           occurs while processing the file.
49     */
50    public YamlWorkflowDefinition(String pathToYamlDefinition)
51      throws IOException {
52      elements = new HashMap<String, WorkflowElement>();
53      FileInputStream stream = new FileInputStream(new File(pathToYamlDefinition));
54      YamlDecoder yamlDecoder = new YamlDecoder(stream);
55      try {
56        while (true) {
57          WorkflowElement element = yamlDecoder
58              .readObjectOfType(WorkflowElement.class);
59          elements.put(element.getUri(), element);
60        }
61      } catch (EOFException e) {
62        System.err.println("Finished reading stream.");
63      } finally {
64        yamlDecoder.close();
65        stream.close();
66      }
67    }
68
69    public int size() {
70      return elements.size();
71    }
72
73    public Iterator<Set<WorkflowElement>> elementsIterator() {
74      return new WorkflowDefinitionIterator(elements);
75    }
76  }
```

Die `WorkflowDefinition` ist jedoch lediglich die programmatische Repräsentation der deklarativen Beschreibung, um diese auszuführen bedarf es einer weiteren Komponente, der *Workflow Engine* [Hollingsworth 1995, S. 53]. Sie wird in COSIMA durch die abstrakte `WorkflowEngine`-Klasse realisiert (→ Listing 4.9). Die jeweiligen Unterklassen müssen dabei entsprechend die `execute()`-Methode überschreiben.

Listing 4.9: Abstrakte `WorkflowEngine`-Klasse

```
12  package de.fhkoeln.cosima.workflow;
24  public abstract class WorkflowEngine {
25
26    private Map<String, String> workflowStore;
27    private WorkflowDefinition definition;
28
29    /**
30     * This method must be implemented by its subclasses to execute the workflow
```

```
31     * defined in the {@link WorkflowDefinition} instance.
32     *
33     * @throws Exception If something went wrong during execution.
34     */
35    public abstract void execute() throws Exception;
36
37    public void setWorkflowDefinition(WorkflowDefinition definition) {
38      this.definition = definition;
39    }
40
41    public WorkflowDefinition getWorkflowDefinition() {
42      return this.definition;
43    }
44
45    public void setWorkflowStore(Map<String, String> workflowStore) {
46      this.workflowStore = workflowStore;
47    }
48
49    public Map<String, String> getWorkflowStore() {
50      return workflowStore;
51    }
52  }
```

Eine erste einfache Implementierung der abstrakten `WorkflowEngine`-Klasse ist in Listing 4.10 dargestellt.

Listing 4.10: Implementierung einer einfachen Workflow Engine

```
12  package de.fhkoeln.cosima.workflow;
30  public class SimpleWorkflowEngine extends WorkflowEngine {
31
32    public SimpleWorkflowEngine() {
33      setWorkflowStore(new HashMap<String, String>());
34    }
35
36    @Override
37    @SuppressWarnings("unchecked")
38    public void execute() throws Exception {
39      // iterate through the workflow definition elements
40      Iterator<Set< WorkflowElement >> elementsIterator = getWorkflowDefinition().elementsIterator();
41
42      while (elementsIterator.hasNext()) {
43        for ( WorkflowElement  element : elementsIterator.next()) {
44
45          List<String> inputList = new ArrayList<String>();
46
47          if (element.needsInput()) {
48            for (Input elementInput : element.getInput()) {
49              if (elementInput.isExternal())
50                inputList.add(elementInput.getData());
51              if (elementInput.isInternal())
52                inputList.add(getWorkflowStore().get(elementInput.getUri()));
53            }
54          }
55
```

```
56          Class<AbstractComponent> elementClass = (Class<AbstractComponent>) Class.forName(element.getUri());
57          AbstractComponent elementComponent = elementClass.newInstance();
58
59          elementComponent.setInput(inputList.toArray(new String[] {}));
60          String output = elementComponent.execute();
61
62          if (output != null)
63            getWorkflowStore().put(element.getOutputUri(), output);
64        }
65      }
66    }
67  }
```

Die Instanziierung der einzelnen Komponenten innerhalb der `SimpleWorkflowEngine` geschieht dabei unter zur Hilfenahme der Java Reflection API (Zeile 49 und 50 in Listing 4.10). Dies gelingt, da in der Ablaufbeschreibung als URI jeder Komponente der vollständig klassifizierende Java-Klassenname angegeben ist. Für den Einsatz in einer verteilten Umgebung ist diese Implementierung jedoch völlig ungeeignet. Später werden die einzelnen Komponenten daher als *Web Services* implementiert werden (→ Abschnitt 4.1.4). Ein weiterer Punkt von Interesse in dieser Implementierung ist die Notwendigkeit, den aktuellen Fortschritt der Ausführung nachzuhalten. Dies wird über die Einführung des lokalen `workflowStore`-Objekts erreicht. Für die Umsetzung komplexerer Abläufe scheint es jedoch sinnvoll zu sein, dass der ausführenden Komponente ein Zustandsautomat zu Grunde gelegt wird, wie es bei [Biornstad u. a. 2006] propagiert wird.

Das Santiago Programm aus den bisherigen Listings lässt sich durch die in diesem Abschnitt eingeführten Erweiterungen allein auf die Instanziierung einer `WorkflowDefinition` und die Ausführung der `SimpleWorkflowEngine` mit dieser Definition reduzieren (Listing 4.11).

Listing 4.11: Santiago Programm mit deklarativer Ablaufbeschreibung

```
19  public static void main(String[] args) throws Exception {
20    if (args.length == 1) {
21      String pathToWorkflowDefinition = args[0];
22      WorkflowDefinition definition = new YamlWorkflowDefinition(pathToWorkflowDefinition);
23
24      WorkflowEngine workflowInstance = new SimpleWorkflowEngine();
25      workflowInstance.setWorkflowDefinition(definition);
26      workflowInstance.execute();
27
28    } else {
29      System.err.println("Path to the workflow definition is needed!");
30      System.exit(-1);
31    }
32  }
```

Nachdem eine deklarative Ablaufbeschreibung eingeführt wurde und die grundsätzliche Möglichkeit geboten ist, diese Beschreibung auszuführen, ist der nächste Schritt die Umsetzung der Medienobjekte-Modellierung. Diese wird im nächsten Abschnitt behandelt.

4.1.3 Explizite Umsetzung des Datenfluss und Einführung des Medienobjekts

In Abschnitt 2.4.8.3 wurde bereits darauf hingewiesen, dass zwischen Kontrollfluss und Datenfluss innerhalb einer SOA unterschieden werden muss. Im vorherigen Abschnitt wurden entsprechend die Komponenten realisiert, die dem Kontrollfluss zuzuordnen sind. In diesem Abschnitt werden die Komponenten implementiert, die zur Umsetzung des Datenfluss notwendig sind.

Im ersten Schritt soll zunächst ein *Value-Object* eingeführt werden, wie in Abschnitt 4.1.1 bereits angekündigt. Dieses soll dazu dienen die notwendigen Ein- und Ausgabewerte einzelner `AbstractComponent`-Instanzen zu kapseln und leichter übertragbar zu machen. Die Implementierung dieses Objekt in Form der IODescriptor-Klasse ist in Listing 4.12 dargestellt[10].

Listing 4.12: `IODescriptor` als Implementierung des *Value-Object* Pattern

```
12 package de.fhkoeln.cosima.services;
26
27   private List<String> descriptorElements;
28
29   public IODescriptor() {
30     this.descriptorElements = new ArrayList<String>();
31   }
32
33   /**
34    * Adds an element to this descriptor which describes an output
35    * location.
36    */
37   public void add(String element) {
38     this.descriptorElements.add(element);
39   }
40
41   /**
42    * @return The size of the descriptorElements in this descriptor.
43    */
44   public int size() {
45     return this.getDescriptorElements().length;
46   }
```

[10] Die Implementierung der `equals()` und `hash()`-Methoden sind zur Erfüllung des Value-Object-Pattern notwending [Fowler 2003, S. 486].

```
47
48   /**
49    * @return The first element in this descriptor.
50    */
51   public String first() {
52     try {
53       return getDescriptorElements()[0];
54     } catch (ArrayIndexOutOfBoundsException e) {
55       // if there are no elements just return null
56       return null;
57     }
58   }
59
60   public void setDescriptorElements(String[] strings) {
61     for (String string : strings) {
62       this.descriptorElements.add(string);
63     }
64   }
65
66   public String[] getDescriptorElements() {
67     return descriptorElements.toArray(new String[] {});
68   }
69
70   /**
71    * @return If the Descriptor is empty.
72    */
73   public boolean isEmpty() {
74     return descriptorElements.isEmpty();
75   }
76
77   @Override
78   public int hashCode() {
79     final int prime = 31;
85
86   @Override
87   public boolean equals(Object obj) {
88     if (this == obj)
102 }
```

Diese Änderungen haben dabei keinerlei Auswirkungen auf das Santiago Programm selbst. Dieses stellt sich immer noch genauso dar wie in Listing 4.11. Änderungen müssen nur, wie in Listing 4.13 dargestellt, an der `SimpleWorkflowEngine`-Klasse vorgenommen werden. Die Anpassungen an der `AbstractComponent`-Klasse, dass die Methoden `execute()` und `setInput()` je eine `IODescriptor`-Instanz zurückgeben beziehungsweise entgegennehmen, werden an dieser Stelle ausgespart.

Listing 4.13: Erweiterung der einfachen Workflow-Engine um den `IODescriptor`

```
12 package de.fhkoeln.cosima.workflow;
30 public class SimpleWorkflowEngineWithIO extends WorkflowEngine {
31
32   public SimpleWorkflowEngineWithIO() {
33     setWorkflowStore(new HashMap<String, String>());
```

```
34    }
35
36    @SuppressWarnings("unchecked")
37    public void execute() throws Exception {
38      // iterate through the workflow definition elements
39      Iterator<Set<WorkflowElement>> elementsIterator = getWorkflowDefinition().elementsIterator();
40
41      while (elementsIterator.hasNext()) {
42        for (WorkflowElement element : elementsIterator.next()) {
43
44          IODescriptor input = new IODescriptor();
45
46          if (element.needsInput()) {
47            for (Input elementInput : element.getInput()) {
48              if (elementInput.isExternal())
49                input.add(elementInput.getData());
50              if (elementInput.isInternal())
51                input.add(getWorkflowStore().get(elementInput.getUri()));
52            }
53          }
54
55          Class<AbstractComponent> elementClass = (Class<AbstractComponent>) Class.forName(element.getUri());
56          AbstractComponent elementComponent = elementClass.newInstance();
57
58          elementComponent.setInput(input);
59          IODescriptor output = elementComponent.execute();
60
61          if (output != null)
62            getWorkflowStore().put(element.getOutputUri(), output.first());
63        }
64      }
65    }
66  }
```

Wesentlich entscheidender für die Realisierung des Datenfluss ist jedoch die Implementierung des Medienobjekts, wie es in 2.4.8.3 konzeptioniert wurde. Diese Erweiterung hat dabei nur Auswirkungen auf die `AbstractComponent`-Klasse und ihre Unterklassen, nicht jedoch auf die jeweiligen `WorkflowEngine`-Implementierungen.

Das Medienobjekt selbst ist wie bereits ausgeführt und in Abbildung 2.8 zu erkennen, nach dem *Composite*-Pattern implementiert. Bei der Umsetzung des Composite-Pattern sind in der Regel[11] drei Klassen zu implementieren:

Component Die Klasse von der die anderen beiden jeweils vererben (hier die `MediaComponent`-Klasse). In dieser Klasse werden auch die Methoden zum traversieren der Composite-Objekte definiert.

[11] Alternativ ließe sich die polymorphische Verwendung von `Composite` und `Leaf`-Objekten über die Einführung eines `Component`-Interface erreichen.

Composite Die Klasse, in der mehrere Component-Objekte zusammengefasst werden (hier die `MediaContainer`-Klasse).

Leaf Die Klasse, die einen Endpunkt in der Hierarchie repräsentiert (hier die `Media`-Klasse).

Zur Realisierung der Santiago-Anwendung ist die Verwendung von `MediaContainer`-Instanzen nicht notwendig, daher wird darauf auch nicht weiter eingegangen. Die Implementierungen der anderen beiden Klassen sind in Listing 4.14 und 4.15 zu sehen.

Listing 4.14: `MediaComponent`-Klasse als *Component* im Composite-Pattern

```
12 package de.fhkoeln.cosima.media;
31 public abstract class MediaComponent implements Serializable {
32
33   private static final long serialVersionUID = -5578809582939403020L;
34
35   private List<Metadata> metadatas;
36   private MediaStore store;
37
38   private String name;
39   private String namespace;
40
41   public MediaComponent() {
42     metadatas = new ArrayList<Metadata>();
43   }
44
45   /**
46    * Constructor to set the metadata object on creation.
47    *
48    * @param metadata The Metadata object to set.
49    */
50   public MediaComponent(List<Metadata> metadatas) {
51     this.metadatas = metadatas;
52   }
53
54   public List<Metadata> getMetadata() {
55     return this.metadatas;
56   }
57
58   public void addMetadata(Metadata metadata) {
59     this.metadatas.add(metadata);
60   }
61
62   public void setName(String name) {
63     this.name = name;
64   }
65
66   public String getName() {
67     return name;
68   }
69
70   public String getUri() {
71     String uri = "/";
```

```
72
73    try {
74      if (getNamespace() != null) {
75        for (int i = 0; i < getNamespace().length; i++) {
76          uri += URLEncoder.encode(getNamespace()[i], "UTF-8");
77          uri += "/";
78        }
79      }
80      uri += URLEncoder.encode(getName(), "UTF-8");
81    } catch (UnsupportedEncodingException e) {
82      e.printStackTrace();
83    }
84
85    return uri;
86  }
87
88  public void setStore(MediaStore store) {
89    this.store = store;
90  }
91
92  public MediaStore getStore() {
93    return store;
94  }
95
96  public void setReferenceToRealData(Object realMediaData) throws UnsupportedOperationException {
97    throw new UnsupportedOperationException();
98  }
99
100 public URI getReferenceToRealData() throws UnsupportedOperationException {
101   throw new UnsupportedOperationException();
102 }
103
104 /**
105  * @return The namespace as String Array
106  */
107 public String[] getNamespace() {
108   return namespace == null ? null : namespace.split("::");
109 }
110
111 /**
112  * Let's a client set a namespace in the form "Foo::Bar".
113  *
114  * @param Namespace of this Media Object.
115  */
116 public void setNamespace(String namespace) {
117   this.namespace = namespace;
118 }
119
120 public int size() throws UnsupportedOperationException {
121   throw new UnsupportedOperationException();
122 }
123
124 public MediaComponent getChild(int atIndex) throws UnsupportedOperationException {
125   throw new UnsupportedOperationException();
126 }
127
128 public void removeMedia(MediaComponent media) throws UnsupportedOperationException {
```

```
129     throw new UnsupportedOperationException();
130   }
131
132   public void addMedia(MediaComponent media) throws UnsupportedOperationException {
133     throw new UnsupportedOperationException();
134   }
135
136   public final String storageKey() {
137     return new Integer(Math.abs(hashCode())).toString();
138   }
139
140   /**
141    * Returns the data which could be processed or played. This is although the
142    * place to put synchronisation logic based on the associated AbstractSync instance.
143    *
144    * @return Something that could be processed or played by an instance of AbstractComponent.
145    */
146   public abstract Object getPlayableData();
147
148   @Override
149   public int hashCode() {
157   }
158 }
```

Listing 4.15: `Media`-Klasse als *Leaf* im Composite-Pattern

```
12  package de.fhkoeln.cosima.media;
23  public class Media extends MediaComponent {
24
25    private static final long serialVersionUID = -7185178004655851316L;
26
27    private Object realMediaData;
28
29    @Override
30    public URI getReferenceToRealData() throws UnsupportedOperationException {
31      return URI.create((String) realMediaData);
32    }
33
34    @Override
35    public void setReferenceToRealData(Object realMediaData) throws UnsupportedOperationException {
36      this.realMediaData = realMediaData;
37    }
38
39    @Override
40    public Object getPlayableData() {
41      // Lazy loading of data. Real data is only requested if it is really needed.
42      return getStore().read(storageKey());
43    }
44
45    @Override
46    public boolean equals(Object obj) {
60    }
61  }
```

Jedes Medienobjekt, unabhängig davon ob es *Leaf* oder *Composite* ist, verfügt über einen Namen und einen Namensraum. Aus diesen beiden Angaben lässt sich die URI über die `getUri()`-Methode ermitteln. Später lässt sich mit dieser URI ein Medienobjekt eindeutig im Kontext eines Medienbroker lokalisieren.

Zur Speicherung der Medienobjekte über den Medienbroker implementiert die `MediaComponent`-Klasse das `java.lang.Serializable`-Interface. Zusätzlich muss jede Unterklasse das Feld `serialVersionUID` definieren.

Zur Persistierung der eigentlichen Mediendaten sowie um Medienobjekte vergleichbar zu machen, wurden zusätzlich die Methoden `equals()` und `hash()` überschrieben.

Des Weiteren hat jedes Medienobjekt eine Reihe von Metadaten assoziiert. Jede Metadata muss dabei das `Metadata`-Interface implementieren. Es definiert dabei lediglich die Funktionalität, dass einem Schlüssel in einem definierten Namensraum ein beliebiger Wert zugewiesen wird (siehe Listing 4.16). Zusätzlich verfügt jedes `Metadata`-Objekt über eine URI zur eindeutigen Identifikation. Zu diesem Zeitpunkt existiert jedoch noch keine Implementierung dieser Schnittstelle, denkbar wäre aber zum Beispiel eine Implementierung auf Basis des Dublin Core Metadaten Standards[12].

Listing 4.16: `Metadata`-Interface als abstrakte Repräsentation von Metadaten

```
12 public interface Metadata {
13
14   public URI getUri();
15
16   public String getNamespace();
17
18   public void setNamespace(String namespace);
19
20   public String getKey();
21
22   public void setKey(String key);
23
24   public String getValue();
25
26   public void setValue(String value);
27
28 }
```

Medienobjekte sind weder „abspielbar“ noch lassen sich auf ihnen medienverarbeitende Operationen durchführen, sie sind wie schon in Abschnitt 2.4.8.3 erwähnt, lediglich reichhaltige Referenzen auf die eigentlichen Daten. Der Zugriff auf eben diese Daten erfolgt über

[12] Die *Dublin Core Metadata Initiative* ist eine offene Organisation mit dem Ziel einen flexiblen und austauschbaren online Standard für Metadaten zu schaffen, der die unterschiedlichsten Anwendungen erlaubt (`http://dublincore.org/`).

die `getPlayableData()`-Methode. Die Methode ist nach dem *Lazy-Load*-Pattern [Fowler 2003, S. 200] implementiert, so dass die potentiell sehr großen Mediendaten erst geladen werden, wenn sie tatsächlich benötigt werden. Wie das Laden uns Speichern von Medienobjekten an sich implementiert ist, stellt der nachfolgende Abschnitt im Detail vor.

4.1.3.1 Speichern und Laden von Medienobjekten

Bei dem Umgang mit Medienobjekten müssen zwei unterschiedliche Aspekte betrachtet werden:

- Die Vermittlung der Medienobjekte
- Die Persistierung der Mediendaten

Die Trennung ist bereits auf Modellebene vorgesehen worden, wie in Abbildung 2.8 dargestellt ist. Diese Trennung soll daher entsprechend auch im Code umgesetzt werden.

Die Vermittlung der Medienobjekte übernimmt ein Medienbroker. Der wesentliche Teil der zugehörigen Schnittstelle ist in Listing 4.17 zu sehen. Im Rahmen dieser Arbeit wird die `MemcachedMediaBroker`-Implementierung des `MediaBroker`-Interface verwendet. Die Speicherung der einzelnen Medienobjekte wird dabei über eine Objektserialisierung und deren Hinterlegung in einem *`memcached`-Server*[13] realisiert. Der `memcached`-Server erlaubt die leichte Vermittlung und Speicherung von Medienobjekten in einem verteilten System und war daher optimal für den Einsatz in einem Prototypen geeignet.

Listing 4.17: Das `MediaBroker`-Interface zur Vermittlung von Medienobjekten

```
12 package de.fhkoeln.cosima.media.mediabroker;
27 public interface MediaBroker {
28
29   /**
30    * @param media The media object to store within the broker.
31    */
32   public URI store(MediaComponent media);
33
34   /**
35    * @param mediaUri The URI of the media object to retrieve from the broker.
36    * @return The found media object (or null if none was found) with the
37    *         MediaStore instance of this broker attached.
38    */
39   public MediaComponent retrieve(String mediaUri);
60
61 }
```

[13] `memcached` ist ein ACID-konformer, hochperformanter verteilter Objektspeicher (siehe `http://www.danga.com/memcached/`)

Die Persistierung der eigentlichen Daten übernimmt die `MediaStore`-Komponente, deren Schnittstelle in Listing 4.18 zu sehen ist. Die Implementierung innerhalb dieser Arbeit speichert die Mediendaten auf dem Dateisystem[14]. Für eine weitere Entwicklung würde sich hier die Integration eines verteilten Dateisystems oder des bereits erwähnten MultiMonster-Medienservers anbieten.

Listing 4.18: Das `MediaStore`-Interface zur Persistierung von Medien

```
12 package de.fhkoeln.cosima.media.mediabroker.storage;
26 public interface MediaStore {
27
28   /**
29    * @param data The media object which should be persisted.
30    * @return The key to find the persisted data again.
31    * @throws IOException If there was an error while writing the media to the persistance layer.
32    */
33   public String write(MediaComponent data) throws IOException;
34
35   /**
36    * @param key The key to find the persisted data.
37    * @return The Reference to the persisted data.
38    */
39   public Object read(String key);
40
41   /**
42    * The location where to store all media.
43    *
44    * @param Store specific location value.
45    */
46   public void setStoreLocation(String location);
47 }
```

Die wesentlichen Methoden des `MediaBroker`-Interface sind die Methoden `store()` und `retrieve()`. Die `store()`-Methode nimmt dabei ein Medienobjekt entgegen, legt es in der Speicherlösung der jeweiligen `MediaBroker`-Implementierung ab und reicht das Medienobjekt an die assoziierte `MediaStore`-Instanz weiter. Am Ende wird die Referenz im übergebenen Medienobjekt auf `null` gesetzt, damit im Anschluss diese Referenz nicht länger genutzt werden kann. Die Realisierung dieser Methode in der `MemcachedMediaBroker`-Implementierung ist in Listing 4.19 zu sehen.

Listing 4.19: Implementierung der `store()`-Methode im `MemcachedMediaBroker`

```
72   public URI store(MediaComponent media) {
73     URI realUri = URI.create(BROKER_URI + media.getUri());
74
75     try {
76       client.set(realUri.toString(), 0, media);
77       mediaStore.write(media);
```

[14] Für den Einsatz in einer verteilten Umgebung lässt sich einfach ein NFS-Mount verwenden.

```
78        // Ensure that the reference is no longer available, because it is not intended to be
79        // used by some one else then the MediaStore instances
80        media.setReferenceToRealData(null);
81      } catch (IOException e) {
82        e.printStackTrace();
83      }
84      return realUri;
85    }
```

Zur besseren Veranschaulichung, wie die Speicherung von Medienobjekt abläuft, dient das in Abbildung 4.2 dargestellte Sequenzdiagramm.

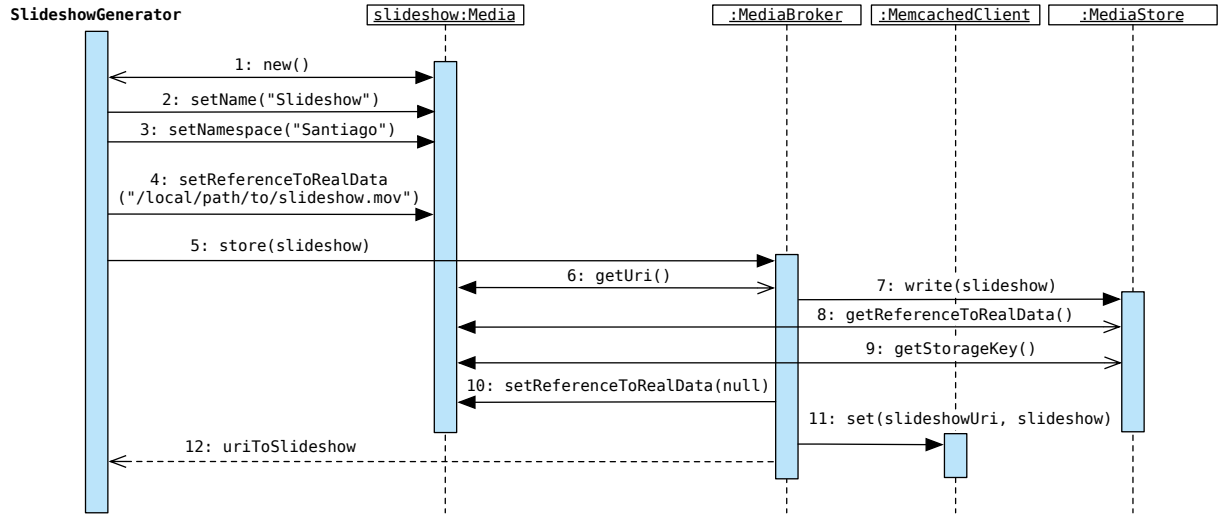

Abbildung 4.2: Sequenzdiagramm über das Speichern von Medienobjekten

Das Laden eines Medienobjekts wird durch die `retrieve()`-Methode umgesetzt. Es wird dazu die URI eines Medienobjekts übergeben, anhand derer das entsprechende Objekt aus der implementierten Speicherlösung geholt wird. Um später über dieses Medienobjekt die eigentlichen Mediendaten über die die `getPlayableData()`-Methode abrufen zu können, wird dem Medienobjekt noch die Referenz auf die `MediaStore`-Instanz des Medienbrokers übergeben. Die Implementierung dieser Methode in der `MemcachedMediaBroker`-Implementierung ist in Listing 4.20 zu sehen.

Listing 4.20: Implementierung der `retrieve()`-Methode im `MemcachedMediaBroker`

```
61  public MediaComponent retrieve(String mediaUri) {
62    MediaComponent storedMedia = (MediaComponent) client.get(mediaUri);
63    storedMedia.setStore(mediaStore);
64    return storedMedia;
65  }
```

Das Sequenzdiagramm aus Abbildung 4.3 verdeutlicht noch einmal die notwendigen Schritte, die beim lesen eines Medienobjekts notwendig sind.

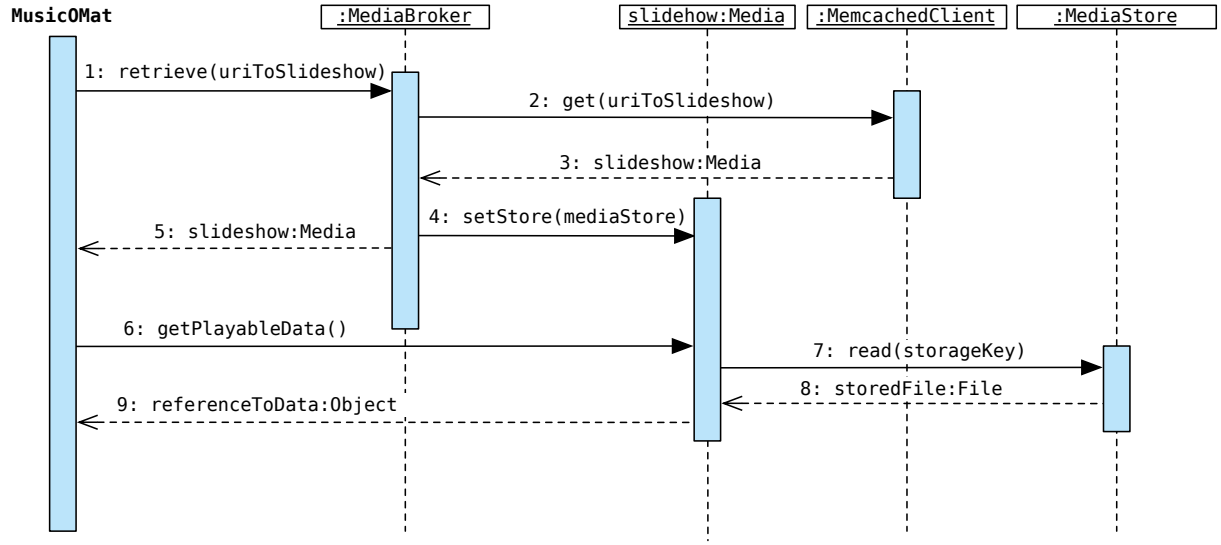

Abbildung 4.3: Sequenzdiagramm über das Lesen von Medienobjekten

Stellvertretend für die anderen medienverarbeitenden Komponenten der Santiago-Anwendung ist in Listing 4.21 die Implementierung der `MusicOMat`-Klasse zur Integration des Medienobjekts dargestellt.

Listing 4.21: Die `MusicOMat`-Klasse unter Verwendung des Medienobjektes

```
26  public IODescriptor _execute() {
27    IODescriptor output = new IODescriptor();
28
29    MediaComponent outputMedia = new Media();
30    outputMedia.setName("SlideshowWithMusic");
31
32    MediaComponent movieFile = getBroker().retrieve(getInput().getDescriptorElements()[0]);
33    MediaComponent audioFile = getBroker().retrieve(getInput().getDescriptorElements()[1]);
34
35    MediaAction action = new FFMpegMerger(movieFile, audioFile, outputMedia);
36    action.performAction();
37
38    URI mediaUri = getBroker().store(outputMedia);
39    output.add(mediaUri.toString());
40
41    return output;
42  }
```

Nachdem sowohl der explizite Kontrollfluss als auch der Datenfluss etabliert worden sind, bleibt als letzter Schritt nur noch die Umsetzung der einzelnen Komponenten als Dienste

und ihre Verwendung in einer verteilten Umgebung. Dies wird im nächsten Abschnitt vorgestellt.

4.1.4 Extraktion der Komponenten als Dienste

Der letzte Schritt in der Umsetzung der Architektur war es, die medienverarbeitenden Komponenten so zu realisieren, dass sie auf unterschiedlichen Plattformen betrieben werden können. Im Rahmen dieser Arbeit wurde dieses Ziel durch die Realisierung der Komponenten als Web Services unter Verwendung von SOAP als Übertragungsprotokoll erreicht. Für die konkrete Implementierung der Web Service Kommunikation und SOAP wurde auf die *Axis2 Web Service Engine*[15] der Apache Group zurück gegriffen. Da die Einbindung von Axis2 sehr transparent geschehen ist, kann leicht eine andere Lösung verwendet werden, wie etwa JAX-WS[16] oder Apache CXF[17]. Eine andere Alternative, die zur Auswahl stand, war die Umsetzung einer REST-Architektur. Da in einer REST-Architektur aber bestimmte Konventionen gelten[18] was nach [Papazoglou 2003] jedoch nicht eine Eigenschaft eines Dienstes sein sollte, weil sie eine lose Kopplung erschwert, wurde gegen diese Alternative entschieden.

4.1.4.1 Definition einer Dienstschnittstelle

Der erste Schritt bei der Umsetzung der Komponenten als Dienste war die Etablierung einer Schnittstelle, die die wesentlichen Methoden eines Dienstes definiert. Bei genauerer Betrachtung lässt sich schnell feststellen, dass nur die Methoden `execute()` und `setInput()` der `AbstractComponent`-Klasse auch für die Veröffentlichung als Dienst von Relevanz sind. Das so entstandene `CoreService`-Interface ist in Listing 4.22 dargestellt.

Listing 4.22: Das `CoreService`-Interface von COSIMA

```
12 package de.fhkoeln.cosima.services;
27 public interface CoreService {
28
29   /**
30    * SERVICE_SET_INPUT_OPERATION The string literal of the operation
31    * which sets the required input of a service component.
32    */
```

[15] `http://ws.apache.org/axis2/`
[16] `https://jax-ws.dev.java.net/`
[17] `http://cxf.apache.org/`
[18] Wie etwa die Übertragung der HTTP-Verben (`post`, `get`, `put`, `delete`) auf die entsprechenden CRUD-Operationen.

```
33    public static final String SERVICE_SET_INPUT_OPERATION = "setInput";
34
35    /**
36     * SERVICE_EXECUTE_OPERATION The string literal of the operation
37     * which executes the service itself.
38     */
39    public static final String SERVICE_EXECUTE_OPERATION = "execute";
40
41    public void setInput(IODescriptor descriptor);
42
43    public IODescriptor execute();
44
45    public String getUri();
46
47    public String getDescription();
48
49 }
```

Die beiden Konstanten `SERVICE_SET_INPUT_OPERATION` und `SERVICE_EXECUTE_OPERATION` dienen dazu, die Informationen über die tatsächlichen Methoden an einer zentralen Stelle im `CoreService`-Interface vorzuhalten. Zusätzlich wurden in dem Interface die beiden Methoden `getUri()` und `getDescription()` definiert, die zur Lokalisierung eines Dienstes über die Service Registry von Bedeutung sind. Diese wird im nächste Abschnitt genauer vorgestellt.

4.1.4.2 Umsetzung der Service Registry

Nachdem die einzelnen Komponenten über eine Schnittstelle verfügten, die sie als Dienste identifiziert, war der nächste Schritt, die Umsetzung der in Abschnitt 2.4.2.1 definierten Eigenschaft nach *location transparency* [Papazoglou 2003]. Der Ort eines Dienstes soll demnach an einer zentralen Stelle hinterlegt werden. Im Prototypen wird diese Funktionalität durch die eine Implementierung des `ServiceRegistry`-Interface aus Listing 4.23 bereitgestellt. Alternativ dazu hätte auch auf die bereits erwähnten Technologien UDDI oder ebXML zurückgegriffen werden können. Wie bereits in Bezug auf den Verzicht von BPEL ist auch hier anzuführen, dass eine Verwendung von UDDI oder ebXML zu aufwändig gewesen wäre. Daher wurde für den Prototypen eine `ServiceRegistry`-Implementierung ebenfalls auf `memcached`-Basis gewählt, ähnlich der des Medienbrokers. Dadurch ist die Service Registry selbst nicht als echter Dienst wie bei [Papazoglou 2003] gefordert implementiert, eignet sich aber dennoch für den Einsatz in einer verteilten Umgebung. Die Verwendung von JNDI war dem gegenüber keine Option, da sie an die Java Plattform

gebunden ist, was mit der Forderung nach Plattformunabhängigkeit [Papazoglou 2003] im Konflikt steht.

Listing 4.23: Das `ServiceRegistry`-Interface von COSIMA

```
12 package de.fhkoeln.cosima.services.registry;
13
14 import de.fhkoeln.cosima.services.CoreService;
15
16 /**
17  * The service registry interface for accessing and publishing {@link CoreService} instances.
18  *
19  * @author Dirk Breuer
20  * @version 1.0  Nov 30, 2008
21  *
22  */
23 public interface ServiceRegistry {
24
25   /**
26    * Publishes the given service in the Service Registry.
27    *
28    * @param service The Service which should be published.
29    */
30   public void publish(CoreService service);
31
32   /**
33    * Queries for a service by the given description.
34    *
35    * @param description A Description of the Service.
36    * @return The URI of the Service.
37    */
38   public String query(String description);
39
40 }
```

Durch die Einführung einer Service Registry ist es auch nicht länger notwendig, die tatsächliche URI zu einer Komponente in der deklarativen Ablaufbeschreibung zu definieren. Es muss aber eine Beschreibung angegebenen werden, anhand derer sich ein passender Dienst in der Registry finden lässt.

4.1.4.3 Web Service-fähige Workflow Engine

Bevor die Inbetriebsetzung der Komponenten als Dienste begonnen werden kann, muss zuvor eine `WorkflowEngine`-Implementierung umgesetzt werden, die in der Lage ist, diese Komponenten via SOAP aufzurufen und die die Verwendung der Reflection API obsolet macht. Zu diesem Zwecke ist die `RemoteWorkflowEngine` implementiert worden, deren vollständiges Listing A.2 im Anhang zu finden ist. Die hier wesentliche Funktionalität für das Aufrufen eines Web Services findet sich in Listing 4.24 wieder.

Listing 4.24: Aufruf eines Web Service aus der `RemoteWorkflowEngine` heraus

```
161  @SuppressWarnings("unchecked")
162  private IODescriptor invokeServiceForElementWithInputDescriptor(WorkflowElement element,
163                                                                  IODescriptor inputDescriptor) {
164
165    IODescriptor output = new OutputDescriptor();
166
167    try {
168      RPCServiceClient client = new RPCServiceClient();
169
170      Options options = client.getOptions();
171
172      String serviceUri = registry.query(element.getDescription());
173      Logger.info("Setting EPR to: " + serviceUri);
174      EndpointReference targetEPR = new EndpointReference(serviceUri);
175      options.setTo(targetEPR);
176
177      // We want to set the timeout a bit higher to handle long running remote services.
178      options.setTimeOutInMilliSeconds(500000);
179
180      // Setting input arguments first
181      QName setInputOperation = new QName(element.getNamespace(), CoreService.SERVICE_SET_INPUT_OPERATION);
182
183      Object[] setInputOperationArgs = new Object[] { inputDescriptor };
184
185      client.invokeRobust(setInputOperation, setInputOperationArgs);
186
187      // Run the service action and get back the output value
188      QName executeOperation = new QName(element.getNamespace(), CoreService.SERVICE_EXECUTE_OPERATION);
189
190      Object[] executeOperationArgs = new Object[] {};
191      Class[] returnTypes = new Class[] { IODescriptor.class };
192
193      Object[] response = client.invokeBlocking(executeOperation, executeOperationArgs, returnTypes);
194
195      output = (IODescriptor) response[0];
196    } catch (AxisFault e) {
197      e.printStackTrace();
198    }
199
200    return output;
201  }
```

Im Zuge dessen wurde auch das `ProcessStore`-Interface aus Listing 4.25 eingeführt, dass die Speicherung des Workflow-Fortschritts kapselt. Über dieses Interface lassen sich in späteren Versionen leichter verteilte Speicherstrategien verwenden, die somit auch eine verteilte Implementierung der `WorkflowEngine` ermöglicht.

Listing 4.25: Das `ProcessStore`-Interface

```
12 package de.fhkoeln.cosima.workflow.storage;
22 public interface ProcessStore {
23
24   /**
25    * @param data
26    *          The data which should be stored under a certain key
27    * @param key
28    *          The key under which the data is stored
29    */
30   public void putDataForKey(String data, String key);
31
32   /**
33    * @param key
34    *          The key under which the data was stored
35    * @return The data for the given key
36    */
37   public String getInputByKey(String key);
38
39 }
```

Zum Abschluss ist der Prozess der Ausführung in Abbildung 4.4 noch einmal schematisch dargestellt.

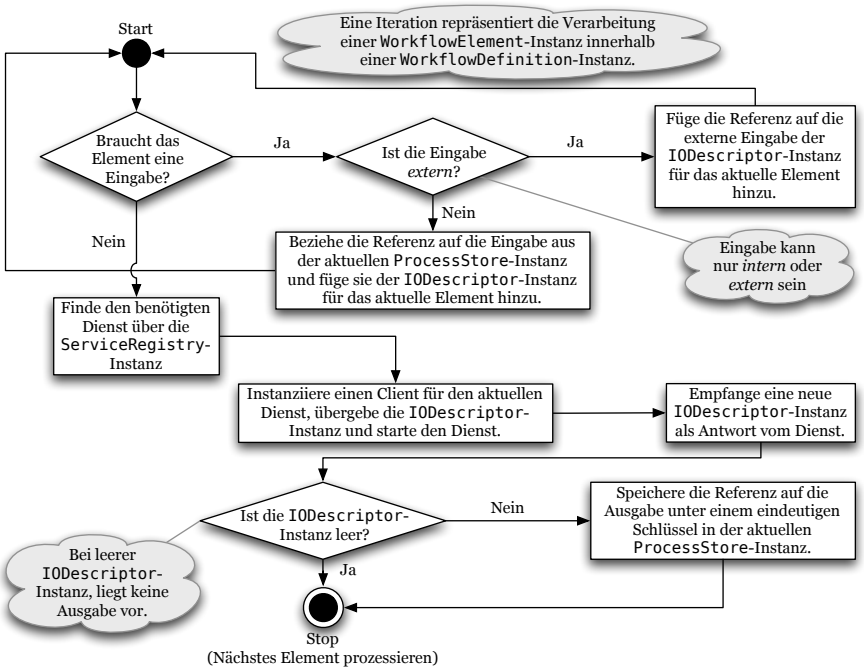

Abbildung 4.4: Abalaufdiagramm für die `RemoteWorkflowEngine`

4.1.4.4 Anpassung der Komponenten

Als letzter Schritt mussten die `AbstractComponent`-Klasse ebenso wie die einzelnen medienverarbeitenden Komponenten angepasst werden. Die `AbstractComponent`-Klasse implementiert nun das `CoreService`-Interface und registriert sich bei Instanziierung selbst in der Service Registry. Das vollständige Listing A.3 der finalen Version der `AbstractComponent`-Klasse mit allen Anpassungen findet sich im Anhang.

Die Implementierung des `MusicOMatService` in Listing 4.26 steht exemplarisch für die Umsetzung der anderen Komponenten als Dienste. Zu beachten ist, dass die Klasse selbst die Adresse und Beschreibung des Dienstes definiert. Diese Informationen werden über den `super()`-Konstruktor in der Service Registry veröffentlicht, so dass sich später der Dienst darüber wieder finden lässt.

Listing 4.26: `MusicOMatService`-Klasse in der finalen Version

```
12 package de.fhkoeln.santiago.services;
26 public class MusicOMatService extends AbstractComponent {
27
28   private final static String URI         = "http://localhost:8080/axis2/services/MusicOMatService";
29   private final static String DESCRIPTION = "Transformer:MusicOMatService";
38
39   protected IODescriptor _execute() {
40     IODescriptor output = new IODescriptor();
41
42     MediaComponent outputMedia = new Media();
43     outputMedia.setName("SlideshowWithMusic");
44
45     Logger.info("--- Slideshow File @ " + this.getInput().getDescriptorElements()[0]);
46     Logger.info("--- Audio File @ " + this.getInput().getDescriptorElements()[1]);
47
48     MediaComponent movieFile = getBroker().retrieve(this.getInput().getDescriptorElements()[0]);
49     MediaComponent audioFile = getBroker().retrieve(this.getInput().getDescriptorElements()[1]);
50
51     Logger.info("--- MediaObject for Slideshow File: " + movieFile.getPlayableData());
52     Logger.info("--- MediaObject for Audio File: " + audioFile.getPlayableData());
53
54     MediaAction action = new FFMpegMerger(movieFile, audioFile, outputMedia);
55     action.performAction();
56
57     URI mediaUri = getBroker().store(outputMedia);
58     output.add(mediaUri.toString());
59
60     return output;
61   }
86
87 }
```

Die Dienste werden anschließen auf einem Tomcat Servlet Container[19] über ein dort befindliches Axis2-Servlet verfügbar gemacht. Die Instanziierung der Dienste und die Auflösung der notwendigen Abhängigkeiten werden im Prototypen durch das Spring Framework[20] auf Basis des *Dependency Injection*-Pattern realisiert [Johnson u. Hoeller 2004, S. 130] und [Fowler 2004]. In Listing 4.27 ist dabei die entsprechende Spring Konfigurationsdatei zu sehen.

Listing 4.27: `applicationContext.xml`-zur Definition der Abhängigkeiten

```
1  <?xml version="1.0" encoding="UTF-8"?>
18 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
19
20 <beans>
21
22   <bean id="applicationContext" class="org.apache.axis2.extensions.spring.receivers.ApplicationContextHolder" />
23
24   <bean id="slideshowGeneratorService"
25     class="de.fhkoeln.santiago.services.SlideshowGeneratorService">
26     <constructor-arg index="0" ref="serviceRegistry" />
27     <property name="broker" ref="mediaBroker" />
28   </bean>
29
30   <bean id="musicProviderService"
31     class="de.fhkoeln.santiago.services.MusicProviderService">
32     <constructor-arg index="0" ref="serviceRegistry" />
33     <property name="broker" ref="mediaBroker" />
34   </bean>
35
36   <bean id="musicOMatService"
37     class="de.fhkoeln.santiago.services.MusicOMatService">
38     <constructor-arg index="0" ref="serviceRegistry" />
39     <property name="broker" ref="mediaBroker" />
40   </bean>
41
42   <bean id="videoPlayerService"
43     class="de.fhkoeln.santiago.services.VideoPlayerService">
44     <constructor-arg index="0" ref="serviceRegistry" />
45     <property name="broker" ref="mediaBroker" />
46   </bean>
47
48   <bean id="mediaBroker"
49     class="de.fhkoeln.cosima.media.mediabroker.MemcachedMediaBroker">
50     <property name="mediaStore" ref="mediaStore" />
51   </bean>
52
53   <bean id="serviceRegistry" class="de.fhkoeln.cosima.services.registry.MemcachedServiceRegistry" />
54   <bean id="mediaStore" class="de.fhkoeln.cosima.media.mediabroker.storage.FileSystemStore" />
55
56 </beans>
```

[19] http://tomcat.apache.org/
[20] http://www.springsource.org/

4.1.4.5 Die fertige Anwendung

Nachdem nun auch die Komponenten als Dienste realisiert sind und über Web Services auf einem Server zur Ausführung bereitstehen, muss nur noch das Santiago-Hauptprogramm wie in Listing 4.28 angepasst werden, so dass die `RemoteWorkflowEngine` verwendet wird. Das Ergebnis ist die Erstellung und Wiedergabe einer Diashow mit Hintergrundmusik in einer verteilten und dienstorientierten Umgebung unter Verwendung des COSIMA-Projekts, wie es im vorherigen Kapitel beschrieben wurde.

Listing 4.28: Die finale Version der Santiago Anwendung

```
8  package de.fhkoeln.santiago;
9
10 import java.io.IOException;
11
12 import de.fhkoeln.cosima.services.registry.MemcachedServiceRegistry;
13 import de.fhkoeln.cosima.services.registry.ServiceRegistry;
14 import de.fhkoeln.cosima.workflow.RemoteWorkflowEngine;
15 import de.fhkoeln.cosima.workflow.WorkflowDefinition;
16 import de.fhkoeln.cosima.workflow.YamlWorkflowDefinition;
17 import de.fhkoeln.cosima.workflow.storage.MapProcessStoreImpl;
18 import de.fhkoeln.cosima.workflow.storage.ProcessStore;
19
20 public class SantiagoApp {
21
22   private static String pathToWorkflowDefinition;
23
24   public static void main(String[] args) throws IOException {
25
26     if (args.length == 1) {
27       pathToWorkflowDefinition = args[0];
28
29       WorkflowDefinition   workflowDefinition = new YamlWorkflowDefinition(pathToWorkflowDefinition);
30       ProcessStore         processStore       = new MapProcessStoreImpl();
31       RemoteWorkflowEngine engine             = new RemoteWorkflowEngine();
32       ServiceRegistry      registry           = new MemcachedServiceRegistry();
33
34       engine.setWorkflowDefinition(workflowDefinition);
35       engine.setProcessStore(processStore);
36       engine.setRegistry(registry);
37
38       engine.execute();
39
40     } else {
41       System.err.println("Path to the workflow definition is needed!");
42       System.exit(-1);
43     }
44   }
45
46 }
```

4.2 Realisierung des Szenarios (Nerstrand)

Nachdem im vorherigen Abschnitt erfolgreich die COSIMA-Architektur realisiert wurde, kann nun das eigentliche Szenario aus Abschnitt 3.4 auf Basis dieser Architektur implementiert werden. Zur Entwicklung und zur besseren Kommunikation hat diese Anwendung den Codenamen *Nerstrand* erhalten. Bevor die wesentlichen Komponenten, die es umzusetzen galt, vorgestellt werden, soll zunächst das eigentliche Anwendungsprogramm aus Listing 4.29 diskutiert werden.

Listing 4.29: Die Nerstrand Anwendung als Umsetzung des Szenario

```
1  package de.fhkoeln.nerstrand;
13 public class NerstrandApp {
14   private static String pathToWorkflowDefinition;
15
16   public static void main(String[] args) throws IOException {
17     if (args.length == 1) {
18       pathToWorkflowDefinition = args[0];
19
20       WorkflowDefinition   workflowDefinition = new YamlWorkflowDefinition(pathToWorkflowDefinition);
21       ProcessStore         processStore       = new MapProcessStoreImpl();
22       RemoteWorkflowEngine engine             = new RemoteWorkflowEngine();
23       ServiceRegistry      registry           = new MemcachedServiceRegistry();
24
25       engine.setWorkflowDefinition(workflowDefinition);
26       engine.setProcessStore(processStore);
27       engine.setRegistry(registry);
28       engine.execute();
29
30     } else {
31       System.err.println("Path to the workflow definition is needed!");
32       System.exit(-1);
33     }
34   }
35 }
```

Wie sich leicht erkennen lässt bestehen keinerlei Unterschiede zum Anwendungsprogramm von Santiago aus Listing 4.28. Auch sonst ist Nerstrand ähnlich umgesetzt worden wie Santiago: Die Dienste werden auch hier über einen Tomcat Servlet Container verfügbar gemacht und die Auflösung der Abhängigkeiten geschieht ebenso über Dependency Injection und per Spring Framework. Der Anwendungsentwickler sollte bei der Realisierung einer Anwendung auf Basis von COSIMA lediglich eine Ablaufbeschreibung definieren und die notwendigen Dienste implementieren müssen. Demzufolge entspricht die Implementierung des Anwendungsprogramms für Nerstrand den Erwartungen.

Die Ablaufbeschreibung in Listing 4.30 ist für die Nerstrand Anwendung ist sogar noch einfacher geworden, als die der Santiago Anwendung, da nur ein Producer- sowie ein Consumer-Dienst benötigt werden.

Listing 4.30: Die Ablaufbeschreibung für die Nerstrand Anwendung

```
---
type: "producer"
description: "Producer:WebcamStreamingService"
uri: "http://localhost:8080/axis2/services/WebcamStreamingService"
namespace: "http://services.nerstrand.fhkoeln.de"
input:
  - type: "external"
    uri: "qtcapture://"
output:
  - uri: "http://localhost:8080/axis2/services/WebcamStreamingService/output"
predecessors: null
successors:
  - uri: "http://localhost:8080/axis2/services/StreamingPlayerService"

---
type: "consumer"
description: "Consumer:StreamingPlayerService"
uri: "http://localhost:8080/axis2/services/StreamingPlayerService"
namespace: "http://services.nerstrand.fhkoeln.de"
input:
  - type: "internal"
    uri: "http://localhost:8080/axis2/services/WebcamStreamingService/output"
output: null
predecessors:
  - uri: "http://localhost:8080/axis2/services/WebcamStreamingService"
successors: null
```

Selbst der Consumer-Dienst unterscheidet sich nur marginal von dem Consumer-Dienst in Santiago. Daher soll an dieser Stelle vor allem der Producer-Dienst vorgestellt werden, der das Webcam Signal abgreift und als Stream bereitstellt. Die Implementierung der wesentlichen Funktionalitäten dieses Dienstes lässt sich Listing 4.31 entnehmen.

Listing 4.31: Der `WebcamStreamingService` der Nerstrand Anwendung

```
package de.fhkoeln.nerstrand.services;
public class WebcamStreamingService extends AbstractComponent {

  private static final String URI         = "http://localhost:8080/axis2/services/WebcamStreamingService";
  private static final String DESCRIPTION = "Producer:WebcamStreamingService";

  public WebcamStreamingService(ServiceRegistry registry) {
    super(registry, URI, DESCRIPTION);
  }
  @Override
  protected IODescriptor _execute() {
    IODescriptor output = new IODescriptor();

```

```
43      MediaComponent stream = new Media();
44      stream.setName("Webcam Stream");
45      stream.setNamespace("Nerstrand::StreamingService");
46
47      MediaOperation streamingOp = new VLCStreamingOperation(getInput().first(), stream);
48      thread(streamingOp, false);
49
50      boolean wait = true;
51      File sdp_file = new File(stream.getReferenceToRealData());
52      while(wait)
53        wait = sdp_file.exists() ? false : true;
54
55      URI mediaUri = getBroker().store(stream);
56      output.add(mediaUri.toString());
57
58      return output;
59    }
92  }
```

Der eigentliche Streamingprozess ist dabei in der `VLCStreamingOperation`-Klasse gekapselt worden. Dies geschah aus zwei Gründen: Zum einen ist die Funktionalität dadurch in Isolation leichter zu testen, zum anderen, und das ist der entscheidende Grund, lässt sich die Operation in einem Thread starten. Dies ist notwendig, da die Operation kein definiertes Ende hat, wie etwa die Operationen in der Santiago Anwendung. Als Lösung wird der Streamingvorgang über die `thread()`-Methode aus Zeile 49 in einem Thread ausgeführt und die `execute()`-Methode kann dadurch regulär beendet werden. Das bedeutet, dass überhaupt erst das `stream`-Medienobjekt im Medienbroker gespeichert werden kann und die `output`-Instanz mit den notwendigen Informationen an den Dienstnutzer zurückgegeben werden wird. Wäre das nicht der Fall, würde der gesamte Workflow zum erliegen kommen, da die Eingabe für den folgenden Dienst nicht gefunden werden kann.

Eine andere Besonderheit bestand bei der Implementierung der `VLCStreamingOperation`. Da der verwendete `MediaStore` nur in der Lage ist Dateien abzulegen, war es notwendig, die Informationen als Datei zu repräsentieren. Als Lösung dieses Problems werden die Informationen über den Stream in einer SDP-Datei abgelegt. SDP steht für *Session Description Protocol* und dient dazu, Informationen wie die Kodierung der Daten, Transportprotokoll, Ziel- und Ursprungsadresse des Streams sowie den verwendeten Port in einer standardisierten Form zu repräsentieren[21]. Diese Datei kann dann von der `MediaStore`-Instanz gespeichert und von dem Consumer-Dienst später dazu verwendet werden, den eigentlichen Stream wiederzugeben.

[21] `http://tools.ietf.org/html/rfc4566`

Die hier beschriebenen Schritte haben ausgereicht, das Anwendungsszenario vollständig prototypisch zu implementieren. Es kann bereits als Indiz dafür gesehen werden, dass die Architektur geeignet ist, Multimediaanwendungen zu realisieren. Eine genauere Betrachtung und vor allem eine Validierung der Implementierung und deren Ergebnisse erfolgt im folgenden Kapitel.

5 Validierung der Architektur

Neben der Konzeption und prototypischen Realisierung von COSIMA ist der dritte Pfeiler dieser Arbeit die Validierung von COSIMA. Im vorherigen Kapitel wurde sowohl die Architektur selbst als auch ein Prototyp für ein Anwendungsszenario implementiert. In diesem Kapitel steht die Validierung dieser Implementierung im Fokus. Bevor aber die Ergebnisse im Detail präsentiert werden können, soll zunächst eine Einordnung und Definition des Begriffs der *Validierung* erfolgen.

5.1 Definition

Als erster Hinweis eine Definition für Validierung zu finden, die in dieser Arbeit Gültigkeit besitzt, soll das offizielle Glossar über Software-Engineering des IEEE liefern:

Definition 32 (Validierung (IEEE))**.** *„The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements. Contrast with: verification.“* [IEEE 1990]

In dieser Definition wird Validierung mit dem Begriff Evaluation erklärt, was die Notwendigkeit aufbringt auch diesen zu definieren:

Definition 33 (Evaluation)**.** *„[Die] sach- und fachgerechte Bewertung“*[1]

Eine „fachgerechte Bewertung“ ist für den gegebenen Kontext zu verstehen, als dass die prototypische Implementierung der Architektur und des Szenario anhand bestimmter Kriterien bewertet wird. Wie in Kapitel 3 jedoch bereits verdeutlicht wurde, wird im Rahmen dieser Arbeit keine Evaluierung der Architektur vorgenommen, sondern eine Validierung. Aus diesem Grunde wird hier die folgende Definition verwendet:

[1] aus: Duden - Deutsches Universal Wörterbuch A-Z, 3. Aufl., 1996

Definition 34 (Validierung (Balzert))**.** *„Unter Validation wird die Eignung bzw. der Wert eines Produktes bezogen auf seinen Einsatzzweck verstanden.“* [Balzert 1998, S. 101]

Sie drückt wesentlich deutlicher aus, dass Inhalt einer Validierung ist, zu überprüfen, ob sich die konzipierte Architektur grundsätzlich für die Konstruktion von verteilten Multimediaanwendungen eignet.

Wie bereits in der Definition des IEEE deutlich wurde, steht der Begriff der *Validierung* in der Regel nicht für sich allein, sondern gemeinsam mit dem Begriff der *Verifikation*:

Definition 35 (Verifikation (IEEE))**.** *„The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. Contrast with: validation.“* [IEEE 1990]

Hier ebenfalls zu der Definition des IEEE ergänzend die Definition von Balzert, ohne die Verwendung des Begriffs der Evaluation:

Definition 36 (Verifikation (Balzert))**.** *„Unter Verifikation wird die Überprüfung der Übereinstimmung zwischen einem Software-Produkt und seiner Spezifikation verstanden.“* [Balzert 1998, S. 101]

Bei [Boehm 1984] wird der Zusammenhang von Verifikation und Validierung noch einmal deutlicher gemacht:

Verifikation „Am I building the product right?“ [Boehm 1984, S. 75]

Validierung „Am I building the right product?“ [Boehm 1984, S. 75]

Durch die gegebenen Definition wird auch deutlich, warum zum gegebenen Zeitpunkt nur eine Validierung stattfinden kann und nicht zusätzlich auch eine Verifikation: Es existiert bis dato keine formale Spezifikation des Produkts, in diesem Fall der Architektur, gegen die eine Verifikation durchgeführt werden könnte.

Nachdem die Begrifflichkeiten geklärt sind, werden im nächste Schritt zunächst die Ergebnisse der Validierung der Implementierung der Architektur auf Basis der Santiago Anwendung vorgestellt. Anschließend werden jene Ergebnisse betrachtet, die sich aus der prototypischen Umsetzung des Anwendungsszenario ergeben haben.

Zum verwendeten Vorgehen bei der Validierung sei an dieser Stelle noch erwähnt, dass sich an dem Grad der Umsetzbarkeit sowohl der Santiago als auch der Nerstrand Anwendung auf Grundlage der Architektur orientiert wurde. Da beide Anwendung einen Anwendungsfall repräsentieren, der mit COSIMA realisierbar sein soll, beantwortet der Grad der Umsetzbarkeit die Frage, ob das richtige Produkt gebaut wird.

5.2 Ergebnisse aus der Santiago Anwendung

Grundsätzlich lässt sich festhalten, dass die Umsetzung der Architektur so gelang, wie sie vorgesehen war. Dennoch kam es bei einigen Punkten zu Abweichungen, welche im Folgenden genauer beschrieben werden sollen.

5.2.1 Verwendung des Nachrichtensystems

Die einzige gravierende Änderung an der ursprünglichen Architektur ist bei der Verwendung des Nachrichtensystems vorgenommen worden. Zwar ist ein Nachrichtensystem mit einer entsprechenden Schnittstelle und deren Implementierung auf JMS Basis über ActiveMQ[2] vorgenommen, am Ende jedoch nicht verwendet worden. Im Konzept ist das Nachrichtensystem als Schnittstelle zwischen der Servicekomposition und den medienverarbeitenden Diensten vorgesehen: Die Servicekomposition sollte die Dienste immer nur über das Nachrichtensystem aufrufen können. Entsprechend dem Konzept wurde das Nachrichtensystem nach dem *Publish-Subscribe*-Pattern realisiert. Dies hat jedoch zu Problemen beim Aufruf der Dienste geführt. Ein Dienst gibt nach Durchführung seiner Funktionalität eine `IODescriptor`-Instanz an den Dienstnutzer zurück; In ihr ist hinterlegt wo der nächste Dienst die entsprechenden Medienobjekte finden kann. Die ausführende Komponente in der Servicekomposition kann also den nächsten Dienst nicht ohne diese Information aufrufen. Es handelt es sich somit um einen *synchronen* Aufruf. Ein Nachrichtensystem wird aber vor allem eingesetzt um eine *asynchrone* Kommunikation zu realisieren. Zwar ließe sich eine funktionierende Funktionsweise der Servicekomposition auf einer asynchronen Kommunikation über das Nachrichtensystem umsetzen, diese wäre aber ungleich komplexer und damit auch fehleranfälliger als eine synchrone Kommunikation.

[2] `http://activemq.apache.org`

Die zur Zeit gewählte Implementierung, dass die Servicekomposition die Dienste direkt über eine Web Service Schnittstelle aufruft, entspricht dabei auch der Vorgehensweise von BPEL und ist demnach als verwendbar einzustufen. Die durch die Verwendung eines Nachrichtensystems erhoffte Entkopplung der Servicekomposition und der einzelnen Dienste wird dabei nicht verworfen, sondern durch den Einsatz der Service Registry erreicht.

Dennoch ist die Implementierung des Nachrichtensystems nicht obsolet, so ließe es sich etwa zur Vermittlung von Synchronisationsspezifikationen verwenden.

5.2.2 Verwendung der Servicekomposition

Die jetzige Implementierung der Servicekomposition erfüllt sowohl die Anforderungen der Santiago als auch der Nerstrand Anwendung. Dennoch zeichnen sich einige Probleme für die weitere Verwendung ab.

Für komplexere Kompositionen, die etwa Schleifen oder eine variable Abfolge beinhalten, ist die jetzige Umsetzung nicht geeignet. Sowohl die Möglichkeiten der verwendeten Ablaufbeschreibung als auch die zur Verfügung stehenden `WorkflowEngine`-Implementierungen sehen solche Fälle nicht vor. Für deren Verwendung wäre zumindest eine `WorkflowEngine` auf Basis eines Zustandsautomaten, wie bei [Biornstad u. a. 2006] beschrieben, zu realisieren. Darüber hinaus sollte der Einsatz einer etablierten Prozessbeschreibungssprache evaluiert werden. Eine erste Prüfung von BPEL für den Einsatz in Multimediaanwendungen ist in diesem Zusammenhang bei [Richter 2008] durchgeführt worden.

Ebenfalls Inhalt einer Weiterentwicklung sollte die Möglichkeit sein, die Servicekomposition verteilt ausführen zu können. In der jetzigen Implementierungen ist dies auch ohne größere Anpassungen in der `RemoteWorkflowEngine` möglich. Es muss lediglich ein verteilter `ProcessStore` realisiert werden.

5.2.3 Integration von Synchronisation

Bereits bei dieser einfachen Anwendung bestand bereits der Bedarf nach Synchronisation von Medien. Die Tranformer-Komponente, die die Diashow und das Musikstück verarbeitet, muss dazu beide Medien synchronisieren. Es findet in diesem Fall eine implizite Synchronisation statt. Eine formale Spezifikation dieses Synchronisationsprozess ist nicht

vorgesehen worden. Es handelt sich aber dennoch um eine valide und empfohlene Umsetzung von Synchronisation in einer verteilten Umgebung nach [Steinmetz 2000, S. 609]. Wie bereits in Abschnitt 2.4.8.2 beschrieben, werden hier zwei synchronisierte Medien als Multiplex-Daten an den Consumer weitergereicht.

Ohne die Umsetzung einer expliziten Synchronisationsspezifikationen bestand auch nicht die Notwendigkeit der Implementierungen des Synchronisation-Composite, wie es im Klassendiagramm des Medienobjekts in Abbildung 2.8 dargestellt ist. In wie weit sich diese Modellierung daher tatsächlich zur Realisierung von Synchronisationsanforderungen in einer verteilten Umgebung eignet, werden weitere Arbeiten zeigen müssen.

5.2.4 Verwendung des Medienbroker

Die Implementierungen des Medienbroker sowie der Medienobjekte ist zu diesem Zeitpunkt als ausreichend zu betrachten. Alle Anforderungen ließen sich damit leicht umsetzen. Jedoch wurde bisher nicht von der `MediaContainer`-Klasse Gebrauch gemacht. Zur Zeit sind aber keine Anzeichen erkennbar, dass die Implementierungen des Medienobjekts nicht ausreichend ist.

Für eine weitere Entwicklung des Medienbroker ist es erstrebenswert, ihn stärker als eigene Anwendung herauszuarbeiten. Dienste sollten die Möglichkeit erhalten, mit dem Medienbroker über ein eigenes Protokoll zu kommunizieren. Ein URI-Schema der Form `cosima://santiago.fh-koeln.de/media/Slideshow` wäre in diesem Fall eine denkbare Möglichkeit.

5.2.5 Nicht-funktionale Aspekte

Obwohl nicht-funktionale Anforderungen nicht Bestandteil der eigentlichen Validierung sind, sollen an dieser Stelle dennoch einige Punkte genannt werden, die während der Implementierungen bereits aufgefallen sind.

Vor allem durch die prototypische Charakteristik der Implementierungen der Architektur werden Fehler nicht immer adäquat behandelt. *Exceptions* werden zwar geworfen, jedoch nicht in einem konsistenten Rahmen. Für die weitere Entwicklung sollte demnach in jedem Fall eine vollständigere Fehlerbehandlung implementiert werden.

Eng mit der unzureichenden Fehlerbehandlung ist die geringe Fehlerrobustheit zu nennen. Bei nicht vollständig konformer Verwendung der Architektur kann nicht mit Sicherheit bestimmt werden, ob die Anwendung korrekt ablaufen wird oder nicht.

Vor allem für die Wartung und Entwicklung ist es notwendig, das momentan verwendete *Logging-Singleton* durch einen robusten *Logging-Dienst* zu ersetzen. Denn auch in einer verteilten Umgebung ist es notwendig, ein zentrales Log der Gesamtapplikation zur Verfügung zu haben.

5.3 Ergebnisse aus der Nerstrand Anwendung

Nachdem im vorherigen Teil die Ergebnisse der Umsetzung von Santiago vorgestellt wurden, liegt der Fokus in diesem Abschnitt auf die Umsetzbarkeit des Anwendungsszenarios.

Wie bereits in Abschnitt 4.2 festgehalten wurde, ließ sich die Nerstrand Anwendung sehr leicht innerhalb der Architektur implementieren. Einzig die momentane Realisierung der `MediaStore`-Funktionalität hat zu einer Erschwerung geführt. Die Etablierung einer `MediaStoreFactory`, die zur Laufzeit die für die jeweiligen Daten optimale Speicherstrategie wählt, würde hier bereits Abhilfe schaffen. In diesem Zusammenhang sollte auch eine weitere Betrachtung des bereits erwähnten MultiMonster-Medienservers erfolgen. Zur Bewertung der einzelnen Daten können darüber hinaus umfangreiche Metadaten notwendig sein, daher muss auch dieser Teil des Medienobjekts weiter ausgearbeitet werden, hierzu sei auch auf [Lehmann 2009] verwiesen.

Ein letzter Punkt, der bei der Umsetzung des Anwendungsszenario auffiel, war, dass es durchaus sinnvoll wäre, wenn sich die Operation eines einzelnen Dienstes wieder stoppen ließe. In der Nerstrand Anwendung stellt der `WebcamStreamingService` solange den Stream bereit, bis der Thread in dem die eigentliche Operation ausgeführt wird, unterbrochen wird.

Insgesamt lässt sich sagen, dass der Grad der Umsetzbarkeit der beiden Anwendung sehr hoch war. Vor allem wurde deutlich, dass es durch COSIMA sehr leicht wird, unterschiedliche Strategien zur Medienverarbeitung einzusetzen. So ist fast in jedem Dienst eine andere technologische Grundlage verwendet worden, um seine jeweilige Funktionalität bereitzustellen. Die meisten davon basieren zudem noch nicht einmal auf der Java Plattform. Somit

kann also festgehalten werden, dass die konzipierte Architektur hinter COSIMA als valide im Rahmen der hier festgelegten Kriterien zu bezeichnen ist.

6 Fazit

In dieser Arbeit wurde die Konzeption, prototypische Implementierung sowie die Validierung einer dienstorientierten Architektur für Multimediaanwendungen behandelt. Bei der Konzeption wurde zunächst ersichtlich, dass eine dienstorientierte Architektur eine Vielzahl von unterschiedlichen Komponenten aufweist, von denen jede einzelne bereits ein gewisses Maß an Komplexität aufweist. Unter der Hinterzunahme der zusätzlichen Eigenschaften, die die Integration von Multimedia gefordert hat, entstand eine komplexe Gesamtarchitektur.

Durch diese Komplexität war es notwendig die jeweiligen Komponenten zunächst isoliert zu betrachten und im Anschluss in den größeren Kontext einzuordnen. Durch dieses Vorgehen wurde zum einen ein sehr viel besseres Verständnis über COSIMA entwickelt, zum anderen lassen sich die Komponenten in weiteren Arbeiten dadurch einzeln leichter betrachten.

In seiner Gesamtheit weist das COSIMA-Projekt schon ein sehr hohes Maß an Vollständigkeit auf. Die wesentlichen Aspekte sind durchweg bekannt und entsprechend diskutiert worden. Lediglich in vertikaler Richtung fehlt es Aspekten, wie der Synchronisation von Medien oder der Behandlung von Metadaten noch an Substanz.

Entsprechend der Konzeption konnte auch die Implementierung in horizontaler Ebene fast vollständig umgesetzt werden. Auch hier wurde für nachfolgende Arbeiten diese Grundlage geschaffen. Dies wurde vor allem auch durch das iterative Entwicklungsvorgehen und durch den Einsatz eines szenariobasierten Ansatzes begünstigt. Durch den vermehrten Einsatz von etablierten Entwurfsmustern und Entwicklungsparadigmen sowie der Verwendung der Programmiersprache Java während der Implementierung wurde auch in diesem Teil eine solide Basis für Folgeprojekte geschaffen.

Es lässt sich also festhalten, dass die beiden aufgestellten Ziele im Rahmen der gestellten Aufgaben für diese Arbeit als erfüllt gelten können. Dennoch müssen einige Punkte kritisch angemerkt werden.

Die Nichtverwendung einer etablierten Prozessbeschreibungssprache stellt eines der Hauptprobleme dar. Zwar erlaubt die gewählte Lösung in dieser Arbeit ein besseres Verständnis über die internen Abläufe bei der Servicekomposition, für eine langfristige Weiterentwicklung sollte diese Komponente jedoch ausgetauscht werden. Stattdessen sollte die Eignung und Anpassung bestehender Lösungen weiterverfolgt werden, wie sie bereits bei [Richter 2008] begonnen wurde.

Ebenfalls nur rudimentär ist die Integration von Synchronisation durchgeführt worden. Es wurde vor allem klar, dass es sich dabei um ein sehr komplexes Themengebiet handelt. In [Antons 2009] findet aber bereits eine intensivere Auseinandersetzung mit der Einbindung von Synchronisation in das COSIMA-Projekt statt. In Zukunft sollten Arbeiten die dort gefundenen Ergebnisse mit der hier vorgestellten Implementierung in Einklang bringen.

Der letzte Punkt, der weitestgehend ausgelassen wurde, ist die Betrachtung von Metadaten. Da dieser aber nicht zur essentiellen Funktionalität des umgesetzten Szenarios gehört, wurde bisher nur eine Schnittstelle vorgesehen. In [Lehmann 2009] findet sich eine dedizierte Betrachtung dieses Themas im Kontext von COSIMA.

Zu den anderen Punkten kann festgehalten werden, dass sie trotz ihres prototypischen Charakters durchaus so implementiert sind, dass sie sich auch für den Einsatz in Folgeprojekten eignen.

Abschließend lässt sich festhalten, dass das gesamte COSIMA-Projekt einen innovativen und breiten Themenbereich im Rahmen der Medieninformatik bietet. Nachdem in dieser Arbeit zum ersten Mal eine funktionierende Implementierung der grundlegenden Architektur geliefert wurde, haben künftige Arbeiten in diesem Bereich eine robuste Basis zur Hand, um in den unterschiedlichen Teilbereichen des Projekts in die Tiefe gehen zu können.

Literaturverzeichnis

Ackermann 1994

Ackermann, P.: Design and implementation of an object-oriented Media Composition Framework. In: *Proceedings of the International Computer Music Conference* International Computer Music Accociation, 1994, S. 220–220

Ackermann u. Eichelberg 1996

Ackermann, Philipp ; Eichelberg, Dominik: *Developing Object-Oriented Multimedia-Software: Based on MET++ Application Framework*. dpunkt-Verlag, Heidelberg, 1996

Alonso 2004

Alonso, G.: *Web Services: Concepts, Architectures and Applications*. Springer, 2004

Antons 2009

Antons, Stefan: *Integration von Streaming-Diensten in die Cologne Service-Oriented Integrative Multimedia Architecture*. Steinmüllerallee 1, 51643 Gummersbach, 2009. – Master Thesis

Arndt u. a. 2007

Arndt, R. ; Troncy, R. ; Staab, S. ; Hardman, L. ; Vacura, M.: COMM: Designing a Well-Founded Multimedia Ontology for the Web. In: *Lecture Notes in Computer Science* 4825 (2007), S. 30–43

Balzert 1999

Balzert, Heide: *Lehrbuch der Objektmodellierung*. Spektrum, Akademischer Verlag, 1999

Balzert 1998

Balzert, Helmut: *Lehrbuch der Software-Technik*. Bd. 2, Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung. Spektrum, Akad. Verl., 1998

Basili u. Turner 2005

Basili, V.R. ; Turner, A.J.: Iterative Enhancement: A Practical Technique for Software Development. In: *Foundations of Empirical Software Engineering: The Legacy of Victor R. Basili* (2005), S. 28–39

Bass u. a. 2003

Bass, L. ; Clements, P. ; Kazman, R.: *Software Architecture in Practice*. 2. Auflage. Addison-Wesley Professional, 2003

Bertino u. Ferrari 1998

Bertino, E. ; Ferrari, E.: Temporal synchronization models for multimedia data. In: *Knowledge and Data Engineering, IEEE Transactions on* 10 (1998), Nr. 4, S. 612–631

Biornstad u. a. 2006

Biornstad, B. ; Pautasso, C. ; Alonso, G.: Control the Flow: How to Safely Compose Streaming Services into Business Processes. In: *Proceedings of the IEEE International Conference on Services Computing table of contents* IEEE Computer Society Washington, DC, USA, 2006, S. 206–213

Blakowski u. Steinmetz 1996

Blakowski, G. ; Steinmetz, R.: A media synchronization survey: reference model, specification, andcase studies. In: *Selected Areas in Communications, IEEE Journal on* 14 (1996), Nr. 1, S. 5–35

Boehm 1986

Boehm, Barry W.: A spiral model of software development and enhancement. In: *SIGSOFT Softw. Eng. Notes* 11 (1986), Nr. 4, S. 14–24

Boehm 1984

Boehm, B.W.: Verifying and Validating Software Requirements and Design Specifications. In: *IEEE Software* 1 (1984), Nr. 1, S. 75–88

Breuer u. a. 2008

Breuer, D. ; Cohnen, S. ; Richter, M.: Konzeption eines verteilten, dienstorienterten Rahmenwerkes für Medienverarbeitung. / Institutsbericht. Institut für Informatik, Fakultät für Informatik und Ingenieurwissenschaften, FH Köln, Campus Gummersbach. Steinmüllerallee 1, 51643 Gummersbach, 2008 (1). – Forschungsbericht

Brooks 1987

Brooks, F.P.: No Silver Bullet: Essence and Accidents of Software Engineering. In: *IEEE Computer* 20 (1987), Nr. 4, S. 10–19

Carroll 2000

Carroll, John M.: Five reasons for scenario-based design. In: *Interacting with Computers* 13 (2000), Nr. 1, S. 43–60

Chappell 2004

Chappell, D.A.: *Enterprise Service Bus*. O'Reilly Media, Inc., 2004

Clements u. a. 2002

Clements, P. ; Kazman, R. ; Klein, M.: *Evaluating Software Architectures - Methods and Case Studies*. Addison-Wesley Professional, 2002

Coulouris u. a. 2001

Coulouris, G. ; Dollimore, J. ; Kindberg, T.: *Distributed systems*. Addison-Wesley Reading, Mass, 2001

Crispen u. Lynn D. Stuckey 1994

Crispen, Robert G. ; Lynn D. Stuckey, Jr.: Structural model: architecture for software designers. In: *TRI-Ada '94: Proceedings of the conference on TRI-Ada '94*. New York, NY, USA : ACM, 1994, S. 272–281

Erl 2008

Erl, Thomas: *SOA: Principles of Service Design*. Prentice Hall, 2008

Fowler 2003

Fowler, M.: *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2003

Fowler 2004

Fowler, Martin: *Inversion of Control Containers and the Dependency Injection pattern*. Version: 2004. `http://martinfowler.com/articles/injection.html`, Abruf: 13.01.2009

Gaggi u. Celentano 2005

Gaggi, O. ; Celentano, A.: Modelling Synchronized Hypermedia Presentations. In: *Multimedia Tools and Applications* 27 (2005), Nr. 1, S. 53–78

Gamma u. a. 1995

Gamma, E. ; Helm, R. ; Johnson, R. ; Vlissides, J.: *Design Patterns: Elements of reusable object-oriented software*. Addison-Wesley Professional, 1995

Gibbs 1995

Gibbs, Simon: Multimedia Component Frameworks. In: Nierstrasz, Oscar (Hrsg.) ; Tsichritzis, Dennis (Hrsg.): *Object-Oriented Software Composition*. Prentice-Hall, 1995, S. 305–319

Hohpe u. a. 2004

Hohpe, G. ; Woolf, B. ; Brown, K.: *Enterprise integration patterns*. Addison-Wesley Boston, 2004

Hollingsworth 1995

Hollingsworth, D: Workflow Management Coalition: The Workflow Reference Model. (1995), Jan

Hüvelmeyer u. a. 2004

Hüvelmeyer, J. ; Irmer, A. von ; Ritterskamp, C. ; Wenk, T.: CampusSource Engine / Medienzentrum der Universität Dortmund. Emil-Figge-Str. 50, 44227 Dortmund, 2004. – Forschungsbericht

IEEE 1990

IEEE: IEEE standard glossary of software engineering terminology. (1990), Nr. IEEE Std 610.12-1990

Ionita u. a.

Ionita, M. ; Hammer, D. ; Obbink, H.: Scenario Based Software Architecture Evaluation Methods: An Overview. In: *Analysis* 5, S. 6–18

Johnson u. Hoeller 2004

Johnson, R. ; Hoeller, J.: *Expert One-on-One J2EE Development without EJB*. John Wiley & Sons, 2004

Josuttis 2007

Josuttis, N.: *SOA in Practice: The Art of Distributed System Design*. O'Reilly Media, Inc., 2007

Kazman u. a. 2000

Kazman, R. ; Klein, M. ; Clements, P.: ATAM: Method for Architecture Evaluation (CMU/SEI-2000-TR-004, ADA382629). (2000)

Kazman u. a. 1996

Kazman, Rick ; Abowd, Gregory ; Bass, Len ; Clements, Paul: Scenario-Based Analysis of Software Architecture. In: *IEEE Software* 13 (1996), Nr. 6, S. 47–55

Kruchten 2004

Kruchten, P.: *The Rational Unified Process: An Introduction*. Addison-Wesley Professional, 2004

Lehmann 2009

Lehmann, Andreas: *Metadaten-Management und semantische Annotation von Medienressourcen in einer dienst-orientierten integrativen Multimedia-Architektur*. Steinmüllerallee 1, 51643 Gummersbach, 2009. – Master Thesis

Liebhart 2007

Liebhart, D.: *SOA goes real*. Hanser Verlag, 2007

Little u. Ghafoor 1991

Little, TDC ; Ghafoor, A.: Multimedia synchronization protocols for broadband integrated services. In: *Selected Areas in Communications, IEEE Journal on* 9 (1991), Nr. 9, S. 1368–1382

Little u. a. 1991

Little, T.D.C. ; Ghafoor, A. ; Chen, C.Y.R. ; Chang, CS ; Berra, P.B.: Multimedia Synchronization. In: *Data Engineering Bulletin* 14 (1991), Nr. 3, S. 26–35

MacKenzie u. a. 2006

MacKenzie, C.M. ; Laskey, K. ; McCabe, F. ; Brown, P.F. ; Metz, R.: Reference Model for Service-Oriented Architecture 1.0. In: *OASIS Standard* (2006)

Masak 2007

Masak, D.: *SOA?: Serviceorientierung in Business und Software*. Springer, 2007

Mellon

Mellon, Software Engineering Institute C.: *SEI - Software Architecture Glossary*. `http://www.sei.cmu.edu/architecture/glossary.html`, Abruf: 03.11.2008

de Mey u. Gibbs 1993

Mey, Vicki de ; Gibbs, Simon: A multimedia component kit: experiences with visual composition of applications. In: *Multimedia '93: Proceedings of the first ACM international conference on Multimedia*. New York, NY, USA : ACM, 1993, S. 291–300

Meyer u. a. 1993

Meyer, T. ; Effelsberg, W. ; Steinmetz, R.: A taxonomy on multimedia synchronization. In: *Distributed Computing Systems, 1993., Proceedings of the Fourth Workshop on Future Trends of*, 1993, S. 97–103

Milanovic u. Malek 2004

Milanovic, N. ; Malek, M.: Current Solutions for Web Service Composition. In: *IEEE Internet Computing* (2004), S. 51–59

Natis 2003

Natis, Y.: Service-Oriented Architecture Scenario. In: *Gartner Research Note AV-19-6751* (2003), S. 6

Papazoglou 2003

Papazoglou, Michael P.: Service-Oriented Computing: Concepts, Characteristics and Directions. In: *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on* (2003), S. 3–12

Papazoglou 2008

Papazoglou, Michael P.: *Web Services: Principles and Technology.* Pearson Education Ltd., 2008

Papazoglou u. a. 2007

Papazoglou, M.P. ; Traverso, P. ; Dustdar, S. ; Leymann, F.: Service-Oriented Computing: State of the Art and Research Challenges. In: *Computer* (2007), S. 38–45

Peltz 2003

Peltz, C.: Web Services Orchestration and Choreography. In: *Computer* (2003), S. 46–52

Reussner u. Hasselbring 2006

Reussner, Ralf ; Hasselbring, Wilhelm: *Handbuch der Software-Architektur.* Dpunkt Verlag, 2006

Richter 2008

Richter, Matthias: *Modellierung von Produktion, Verarbeitung und Wiedergabe audiovisueller Medien auf Basis von dienstorientierten Prozess-Beschreibungssprachen.* Steinmüllerallee 1, 51643 Gummersbach, 2008. – Master Thesis

Rosson u. Carroll 2002

Rosson, M.B. ; Carroll, J.M.: *Usability Engineering: Scenario-Based Development of Human-Computer Interaction*. Morgan Kaufmann, 2002

Scherp u. Boll 2006

Scherp, Ansgar ; Boll, Susanne ; Reussner, Ralf (Hrsg.) ; Hasselbring, Wilhelm (Hrsg.): *Framework-Entwurf*. Dpunkt Verlag, 2006 (Handbuch der Software-Architektur)

Schwabe u. a. 2001

Schwabe, G. ; Streitz, N. ; Unland, R.: *CSCW-kompendium: Lehr-und Handbuch zum computerunterstützten Kooperativen arbeiten*. Springer, 2001

Starke 2008

Starke, G.: *Effektive Software-Architekturen - Ein praktischer Leitfaden*. 3. Auflage. Carl Hanser Verlag, 2008

Steinmetz 1990

Steinmetz, R.: Synchronization properties in multimedia systems. In: *Selected Areas in Communications, IEEE Journal on* 8 (1990), Nr. 3, S. 401–412

Steinmetz 2000

Steinmetz, R.: *Multimedia-Technologie: Grundlagen, Komponenten und Systeme*. Springer, 2000

Steinmetz u. Meyer 1992

Steinmetz, R. ; Meyer, T.: Multimedia Synchronization Techniques: Experiences Based on Different System Structures. In: *Multimedia Communications, 1992., 4th IEEE ComSoc International Workshop on*, ACM New York, NY, USA, 1992, S. 306–314

Steinmetz u. Nahrstedt 1995

Steinmetz, R. ; Nahrstedt, K.: *Multimedia: Computing, Communications and Applications*. Prentice Hall PTR, 1995

Suchomski u. a. 2004

Suchomski, M. ; Meyerhöfer, M. ; Meyer-Wegener, K.: Open and Reconfigurable Multimedia Server Architecture. In: *Proc. 1st Int. Workshop on Multimedia Information Systems Technology (MMISTech 2004, Szklarska Poreba, Poland).*, 2004, S. 83–7085

Sun Microsystems

Sun Microsystems, Inc.: *JMF FAQ*. `http://java.sun.com/javase/technologies/desktop/media/jmf/reference/faqs/index.html#what`, Abruf: 02.12.2008

Weidenhaupt u. a. 1998a

Weidenhaupt, K. ; Pohl, K. ; Jarke, M. ; Haumer, P.: Scenario usage in system development: A report on current practice IEEE Computer Society Washington, DC, USA, 1998, S. 222–241

Weidenhaupt u. a. 1998b

Weidenhaupt, K. ; Pohl, K. ; Jarke, M. ; Haumer, P. ; Aachen, T.H.: Scenarios in system development: current practice. In: *Software, IEEE* 15 (1998), Nr. 2, S. 34–45

Wu u. a. 2001

Wu, D. ; Hou, YT ; Zhu, W. ; Zhang, Y.Q. ; Peha, JM: Streaming video over the Internet: approaches and directions. In: *Circuits and Systems for Video Technology, IEEE Transactions on* 11 (2001), Nr. 3, S. 282–300

A Weitere Listings

In diesem Anhang finden sich solche Listings, die eine zu umfangreich waren, um im laufenden Text untergebracht zu werden, dennoch aber von Interesse während des Lesens der Arbeit sind und daher nicht nur auf der Begleit-CD zu finden sein sollten.

Listing A.1: Die `WorkflowElement`-Klasse

```
12 package de.fhkoeln.cosima.workflow;
13
14 import java.util.ArrayList;
15 import java.util.HashMap;
16 import java.util.Iterator;
17 import java.util.List;
18 import java.util.Map;
19
20 import de.fhkoeln.cosima.components.AbstractComponent;
21
22 /**
23  * This class represents a single workflow element within a workflow
24  * definition. The {@link WorkflowDefinition} is responsible for
25  * creating {@link WorkflowElement} instances and hold them through
26  * out the life cycle of a workflow definition. A single workflow
27  * element holds all information to find, allocate and invoke a
28  * service component. Furthermore it has information about what its
29  * position is in the overall workflow.
30  *
31  * @author Dirk Breuer
32  * @version 1.0 Jul 11, 2008
33  */
34 public class WorkflowElement {
35
36   /**
37    * A simple inner class to represent a successor of the current
38    * element. The Successor class just wraps the URI of the successing
39    * element to have a better semantic.
40    */
41   class Successor {
42
43     /**
44      * uri The URI of the successor. A URI of the same successor can
45      * never change during runtime.
46      */
47     private final String uri;
48
```

```
49      /**
50       * The default constructor which sets the URI of the successor.
51       *
52       * @param uri
53       */
54      public Successor(String uri) {
55        this.uri = uri;
56      }
57
58      /**
59       * @return The URI of this successor.
60       */
61      public String getUri() {
62        return this.uri;
63      }
64    }
65
66    /**
67     * A simple wrapper class describing the input of an element. The
68     * input for an element be have one out of two characteristics:
69     *
70     * <ul>
71     *   <li>
72     *     The input is internal. Due to this the URI is a reference
73     *     to some internal media storage system (like a media broker)
74     *   </li>
75     *   <li>
76     *     The input is external. In this case the URI points to file
77     *     or something else which is not stored in the system so far.
78     *     This type of input is only acceptable for "producer" components.
79     *   </li>
80     * </ul>
81     */
82    public class Input {
83
84      private final String uri;
85      private final String data;
86
87      public Input(String uri, String data) {
88        this.uri = uri;
89        this.data = data;
90      }
91
92      public String getUri() {
93        return this.uri;
94      }
95
96      public String getData() {
97        return this.data;
98      }
99
100     /**
101      * TODO: This should be more explicit!
102      *
103      * @return If the input is external or not
104      */
105     public boolean isExternal() {
```

```
106       return (data != null);
107     }
108
109     /**
110      * TODO: This should be more explicit!
111      *
112      * @return If the input is internal or not
113      */
114     public boolean isInternal() {
115       return (data == null);
116     }
117   }
118
119   private String type;
120   private String uri;
121   private String description;
122   private String className;
123   private String namespace;
124   private Class<AbstractComponent> elementClass;
125   private List<Input> input;
126   private List<Map<String, String>> output;
127   private List<String> predecessors;
128   private List<Successor> successors;
129
130   public String getType() {
131     return this.type;
132   }
133
134   public void setType(String type) {
135     this.type = type;
136   }
137
138   public String getUri() {
139     return this.uri;
140   }
141
142   public void setUri(String uri) {
143     this.uri = uri;
144   }
145
146   public String getClassName() {
147     return this.className;
148   }
149
150   public void setClassName(String className) {
151     this.className = className;
152   }
153
154   public String[] getInputKeys() {
155     String[] inputKeys = new String[this.input.size()];
156     int i = 0;
157     for (Input input : this.input) {
158       inputKeys[i] = input.getUri();
159       i++;
160     }
161     return inputKeys;
162   }
```

```
163
164   public List<Input> getInput() {
165     return this.input;
166   }
167
168   public void setInput(List<Map<String, String>> rawInput) {
169     this.input = new ArrayList<Input>();
170     Input input;
171
172     for (Map<String, String> singleRawInput : rawInput) {
173       if (singleRawInput.get("type").equalsIgnoreCase("external")) {
174         input = new Input(getUri() + "/input", singleRawInput.get("uri"));
175       } else {
176         input = new Input(singleRawInput.get("uri"), null);
177       }
178       this.input.add(input);
179     }
180   }
181
182   public List<Map<String, String>> getOutput() {
183     return this.output;
184   }
185
186   public void setOutput(List<Map<String, String>> output) {
187     this.output = output;
188   }
189
190   public List<String> getPredecessors() {
191     return this.predecessors;
192   }
193
194   public void setPredecessors(List<String> predecessors) {
195     this.predecessors = predecessors;
196   }
197
198   public List<Successor> getSuccessors() {
199     return this.successors;
200   }
201
202   public boolean hasSuccessors() {
203     if (this.successors != null && !this.successors.isEmpty()) {
204       return true;
205     } else {
206       return false;
207     }
208   }
209
210   public void setSuccessors(List<HashMap<String, String>> successors) {
211     this.successors = new ArrayList<Successor>();
212     for (Iterator<HashMap<String, String>> iterator = successors.iterator(); iterator
213         .hasNext();) {
214       Successor successor = new Successor(iterator.next().get("uri"));
215       this.successors.add(successor);
216     }
217   }
218
219   /**
```

```
220     * @return The AbstractComponent subclass representing this WorkflowElement instance.
221     * @throws ClassNotFoundException If the class could be found.
222     */
223    @SuppressWarnings("unchecked")
224    @Deprecated
225    public Class<AbstractComponent> getElementClass()
226                                                    throws ClassNotFoundException {
227      if (this.elementClass != null)
228        return this.elementClass;
229
230      try {
231        this.elementClass =
232            (Class<AbstractComponent>) Class.forName(getClassName());
233      } catch (ClassNotFoundException e) {
234        throw new ClassNotFoundException("Class " + getClassName()
235            + " could not be found.");
236      }
237      return this.elementClass;
238    }
239
240    /**
241     * @return Either this element needs input or not.
242     */
243    public boolean needsInput() {
244      return !input.isEmpty();
245    }
246
247    /**
248     * TODO: This is not so nice. Every element can only have one output
249     * port anyway. So we can either - store it into the workflow
250     * definition file or - make this implicit
251     *
252     * @return The URI of the output of this element
253     */
254    public String getOutputUri() {
255      return output.get(0).get("uri");
256    }
257
258    public void setNamespace(String namespace) {
259      this.namespace = namespace;
260    }
261
262    public String getNamespace() {
263      return namespace;
264    }
265
266    public void setDescription(String description) {
267      this.description = description;
268    }
269
270    public String getDescription() {
271      return description;
272    }
273  }
```

Listing A.2: Die `RemoteWorkflowEngine`-Klasse

```
12 package de.fhkoeln.cosima.workflow;
13
14 import java.util.Iterator;
15 import java.util.Set;
16
17 import javax.xml.namespace.QName;
18
19 import org.apache.axis2.AxisFault;
20 import org.apache.axis2.addressing.EndpointReference;
21 import org.apache.axis2.client.Options;
22 import org.apache.axis2.rpc.client.RPCServiceClient;
23
24 import de.fhkoeln.cosima.services.CoreService;
25 import de.fhkoeln.cosima.services.IODescriptor;
26 import de.fhkoeln.cosima.services.OutputDescriptor;
27 import de.fhkoeln.cosima.services.registry.ServiceRegistry;
28 import de.fhkoeln.cosima.workflow.WorkflowElement.Input;
29 import de.fhkoeln.cosima.workflow.storage.ProcessStore;
30
31 /**
32  * A {@link RemoteWorkflowEngine} instance runs a workflow specified by its
33  * {@link WorkflowDefinition}. The single workflow elements are web
34  * services which have to be invoked remotely. Their URI is specified
35  * in the WorkflowDefinition. Every runner instance needs, in addition
36  * to a WorkflowDefinition reference, also a reference to a
37  * ProcessStore. In a ProcessStore the workflow can store the return
38  * values of a component, so it can be assinged later to another
39  * component.
40  *
41  * @author Dirk Breuer
42  * @version 1.0 Sep 25, 2008
43  */
44 public class RemoteWorkflowEngine extends WorkflowEngine {
45
46   /**
47    * The ProcessStore reference of this runner instance.
48    */
49   private ProcessStore processStore;
50   /**
51    * The Service Registry instance to query for service URIs.
52    */
53   private ServiceRegistry registry;
54
55   /**
56    * This is the main method of every enigne. After successful
57    * initialization of the runner instance the run methods starts the
58    * process. Every element will be invoked as described in the
59    * definition object. While working through the definition elements
60    * certain the following steps are performed by the run method:
61    *
62    * <ul>
63    *   <li>It is asked if the element needs any input (in most of
64    *       the cases this is true)</li>
65    *       <ul>
66    *         <li>If yes and the input is external add the reference
```

```
67    *             to the external content IODescriptor instance for
68    *             the current element.</li>
69    *         <li>If yes and the input is internal get the reference
70    *             from the {@link ProcessStore} reference and add it
71    *             to the IODescriptor instance.</li>
72    *         <li>If no, which means, there is no more input required
73    *             for this component, proceed with execution</li>
74    *       </ul>
75    *       <li>Setup the WS client.</li>
76    *       <li>Set the IODescriptor instance at the remote service.</li>
77    *       <li>Call the <code>execute()</code> method of the service.</li>
78    *       <li>Retrieve the result which is an {@link IODescriptor}</li>
79    *       <li>If the IODescriptor contains any elements, they represent
80    *           the output of the service. Store the reference of the output
81    *           in the ProcessStore.</li>
82    *       <li>Proceed to the next element in the workflow definition</li>
83    * </ul>
84    *
85    * (Further information is provided in the 'WorkflowEngine_Flowchart.graffle' Document.
86    */
87   public void execute() {
88
89     // iterate through the workflow definition elements
90     Iterator<Set<WorkflowElement>> elementsIterator = getWorkflowDefinition().elementsIterator();
91
92     while (elementsIterator.hasNext()) {
93        for (WorkflowElement element : elementsIterator.next()) {
94         // define a default inputDescriptor
95         IODescriptor inputDescriptor = new IODescriptor();
96
97         if (element.needsInput()) {
98
99           for (Input elementInput : element.getInput()) {
100            if (elementInput.isExternal()) {
101              inputDescriptor.add(elementInput.getData());
102            } else if (elementInput.isInternal()) {
103              inputDescriptor.add(getStorage().getInputByKey(
104                  elementInput.getUri()));
105            }
106          }
107
108        }
109
110        IODescriptor outputDescriptor = invokeServiceForElementWithInputDescriptor(element, inputDescriptor);
111
112        if (!outputDescriptor.isEmpty()) {
113          getStorage().putDataForKey(outputDescriptor.first(), element.getOutputUri());
114        }
115      }
116    }
117
118    System.out.println("-> Workflow successfully executed!");
119  }
120
121  /**
122   * @return The store which holds temporary information of the process
123   */
```

```
124   public ProcessStore getStorage() {
125     return processStore;
126   }
127
128   /**
129    * @param processStore
130    *          The ProcessStore instance which should be used during
131    *          the runtime of this workflow.
132    */
133   public void setProcessStore(ProcessStore processStore) {
134     this.processStore = processStore;
135   }
136
137   /**
138    * Sets the concrete Service Registry implementation.
139    *
140    * @param registry A service registry implementation.
141    */
142   public void setRegistry(ServiceRegistry registry) {
143     this.registry = registry;
144   }
145
146   /**
147    * This method encapsulated the call of the remote service. So far it only can handle SOAP services.
148    * All information of connecting to the service component are encapsulated in the workflow element.
149    *
150    * FIXME: If we receive a timeout from the remote service we do not get the {@link IODescriptor} instance
151    * we were expecting. Due to this follow up operations can fail. This should be handled in a reasonable way.
152    * At the moment we just set a higher timeout value to handle this effects.
153    *
154    * @param element
155    *          The WorkflowElement of which we call its service
156    * @param inputDescriptor
157    *          The InputDescriptor for the service
158    * @return An OutputDescriptor which holds the result of the service
159    *         invocation.
160    */
161   @SuppressWarnings("unchecked")
162   private IODescriptor invokeServiceForElementWithInputDescriptor(WorkflowElement element,
163                                                                   IODescriptor inputDescriptor) {
164
165     IODescriptor output = new OutputDescriptor();
166
167     try {
168       RPCServiceClient client = new RPCServiceClient();
169
170       Options options = client.getOptions();
171
172       String serviceUri = registry.query(element.getDescription());
173       Logger.info("Setting EPR to: " + serviceUri);
174       EndpointReference targetEPR = new EndpointReference(serviceUri);
175       options.setTo(targetEPR);
176
177       // We want to set the timeout a bit higher to handle long running remote services.
178       options.setTimeOutInMilliSeconds(500000);
179
180       // Setting input arguments first
```

```
181       QName setInputOperation = new QName(element.getNamespace(), CoreService.SERVICE_SET_INPUT_OPERATION);
182
183       Object[] setInputOperationArgs = new Object[] { inputDescriptor };
184
185       client.invokeRobust(setInputOperation, setInputOperationArgs);
186
187       // Run the service action and get back the output value
188       QName executeOperation = new QName(element.getNamespace(), CoreService.SERVICE_EXECUTE_OPERATION);
189
190       Object[] executeOperationArgs = new Object[] {};
191       Class[] returnTypes = new Class[] { IODescriptor.class };
192
193       Object[] response = client.invokeBlocking(executeOperation, executeOperationArgs, returnTypes);
194
195       output = (IODescriptor) response[0];
196     } catch (AxisFault e) {
197       e.printStackTrace();
198     }
199
200     return output;
201   }
202 }
```

Listing A.3: Die vollständige `AbstractComponent`-Klasse

```
12 package de.fhkoeln.cosima.components;
13
14 import de.fhkoeln.cosima.media.MediaComponent;
15 import de.fhkoeln.cosima.media.mediabroker.MediaBroker;
16 import de.fhkoeln.cosima.services.CoreService;
17 import de.fhkoeln.cosima.services.IODescriptor;
18 import de.fhkoeln.cosima.services.registry.ServiceRegistry;
19 import de.fhkoeln.cosima.util.Logger;
20
21 /**
22  * This is the parent class of all components which are used to transform,
23  * produce or consume {@link MediaComponent} instances and their data. The class
24  * implements the {@link CoreService} interface to enable itself as a service
25  * within the application.
26  *
27  * @author Dirk Breuer
28  * @version 1.0 Nov 24, 2008
29  */
30 public abstract class AbstractComponent implements CoreService {
31
32   private MediaBroker broker;
33   private IODescriptor input;
34
35   private final ServiceRegistry registry;
36   private final String description;
37   private final String uri;
38
39   /**
40    * The constructor which should be used to instantiate any component. It
41    * ensures the publication of the service's relevant information to the given
42    * {@link ServiceRegistry} instance.
43    *
44    * @param registry The {@link ServiceRegistry} instance to publish the information.
45    * @param uri The URI of this service.
46    * @param description The description of this service.
47    */
48   public AbstractComponent(ServiceRegistry registry, String uri,
49       String description) {
50     Logger.info("Booting Service: " + getClass().getName());
51     this.registry = registry;
52     this.uri = uri;
53     this.description = description;
54
55     this.registry.publish(this);
56   }
57
58   /*
59    * TODO: This method should be final. Due to issues with Axis2 at the time of coding
60    * it was not able that this method was cleanly inherited and so could be
61    * final.
62    */
63   public IODescriptor execute() {
64     IODescriptor output = _execute();
65     return output;
66   }
```

```
67
68    protected abstract IODescriptor _execute();
69
70    public void setInput(IODescriptor descriptor) {
71      this.input = descriptor;
72    }
73
74    public IODescriptor getInput() {
75      return this.input;
76    }
77
78    public MediaBroker getBroker() {
79      return this.broker;
80    }
81
82    public void setBroker(MediaBroker broker) {
83      this.broker = broker;
84    }
85
86    public String getDescription() {
87      return this.description;
88    }
89
90    public String getUri() {
91      return this.uri;
92    }
93  }
```

B Inhalt der Begleit-CD

Auf der Begleit-CD findet sich der gesamte Quellcode des COSIMA-Projekts, der Santiago sowie Nerstrand Anwendung. Zusätzlich finden sich auf der CD alle verwendeten Grafiken, sowohl im PDF- als auch dem Originalformat. Ebenso sind das Maven 2 Repository mit allen verwendeten Bibliotheken und die notwendigen Programme zur Ausführung der medienverarbeitenden Komponenten in Mac OS X Version beigelegt.

Die Verzeichnisstruktur ist im Folgenden abgebildet:

Listing B.1: Verzeichnisstruktur der Begleit-CD

```
Begleit-CD
|-- Abhaengigkeiten
|   |-- Maven 2 Repository
|   |-- QuickTime Player.app
|   |-- VLC.app
|   |-- apache-tomcat-6.0.18
|   |-- axis2.war
|   |-- ffmpeg
|   |-- ffmpegX.app
|   |-- mencoder
|   `-- mplayer
|-- Arbeit
|   `-- Master Thesis.pdf
|-- Grafiken
`-- Quellcode
    |-- COSIMA
    |-- README.html
    |-- README.mdown
    |-- Santiago
    `-- nerstrand
```

Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben.

Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsgbehörde vorgelegen.

Dirk Breuer — Köln, den 20. Januar 2009

Made on a Mac. Typesetted with LaTeX.