Generisches Kompositionsmodell für UI-Mashups

Großer Beleg
Technische Universität Dresden
Oktober 2009

Jan Reimann

Betreuer: Dipl.-Medieninf. Stefan Pietschmann
Dipl.-Medieninf. Vincent Tietz
Hochschullehrer: Prof. Dr.-Ing. Klaus Meißner

Fakultät Informatik
Institut für Software- und Multimediatechnik
Professur für Multimediatechnik

Erklärung

Hiermit erkläre ich, Jan Reimann, den vorliegenden Großen Beleg zum Thema

Generisches Kompositionsmodell für UI-Mashups

selbstständig und ausschließlich unter Verwendung der im Quellenverzeichnis aufgeführten Literatur- und sonstigen Informationsquellen verfasst zu haben.

Dresden, 31. Oktober 2009

Unterschrift

Aufgabenstellung

Diese Seite wird beim Druck durch die originale Aufgabenstellung ersetzt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Abkürzungsverzeichnis

API	Application Programming Interface
ATL	Atlas Transformation Language
BPEL	Business Process Execution Language
CC	Context Component
CCM	CRUISe Composition Model
CDO	Connected Data Objects
CIM	Computation Independent Model
CMM	CRUISe Metamodel
CRUISe	Composition of Rich User Interface Services
DSL	Domain Specific Language
EMF	Eclipse Modeling Framework
GEF	Graphical Editing Framework
IDE	Integrated Development Environment
LC	Logic Component
M2C	Model to Code
M2M	Model to Model
MDA	Model Driven Architecture
MDL	MashArt Description Language
MOF	Meta Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
OO-H	Object-Oriented Hypermedia
OOWS	Object Oriented Web Solution
PIM	Platform Independent Model
PIV	Platform Independent Viewpoint
PSM	Platform Specific Model
PSV	Platform Specific Viewpoint
QVT	Query View Transformation
QVTO	QVT Operational

REST	REpresentational State Transfer
RIA	Rich Internet Application
SaaS	Software-as-a-Service
SAC	Service Access Component
SOA	Service-orientierte Architektur
SoC	Separation of Concerns
UCL	Universal Composition Language
UCM	Universal Composition Model
UI	User Interface
UIC	User Interface Component
UIS	User Interface Service
UISDL	User Interface Service Description Language
UML	Unified Modeling Language
URI	Uniform Resource Identifier
UWE	UML-based Web Engineering
W3C	World Wide Web Consortium
WADL	Web Application Description Language
WebML	Web Modeling Language
WSDL	Web Services Description Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XSD	XML Schema Definition

1 Einleitung

1.1 Motivation

Ein Trend der Software-Entwicklung ist die Nutzung serviceorientierter Architekturen (SOA) [@Neu09]. Dabei werden Systeme strukturiert und bestimmte Funktionalitäten als Service zur Verfügung gestellt. SOA bietet eine einfache, skalierbare Möglichkeit, Netzwerke unterschiedlicher Systeme zu organisieren, die miteinander interoperieren [@MLM+06]. Dieser Ansatz bringt den Vorteil der Flexibilität und der losen Kopplung.

Das SOA-Paradigma wird mittlerweile auch bei der Entwicklung von Web-Anwendungen genutzt. Sogenannte *Mashups* sind Web-Applikationen, die Daten und Funktionalität aus verschiedenen Quellen außerhalb ihrer Organisationsstruktur abrufen und zu neuen innovativen Diensten kombinieren [@Mer06]. Ein Beispiel dafür ist, Google Maps mit aktuellen Flugdaten zu koppeln, wie es die Zürcher Hochschule für Angewandte Wissenschaften in einer Diplomarbeit entwickelt hat[1].

Im Forschungsprojekt CRUISe[2] wird der serviceorientierte Ansatz auf die Präsentationsschicht von Mashups erweitert. Visuelle Komponenten der Benutzerschnittstelle (UI) werden als User Interface Services (UISs) bereitgestellt. Deshalb kann man von *UI-Mashups* sprechen. Die Serviceorientierung in CRUISe bringt die Vorteile der Wiederverwendbarkeit und Anpassbarkeit der UI-Komponenten sowie Technologieunabhängigkeit während des Entwurfs eines UI-Mashups [PVM09].

Um Services zu orchestrieren, bedarf es einer Kompositionssprache. Im Bereich von Geschäftsprozessen gibt es die Business Process Execution Language (BPEL), mit der es möglich ist, Abläufe zu beschreiben, die aus Webservices bestehen. Doch wie kann dies im Bereich Mashups realisiert werden?

Für die UI-Mashups von CRUISe gibt es kein Kompositionsmodell. Dieser Umstand bringt mehrere Nachteile mit sich. Zum einen muss der Autor die Beschreibung des UI-Mashups per Hand anfertigen, so dass er die verfügbaren Dienste und deren Schnittstellen kennen muss sowie frühzeitig an die zu verwendende Technologie gebunden ist. Zum anderen sind viele Vorteile, die durch eine Model Driven Architecture (MDA) geboten werden, nicht nutzbar. Es ist nicht möglich, die Aspekte Technologie und Funktionialität zu trennen sowie schon entworfene Modelle wiederzuverwenden. Außerdem bietet MDA aufgrund der höheren Abstraktion und der Trennung von Verantwortlichkeiten (SoC) bessere Wartbarkeit eines Systems; durch das Fehlen eines Kompositionsmodells ist dies nicht möglich, denn bei der kleinsten Änderung von Anforderungen müsste das UI-Mashup neu bereitgestellt werden. Speziell bezogen auf die UI einer Web-Applikation, ist ein modellbasierter Entwurf zudem systematischer und einfacher als mit traditionellen UI-Entwicklungswerkzeugen [LW07].

[1] http://radar.zhaw.ch
[2] http://www-mmt.inf.tu-dresden.de/CRUISe

Auch vorhandene Kompositionssprachen sind eingeschränkt. So ist es beispielsweise mit der *eXtensible user Interface Component Language* möglich UIs und visuelle Komponenten auf Runtime-Level zu beschreiben [LdS04], wodurch es jedoch nicht möglich ist, Komponenten auszutauschen, ohne ein neues Deployment der Web-Anwendung durchzuführen.

1.2 Zielstellung

Ziel dieser Arbeit ist die Konzeption eines generischen Kompositionsmodells, das einen MDA-Ansatz für UI-Mashups ermöglicht. Dazu gehört, typische Eigenschaften von UIs spezifizieren zu können. Zu nennen sind die Kopplung von UI zu Datendiensten, Layout, Konfiguration und Datenfluss von UI-Komponenten sowie der Kontrollfluss des gesamten UI-Mashups.

Um dieses Ziel erreichen zu können, ist es zunächst erforderlich, ein grundlegendes Verständnis für Kompositionsmodelle sowie für MDA und den damit verbundenen Möglichkeiten zu schaffen. Dafür ist es nötig, existierende Kompositionsmodelle der Forschung und Technik bezüglich der oben genannten typischen UI-Eigenschaften zu untersuchen. Des Weiteren muss analysiert werden, inwiefern existierende Modelle einen Abstraktionsgrad aufweisen, um technologieunabhängig UISs komponieren zu können. Aus der Analyse ergeben sich Vor- und Nachteile der Modelle. Diese werden genutzt, um wichtige Erkenntnisse für diese Arbeit zu gewinnen. Dazu zählt auch die Untersuchung, ob das hier entwickelte Kompositionsmodell in existierende integriert werden kann, wenn diese erweiterbar sind und sonst auch den Anforderungen entsprechen.

Das entwickelte Kompositionsmodell wird anschließend mit einem geeigneten Modellierungs-Framework prototypisch umgesetzt. Im Sinne des MDA-Ansatzes werden außerdem mögliche Modelltransformationen implementiert. Abschließend werden Kompositionsmodell und Transformationen an einer Beispielkomposition eines CRUISe-Anwendungsfalls validiert.

1.3 Aufbau der Arbeit

Die Arbeit wird wie folgt gegliedert. Das Kapitel 2 legt die Grundlagen. Es erfolgen Definitionen und Erläuterungen zur modellgetriebenen Software-Entwicklung. Es wird auf CRUISe und die konkrete Rolle des Kompositionsmodells darin eingegangen. Abschließed werden darauf aufbauend die Anforderungen, die es zu erfüllen gilt, aufgestellt. Im nachfolgenden Kapitel 3 werden bestehende Ansätze analysiert sowie ihre Vor- und Nachteile aufgezeigt. Hierbei werden aktuelle Projekte hinsichtlich der in 1.2 erläuterten Ziele untersucht. Aufbauend auf die an die Beschreibungssprache gestellten Anforderungen, wird diese in Kapitel 4 entworfen. Zusätzlich werden Hinweise für angrenzende Arbeiten gegeben, wie zum Beispiel für einen grafischen Editor oder den in CRUISe enthaltenen *Application Generator*. Um das entwickelte Kompositionsmodell zu validieren, werden die gewonnenen Erkenntnisse im Kapitel 5 in einem Prototyp umgesetzt und Designentscheidungen erläutert. Abschließend enthält Kapitel 6 eine Zusammenfassung der Arbeit und bewertet die Ergebnisse.

2 Begriffsklärung und Anforderungsanalyse

In diesem Kapitel wird zunächst der essentielle Begriff *UI-Mashup* definiert. Dies ist notwending, da in der Literatur und auf seriösen Internetseiten noch keine aussagekräftige Definition existiert. Dazu werden Erklärungen des Begriffes *Mashup* herangezogen und, darauf aufbauend, wird eine Definition hergeleitet. Danach wird erläutert, was ein Kompositionsmodell ausmacht und welche Rolle es im CRUISe-Projekt einnimmt. Zu diesem Zweck werden die CRUISe-Architektur sowie der Integrationsprozess der UISs bis hin zum fertigen UI-Mashup vorgestellt. Da es sich um einen modellgetriebenen Ansatz handelt, wird in Kapitel 2.5 erklärt, was darunter zu verstehen ist. Dafür wird zuvor in Abschnitt 2.4 gezeigt, in welcher Metamodell-Ebene das Kompositionsmodell anzusiedeln ist. Eine Anforderungsanalyse des zu konzipierenden Kompositionsmodells schließt dieses Kapitel ab. Sie beinhaltet funktionale und nichtfunktionale Anforderungen an die Qualität des Kompositionsmodells. Diese dienen im Kapitel 3 dazu, vorhandene Ansätze zu bewerten.

2.1 UI-Mashup

Mashups sind besondere Arten von Web-Applikationen. Duane Merrill beschreibt diese in [@Mer06] als Web-Applikationen, die Daten und Funktionalitäten aus verschiedenen, außerhalb ihrer Organisationsstruktur liegenden Quellen abrufen und kombinieren, um neue innovative Services zu erschaffen. Fischer et al. bilden in [FBN08] darauf aufbauend die folgende Definition:

> *»A mashup can be defined as a situational Web application that extracts and combines data from different sources to support special user needs and tasks.«*

In den Grundzügen gehen Fischer et al. mit der Beschreibung von Merrill konform, erweitern den Begriff zusätzlich noch um den Aspekt der Situationsabhängigkeit (*situational).* In [FBN08] führen sie an, dass es mehr und mehr Frameworks gibt, die es dem Nutzer ermöglichen, manuell Mashups zu erstellen. Dies bedeutet, dass Mashups in Abhängigkeit von den Anforderungen des Nutzers in verschiedenen Situationen anders gestaltet sein können, sie also einen kurzlebigen Charakter haben können.

In [ZRN08] geben Zang et al. folgende Definition an:

> *»[. . .]mashups - software applications that merge separate APIs and data sources into one integrated interface.«*

Nach Wong und Hong in [WH07] heißt es:

> *»[...]mashups [...] combine existing webbased content and services to create new applications.«*

Allen Definitionen gemein ist, dass Mashups externe Daten oder Services kombinieren und unter einer Schnittstelle dem Benutzer zur Verfügung stellen.

In dieser Arbeit handelt es sich jedoch um den Entwurf eines Kompositionsmodells für Web-Anwendungen, die Komponenten der Benutzerschnittstelle mittels UI-Services erhalten. Somit wird, ausgehend von den oben genannten Definitionen, der Begriff **UI-Mashup** wie folgt definiert:

> *Ein UI-Mashup ist eine Web-Applikation, die ihre UI-Komponenten über externe UI-Services konsumiert. UI-Services werden auf Präsentations-Ebene komponiert und UI-Komponenten können an Backend-Dienste, die zu repräsentierende Daten oder Funktionalität liefern, gekoppelt werden. Durch die Kombination von UI-Services entstehen neue Web-Anwendungen mit integrierten, servicebasierten Benutzerschnittstellen.*

Aus dieser Definition folgt, dass UI-Komponenten von unterschiedlichen Service-Anbietern bereitgestellt werden können. Der Autor des UI-Mashups muss sich nicht um die Implementierung der einzelnen UI-Komponenten, sondern einzig um die Komposition der Benutzerschnittstelle kümmern. Das bedeutet auch, dass sich der Autorenprozess in die UI-Komposition und die Komposition der Backend-Dienste unterteilt. Da die Backend-Dienste autonom orchestriert werden können, beispielsweise mit Mashup-Diensten anderer Anbieter, sollen UI-Mashups nur auf die UI-Komposition beschränkt sein. Das heißt, dass zwar eine Bindung an Backend-Dienste stattfinden kann, diese jedoch nicht innerhalb der UI-Mashups orchestriert werden. Einer Beschreibung der UI-Komposition liegt ein Modell zugrunde. Der Begriff des *Kompositionsmodells* wird im Abschnitt 2.2 definiert und näher erläutert.

2.2 Kompositionsmodell

UI-Mashups sind, entsprechend der oben genannten Definition, aus UI-Komponenten zusammengesetzt, die durch UI-Services bereitgestellt wurden. Wie bei der Sprache BPEL, mit der man Kompositionen von Webservices für Geschäftsprozesse beschreiben kann, müssen UI-Services orchestriert werden können, um deren Interaktion und die Essenz des UI-Mashups beschreiben zu können. Damit die Orchestrierung maschinell verarbeitet werden kann, bedarf es einer Kompositionssprache für UI-Mashups. Die Begriffe **Kompositionssprache** und **Kompositionsmodell** werden in dieser Arbeit synonym benutzt, da sie in der Architektur der Meta Object Facility (MOF), einer Architektur zur Metamodellierung, auf derselben Ebene stehen und als gleich angesehen werden können [Fla02, S. 330]. Im Abschnitt 2.4 wird näher darauf eingegangen.

Aßmann definiert in [Aß03, S. 2] den Begriff der *Kompositionssprache* als eine Sprache, mit der man beschreiben kann, wie ein System aus Komponenten aufgebaut wird und welcher Architektur es zu Grunde liegt. Nierstrasz und Meijler beziehen sich eher auf den durch eine Kompositionssprache erreichten Mehrwert und wählten in [NM95a] eine abstraktere Definition:

> »*A composition language supports the technical requirements of a component-oriented development approach by shifting emphasis from programming and inheritance of classes to specification and composition of components.*«

Angelehnt an diese Definitionen, kann nun der Begriff **Kompositionsmodell für UI-Mashups** definiert werden, um eine gemeinsame Ausgangsposition zu schaffen:

> *Ein Kompositionsmodell für UI-Mashups ist ein Metamodell, das angibt, wie man die Komposition von UI-Services beschreiben kann.*

Nach dieser Definition ist ein Kompositionsmodell für UI-Mashups ein Modell, um andere Modelle zu beschreiben. Diese stellen Instanzen des Kompositionsmodells dar. Aus Gründen einer rationelleren Darstellungsweise sei in dieser Arbeit jedoch von einem Metamodell ausgegangen, das im Sinne der Aufgabenstellung das Kompositionsmodell umfasst.

Eine Komposition von UI-Services umfasst insbesondere die Spezifikation der Kommunikation zwischen bereitgestellten UI-Komponenten. Außerdem muss beschrieben sein, an welche Backend-Dienste (vgl. 2.1) die UI-Komponenten gekoppelt werden. Diese und weitere Anforderungen werden in Kapitel 2.6 im Detail erläutert. Im nächsten Abschnitt wird dargestellt, welche Rolle ein Kompostitionsmodell im CRUISe-Projekt einnimmt.

2.3 Einordnung in das Forschungsprojekt CRUISe

Es ist zu beobachten, dass Anwendungsentwicklung vom Desktop ins Web übergeht [PVM09]. Infolgedessen werden immer mehr Applikationen als Software-as-a-Service (SaaS) über das Internet zur Verfügung gestellt. In CRUISe geht man noch einen Schritt weiter und argumentiert, dass Web-basierte Anwendungen in Zukunft gänzlich aus Services bestehen. Das heißt, dass Daten, Geschäftslogik und auch Benutzerschnittstellen als Service bereitgestellt werden. Die zentrale Idee von CRUISe ist, das SOA-Paradigma auf die Präsentationsschicht zu übertragen, um die Entwicklung kontextsensitiver, serviceorientierter Web-Applikationen zu vereinfachen. Durch die Serviceorientierung der UI werden Wiederverwendung und Technologieunabhängigkeit Kernkonzepte des Web-Engineerings [PVRM09]. Existierende Ansätze unterstützen derzeit diesen universellen Ansatz nicht, wodurch Entwickler mit vielen Programmiersprachen, Frameworks und Technologien konfrontiert werden [PVM09].

In CRUISe werden die Elemente der Benutzerschnittstelle **User Interface Components (UICs)** genannt. Diese kapseln Komponenten der Benutzerschnittstelle und machen diese über eine Schnittstelle zugänglich. Ausgeliefert werden sie als JavaScript, das alle notwendigen Informationen zur Instanziierung enthält und gegebenenfalls noch benötigte Ressourcen nachlädt. UICs können alle erdenklichen Technologien enthalten, die für die visuellen Komponenten genutzt werden. Diese werden in CRUISe über **UISs** verfügbar gemacht, wodurch Unabhängigkeit der verwendeten Technologien erreicht wird. UISs bieten eine Schnittstelle, um die dahinter gekapselten UICs zu konfigurieren. Über diese Schnittstelle werden die UICs auch zur Verfügung gestellt. Wie bei konventionellen Webservices ist ein UIS über eine eindeutige Adresse ansprechbar. Doch UICs, die hinter UISs verborgen sind, müssen in die UI integriert werden, um ein UI-Mashup zu erhalten. Dafür wurde in

CRUISe eine Architektur entworfen, mit der UI-Mashups entwickelt werden können. In Abbildung 2.1 ist eine Übersicht der CRUISe-Architektur dargestellt.

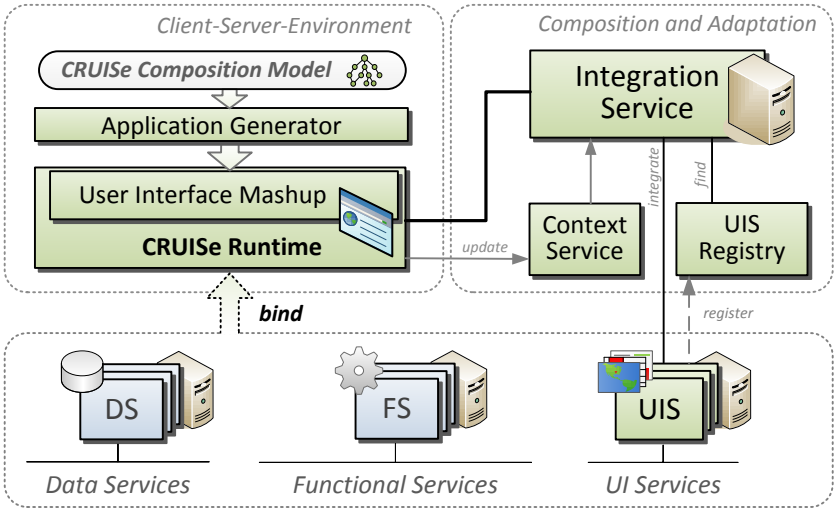


Abbildung 2.1: CRUISe-Architektur [PVRM09]

Die Infrastruktur besteht aus drei Teilen: *Client-Server-Environment*, *Composition and Adaptation* und der Serviceschicht. Anhand der Abbildungen 2.1 und 2.2 wird der Integrations- und Kompositionsprozess erläutert.

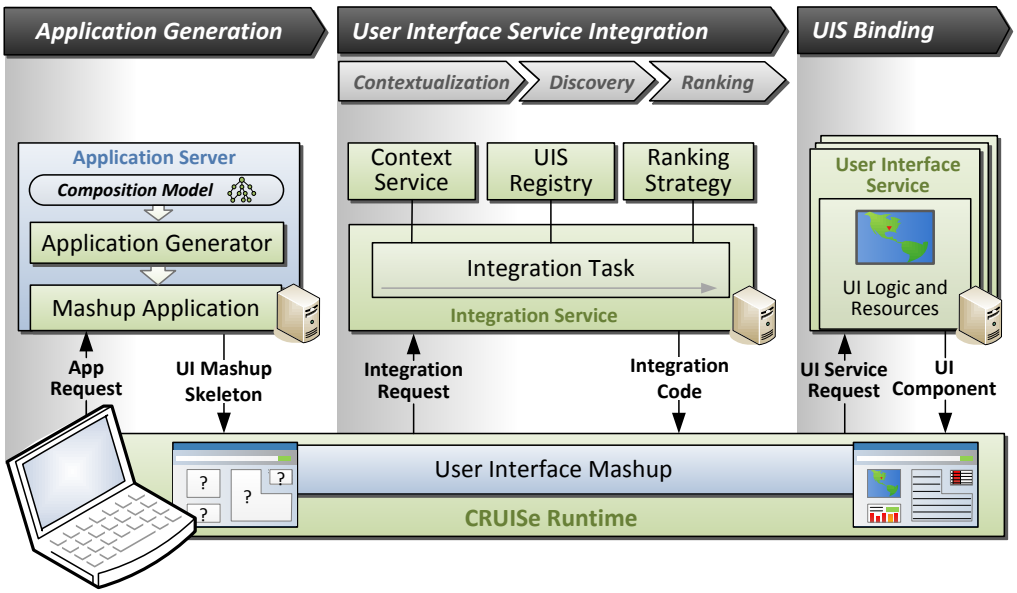


Abbildung 2.2: Ablaufsteuerung in CRUISe [PVRM09]

Im *Client-Server-Environment* befindet sich ein Nutzer (Client), der ein UI-Mashup anfordert und dadurch einen Request an den Server stellt. Auf dem Server befindet sich das CRUISe Composition Model (CCM), in dem die Komposition der UISs beschrieben ist. Das CCM ist eine Instanz des in dieser Arbeit zu entwerfenden

Kompositionsmodells, welches im CRUISe-Projekt CRUISe Metamodel (CMM) genannt wird. Das CCM wird durch den Request vom *Application Generator* verarbeitet. Dieser erzeugt eine lauffähige Web-Applikation (*UI Skeleton*), die an die *CRUISe Runtime* übertragen wird. In diesem Stadium enthält die Web-Applikation Platzhalter, die noch gebunden werden müssen. Dafür sendet die *CRUISe Runtime* Anforderungen und Parameter an den *Integration Service*. Dieser ist verantwortlich für das Auffinden in Frage kommender UISs und kommuniziert deshalb mit der *UIS Registry*, welche Metadaten über die UISs enthält. Vom *Integration Service* wird für gefundene UISs, in Abhängigkeit von den zutreffenden Anforderungen und dem Kontext, eine Rangfolge ermittelt und der beste UIS ausgewählt. Die dahinter gekapselte UIC wird dann an die *CRUISe Runtime* zurückgesendet. Dort werden jetzt die Platzhalter gebunden, und das UI-Mashup wird an den Client ausgeliefert. Die *CRUISe Runtime* führt außerdem den Event- und Datenfluss aus, so wie sie im CCM spezifiziert wurden.

Durch das beschriebene Vorgehen ist es in CRUISe möglich, zur Laufzeit Komponenten auszutauschen oder neu zu konfigurieren. Im Gegensatz zu anderen Ansätzen ist dadurch in CRUISe kein Redeployment des UI-Mashups nötig [PVM09]. Ein großer Vorteil, der dadurch erreicht wird, ist die Tatsache, dass der Autor zur Zeit des Entwurfs des UI-Mashups vollkommen unabhängig von Technologien und Frameworks ist. Dadurch kann man Modelle, die konkrete CCMs beschreiben, wiederverwenden, und es werden alle Vorteile einer modellgetriebenen Entwicklung wirksam. Auf MDA wird im Abschnitt 2.5 näher eingegangen.

2.4 Meta Object Facility

Wie schon in 2.2 erwähnt, ist die MOF eine Architektur zur Metamodellierung. Durch diese werden die Grundlagen für die Austauschbarkeit von Modellen gelegt. Die MOF ist seit 1997 ([SBPM08, S. 40]) ein Standard der Object Management Group (OMG) und liegt derzeit in Version 2.0 vor.

MOF besteht aus einem Kern, dessen Spezifikation in [OMG06a] zu finden ist. Der Ursprung dieser Spezifikation war, dass Datenaustausch zwischen verschiedenen Systemen von der Kompatibilität ihrer Metadaten abhängt. In vielen Systemen werden proprietäre Modelle der Metadaten genutzt, was den Datenaustausch über Systemgrenzen hinaus sehr behindert. MOF bietet dafür ein *Metadata Management Framework*, um die Entwicklung und Interoperabilität von modell- und Metadaten-getriebenen Systemen zu ermöglichen [OMG06a, S. 5]. MOF besteht aus einer Metamodell-Architektur, in welcher Elemente einer konzeptionellen Ebene Elemente der darunter liegenden beschreiben [OMG02]. In Abbildung 2.3 ist beispielhaft die MOF-Architektur dargestellt, angelehnt an eine Abbildung in [KK02].

Die erste Spalte in dieser Abbildung enthält den Namen der MOF-Ebene, die Art des beschriebenen Modells dieser Ebene sowie die Angabe der Sprache, die der Ebene entspricht [Fla02]. In der zweiten Spalte ist die Instanziierungshierarchie an einem Beispiel dargestellt. Die dritte Spalte beinhaltet ein Beispiel eines Modells der jeweiligen Ebene. Die Abbildung ist von oben nach unten zu lesen. So kann mit einer Metasprache ein Metamodell beschrieben werden. Ein Modell ist Instanz eines Metamodells. So siedelt sich das in dieser Arbeit zu entwickelnde Kompositionsmodell, das *CRUISe Metamodel*, in der M2-Ebene an. Mit einer Kompositionssprache kann

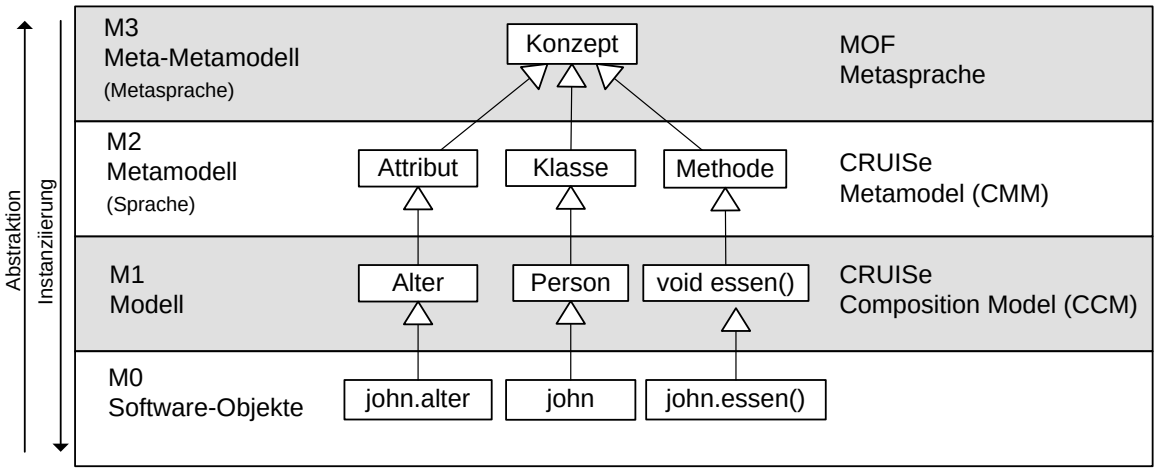


Abbildung 2.3: MOF-Architektur

man ein konkretes CCM eines UI-Mashups angeben. Wie schon in 2.2 erwähnt, liegt demzufolge auch die Kompositionssprache in M2 und kann synonym zum Kompositionsmodell betrachtet werden [Fla02].

XML Metadata Interchange

Wie oben schon erwähnt, trägt MOF zur Interoperabilität Metadaten-getriebener Systeme bei. Dafür wurde von der OMG zusätzlich ein Austauschformat entworfen, dessen ihm zugrunde liegendes Modell selbst eine MOF-Instanz ist [OMG07b, S. 8]. Dieses Format heißt XML Metadata Interchange (XMI) und gestattet den Austausch von serialisierten Daten, welche auf Metamodellen basieren und MOF-konform sind. Die aktuelle Version 2.1.1 ist ein Standard der OMG. XMI ist ein XML-Format, wodurch leichte Verarbeitung gesichert wird. Erzeugen IDEs Modelle, die sich mit MOF ausdrücken lassen, so ist es möglich, diese Modelle in IDEs weiterzuverarbeiten, die mit XMI umgehen können. An der Fähigkeit, mit dem XMI-Standard umzugehen, werden moderne IDEs untereinander gemessen.

2.5 Model Driven Architecture

In dieser Arbeit wird ein Kompositionsmodell entworfen, das einen modellgetriebenen Ansatz ermöglichen soll. Diese Art der Entwicklung eines UI-Mashups ist nach [FBN09] ein semi-automatisches Vorgehen. Das bedeutet, dass bei der Erzeugung der Komposition automatisch Modelle transformiert werden, aber der Autor diese auch manuell verfeinert. Durch Automatismen wird Unabhängigkeit bezüglich der zu verwendenden Technologie erreicht, und der Fokus kann auf die Entwicklung des UI-Mashups, anstatt auf die Implementierung des Glue-Codes der kombinierten UI-Services gelegt werden. Automatische Transformationen und Technologieunabhängigkeit sind Eigenschaften einer modellgetriebenen Entwicklung. Ein modellgetriebenes Vorgehen ist nach Abrams und Helms essentiell, um die Komplexität einer Benutzerschnittstelle handhaben und vollständig erfassen zu können [AH04]. Aus

diesen Gründen wird in diesem Abschnitt die MDA vorgestellt und ihr Nutzen erläutert.

2.5.1 Grundlagen

MDA hat als Ursprung die Idee, die Spezifikation der Operationen eines Systems von den Details der Plattform, auf der das System ausgeführt werden soll, zu trennen. MDA bietet also einen Ansatz, ein System unabhängig von der Plattform, die es unterstützt, zu spezifizieren. Die drei Hauptziele sind Portabilität, Interoperabilität und Wiederverwendbarkeit durch architektonische Separation of Concerns (SoC) [OMG03]. In der Spezifikation wird ein **Modell** wie folgt definiert:

> »*A model of a system is a description or specification of that system and its environment for some certain purpose. A model is often presented as a combination of drawings and text.*«

MDA ist ein Ansatz der Systementwicklung, der die Stärke von Modellen darin erhöht. Dieser Ansatz heißt **modellgetrieben**, weil dadurch Hilfsmittel bereitgestellt werden, mit denen der Verlauf des Verstehens, des Entwurfs, der Konstruktion, des Betriebs, der Wartung und der Modifikation gelenkt werden kann [OMG03].

Die **Plattform**, von der die Modellierung der Essenz eines Systems unabhängig sein soll, wird in [OMG03] wie folgt definiert:

> »*A platform is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented.*«

MDA beruht auf dem Setzen von unterschiedlichen **Gesichtspunkten**. Durch einen Gesichtspunkt wird der Fokus in einem System auf ein bestimmtes Interesse gesetzt, wodurch vom Gesamtsystem abstrahiert wird. In der MDA gibt es ursprünglich drei verschiedene Gesichtspunkte, denen jeweils entsprechende Modelle zugeordnet sind. Zu jedem Gesichtspunkt können mehrere Modelle gehören. Wie viele es sind, hängt vom zu modellierenden System und dem gewünschten Abstraktions- und Trennungsgrad ab. Welche Gesichtspunkte es gibt und welche Modelle sie mindestens enthalten, wird im Folgenden kurz erläutert.

Computation Independent Viewpoint: Dieser Gesichtspunkt visiert die Umgebung eines Systems und seine Anforderungen an. Dabei sind die Details der Struktur und Entwicklung unerheblich oder noch nicht bestimmt. Diesem Gesichtspunkt enstpricht das Computation Independent Model (CIM).

Platform Independent Viewpoint: Der Plattform-unabhängige Gesichtspunkt (PIV) steht für die Operation eines Systems, während Details einer bestimmten Plattform verborgen bleiben. Es wird der Teil einer Systemspezifikation gezeigt, der sich von Plattform zu Plattform nicht ändert. Zum PIV gehört das Plattformunabhängige Modell (PIM).

Platform Specific Viewpoint: Der Plattform-spezifische Gesichtspunkt (PSV) kombiniert den Plattform-unabhängigen mit einem zusätzlichen Fokus auf eine spezifische Plattform für das System. Zu diesem Gesichtspunkt gehört das Plattform-spezifische Modell (PSM).

2.5.2 Ablauf

Modelle unterschiedlicher Gesichtspunkte können von verschiedenen Modellierern entworfen werden. Beispielsweise muss der Autor des CIMs nichts über die Modelle oder Artefakte wissen, die benötigt werden, um Funktionalitäten, die in diesem Modell angegeben wurden, zu spezifizieren. Das erste Modell einer MDA hat meist den Charakter eines Domänenmodells oder eines Vokabulars für das System. Es spielt eine wichtige Rolle, da es die Lücke zwischen Experten der Domäne mit ihren Anforderungen einerseits und Experten des Entwurfs und der Konstruktion andererseits schließt.

Bei einer modellgetriebenen Vorgehensweise werden abstraktere Modelle in konkretere eines Systems konvertiert. Dieser Prozess wird **Transformation** genannt. In Abbildung 2.4 ist dargestellt, wie ein MDA-Prozess typischerweise verläuft.

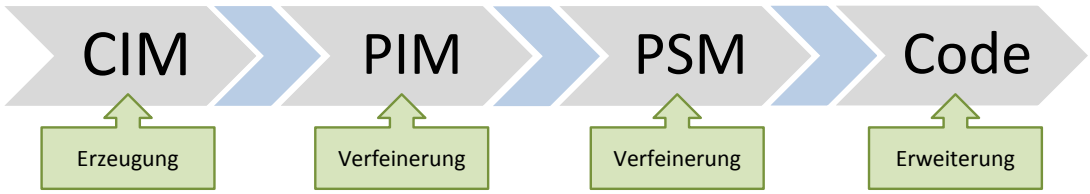


Abbildung 2.4: Transformationsprozess

Dabei entsprechen hellblaue Pfeile den Transformationen zwischen den Modellen. Grüne Rechtecke stehen für den Einfluss der Modellierer. Am Ende einer Transformationskette steht meistens die Generierung von Code für die Zielplattform des Systems. Ein wesentlicher Vorteil solcher Transformationsketten ist, dass Änderungen am System immer in den Modellen getätigt werden, wodurch der Code neu generiert wird. Dadurch bleibt der Code valide, sowie Code und Modelle laufen nicht auseinander.

2.6 Anforderungen

In diesem Abschnitt werden die Anforderungen an ein generisches Kompositionsmodell für UI-Mashups aufgestellt. Die Anforderungen werden unterteilt in spezifische Anforderungen an ein Kompositionsmodell und spezifische Anforderungen, die sich speziell für UI-Mashups ergeben. Zusätzlich wird im letzteren noch diskutiert, welche Anforderungen nötig sind, um ein *generisches* Kompositionsmodell zu erhalten. Diese müssen aufgeschlüsselt werden, da die Anforderungen sonst, bezogen auf CRUISe, zu allgemein ausfallen. Außerdem dienen sie zur Bewertung, existierender Ansätze in Kapitel 3. Abschließend werden nichtfunktionale Anforderungen, bezüglich der Qualität des Kompositionsmodells, aufgestellt.

2.6.1 Anforderungen an ein Kompositionsmodell

In [CWD00] diskutieren die Autoren Kompositionssprachen für den komponentenbasierten Entwurf von Software-Systemen. Damit zielen sie in erster Linie nicht auf Web-Applikationen ab, sondern auf konventionelle Anwendungssoftware und Produktlinien. Curbera et al. beziehen sich im Wesentlichen auf [Aß03, NM95a, NM95b]. Im Folgenden werden dort aufgeführte Anforderungen, die von Bedeutung für das hier zu konzipierende Kompositionsmodell sind, aufgegriffen und erweitert. Im Abschnitt 2.6.4 werden weitere dort genannte Anforderungen erläutert, die aber für diese Arbeit unerheblich sind.

1) Spezifikation von Kommunikationskanälen: Es muss möglich sein, mit einem Kompositionsmodell Kommunikationskanäle zwischen den zu integrierenden UICs zu spezifizieren. UICs reagieren bei eintretender Kommunikation auf Ereignisse. Jedoch ist es möglich, zwischen Ereignissen zu unterscheiden. Zum einen gibt es den Datenfluss, der sich durch das Senden von Daten von einer UI-Komponente zu anderen auszeichnet. Zum anderen gibt es den Kontrollfluss, durch den bestimmte Operationen von UICs initiiert werden können. Es muss aber hier nicht zwischen diesen beiden Arten differenziert werden, da sie sich nur darin unterscheiden, dass innerhalb vom Datenfluss zusätzlich zum Event noch Daten gesendet werden. Aus diesem Grund wird in dieser Arbeit nicht zwischen diesen beiden Arten unterschieden, sondern nur vom **Event-Fluss** gesprochen. Die Forderung, dass zwischen Komponenten Events kommuniziert werden müssen, führen auch die Autoren von [DYB+07] an. Auf das Thema der Kommunikation wird in der Diplomarbeit [Krü09] detailliert eingegangen. Außer der Kommunikation zwischen verschiedenen UICs müsssen auch alle unter 2.6.2 genannten Komponenten kommunizieren können.

2) Rekursive Komposition: UI-Komponenten können in sich selbst ausgereifte Web-Anwendungen sein. Das bedeutet, dass diese Applikation, nach der Fertigstellung ihres Kompositionsmodells, zu einer UI-Komponente abstrahiert werden kann. Diese Abstraktion stellt eine leistungsfähige Technik zum Erstellen von Kompositionen dar. Dadurch ist es möglich, Modelle nicht nur aus UI-Komponenten zu erzeugen, sondern auch andere Kompositionen als ihre Bausteine zu nehmen. Dieses rekursive Modellieren ermöglicht es einerseits, auf verschiedenen Ebenen eines Kompositionsmodells zu abstrahieren (Bottom-Up), und andererseits auch in einem Top-Down-Prozess immer mehr zu verfeinern. Durch rekursive Komposition ist es demnach möglich, Kompositionen wieder zu komponieren und auch gleiche Kompositionen auf unterschiedlichen Abstraktionsstufen wiederzuverwenden, wodurch zusätzlich ein hohes Maß an Skalierbarkeit erreicht wird. Demnach sollte es, in dem zu entwerfenden Kompositionsmodell möglich sein, auch rekursive Komposition zu ermöglichen.

3) Konfiguration: UICs zeichnen sich dadurch aus, dass sie dynamisch an den Kontext angepasst werden, generisch und wiederverwendbar sind. Aus diesen Gründen müssen sie parametrisiert sein, und mit dem Kompositionsmodell muss es möglich sein, für diese Parameter initial Werte anzugeben.

4) Erweiterbarkeit: Um sich in das flexible Paradigma von UI-Mashups einzufügen, sollte das zu entwerfende Kompositionsmodell auch ein hohes Maß an Flexi-

bilität bieten. Das bedeutet, dass es unterstützt werden muss, seine Modellierungsmöglichkeiten zu erweitern. Das Kompositionsmodell sollte demnach so entworfen werden, dass Modellierungselemente hinzugefügt werden können, wodurch neue Semantiken im Modell abgebildet werden können, wenn sich die Forschung weiterentwickelt und neue Konzepte in Web-Applikationen möglich sind. Außerdem käme in Betracht, existierende Modellierungselemente zu erweitern. Dadurch wäre es beispielsweise möglich, zusätzliche Attribute hinzuzufügen, wodurch neue Eigenschaften modelliert werden können. Generell müssen Erweiterungen des Kompositionsmodells einfach durchführbar sein, da sonst der Aufwand nicht gerechtfertigt ist.

5) Geringer Lernaufwand: Der Aufwand, der betrieben werden muss, um mit einem Kompositionsmodell umgehen zu können, ist auf ein Minimum zu reduzieren. Wann immer es möglich ist, sollten schon vorhandene Sprachen sowie syntaktische und semantische Konventionen benutzt werden. Dabei ist es von besonderer Bedeutung, die Zielgruppe im Fokus der Aufmerksamkeit zu behalten. Wie in 2.5.2 schon erwähnt, können Modelle von verschiedenen Personen bearbeitet werden. Da diese nicht zwingend auch Programmierer sein müssen, muss darauf geachtet werden, dass nicht zu hohe technische Anforderungen an den Autor gestellt werden müssen. Es ist aber trotzdem davon auszugehen, dass die modellierende Person eine Affinität zur Modellierung von Software mitbringt und auch ein Verständnis gegenüber Web-Applikationen besitzt.

6) Werkzeugunterstützung: Die Notwendigkeit neuer Werkzeuge sollte so klein wie möglich gehalten werden. Existierende Entwicklungsumgebungen (IDE) müssen mit der Sprache ohne großen Aufwand umgehen können. Diese Anforderung hängt eng mit der zuvor beschriebenen zusammen. Kann der Autor sich in seinem gewohnten Umfeld bewegen, so erhöhen sich sowohl die Effektivität des Autorenprozesses als auch die Akzeptanz des Autors.

7) Separation of Concerns: Der Autor des UI-Mashups, bzw. der Komposition, muss vollkommen unabhängig von den Interna der zu komponierenden UI-Komponenten sein. Er darf nichts von der Implementierung wissen müssen, was bedeutet, dass die Veranwortlichkeiten zwischen der Person, die komponiert, und der Person, die Komponenten bereitstellt, klar getrennt sein müssen.

2.6.2 Anforderungen für UI-Mashups

In diesem Abschnitt werden Anforderungen aufgestellt, die speziell für UI-Mashups benötigt werden. Teilweise wurden sie schon in den Arbeiten [Voi08, Voc09] erwähnt. Diese werden hier aufgegriffen, erweitert und verfeinert:

8) Kopplung an Backend-Dienste: UICs sind im Normalfall keine statischen Elemente der UI, sondern visualisieren Daten oder greifen auf externe Funktionalitäten zu. Beispielsweise ist es denkbar, dass eine UI-Komponente Wetterdaten einer bestimmten Region anzeigen soll. Hierfür kann sie zum Beispiel die benötigten Daten von einem RSS-Feed oder Webservice konsumieren, der Wetterinformationen in Abhängigkeit von der Postleitzahl bereitstellt. Dafür muss es möglich sein, zu spezifizieren, woher diese Backend-Dienste kommen

beziehungsweise, wie sie zu erreichen sind. In CRUISe werden für diese Bindungen eigene Komponenten entwickelt: **Service Access Components (SACs)**. Diese sind dafür zuständig, Dienste einzubinden, die den UICs insbesondere Daten zur Verfügung stellen. Demnach müssen im Kompositionsmodell SACs spezifiziert werden können.

9) Aufbereitung von Daten: Wie oben erwähnt, stellen SACs Daten zur Verfügung. Sehr häufig kommt es vor, dass Daten, die als Eingabe für andere Komponenten dienen, nicht dem von der konsumierenden UIC geforderten Datentyp entsprechen. Als Beispiel sei zu nennen, wenn eine SAC eine Liste von Adressen liefert, die konsumierende UI-Komponente allerdings nur Straßen benötigt. Oder aber die gelieferten Daten enthalten nur implizit die benötigten Informationen. Dies wäre beispielsweise der Fall, wenn in einem UI-Mashup Informationen auf einer Weltkarte[1] angezeigt werden sollen und von einer SAC Adressdaten geliefert werden. Demnach müssen die Adressen erst in Geoinformationen umgewandelt werden. Für solche Fälle muss man mit dem Kompositionsmodell angeben können, dass gelieferte Daten aufbereitet werden sollen. In CRUISe existieren für diesen Zweck **Logic Components (LCs)**. Diese können als Adapter (vgl. [GHJV94]) zwischen inkompatiblen Datentypen dienen oder aber zu deren Transformation, Filterung und Aggregation. Demnach müssen mit dem Kompositionsmodell LCs zu spezifizieren sein.

10) Sammeln von Kontextdaten: Wie schon in Kapitel 2.3 dargestellt, werden UISs in Abhängigkeit von den Anforderungen und den Kontextdaten des Nutzers ermittelt. Im Ergebnis dieses Prozesses wird eine Rangfolge bestimmt, so dass dann der treffendste UIS ausgewählt werden kann. Die für das Ranking notwendigen Daten kommen vom Client, weshalb sie im Kompositionsmodell berücksichtigt werden müssen, da das UI-Mashup, welches aus einer CCM erzeugt wurde, im Browser des Anwenders läuft. Die Idee der Sensoren für den Client wird in CRUISe mittels **Context Components (CCs)** realisiert werden. Ihre Aufgabe ist es, im Hintergrund Kontextdaten zu sammeln, die über die *CRUISe Runtime* an den *Context Service* gesendet werden können. CCs müssen demnach im Kompositionsmodell spezifiziert werden können. Außerdem kann es auch möglich sein, dass UI-Komponenten Kontextdaten als Input bekommen. Diese müssen für solche Fälle in den oben erläuterten Kommunikationskanälen spezifiziert werden können.

11) Kontrollfluss der Anwendung: Wie schon in Anforderung 1 erwähnt, verläuft die Kommunikation zwischen UICs über Ereignisse. Jedoch können auch Ereignisse eintreten, wenn der Benutzer des UI-Mashups mit der UI interagiert (beispielsweise durch Betätigen eines Knopfes). In solchen Fällen ist es möglich, dass sich die komplette Ansicht des Benutzers ändert. Als Beispiel sei an dieser Stelle genannt, wenn der Nutzer durch einen Wizard geführt wird. Dabei ändert sich beim vollständigen Abarbeiten eines Formulars die komplette Ansicht, und er gelangt zur nächsten. Möglich ist auch, dass bei einem eintretenden Ereignis Teile der UI verschwinden oder hinzukommen, sei es durch

[1] beispielsweise mittels Google Maps: maps.google.com

ein Ereignis, ausgelöst durch den Benutzer, oder aber ausgelöst von einer anderen Komponente. Diesen Aspekt von UI-Mashups nennt man Kontrollfluss. Im Kompositionsmodell muss also spezifizierbar sein, welche Übergänge verschiedener Sichten existieren und wann sie eintreten.

12) Adaptivität: Im Zuge des wachsenden Umfangs der verfügbaren Informationen im Web ist es nicht mehr sinnvoll, jedem Nutzer dieselben Informationen anzuzeigen [BKN07, S. V]. So ist es sinnvoll, wenn auf einer Nachrichtenseite Benutzern mit unterschiedlichen Interessen unterschiedliche aktuelle Nachrichten als Erstes angezeigt werden. Diese Art von Adaptivität wird *Datenadaptivität* genannt. In UI-Mashups geht es jedoch um die Flexibilität der Benutzerschnittstelle, weshalb mit dem Kompositionsmodell *UI-Adaptivität* spezifiziert werden soll. Die UI kann sich also in ungleichen Kontexten ändern, sei es nutzerübergreifend oder in verschiedenen Kontexten desselben Nutzers. Um Kontextinformationen zu sammeln, werden in CRUISe, wie oben schon erwähnt, CCs verwendet. Doch im Kompositionsmodell sollte zusätzlich noch die Möglichkeit geschaffen werden, anzugeben, welche Elemente der UI sich in Abhängigkeit von welchem Kontext wie anpassen sollen. Konkret bedeutet dies, dass sich Aspekte des UI-Mashups oder einzelne integrierte UI-Komponenten unter bestimmten Bedingungen ändern können. Als Beispiel ist denkbar, dass eine komplette UI-Komponente nur noch minimiert angezeigt wird, wenn eine CC über diese Komponente keine Kontextdaten liefert. Dies wäre der Fall, wenn ein Nutzer nicht mit dieser UIC interagiert. Oder aber die Anordnung der UI-Komponenten im Layout des UI-Mashups ändert sich dahingehend, dass die meistgenutzten Komponenten oben erscheinen und andere unten. So eine Art von Adaption wird *Laufzeit-Adaption* genannt, da spezifizierte Teile des UI-Mashups erst zur Laufzeit angepasst werden. Das bedeutet, dass man zur Design-Zeit angeben können muss, unter welchen Bedingungen angepasst wird und welche Teile angepasst werden sollen.

13) Layout: Da es sich um visuelle Komponenten handelt, muss es möglich sein, ihr Layout zu beeinflussen. Beispielsweise muss der Autor angeben können, welche Abmessungen die UI-Komponenten besitzen. Dies ist ein wichtiger Aspekt im Autorenprozess, da davon die Übersichtlichkeit des UI-Mashups abhängt. Neben dem Layout der UICs muss auch das Layout des gesamten UI-Mashups spezifiziert werden können. Fragen wie »Wie sind UI-Komponenten im Gesamt-Layout angeordnet?« und »Wie sieht ein solches aus?« müssen mit dem Kompositionsmodell beantwortet werden können.

14) Angabe von Klassen: Um ein *generisches* Kompositionsmodell zu erhalten, bedarf es einer Klassifikation von UISs. In der Arbeit [Bau09] wird eine solche Klassifikation entworfen. Damit ist es möglich, im Autorenprozess des UI-Mashups statt eines konkreten User Interface Services eine Klasse anzugeben. Mehrere verschiedene, aber vom Wesen her gleiche, UISs (beispielsweise eine Google Map und eine Yahoo Map[2]) fallen dann in eine Klasse und können gleich behandelt werden. Dadurch ist es möglich, UICs gleicher Klassen zur Laufzeit auszutauschen. Dieser Austausch kann durch Events, ausgelöst durch

[2] de.maps.yahoo.com

Komponenten oder den Benutzer, initiiert werden. Mit dem Kompositionsmodell muss es demnach möglich sein, Klassen anzugeben, die für UISs stehen.

15) Einheitliches Look & Feel: Es kann erwünscht sein, alle UICs eines UI-Mashups in einem einheitlichen Erscheinungsbild darzustellen, welches beispielsweise dem Corporate Design einer Firma entspricht. Hierfür sollte die Möglichkeit bestehen, Designinformationen auch für das gesamte UI-Mashup zu spezifizieren. Dazu gehören beispielsweise Angaben zu gemeinsamen Schriftarten, Schriftformen oder Schriftgrößen, sowie Farben, die von allen UI-Komponenten einheitlich benutzt werden sollen.

2.6.3 Nichtfunktionale Anforderungen

Die oben dargestellten Anforderungen sind hauptsächlich funktionaler Natur. Im folgenden werden auf Grundlage von [RR06, S. 476ff] weitere nichtfunktionale Anforderungen ergänzt:

16) Wiederverwendbarkeit: Da bei der Konzeption ein MDA-Ansatz verfolgt werden soll, muss mit dem Kompositionsmodell auch gewährleistet werden, dass CCMs oder einzelne Teile davon wiederverwendet werden können. Demnach muss darauf geachtet werden, dass bei Wiederverwendung einzelner Modelle ohne großen Aufwand neue UI-Mashups möglich sind. Beispielsweise könnte man die Teile einer Komposition, die Kommunikationskanäle, Layout und Design, sowie die enthaltenen Komponenten beschreiben, wiederverwenden. Zusätzlich kann dann ein anderer Kontrollfluss des UI-Mashups spezifiziert werden, und man erhält mit geringem Aufwand, durch Wiederverwendung von Modellen, eine neue Web-Applikation.

17) Technologieunabhängigkeit: Um ein hohes Maß an Wiederverwendbarkeit zu erreichen, ist es erforderlich, CCMs vollkommen unabhängig von der Technologie der Zielplattform und auch von der Programmiersprache des Glue-Codes zu gestalten. Erst in einem letzten Transformationsschritt durch den *Application Generator* wird das finale UI-Mashup mit allen notwendigen Technologien erzeugt.

18) Testbarkeit: Mit dem Kompositionsmodell entworfene CCMs müssen so einfach wie möglich zu testen sein. Dazu zählt nicht das Testen der einzelnen UI-Komponenten, da diese autonom entworfen wurden und demnach ausreichend getestet sind. Mit Testbarkeit ist beispielsweise das Testen des Kontrollflusses des UI-Mashups gemeint, dahingehend, ob alle Navigationspfade erreichbar sind und der Nutzer etwa eine Ansicht nicht verlassen kann. So einen Fall kann man als *Deadlock* bezeichnen und kann beispielsweise mit Model-Checkern überprüft werden.

19) Unterstützung von Standards: Das Kompositionsmodell sollte weitestgehend auf Standards beruhen und diese unterstützen, da dadurch ein hohes Maß an Qualität gesichert werden kann. Standards haben einen Reifeprozess durchlaufen und finden Anklang bei einer breiten Masse. Dadurch wird eine große Gruppe von Entwicklern erreicht, so dass das Kompositionsmodell auch in Zukunft weiterentwickelt werden könnte.

20) Lesbarkeit: Die Bezeichnungen der Modellierungselemente, die dem Autor des UI-Mashups zur Verfügung gestellt werden, sollten weitestgehend selbsterklärend sein. Der Autor sollte möglichst ohne Dokumentation in der Lage sein, UI-Mashups zu komponieren.

21) Leichte Integration in den Entwicklungsprozess: Im Allgemeinen wird Software in einem iterativen Software-Entwicklungsprozess[3] implementiert. Demnach muss das Kompositionsmodell leicht in den gewohnten Entwicklungsprozess eines Entwickler-Teams integrierbar sein, da sonst die Akzeptanz sinkt. Diese Anforderung geht einher mit denen der Punkte 5, 6 und 7.

2.6.4 Abgrenzung

Wie schon in 2.6.1 erläutert, wurden in [CWD00] Anforderungen an Kompositionssprachen entwickelt, die sich auf konventionelle Software beziehen. Im Kontext von UI-Mashups spielen einige von ihnen keine Rolle, weshalb diese hier abgegrenzt werden:

Spezifikation von Glue-Code: Nach [CWD00] müssen Kompositionssprachen es erlauben Glue-Code zu spezifizieren, um unterschiedliche Schnittstellen zweier Komponenten zur Laufzeit adaptieren zu können. Dieser Aspekt ist in UI-Mashups nicht notwendig, da Schnittstellenadaption mittels Kommunikationskanälen (vgl. Anforderung 1) und dementsprechenden LCs (vgl. Anforderung 9) zu modellieren ist. Eine Runtime, die das UI-Mashup lauffähig macht, ist dafür verantwortlich, die Kommunikations- und Adaptionslogik der LC so auszuführen, wie sie in dem CCM spezifiziert wurde. Der Glue-Code existiert demnach schon in der Architektur.

Backend-Komposition: Wie schon unter 2.1 erwähnt, besteht der Autorenprozess aus der UI-Komposition und der Komposition der Backend-Dienste. Für Komponenten von UI-Mashups sind die Backend-Kompositionen allerdings unerheblich, da sie orchestrierte Services lediglich konsumieren und, bezogen auf deren Struktur, nicht erfahren, dass es komposite Dienste sind. Demnach ist die Komposition der Backend-Dienste ein autonomer Prozess und also unabhängig von der Komposition der UISs. Folglich kann dieser Prozess ausgelagert und in Architekturen anderer Anbieter[4] durchgeführt werden, die letzen Endes referenziert werden.

[3] beispielsweise mit dem V-Modell XT: www.v-modell.iabg.de

[4] beispielsweise mit dem *JackBe Presto Mashup Server* – siehe `http://www.jackbe.com/products/server.php`

3 Analyse existierender Ansätze

Mit der steigenden Komplexität von Web-Applikationen und den höheren Ansprüchen, die man an diese stellt, sind Ad-hoc-Methoden nicht mehr geeignet, solche Systeme zu entwickeln [Mur08]. Wie schon in 2.5 erwähnt, ist ein modellgetriebenes Vorgehen unabdingbar, um die Komplexität zu bewältigen. Als Wurzel aller Ansätze gelten die Daten-Modellierung und die Software-Entwicklung [SK03, S. 51], aus denen sich verschiedene Paradigmen der Entwicklung von Web-Applikationen herauskristallisiert haben. Dazu gehören der datenorientierte, objektorientierte und komponentenbasierte Ansatz. Was die meisten Ansätze gemein haben, ist die Nutzung der *Unified Modeling Language (UML)* [SK03, S. 51] zur Modellierung bestimmter Aspekte von Web-Applikationen. Die Autoren führen an, dass die UML unbestritten als Notationssprache erhalten bleibt [SK03, S. 74]. Aufgrund der häufigen Verwendung der UML und der Voraussage, dass sie als Quasistandard in der Modellierung von Web-Applikationen auch in Zukunft genutzt werden wird, wird sie zunächst im Abschnitt 3.1 vorgestellt. Danach wird für jedes der oben erwähnten modellgetriebenen Paradigmen eine Methodik präsentiert sowie deren Stärken und Schwächen, bezogen auf die im Kapitel 2.6 ermittelten Anforderungen. Nachdem die Ansätze zunächst neutral beschrieben werden, folgt im Weiteren eine Bewertung jedes einzelnen Ansatzes. Dies geschieht mit folgenden Bewertungskriterien: + steht für eine vorhandene Eigenschaft, die mit nur minimalen Anpassungen übernommen werden könnte; **o** bezeichnet eine vorhandene Eigenschaft, die jedoch nur bedingt nutzbar ist; **-** kennzeichnet eine nicht vorhandene Eigenschaft beziehungsweise einen Ansatz, der nicht nutzbar ist.

3.1 Unified Modeling Language

Die UML ist eine Sprache für die Modellierung von Software und komplexen Systemen. Sie wurde von der OMG entworfen und liegt derzeit in der Version 2.2 vor.

Die UML wurde aus dem Bedürfnis heraus entwickelt, Systemarchitekten, Programmierer und Softwareentwickler bei Analyse, Entwurf und Implementierung von Software-basierten Systemen und Prozessen zu unterstützen [OMG09]. Zwei der Ziele der UML sind es, die visuelle Modellierung und den Austausch von Modellen zwischen verschiedenen Modellierungswerkzeugen zu ermöglichen. Demzufolge besteht die UML aus einer formalen Definition eines Metamodells, das die abstrakte Syntax spezifiziert, einer detaillierten Erklärung der Semantiken jedes Modellierungskonzeptes sowie der Spezifikation einer für Menschen lesbaren Notation für die Repräsentation in den verschiedenen definierten Diagrammarten.

Im Folgenden werden drei Diagramm- und Modellarten erläutert, da sie in den meisten Ansätzen Verwendung finden. Für Details sei auf [OMG09] verwiesen. Außerdem benutzen alle Methoden die ebenfalls mit der UML spezifizierte Sprache *Object Constraint Language (OCL)*, um Bedingungen und Beschränkungen für Mo-

delle zu formulieren. Zum besseren Verständnis wird auch diese im vorliegenden Kapitel vorgestellt.

3.1.1 UseCase-Diagramm

UseCase-Modelle gehören zu den Verhaltensdiagrammen der UML. Ein *UseCase*-Modell wird beispielsweise als erstes Modell in dem in Kapitel 3.3 vorgestellten Ansatz eingesetzt. *UseCase*-Modelle werden typischerweise dazu benutzt, die Anforderungen an den Funktionsumfang eines Systems zu modellieren. Die Hauptbestandteile von *UseCase*-Diagrammen sind *Actors*, *UseCases* und *Subjects*. Ein *Subject* stellt das zu beschreibende System dar, während *Actors* Benutzer und andere interagierende Systeme modellieren. Das Verhalten des Systems wird durch *UseCases* modelliert. Abbildung 3.1 zeigt ein Beispiel-Diagramm.

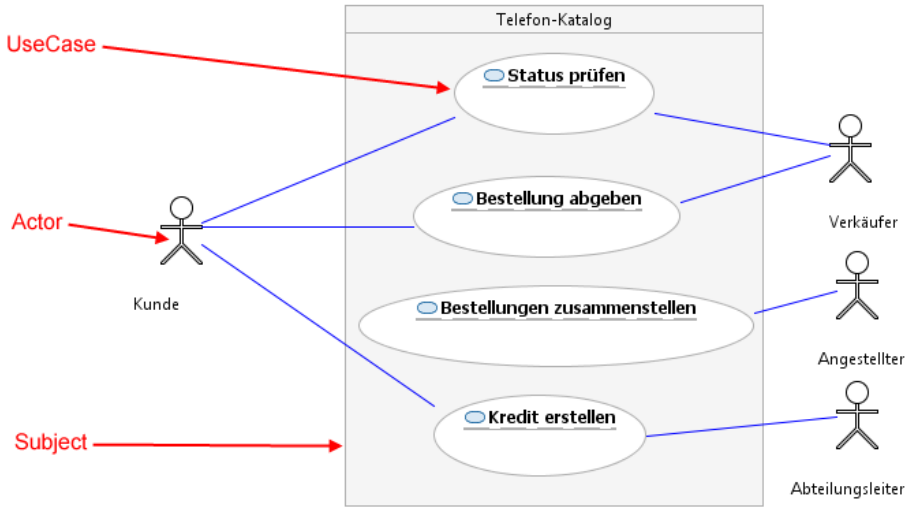


Abbildung 3.1: Beispiel-*UseCase*-Diagramm

3.1.2 Klassendiagramm

Das Klassendiagramm ist das Hauptdiagramm der UML und gehört zu den Strukturdiagrammen. Der Begriff der *Klasse* bezeichnet in der Objektorientierung einen abstrakten Oberbegriff für Objekte gleichen Verhaltens und gleicher Struktur. Das Metamodell der UML selbst wurde mit Klassendiagrammen beschrieben. Diese beschreiben Klassen mit ihren Attributen und Methoden sowie deren Beziehungen zueinander. Dazu gehören beispielsweise *Vererbung* und die *Aggregation*. Abbildung 3.2 zeigt ein Beispiel-Diagramm.

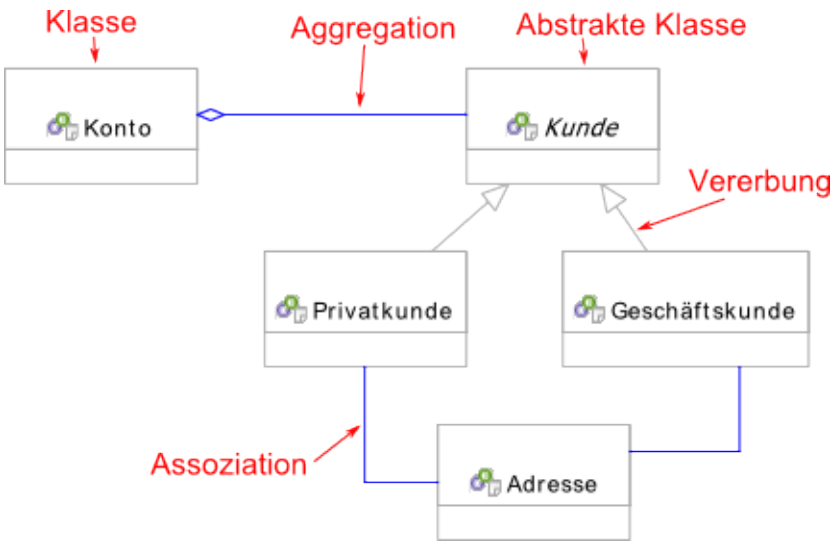


Abbildung 3.2: Beispiel-Klassendiagramm

In dieser Abbildung ist zu sehen, dass ein Konto einem abstrakten Kunden zugeordnet ist. Hiervon gibt es zwei Unterklassen, Privat- und Geschäftskunde, die jeweils eine Adresse besitzen.

Das Klassendiagramm wird in allen im Folgenden vorgestellten Methodiken verwendet, um beispielsweise das Datenmodell zu spezifizieren oder aber um die Navigation innerhalb einer Web-Applikation anzugeben.

3.1.3 Zustandsdiagramm

Zustandsdiagramme gehören auch zu den Verhaltensdiagrammen, die das diskrete Verhalten mittels endlicher Zustandsübergangssysteme modellieren. Damit kann das Verhalten von Entitäten anderer Modelle (beispielsweise von Klassen) beschrieben werden. Elemente dieses Diagrammtyps können zum Beispiel *Zustände*, *Transitionen* oder *Choices* sein. Abbildung 3.3 zeigt ein Beispiel-Diagramm.

Zustandsdiagramme werden beispielsweise dazu benutzt, das Verhalten von *Use-Cases* einer Web-Applikation zu konkretisieren.

Außer den Diagramm- und Modellarten ist in der UML auch ein Erweiterungsmechanismus spezifiziert. Dieser wird im folgenden Kapitel erläutert.

3.1.4 UML-Profile

UML-Profile stellen einen leichtgewichtigen Erweiterungsmechanismus zur Verfügung. Dies bedeutet, dass das Metamodell der UML ergänzt werden kann, ohne es selbst zu verändern. Auf diese Weise ist es möglich, neue Metaklassen für neue Domänen zu entwerfen, wodurch damit modellierte Elemente eine konkretere Bedeutung erhalten. Das Ziel dieses Vorgehens besteht in der Spezialisierung der UML für bestimmte Domänen, Plattformen oder Methoden. Um eine Metaklasse zu erweitern, wird ein sogenannter *Stereotyp* erstellt. Diesem können dann weitere Attribute hinzugefügt werden. In Abbildung 3.4a beispielsweise wird die Metaklasse *Interface* um zwei Stereotypen erweitert: *Remote* und *WebService*. Diese Stereotypen können nun in Modellelementen appliziert werden, die vom Typ der Metaklasse (Interface)

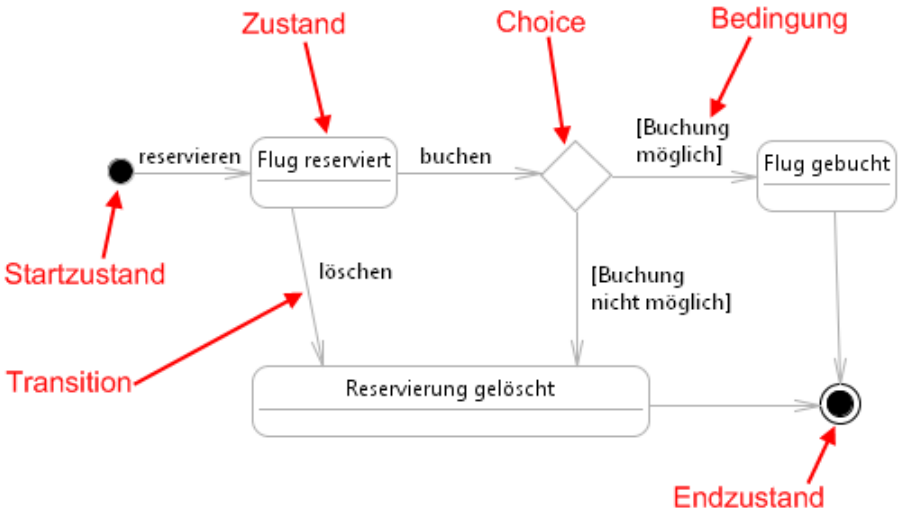
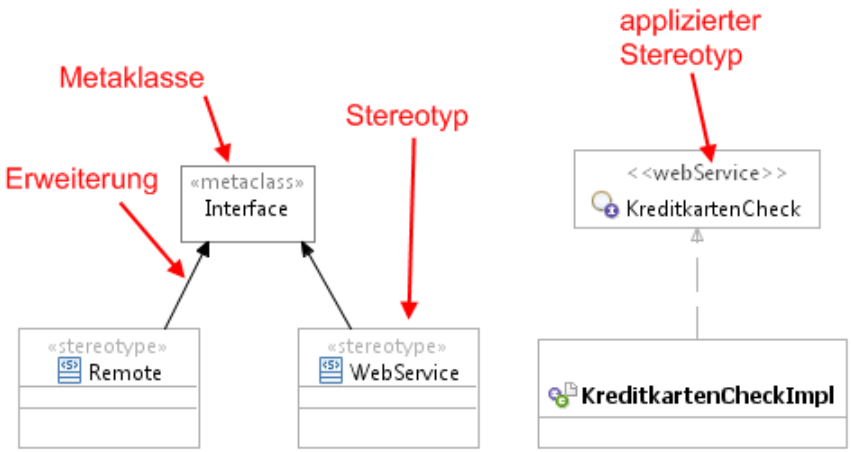


Abbildung 3.3: Beispiel-Zustandsdiagramm

sind. Der erstellte Stereotyp *WebService* wird in Abbildung 3.4b dem Interface *KreditkartenCheck* appliziert, wodurch dieses Interface semantisch konkretisiert wurde.



(a) Profil mit neuen Stereotypen

(b) Applizierung eines Stereotyps

Abbildung 3.4: Beispiel eines UML-Profils

Jim Conallen hat in [Con99] gezeigt, dass Webseiten auf Klassen eines Klassendiagramms übertragen werden können. Hyperlinks entsprechen Assoziationen, und Klassenmethoden können als Skripte einer Webseite angesehen werden. Conallen führte ein UML-Profil ein, mit dem er Klassen mit Stereotypen wie *Client Page*, *Applet* oder *JavaScript* konkretisieren konnte. Diese Idee wurde vielfach aufgegriffen und zu ausgereiften Varianten der Modellierung von Web-Applikationen weiterentwickelt. Anhand von Stereotypen ist es in einigen vorgestellten Ansätzen möglich, Modelle zu transformieren, da nun gezielt nach Stereotypen gesucht werden kann,

die einem bestimmten Zweck entsprechen und in Abhängigkeit davon Code generiert werden kann.

Wie vorher schon erwähnt, können für Modelle Bedingungen und Einschränkungen festgelegt werden. Das nächste Kapitel wendet sich diesem Teil der UML-Spezifikation zu.

3.1.5 Object Constraint Language

Zur im vorangegangenen Abschnitt genannten Erweiterbarkeit der UML zählt auch die Möglichkeit, Einschränkungen anzugeben. Die OMG hat in [OMG06b] eine formale Sprache entworfen, mit deren Hilfe solche Ausdrücke für UML-Modelle spezifiziert werden können, die typischerweise für ein System zwingend geltende Invarianten darstellen.

OCL liegt derzeit in der Version 2.0 vor und ist seit der UML-Version 1.1 auch Bestandteil derselben. Sie wurde von der OMG entwickelt und maßgeblich von vielen Firmen und Universitäten vorangetrieben, zu denen auch die Technische Universität Dresden zählt [OMG06b].

Mit den bisher gebräuchlichen Modellierungsmöglichkeiten der UML ist es oft nicht möglich, Diagramme so weit zu verfeinern, dass sie alle relevanten Aspekte der Spezifikation enthalten. Oft werden zusätzliche Bedingungen in natürlicher Sprache formuliert, wodurch es zu Doppeldeutigkeiten kommen kann. Aus diesem Grund wurde die OCL entwickelt. Es handelt sich hierbei um eine reine Spezifikationssprache, weshalb es keine Seiteneffekte dahingehend gibt, dass am Modell Änderungen bei der Auswertung von OCL-Ausdrücken durchgeführt werden. Wenn von der Einschränkbarkeit von Modellen gesprochen wird, dann sind natürlich auch Metamodelle gemeint. Das bedeutet, dass OCL auch dazu benutzt werden kann, Regeln für ein UML-Profil zu spezifizieren, die dann im Modell, welches das Profil appliziert, gelten müssen. Am meisten Verwendung findet die OCL in der Spezifikation von *Invarianten*. Außerdem gibt es *Vor- und Nachbedingungen*, sowie *Initiale und Abgeleitete Werte*. Jede OCL-Regel gilt immer in einem bestimmten Kontext, welcher eine beliebige Entität in einem Modell darstellt. Am Beispiel von Abbildung 3.5 werden im Folgenden die wichtigsten OCL-Regeln erklärt.

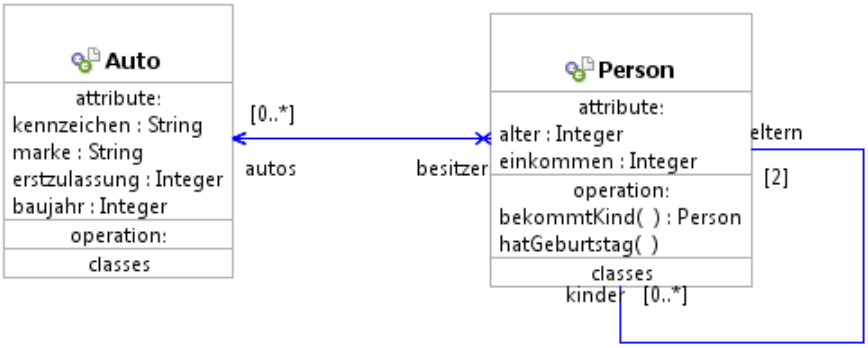


Abbildung 3.5: Beispieldiagramm für OCL-Regeln

3.1.5.1 Invarianten

Invarianten können mit Typen assoziiert werden. Dann müssen Invarianten für alle Instanzen des Typs zu jedem Zeitpunkt gelten. Zum Beispiel spezifiziert Listing 3.1, dass das Alter einer Person immer größer als Null sein muss und die Eltern immer älter als man selbst.

Listing 3.1: OCL-Invarianten

```
context Person inv:
        self.alter >=0 and
        self.eltern->forAll(e|e.alter>self.alter)
```

3.1.5.2 Vor- und Nachbedingungen

Vor- und Nachbedingungen beziehen sich immer auf den Kontext einer bestimmten Operation. Die Bedingung muss also vor, respektive nach, der Ausführung der Operation gelten. Im Listing 3.2 ist beispielsweise spezifiziert, dass das Alter nach der Ausführung von `hatGeburtstag()` um Eins größer sein muss als vorher. Die zweite Regel gibt an, dass die Anzahl der Kinder nach der Ausführung von `bekommtKind()` größer sein muss als vorher.

Listing 3.2: OCL-Nachbedingungen

```
context Person::hatGeburtstag() post:
        self.alter=self.alter@pre+1

context Person::bekommtKind() post:
        self.kinder->notEmpty() and self.kinder->size() > self.kinder@PRE->size()
```

3.1.5.3 Initiale und Abgeleitete Werte

OCL-Ausdrücke können auch benutzt werden, um Initialwerte oder abgeleitete Werte von Attributen oder Assoziationsenden zu bestimmen. So ein spezifizierter Ausdruck muss mit dem Typ konform gehen, der dem Attribut entspricht. Beispielsweise wird im Listing 3.3 spezifiziert, dass der initiale Wert des Einkommens einer Person 0,5% der Summe der Elterngehälter entspricht. Dies kann zum Beispiel als Taschengeld eines Kindes gesehen werden. Wenn das Kind über 16 Jahre alt ist, kann es selber Geld verdienen, weshalb das Einkommen dann durch das eigene Gehalt bestimmt wird.

Listing 3.3: OCL-Initial and abgeleitete Werte

```
context Person::einkommen : Integer
        init: eltern.einkommen->sum() * 0.5%
        derive: if alter<16
                        then eltern.einkommen->sum() * 0.5%
                        else einkommen
                        endif
```

Mit der Darstellung der UML und der damit in Zusammenhang stehenden OCL ist der Grundstein für die Modellierung konventioneller Softwaresysteme gelegt. Damit können nun auch existierende Ansätze zur Modellierung von Web-Applikationen vorgestellt werden.

3.2 Datenorientierte Modellierung am Beispiel von WebML

Datenorientierte Methoden haben ihren Ursprung im Bereich der Datenbanksysteme. Demnach liegt ihr Fokus auf der Modellierung datenintensiver Web-Applikationen, und sie beruhen hauptsächlich auf der Modellierung von Daten mit dem Entity-Relationship-Modell (ER-Modell) [SK03, S. 70]. Bekannteste Vertreter des datenorientierten Paradigmas sind Hera [HvdSB+08] und die *Web Modeling Language (WebML)*. Da WebML bei Firmen wie *Acer* und Universitäten, wie beispielsweise der *Politecnico di Milano*, bereits in der Praxis verwendet wurden, kann man Praxistauglichkeit annehmen. Deshalb wird WebML als Vertreter dieses Paradigmas gewählt und analysiert.

Die Gruppe der WebML-Entwickler setzt sich aus Forschern des *Dipartimento di Elettronica e Informazione* an der *Politecnico di Milano* zusammen. Ihr ursprüngliches Ziel war es, Design und Implementierung datenintensiver Web-Applikationen zu unterstützen, weshalb existierende konzeptuelle Datenmodelle übernommen wurden und eine neue Notation erschaffen wurde, um Hypertext-Navigation zu modellieren [BCFM08]. Im Gegensatz zu anderen Methoden wird in WebML der Ansatz verfolgt, ein möglichst minimales Repertoire an Modellierungselementen anzubieten, die dann ihre Stärke in ihrer Zusammensetzung ausspielen.

WebML ist eine visuelle Sprache, um die Struktur des Inhalts von Web-Applikationen, sowie die Organisation und Präsentation des Inhalts im Hypertext der Web-Seiten zu spezifizieren. Es handelt sich also hierbei nicht nur um eine Sprache, sondern auch um eine Vorgehensweise. So werden in der Phase der Anforderungsanalyse Benutzergruppen identifiziert, funktionale Anforderungen aufgestellt, Hauptinformationen, die Nutzern zugänglich gemacht werden sollen, gesammelt, sowie *Site Views* identifiziert, welche bestimmten funktionalen Aspekten einer Web-Applikation entsprechen und in der nächsten Phase konkret spezifiziert werden. Die konzeptuelle Modellierung umfasst wiederum zwei Phasen: Datenmodellierung und Hypertext-Modellierung. Die darin enthaltenen Modelle werden in den folgenden Abschnitten vorgestellt.

Data Model

In diesem Modell werden Hauptinformationsobjekte aus der Analyse in Datenschemata überführt. Für den Datenentwurf existieren sehr gut etablierte Modellierungsmöglichkeiten, wie das ER-Modell oder das UML-Klassendiagramm. Da beide in der Modellierung von Daten äquivalent genutzt werden können, sind auch beide in WebML nutzbar. Die wichtigsten Elemente des Datenmodells sind somit Entitäten mit Attributen und Relationen mit Kardinalitäten. So zeigt Abbildung 3.6 beispielsweise die Entität `Movie` mit ihren Attributen sowie den Sachverhalt, dass zur Entität `UserComment` eine 1:N-Beziehung besteht. In WebML können, wie im Bereich der Datenbanken, abgeleitete Attribute modelliert werden. Diese ergeben sich aus anderen Attributen, werden mit `/` gekennzeichnet und können mittels OCL beschrieben werden.

Abbildung 3.6 zeigt das abgeleitete Attribut `/NumOfComments`, welches die Anzahl der Kommentare zählt, die mit einem Film assoziiert sind.

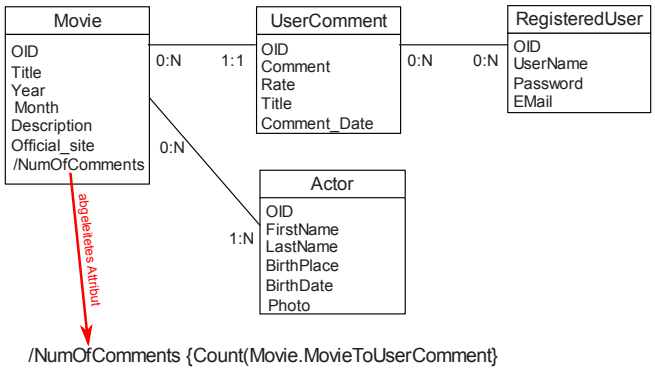


Abbildung 3.6: Beispiel eines WebML *Data Models* [BCFM08]

Hypertext Model

In diesem Modell wird die UI spezifiziert und beinhaltet die Definition der Komponenten, die den Inhalt anzeigen, sowie Links zwischen den Seiten. Die oben schon erwähnten *Site Views* werden dabei verwendet, um eine Menge von Anforderungen abzudecken. Ein *Site View* besteht aus *Areas*, *Pages* und *Content Units*. *Pages* enthalten die für den Nutzer sichtbaren Elemente. Für jede vorher identifizierte Benutzergruppe gibt es mindestens einen *Site View*. Das bedeutet, dass mehrere *Site Views* für ein und dasselbe *Data Model* definiert werden können. Abbildung 3.7 zeigt ein Beispiel.

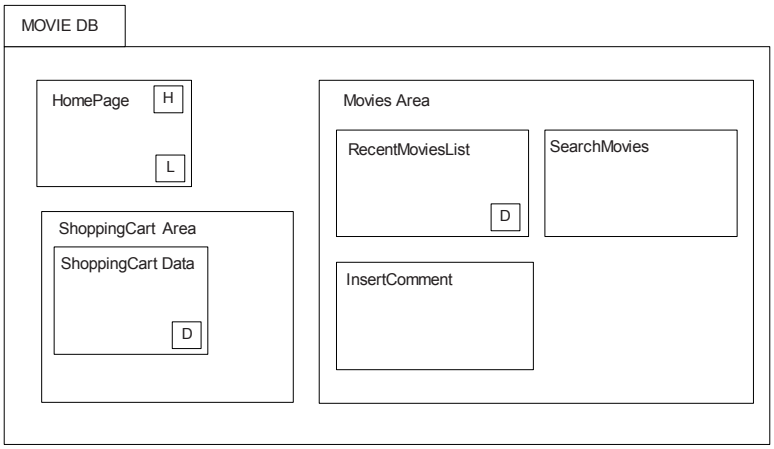


Abbildung 3.7: Beispiel eines WebML *Site Views* [BCFM08]

In der Abbildung sind die Markierungen `H`, `D` und `L` zu sehen. Diese stehen für die *Page*, die als Startseite der Adresse der *Site View* angezeigt wird (*Home Page*), für die anzuzeigende *Page* der einschließenden *Area* (*Default Page*) beziehungsweise für eine Seite, die von allen anderen *Pages* erreichbar ist (*Landmark Page*).

Wie werden nun der konkrete Inhalt der *Pages* sowie die Übergänge modelliert? In WebML gibt es fünf vordefinierte *Content Units*, die jeweils mit Entitäten des *Data Models* verbunden werden und die anzuzeigenden Daten mittels Abfragen einschränken können. In der folgenden Übersicht werden die *Content Units* kurz erläutert. Abbildung 3.8 zeigt alle in einem praktischen Anwendungsbeispiel.

Data Unit: Repräsentiert Attribute einer Entität des *Data Models*.

Multidata Unit: Repräsentiert Attribute einer Menge von Entitäten.

Index Unit: Steht für eine Liste von Attributen einer Menge von Instanzen einer Entität.

Scroller Unit: Ermöglicht das Browsen einer Menge von Objekten.

Entry Unit: Ermöglicht Eingaben vom Benutzer.

Die Navigation wird in WebML über *Links* modelliert. Es gibt zwei Arten: *contextual*, die als Verbindung zweier *Units* dienen und Daten von der Quelle zum Ziel transportieren, und *noncontextual*, die zwei *Pages* miteinander verbinden. Abbildung 3.8 zeigt ein Beispiel beider Arten.

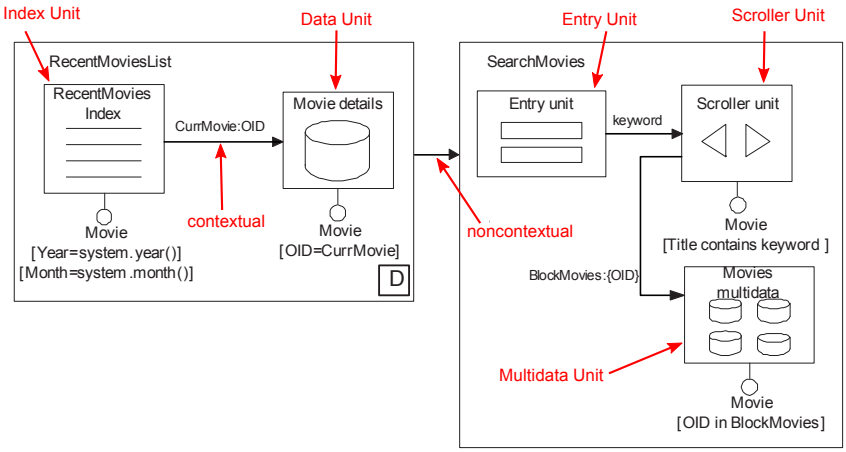


Abbildung 3.8: Beispiel der WebML Navigation [BCFM08]

Eine andere Art von Verbindungen sind *Transport Links*. Diese transportieren kontextuelle Informationen von einer *Unit* zur anderen, sind jedoch nicht als Link auf der finalen Webseite gerendert, für den Nutzer demnach unsichtbar.

Um Daten intern für alle *Pages* zur Verfügung zu stellen, werden *Global Parameter Units* verwendet. Mit *Set Units* können sie gesetzt und mit *Get Units* ausgelesen werden. Um Daten nicht nur anzuzeigen, sondern auch zu manipulieren, gibt es in WebML *Operation Units*. Instanzen von Entitäten des *Data Models* können beispielsweise mit den entsprechenden *Units* erzeugt (*Create Unit*), verändert (*Modify Unit*) oder gelöscht (*Delete Unit*) werden. *Operation Units* können mehrere kontextuelle Links als Eingabe bekommen, wodurch Parameter übergeben werden. Als Ausgang haben sie immer einen *OK Link*, für die erfolgreiche Abarbeitung der Operation, und einen *KO Link*, für den Fehlerfall. Diese können dann wieder als Auslöser weiterer *Operation Units* dienen.

Adaptivität

Ceri et al. stellen in [CDMF07] den Adaptivitätsansatz von WebML vor. Darin wird das *Data Model* um drei zusätzliche Schemata erweitert, wodurch Kontextdaten des Nutzers gesammelt und darin gespeichert werden können. Das *User Subschema* beinhaltet Nutzerdaten und deren Zugriffsrechte auf die Web-Applikation. Das *Personalization Subschema* enthält alle Entitäten des *Data Models*, die in Zusammenhang mit dem Benutzer der Web-Anwendung stehen. Ein Beispiel aus Abbildung

3.6 wäre die Entität `Movie`. Hat der Nutzer das 18. Lebensjahr noch nicht vollendet, sollten FSK-18-Filme nicht mit angezeigt werden. Das *Context Subschema* enthält Entitäten zum Sammeln von Daten wie Ort, Gerät oder Aktivität. Diese Dreiteilung der Schemata stellt im Prinzip SoC auf Ebene der Adaptivität dar. Weiterhin können in WebML *Pages* als kontextsensitiv (`C`) markiert werden, für die *Context Clouds* spezifiziert werden. In Abbildung 3.9 sind beispielhaft zwei *Context Clouds* abgebildet.

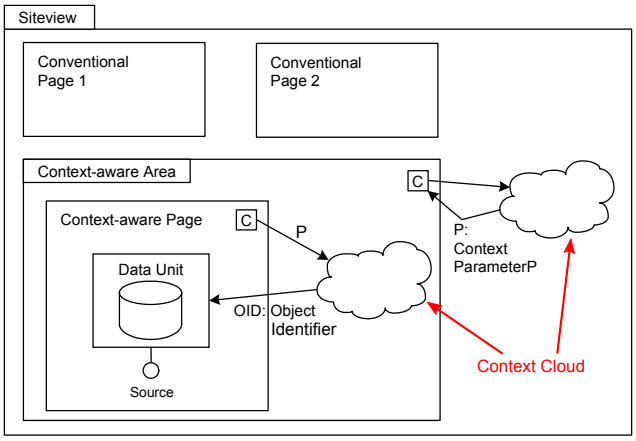


Abbildung 3.9: WebML-*Area* und -*Page* mit *Context Clouds* [CDMF07]

In *Context Clouds* werden Adaptivitätsaktionen angegeben. Diese werden vor der Ausführung der als kontextsensitiv markierten Seite oder *Area* ausgewertet, und es werden entsprechend Anpassungen vorgenommen. Solche Änderungen können beispielsweise die Adaption des Inhalts einer Seite oder die Änderung des vordefinierten Navigationsflusses [CDMF07] sein.

Werkzeugunterstützung

Mit WebRatio[1] ist ein kommerzielles Werkzeug auf dem Markt, das zusammen mit WebML entwickelt wurde und das WebML-Modell in sich integriert. WebRatio besteht im Kern aus der IDE Eclipse[2] und wurde um Funktionen erweitert, die das Modellieren und Entwickeln von Web-Applikationen mit WebML ermöglichen. Wie die Infrastruktur in Abbildung 3.10 zeigt, wird in WebRatio ausschließlich Code generiert, der auf Java-fähigen Applikationsservern zur Ausführung gebracht wird. Konkret bedeutet dies, dass Java Beans für die Komponenten generiert werden und JSP-Code für die Webseiten.

Außerdem können eigene WebML-*Units* entwickelt sowie deren ausführbare Repräsentationen als Komponente (vgl. Abbildung 3.10) in einer Runtime ausgeführt werden. Für das *Data Model* werden alle Datenbanken unterstützt, die einen JDBC[3]-Treiber anbieten, wodurch ein weites Feld abgedeckt ist. Außerdem ist es möglich, aus einem vorhandenen Datenbank-Schema ein *Data Model* zu erzeugen und umgekehrt. Layout und *Look & Feel* werden in Templates definiert, die dann in beliebige Rendering-Sprachen überführt werden können.

[1] http://www.webratio.com
[2] http://www.eclipse.org
[3] http://java.sun.com/javase/technologies/database/

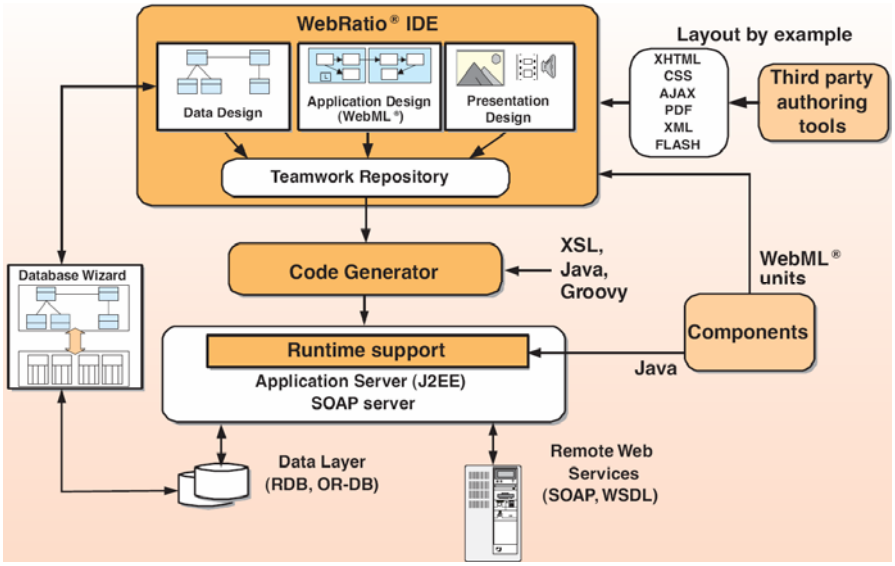


Abbildung 3.10: Infrastruktur von WebRatio [@WebR07]

Bewertung

Viele der in Kapitel 2.6 aufgestellten Anforderungen werden in WebML erfüllt. So können die *Units* mit Schlüssel-Wert-Paaren konfiguriert, Backend-Dienste können angebunden und transportierte Daten mit *Operation Units* aufbereitet werden. Außerdem fällt es leicht, erste Modelle zu entwerfen, da WebML eine *Domain Specific Language (DSL)* ist, demnach einzig für den Zweck der Modellierung von Web-Applikationen entworfen wurde. In der Tabelle 3.1 erfolgt die Bewertung aller Anforderungen, bezogen auf WebML. Die wichtigsten werden danach erläutert.

Tabelle 3.1: Bewertung von WebML

Bezeichnung	**Wertung**	**Bemerkungen**
1) Kommunikationskanäle	o	unterstützt mittels Links, jedoch nur 1:1-Beziehungen möglich
2) Rekursive Komposition	-	nicht vorgesehen
3) Konfiguration der Komponenten	+	über Schlüssel-Wert-Paare
4) Erweiterbarkeit	-	neue Units können nur in WebRatio hinzugefügt werden
5) Geringer Lernaufwand	+	leicht erlernbar, da DSL
6) Werkzeugunterstützung	-	nur WebRatio; außerdem kein XMI-Format
7) Separation of Concerns	+	gute Trennung
8) Kopplung an Backend-Dienste	+	können angebunden werden
9) Aufbereitung von Daten	+	mit *Operation Units* möglich
10) Sammeln von Kontextdaten	o	zu eingeschränkt, da nur Daten für die drei Subschemata gesammelt werden können
11) Kontrollfluss der Anwendung	o	Transitionen möglich, jedoch kein Datentransport
12) Adaptivität	o	modellierbar, jedoch *Content Units* nicht unabhängig von der Adaptivitätsmodellierung
13) Layout	+	mit dem aus Java bekannten GridLayout realisierbar
14) Angabe von Klassen	-	nicht vorgesehen
15) Einheitliches *Look & Feel*	-	nicht vorgesehen
16) Wiederverwendbarkeit	o	nicht ohne Aufwand, da kein XMI-Format

17) Technologieunabhängigkeit	+	keine frühe Bindung an Technologien
18) Testbarkeit	o	nicht ohne Aufwand, da proprietäres Format
19) Unterstützung von Standards	o	UML
20) Lesbarkeit	+	sehr gut lesbar und übersichtlich
21) Integrierbarkeit	o	WebRatio muss in die Werkzeug-Landschaft mit aufgenommen werden

Mit WebRatio wird zwar ausschließlich Java- und JSP-Code generiert, jedoch ist WebML selbst unabhängig von der Technologie der Zielplattform und des Glue-Codes. Durch Erweiterungen von WebRatio wäre es möglich, Code für jede erdenkliche Plattform zu generieren.

Negativ zu bewerten ist, dass *Areas* und *Pages* explizit als kontextsensitiv markiert werden, damit Adaptivität modelliert werden kann. Dadurch sind die *Content Units* nicht unabhängig von der Adaptivitätsmodellierung und die Wiederverwendbarkeit leidet, wenn das Modell in anderen Projekten benutzt werden soll und dort andere Adaptivitätsaspekte existieren. Adaptivität sollte lose modelliert werden, da die anzupassenden Elemente des Modells unabhängig davon sind. So sollte eine Änderung des Modells möglichst auch dann nicht erforderlich sein, wenn Adaptivität nicht angewendet wird.

Ein weiterer Nachteil von WebML ist, dass Kommunikationskanäle zwar unterstützt werden (repräsentiert als *contextual* und *noncontextual Links*), sie jedoch nicht ausdrucksstark genug sind, um Kanäle in UI-Mashups darzustellen. In WebML sind diese Kanäle immer nur 1:1-Beziehungen zweier Elemente, wohingegen es in UI-Mashups notwendig ist, auf einem Kanal mehrere Sender und Empfänger zuzulassen. Nachteilig ist weiterhin, dass zwischen *Pages* nur *noncontextual Links* modellierbar sind, wodurch es nicht möglich ist, beim Übergang von einer Seite zu einer anderen Daten zu transportieren. Außerdem können Übergänge nur durch manuelle Clicks ausgelöst werden und nicht allgemein durch Events innerhalb der Anwendung. Es ist somit nicht möglich, Übergänge zweier Seiten zu modellieren, ohne den Nutzer mit einzubeziehen.

Zusammenfassend kann man sagen, dass WebML sehr gut geeignet ist, um konventionelle Web-Applikationen zu modellieren. Dazu gehören auch komplexe Anwendungen, die jedoch in der Benutzerschnittstelle keine UI-Komponenten in dem Sinne enthalten, dass diese autonom und abgegrenzt sind. Um komponentenbasierte UI-Mashups zu modellieren, ist WebML nicht ausdrucksstark genug, könnte aber dazu benutzt werden, die UI-Komponenten für UI-Mashups zu entwickeln.

3.3 Objektorientierte Modellierung am Beispiel von UWE

Objektorientierte Methoden haben ihren Ursprung in der Modellierung mit der UML [SK03, S. 71]. Wie schon in Kapitel 3.1 erwähnt, werden damit Software und komplexe Systeme modelliert. Ganz nach Conallens Vorbild [Con99] wird die UML in objektorientierten Methoden um Profile erweitert, wodurch neue Stereotypen zur Verfügung stehen. Die bekanntesten heutigen Vertreter dieses Ansatzes sind *UML-based Web Engineering (UWE)*, *Object Oriented Web Solution (OOWS)* [PFPA06]

und *Object-Oriented Hypermedia (OO-H)* [GCP01]. Aufgrund der zahlreichen Veröffentlichungen zu UWE in den letzten Jahren und der Präsenz auf internationalen Konferenzen und Workshops [@UWE09] wird UWE als Repräsentant des objektorientierten Ansatzes gewählt und vorgestellt. Zwar wird OO-H in der Literatur als leistungsfähiger bewertet [DCBS09], jedoch wird es seit einigen Jahren nicht mehr weiterentwickelt. Außerdem ist OO-H aus WebML und UWE entstanden, weshalb es an dieser Stelle nicht als Repräsentant gewählt wurde.

UWE wurde am *Institut für Informatik* an der *Ludwig-Maximilians-Universität München* entwickelt. Die Entwickler legen großen Wert auf die Verwendung von etablierten Standards wie UML, den Erweiterungsmechanismus der Profile und die Anwendung der in Kapitel 2.5 dargestellten MDA-Prinzipien [KKWZ07]. UWE beinhaltet neben einem Vorgehensmodell auch das UWE-UML-Profil [KKZB08]. Es dient als DSL für Web-Applikationen und ist MOF-konform, wodurch UWE-Projekte nicht an eine IDE gebunden sind, sondern über XMI ausgetauscht werden können [KKK07]. In dem UWE-Profil sind für die meisten Stereotypen kleine Icons definiert, wodurch sie auch visuell in IDEs leicht zu erkennen sind. Im UWE-Prozess werden mehrere Modelle erzeugt, die teilweise automatisch aus den vorhergehenden generiert werden. In Abbildung 3.11 ist dieser Prozess dargestellt.

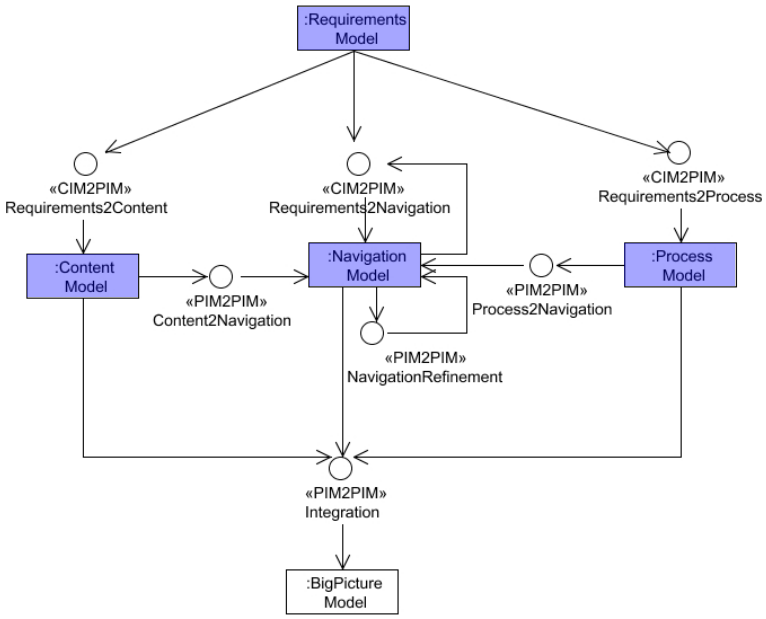


Abbildung 3.11: UWE-Prozess der Modelltransformationen [KKWZ07]

Hellblau unterlegte Modelle werden in den nächsten Abschnitten vorgestellt. Das *BigPicture Model* zählt nicht dazu, weil in diesem Modell alle anderen zu einem Zustandsdiagramm integriert werden und demnach konzeptuell keine neuen Erkenntnisse zu erwarten sind. Wichtig zu erwähnen ist jedoch, dass dieses Zustandsdiagramm anschließend mit dem Modelchecker Hugo/RT[4] verifiziert werden kann [KKWZ07]. Alle im Folgenden vorgestellten Modelle können mit OCL-Regeln konkretisiert werden.

[4]siehe `http://www.pst.ifi.lmu.de/projekte/hugo/`

Requirements Model

Dieses Modell dient in UWE als Ausgangspunkt und entspricht einem *UseCase*-Modell. Es enthält nicht nur Anforderungen, sondern berücksichtigt auch Navigation und Personalisierung. Demnach werden drei Arten von Anwendungsfällen unterschieden: *Navigation*, *Personalized* und *Process Use Cases*. *Navigation Use Cases* werden benutzt, um typisches Nutzerverhalten bei der Interaktion mit der Web-Applikation zu modellieren. Der dafür verwendete Stereotyp heißt `«navigation»`. *Personalized Use Cases* sind Anwendungsfälle, die von den Personalisierungsmechanismen des Systems beeinflusst werden, wie zum Beispiel, wenn in einem Musikportal in Abhängigkeit vom Nutzerverhalten unterschiedliche Empfehlungen angezeigt werden. Diese Anwendungsfälle werden mit dem Stereotyp `«personalized»` markiert. *Process Use Cases* repräsentieren Anwendungsfälle, in denen der Nutzer mit dem System interagiert und für gewöhnlich Transaktionen in der darunter liegenden Datenbank auslöst. Solche Anwendungsfälle entsprechen denen konventioneller Software und besitzen in UWE keinen Stereotyp. Alle Anwendungsfälle können UML-typisch um Aktivitätsdiagramme erweitert werden. Dadurch können sowohl detailliertere Semantik modelliert als auch die Anwendungsfälle in ihrem Verhalten konkretisiert werden.

Content Model

In diesem Modell werden die von der Web-Applikation zu verwaltenden Daten in UML-Klassendiagrammen visuell spezifiziert. Klassen sind dabei verfügbare Entitäten, und Assoziationen sind Beziehungen zwischen ihnen. Dabei wird meist zwischen Daten der zu modellierenden Domäne (*Content Model*) und Daten über den Nutzer (*User Model*) unterschieden. Die Nutzerdaten können später bei der Modellierung der Adaptivität eine Rolle spielen. Objekte, die in den Aktivitätsdiagrammen des *Requirements Models* verwendet werden, sind oft Kandidaten für das *Content Model*. Dieses Modell ist vergleichbar mit dem *Data Model* von WebML, weshalb hier auf eine Abbildung verzichtet wird.

Navigation Model

Ausgehend von *Requirements* und *Content Model*, wird in UWE ein Grundgerüst des *Navigation Models* generiert. Dieses basiert auf dem UML-Klassendiagramm und repräsentiert die Navigationsmöglichkeiten für den Nutzer. Assoziationen zwischen Klassen in diesem Modell sind immer gerichtet und stehen für die Navigationspfade, die der Nutzer am jeweiligen Knoten (Klasse) hat. Der Autor der Web-Applikation hat nun die Möglichkeit, das Modell um Klassen und Assoziationen zu erweitern. Dabei gibt es verschiedene Stereotypen, die appliziert werden können und bestimmte Metaphern von Web-Applikationen repräsentieren. Diese sind in Abbildung 3.12 dargestellt und werden in der folgenden Übersicht erläutert.

`«navigation»`: Repräsentiert den eigentlichen Inhalt aus dem *Content Model*. Für jede Klasse des *Content Models* wird in diesem Modell eine korrespondierende Klasse erzeugt.

`«menu»`: Wenn verschiedene Navigationspfade gewählt werden können, wird dieser Stereotyp verwendet.

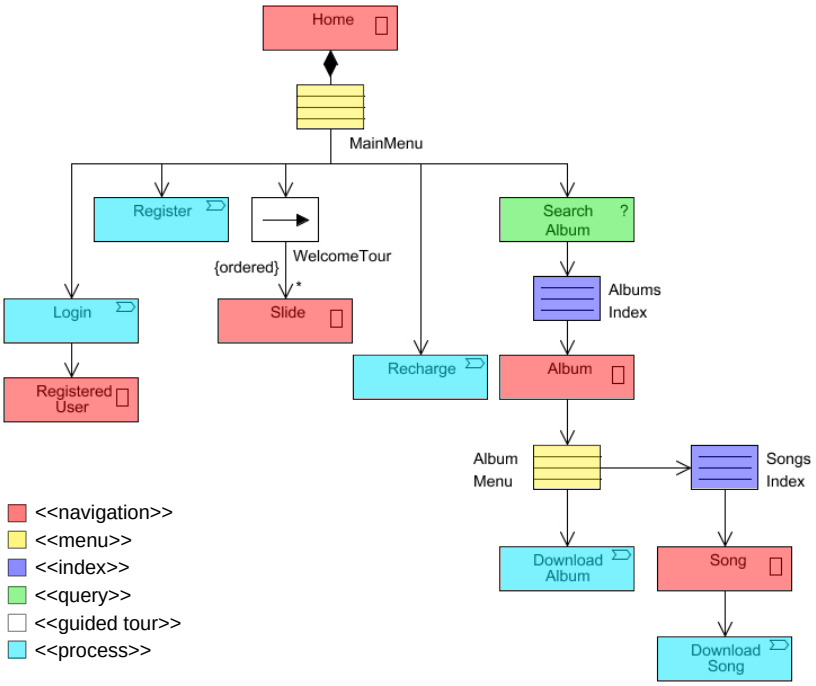


Abbildung 3.12: *Navigation Model* in UWE für ein Musikportal [KKWZ07]

«index»: Repräsentiert eine Menge von Instanzen einer Klasse des *Content Models*, weshalb eine Klasse dieses Stereotyps immer an eine «navigation»-Klasse gekoppelt ist.

«query»: Diese Klasse wird benutzt, um eine «index»-Menge einzuschränken.

«guided tour»: Stellt eine geführte Tour durch die damit verbundenen «navigation»-Klassen dar.

«process»: Enthält eine Web-Applikation Geschäftslogik, so muss diese in das *Navigation Model* integriert werden. Als Ein- und Austrittspunkt solcher Prozesse dient dieser Stereotyp. Die Logik selbst wird im Process Model spezifiziert.

Presentation Model

Dieses Modell stellt eine abstrakte Sicht der UI der Webanwendung dar. Das UML-Klassendiagramm dient auch hier wieder als Grundlage. In einem Transformationsschritt wird aus dem *Navigation Model* ein Grundgerüst dieses Modells generiert. Dabei wird für jede Klasse eine mit dem Stereotyp «presentation» korrespondierende erzeugt. Der Autor hat wieder die Möglichkeit, das Modell zu verfeinern, indem er zusätzliche Klassen mit Stereotypen für Bilder, Texte, Verweise, Knöpfe oder Formulare hinzufügt. Der Stereotyp «page» kann dabei auf Klassen appliziert werden, die einer Seite entsprechen. Damit können die Klassen der korrespondierenden Navigationsknoten verknüpft werden, da davon im Allgemeinen mehrere gleichzeitig auf einer Seite angezeigt werden. Das Layout einer Seite wird in einem *Concrete Presentation Model* mittels UML-Dependency-Relationen auf den Klassen modelliert, in denen andere Elemente enthalten sind. Abbildung 3.13 zeigt einen Ausschnitt eines

Concrete Presentation Models, in dem ein «text» in einer «presentationGroup» enthalten ist.

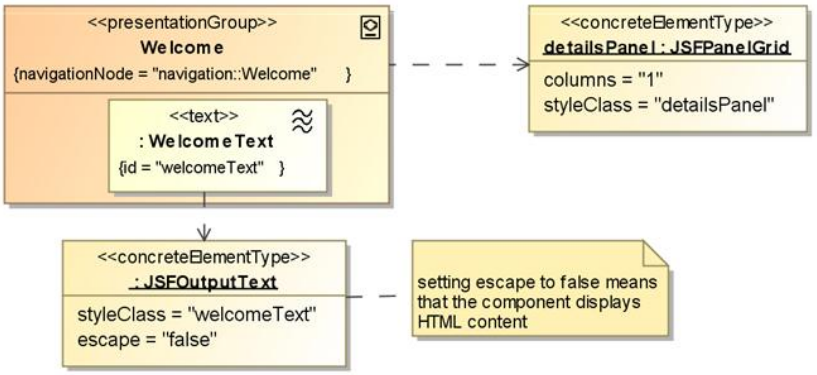


Abbildung 3.13: Ausschnitt eines UWE *Concrete Presentation Models* [Kro08]

Dependency-Relationen stehen für abgeschwächte Assoziationen zwischen Klassen. Im *Concrete Presentation Model* werden damit Abhängigkeiten zu Klassen modelliert, an denen der Stereotyp «concreteElementType» appliziert wurde [Kro08]. Dieser dient dazu, konkrete Attribute von den zu verwendenden Layout-Managern zu setzen, wie zum Beispiel `columns` eines JSF[5] PanelGrids.

Process Model

Das *Process Model* stellt eine Verfeinerung der «process»-Klassen des *Navigation Models* dar. Mittels des UML-Aktivitätsdiagramms wird die Geschäftslogik modelliert. Entitäten, die noch nicht im *Content Model* modelliert wurden, jedoch in einem Geschäftsprozess benötigt werden, müssen in einem neuen Klassendiagramm, dem *Process Structure Model*, modelliert werden. Mit diesem Modell wird der Datenfluss der Web-Applikation entworfen.

Adaptivität

In UWE sollen, abhängig von den Benutzerdaten aus dem *Content Model*, den Nutzungsdaten, wie Navigationshistorie, oder den Umgebungsdaten, wie benutzter Browser, die Präsentation der Inhalte oder die Navigationsstruktur angepasst werden können. Da die Anpassbarkeit mit den anderen Modellen eng verzahnt ist, wird in UWE eine Technik der aspektorientierten Adaptionsmodellierung verwendet, wodurch man einer zu engen Verzahnung mit den vorhandenen Modellen aus dem Weg geht. Dafür wurde der Stereotyp «aspect» entworfen, der dem UML-Element *Package* appliziert werden kann und einen «pointcut» und einen «advice» enthält. Eine «pointcut»-Klasse spezifiziert über ein Muster Elemente aus den anderen Modellen und bedeutet, dass diese angepasst werden sollen. Eine «advice»-Klasse spezifiziert die Veränderung, die eintreten soll. Gewöhnlich entsprechen solche Anpassungen dem Hinzufügen von zusätzlichen Eigenschaften an die spezifizierten Elemente in der «pointcut»-Klasse. Durch einen Aspektweber werden die so definierten Aspekte in die anderen Modelle integriert, wodurch sie ihre kompletten Eigenschaften erhalten.

[5] siehe `http://java.sun.com/javaee/javaserverfaces/`

Werkzeugunterstützung

Da das Metamodell von UWE ein UML-Profil darstellt, ist UWE prinzipiell in allen Werkzeugen nutzbar, die mit UML umgehen können, insbesondere Profile unterstützen. Die Entwickler von UWE haben allerdings für bekannte IDEs, für die bessere Unterstützung von UWE, Erweiterungen in Form von Plugins veröffentlicht. Zu den unterstützenden Entwicklungsumgebungen gehören das kommerzielle MagicDraw[6] und das freie Eclipse, sowie weitere, die allerdings nicht mehr weiterentwickelt werden. Das um UWE erweiterte MagicDraw wird von den Entwicklern MagicUWE genannt – dieser Begriff wird auch im Folgenden benutzt. Die Eclipse-Version heißt UWE4JSF und ist dafür gedacht, einen letzten Transformationsschritt aus schon vorhandenen Modellen zu Code durchzuführen. Die Modelle dafür werden mit MagicUWE erstellt. MagicUWE ist eine ausgereifte IDE für alle Arten von Softwareprojekten. Es bringt zahlreiche Editoren für die verschiedensten Modellarten mit, wie zum Beispiel UML-Modelle. Dadurch befindet man sich als Entwickler in einem gewohnten Umfeld und kann schnell UWE-spezifische Modelle erzeugen. Dies gilt für alle beschriebenen Modelle. Lediglich die Modellierung von Adaptivitätsaspekten ist noch nicht integriert. MagicUWE bringt mehrere *Model to Model (M2M)*-Transformationen mit, die in ATL[7] implementiert wurden. Zum einen kann das *Navigation Model* aus dem *Content Model* generiert werden und zum anderen *Presentation* und *Process Models* aus dem *Navigation Model*.

Bewertung

In der Tabelle 3.2 wird UWE anhand der Anforderungen aus Kapitel 2.6 bewertet. Das größte Manko bei UWE ist, dass der Vorteil der Werkzeugunabhängigkeit, der durch die Nutzung eines UML-Profils entsteht, wieder zunichte gemacht wird durch die permanente Erzeugung von redundanten Klassen in den unterschiedlichen Modellen. Diese redundanten Klassen stehen eigentlich in Verbindung zueinander, da sie dasselbe konzeptuelle Element der Web-Applikation repräsentieren. Jedoch besteht zwischen ihnen keine physische, sondern nur eine sehr schwache Verbindung, die durch Benutzung gleicher Bezeichnungen ausgedrückt wird. Dadurch muss das Werkzeug zusätzlich die Konsistenz der Modelle sicherstellen und beobachten, ob nicht Klassen in einem Modell umbenannt wurden und ihre korrespondierenden in anderen Modellen nicht.

Tabelle 3.2: Bewertung von UWE

Bezeichnung	**Wertung**	**Bemerkungen**
1) Kommunikationskanäle	o	nur Datenfluss im *Process Model*
2) Rekursive Komposition	-	nicht vorgesehen
3) Konfiguration der Komponenten	o	keine Benutzung von Komponenten, jedoch sind Modellelemente teilweise mit Dependency-Relation im *Presentation Model* konfigurierbar
4) Erweiterbarkeit	+	neue Stereotypen durch erweiterndes Profil möglich
5) Geringer Lernaufwand	+	sehr gut, da UML bekannt

[6] siehe http://www.magicdraw.com

[7] entstand im Zuge einer Auschreibung für eine Transformationssprache von der OMG– Details siehe http://www.eclipse.org/m2m/atl

6) Werkzeugunterstützung	+	aufgrund der Umsetzung als UML-Profil in jedem UML-Tool nutzbar
7) Separation of Concerns	+	gute Trennung
8) Kopplung an Backend-Dienste	-	nicht vorgesehen
9) Aufbereitung von Daten	-	nicht vorgesehen, da keine Komponenten modellierbar
10) Sammeln von Kontextdaten	o	zu eingeschränkt (ähnlich zu WebML)
11) Kontrollfluss der Anwendung	o	kein Datentransport bei Transitionen möglich
12) Adaptivität	+	theoretisch abgehandelt, jedoch nicht in den Tool-Erweiterungen umgesetzt
13) Layout	-	mit Dependency-Relationen möglich, aber für jede Zielplattform redundantes *Concrete Presentation Model* nötig
14) Angabe von Klassen	-	nicht vorgesehen
15) Einheitliches *Look & Feel*	-	im UWE-Profil nicht vorgesehen, aber im *Concrete Presentation Model* Angabe von CSS Stylesheets möglich
16) Wiederverwendbarkeit	o	Modelle enthalten hauptsächlich Low-Level-Elemente (Texte, Bilder, Buttons); dadurch sehr spezifisch und nur theoretisch wiederverwendbar
17) Technologieunabhängigkeit	o	Abhängigkeit entsteht erst durch *Concrete Presentation Model*
18) Testbarkeit	+	da UML die Basis, kann *Model Checking* durchgeführt werden (beispielsweise bei den *Process Models*)
19) Unterstützung von Standards	+	sehr gut (UML, MOF, XMI)
20) Lesbarkeit	-	durch immer neue Erzeugung von redundanten korrespondierenden Klassen, anstatt zu referenzieren, werden die Dateien sehr groß; auch die Applizierung der Stereotypen ist in der Datei nicht leicht ersichtlich, da nur mittels ID's
21) Integrierbarkeit	+	aufgrund von UML kaum Umstellung nötig und leicht integrierbar

Das Metamodell von UWE kann zwar um Stereotypen erweitert werden, ist jedoch trotzallem an das Metamodell der UML gebunden. Dadurch ist es nur möglich, Metaklassen zu erweitern, hingegen aber nicht, neue Assoziationen zu erstellen. Somit ist die UML in der Form nicht ausdrucksstark genug, um Konzepte von UI-Mashups mit dem leichtgewichtigen Erweiterungsmechanismus der Profile zu modellieren.

Abschließend kann man sagen, dass UWE auf dem Gebiet einfacher Web-Applikationen sehr gut einsetzbar ist. Für den Bereich der UI-Mashups jedoch überwiegen die Schwächen. Dennoch sind einige Ansätze von UWE generell gut geeignet und sollten beim zu entwickelnden Kompositionsmodell in Betracht gezogen werden. So ist die aspektorientierte Adaptivitätsmodellierung sehr gut geeignet, Adaptivität außerhalb der zu adaptierenden Elemente zu spezifizieren. Dadurch entsteht eine lose Adaptivitätsmodellierung, wie schon in der Bewertung von WebML erwähnt (siehe Abschnitt 3.2).

3.4 Komponentenorientierte Modellierung am Beispiel von mashArt

Komponentenorientierte Methoden übertragen den Ansatz der komponentenbasierten Entwicklung konventioneller Software ins Web. Dabei zielen sie nicht auf Web-Applikationen ab, die aus Texten, Bildern und Knöpfen bestehen, sondern auf *Rich Internet Applications (RIAs)*, die viel allgemeiner aus Komponenten bestehen. Diese können natürlich Elemente der nicht so abstrakten Stufe enthalten. Diese Anschauungsweise entspricht genau der von UI-Mashups und hat mit mashArt und CRUISe derzeit zwei Vertreter. Bis auf mashArt existieren keine weiteren veröffentlichten Ansätze [DCBS09]. Aus diesem Grund wird die komponentenorientierte Modellierung am Beispiel von mashArt vorgestellt und bewertet.

MashArt wird von mehreren Forschern aus Italien, Australien und den USA entwickelt. Es beruht grundsätzlich auf den Erkenntnissen des Mixup-Projekts [YBCD08, YBSP+07, YBC+07], das sich mit der Integration und Synchronisation von UI-Komponenten beschäftigt hat. Ausgehend von der Tatsache, dass mehr und mehr Web-Applikationen ihre UI als Komponenten bereitstellen oder offene Programmierschnittstellen (APIs) anbieten, führen die Entwickler an, dass eine universelle Integration, aufgrund der heterogenen Komponentenlandschaft, schnell an Bedeutung gewinnt [DCBS09]. So wird in mashArt, ähnlich wie in CRUISe, die Komposition von UI-Komponenten in gleicher Weise angegangen wie die Komposition von Daten oder Funktionalität [DCS+09]. Ein Hauptziel von mashArt ist die Einfachheit. Es sollen nicht nur Programmierer Web-Applikationen entwickeln können, sondern auch fortgeschrittene Web-Nutzer. Deshalb argumentieren die Entwickler, dass ein Kompositionsmodell so einfach wie möglich gehalten werden soll [YBCD08]. Im Gegensatz zu WebML und UWE beinhaltet das mashArt-Kompositionsmodell nur ein Modell. Da Kompositionen durch ihre Komponenten charakterisiert werden, wird zusätzlich ein Komponentenmodell benötigt, welches nachfolgend näher betrachtet wird.

mashArt-Komponentenmodell

MashArt-Komponenten halten UI-, Daten- und Funktionsservices transparent. Das dafür notwendige universelle Komponentenmodell ist in Abbildung 3.14 dargestellt. Eine Komponente besitzt dabei mehrere Abstraktionen, deren jede für die Interaktion mit anderen wichtig ist. Diese werden in der folgenden Auflistung beschrieben.

Zustand (*State Variable*): Besteht aus Schlüssel-Wert-Paaren, deren Änderung relevant für andere Komponenten sind. Beispielsweise sollte eine sich ändernde Farbe einer UI-Komponente nicht Bestandteil ihres Zustandes sein, wenn sie keine Auswirkung auf andere Komponenten hat.

Ereignisse (*Event*): Kommunizieren Zustandsänderungen nach außen. Wenn eine Komponente ihren Zustand ändert, wird ein Ereignis erzeugt, welches den neuen Zustand als Schlüssel-Wert-Paare enthält.

Operationen (*Operation*): Werden nach der Initiierung von Ereignissen aufgerufen. Sie müssen Parameter besitzen, die dem übertragenen Zustand entsprechen.

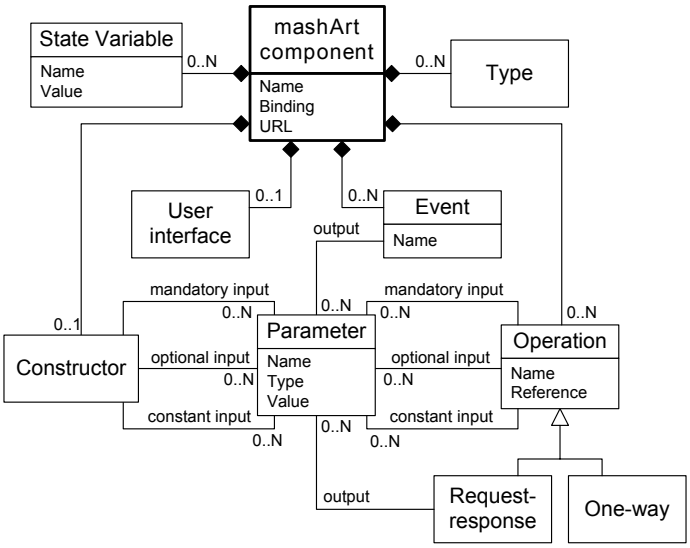


Abbildung 3.14: Metamodell der mashArt-Komponenten [DCBS09]

Konfiguration (*Constructor*): Beinhaltet beliebige Konfigurationsinformationen für die jeweilige Komponente. Diese können beispielsweise Informationen zum Layout sein oder Zugangsdaten zu Services.

Um Komponenten beschreiben zu können, wurde in mashArt ein abstrakter *Component Descriptor* eingeführt, welcher mit der *MashArt Description Language (MDL)* spezifiziert wurde. Mit der MDL können für konkrete Komponenten alle oben aufgeführten Eigenschaften beschrieben werden. Sie ist für mashArt-Komponenten, was die *Web Services Description Language (WSDL)*[8] für Webservices darstellt. Konkrete *Component Descriptors* werden im mashArt-Kompositionsmodell verwendet, um die Komponenten dort nicht mehr spezifizieren zu müssen, sondern nur noch zu referenzieren. Somit entsteht eine klare Trennung zwischen Komponenten- und Kompositionsentwickler.

mashArt-Kompositionsmodell

Da in mashArt ein universeller Ansatz verfolgt wird, wird dieses Modell *Universal Composition Model (UCM)* genannt. Dieses wird mit der *Universal Composition Language (UCL)* implementiert, welche das Metamodell für ein UCM darstellt und wofür online ein XML-Schema verfügbar ist[9]. Aufgrund der Einfachheit als Hauptziel von mashArt wird in diesem Modell ein Datenfluss-Ansatz verfolgt, um die Kommunikation zwischen den Komponenten zu beschreiben. Im Gegensatz zu variablenbasierten Ansätzen, wie sie in Programmiersprachen verwendet werden, ist dieser für Nichtprogrammierer leichter zu verstehen [DCBS09]. Die graphische Notation der Bestandteile einer Komposition ist in Abbildung 3.15 an einem Beispiel dargestellt. Darin ist beispielsweise zu sehen, dass das Ereignis `Process Selected` in der UI-Komponente `Process` erzeugt werden kann. Dies ist der Fall, wenn der Benutzer darin einen Click tätigt. Daraufhin wird die benachbarte UI-Komponente `Analysis` synchronisiert, indem ihre Operation `Show Analysis` aufgerufen wird. Außerdem

[8] eine Beschreibungssprache für Webservices – siehe http://www.w3.org/TR/wsdl
[9] siehe http://mashart.org/schema/ucl.xsd

wird synchron dazu die Service-Komponente `Repository` aktiviert. In der folgenden Auflistung werden die modellierbaren Konzepte eines mashArt-Kompositionsmodells beschrieben.

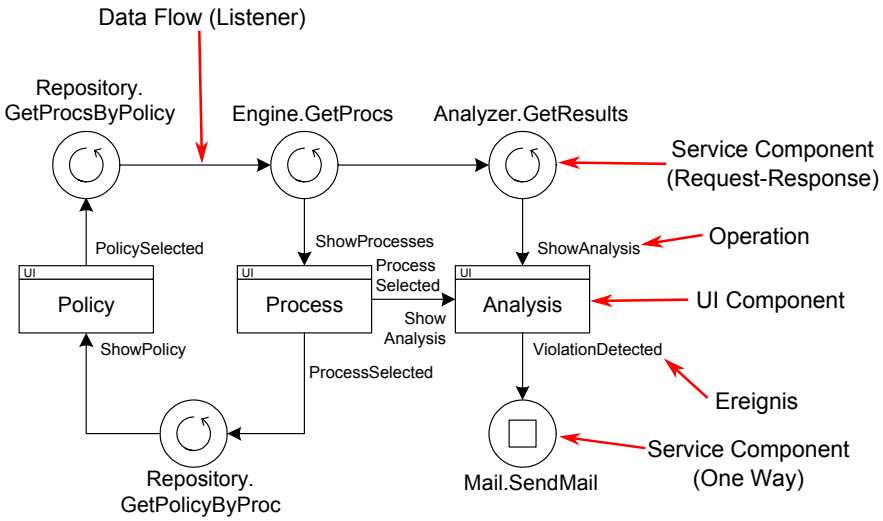


Abbildung 3.15: Beispiel eines mashArt-Kompositionsmodells [DCBS09]

Component Declaration: Komponenten werden anhand ihrer MDL-Deskriptoren deklariert. Optional können konkrete Werte für die Parameter ihrer Konstruktoren festgelegt werden.

Listeners: Verbinden Ereignisse mit Operationen und entsprechen dem Publish/Subscribe-Prinzip. Dadurch entsteht eine lose Kopplung zwischen Sender und Empfänger, so dass diese unabhängig voneinander sind. Im *Listener* werden Eingänge (Ereignisse) und Ausgänge (Operationen) miteinander über die Namen der Parameter verbunden. Dadurch entsteht der *Data Flow*. Zusätzlich können am *Listener* Bedingungen als XPath[10]-Ausdruck über die Eingangs-Parameter definiert werden. Nur wenn diese Bedingungen wahr sind, werden die Daten kommuniziert.

Type Definitions: Komplexe Datentypen für Parameter können hier definiert werden.

Komplexere Datentransformationen, wie beispielsweise das Auftrennen oder Zusammenführen von Daten, sind derzeit nicht modellierbar, können jedoch beispielsweise durch eigene Services, implementiert als mashArt-Komponente, oder als BPEL-Prozess, welcher in einer Service-Komponente aufgerufen wird, nachgerüstet werden.

Werkzeugunterstützung

Im mashArt-Projekt wurde eine Web-basierte Plattform entwickelt, mit der es möglich ist, Kompositionen zu erstellen. Diese heißt *mashArt Editor* und läuft im Browser des Nutzers. Verfügbare Komponenten werden in eine Registry aufgenommen und stehen dem Autor dann zur Verfügung. Für den Editor gilt wieder das Prinzip

[10] eine Abfragesprache für XML-Dokumente – siehe http://www.w3.org/TR/xpath

der Einfachheit, weshalb lediglich Komponenten auf eine Bühne gezogen und die Verbindungen zwischen ihnen gesetzt werden müssen. Weiterhin kann das Layout der Web-Applikation hier mittels HTML-Templates definiert werden. Es gibt vordefinierte Layouts, auf denen die ausgewählten Komponenten abgelegt werden können. Nach der Fertigstellung einer Komposition wird diese in der Registry gespeichert und erhält eine eigene Adresse, über die sie dann online verfügbar ist.

Bewertung

Der komponentenorientierte mashArt-Ansatz erfüllt viele der in Kapitel 2.6 aufgestellten Anforderungen. So sind die Spezifikation von Kommunikationskanälen oder die rekursive Komposition mit mashArt möglich. In Tabelle 3.3 erfolgt die Bewertung aller Anforderungen.

Tabelle 3.3: Bewertung von mashArt

Bezeichnung	**Wertung**	**Bemerkungen**
1) Kommunikationskanäle	+	über Publish/Subscribe mit Ereignissen und Operationen
2) Rekursive Komposition	+	erzeugte Kompositionen können gespeichert und in neuen wiederverwendet werden
3) Konfiguration der Komponenten	+	über Schlüssel-Wert-Paare
4) Erweiterbarkeit	+	das Metamodell ist erweiterbar, da es ein XML-Schema darstellt, welches erweitert werden kann
5) Geringer Lernaufwand	+	Kompositionen sehr einfach zu erstellen
6) Werkzeugunterstützung	-	derzeit nur online auf der mashArt-Plattform nutzbar
7) Separation of Concerns	+	klare Trennung zwischen Komponentenentwickler und Kompositionsentwickler
8) Kopplung an Backend-Dienste	+	mittels *Service Components*
9) Aufbereitung von Daten	o	derzeit nur über den Umweg von Wrapper-Komponenten oder angebundenen BPEL-Prozessen möglich
10) Sammeln von Kontextdaten	-	nicht vorgesehen
11) Kontrollfluss der Anwendung	-	nicht vorgesehen
12) Adaptivität	-	nicht vorgesehen
13) Layout	-	im Metamodell nicht enthalten, aber im mashArt-Editor mittels HTML-Templates möglich
14) Angabe von Klassen	-	nicht vorgesehen
15) Einheitliches *Look & Feel*	-	siehe Layout
16) Wiederverwendbarkeit	+	sehr gut umgesetzt — Modelle und Komponenten können unabhängig voneinander wiederverwendet werden
17) Technologieunabhängigkeit	+	durch die UCL sehr gut umgesetzt
18) Testbarkeit	o	dadurch, dass nur online komponiert werden kann, ist externes Testen nicht möglich — die mashArt-Plattform bringt jedoch ein Monitoring- und Analyse-Werkzeug mit, womit zumindest zur Laufzeit getestet werden kann
19) Unterstützung von Standards	o	XML-Schema
20) Lesbarkeit	+	nur graphische Notation für den Nutzer verfügbar, diese ist jedoch sehr intuitiv und leicht verständlich

21) Integrierbarkeit	o	eine Integration ist nicht trivial, da die mashArt-Plattform benutzt werden muss

Der Ereignis-Operation-Ansatz der Kommunikation entspricht sehr stark der Anforderung 1 und beinhaltet zusätzlich die Möglichkeit, Bedingungen mittels XPath anzugeben. Obwohl dieser Ansatz vielversprechend ist, stellt sich die Frage, wie fortgeschrittene Web-Nutzer derartige Bedingungen modellieren können. Da mashArt auf Einfachheit auf Modellebene setzt, erfordert die einfache Verwendung von XPath Unterstützung durch das Werkzeug. Diese ist im *mashArt Editor* jedoch nicht ersichtlich. Es ist nicht davon auszugehen, dass Nichtprogrammierer der XPath-Syntax und -Semantik mächtig sind, da hierfür ein Mindestmaß an Kenntnissen von Programmiersprachen vorausgesetzt wird.

Eine große Schwäche von mashArt ist, dass die Modellierung von Adaptivität gar nicht berücksichtigt wurde, stellt sie doch einen wichtigen Aspekt für UI-Mashups dar. Die Bereitstellung einer Online-Kompositionsplattform ist aufgrund der Konzentration auf Nichtprogrammierer als Zielgruppe ein nachvollziehbarer Schritt. Nachteilig jedoch ist, dass die für Entwickler gewohnten IDEs, alternativ zur Plattform, nicht benutzt werden können.

Des Weiteren ist der Ansatz der Einfachheit des Kompositionsmodells nicht ausreichend, um alle wichtigen Aspekte von UI-Mashups zu modellieren. Als Folge dessen wurde der Kontrollfluss gar nicht berücksichtigt, wodurch es nicht möglich ist, unterschiedliche Sichten auf die Web-Applikation darzustellen.

3.5 Fazit

In den vorangegangenen Kapiteln wurden drei verschiedene Ansätze der modellbasierten Komposition von Web-Applikationen vorgestellt und deren Stärken und Schwächen analysiert. So hat WebML die Modellierung des Layouts sehr gut umgesetzt, da technologieunabhängig ein an das GridLayout von Java angelehnter Ansatz benutzt wird. Darüber hinaus können Daten sehr einfach mit Hilfe der *Operation Units* transformiert werden, weshalb diesbezüglich Überlegungen in das folgende Konzept mit einfließen sollten. UWE hingegen hat großes Potential in der Technologieunabhängigkeit, da dieser Ansatz einzig auf UML-Profilen und der Transformation anhand von Stereotypen beruht. Außerdem entspricht der aspektorientierte Ansatz der Adaptivitätsmodellierung der Natur von Adaptivität, dahingehend, dass sie sich durch alle Ebenen der Modelle ziehen kann. Dieser Ansatz sollte im zu entwickelnden Konzept Verwendung finden. Der komponentenbasierte Ansatz von mashArt ist, bezüglich der Ausrichtung auf Komponenten in UI-Mashups und nicht auf Low-Level-Elemente, als stärkster zu werten. Er setzt die wichtigsten Anforderungen von der Spezifikation der Kommunikationskanäle, über rekursive Komposition, Konfigurierbarkeit der Komponenten bis hin zur Erweiterbarkeit sehr gut um. Die mashArt-Konzepte bauen am meisten auf den Eigenschaften von Komponenten auf, weshalb sie in dieser Arbeit aufgegriffen werden sollten. Dennoch sind die Vorteile in ihrer Gesamtheit noch nicht zufriedenstellend, da nicht alle Anforderungen abgedeckt werden oder sie nur bedingt im Bereich von UI-Mashups eingesetzt werden können. So wurden beispielsweise in keinem der Ansätze die Angabe von Klassen,

um Arten von UI-Komponenten anzugeben und sich von konkreten Komponenten zu lösen, oder ein einheitliches *Look & Feel* berücksichtigt. Ersteres spielt bei WebML und UWE keine Rolle, da diese Methoden nicht komponentenorientiert sind, weshalb keine Austauschbarkeit von Komponenten gewährleistet zu werden braucht. Bei mashArt allerdings wären Klassen sinnvoll, da man dadurch von konkreten Komponenten unabhängig wird und diese somit austauschbar sind. Auch die Modellierung des Kontrollflusses wird von bestehenden Ansätzen nur ungenügend umgesetzt. Der größte Nachteil hierbei ist, dass entweder bei Transitionen zwischen Sichten der Web-Applikation keine Daten transportiert werden können, oder dass Transitionen nur von einem Nutzer der Web-Applikation, und nicht allgemein von Events, ausgelöst werden können. Insgesamt konnten in diesem Kapitel viele Erkenntnisse gewonnen werden, auf die im Folgenden aufgebaut werden wird und weitere Ideen generiert werden.

4 Konzeption eines generischen Kompositionsmodells

Im vorangegangenen Kapitel wurden existierende Ansätze zur Modellierung von Web-Applikationen vorgestellt und bezüglich der in Kapitel 2.6 aufgestellten Anforderungen bewertet. Da keines der vorgestellten Konzepte allen Anforderungen gerecht wird, wird aus den dort gewonnenen Erkenntnissen und vorgestellten Konzepten in diesem Kapitel ein generisches Kompositionsmodell für UI-Mashups konzipiert.

Dafür wird zunächst das Komponentenmodell von CRUISe vorgestellt, dem alle Komponenten unterliegen. Daraus ergibt sich, welche konkreten Eigenschaften mit dem zu entwerfenden Kompositionsmodell modellierbar sein müssen. Weiterhin wird dargestellt, welche konzeptuellen Modelle in UI-Mashups sinnvoll sind und was sie beinhalten müssen. Darauf aufbauend wird erläutert, welche Modelltransformationen durchgeführt werden können und inwiefern sie sich in den klassischen MDA-Prozess (siehe Kapitel 2.5) der Softwareentwicklung eingliedern. Abschließend wird auch das hier entworfene Konzept bezüglich der Anforderungen bewertet.

4.1 CRUISe Komponentenmodell

Das CRUISe-Komponentenmodell stellt, ähnlich wie in mashArt, eine Abstraktion aller in ein UI-Mashup integrierbaren Komponenten dar. Demzufolge ist es universeller Natur und muss deren Aufbau, Struktur und die Schnittstellen beschreiben.

CRUISe-Komponenten können einen internen Zustand haben. Dieser kann sich beispielsweise durch die Ausführung bestimmter Funktionen oder durch Nutzerinteraktion (bei UI-Komponenten) ändern. Sind diese Zustandsänderungen relevant für andere Komponenten, dann werden dafür Ereignisse (*Events*) erzeugt und können so nach außen propagiert werden. Ereignisse transportieren neben ihrem Namen zusätzlich beliebige Daten. Damit Ereignisse ein Zusammenwirken zwischen Komponenten hervorrufen können, besitzen die CRUISe-Komponenten außerdem Operationen (*Operations*). Sie können wie bei mashArt durch die Initiierung von Ereignissen aufgerufen werden und führen dann interne Funktionen aus. Auch zustandslose Komponenten verfügen über Ereignisse und Operationen. Eine Trennung bezüglich des Zustandes der Komponenten muss jedoch nicht durchgeführt werden, denn es ergeben sich keine Unterschiede in der Modellierung, da die Zustände unerheblich für das UI-Mashup sind. Das Metamodell der CRUISe-Komponenten ist in Abbildung 4.1 schematisch dargestellt und wird nachfolgend weiter beschrieben.

Wie in Abbildung 4.1 zu sehen, können Operationen und Ereignisse getypte Parameter besitzen. So ist es möglich, dass Komponenten komplexe Daten von anderen Komponenten verarbeiten können. Ein Unterschied zu mashArt ist das Konzept der Konfiguration (*Configuration*) der Komponenten. Die Konfiguration wird benutzt,

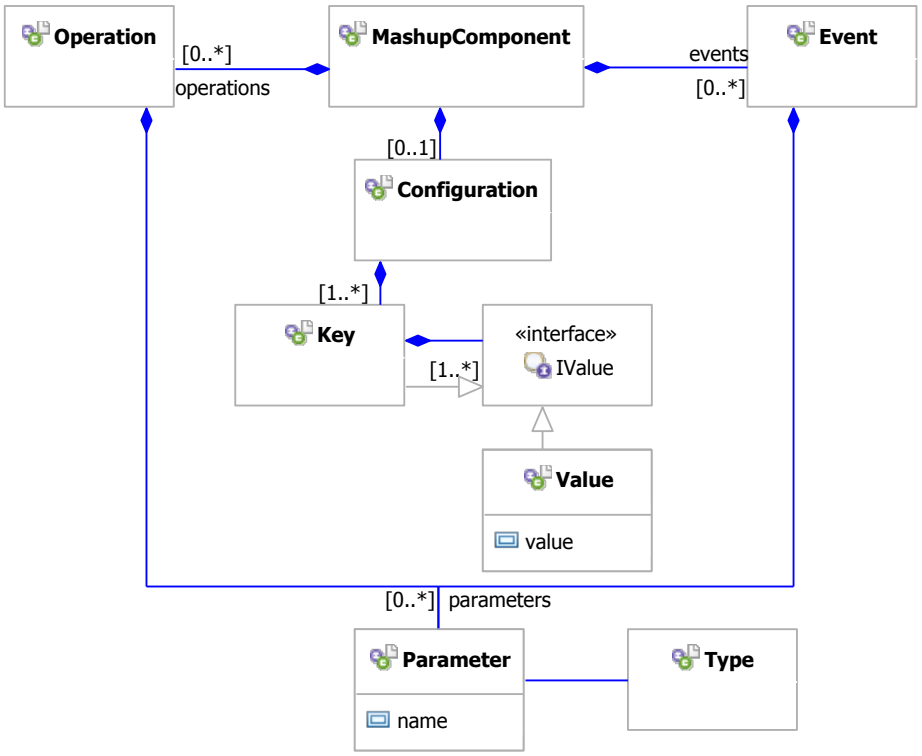


Abbildung 4.1: Komponentenmodell von CRUISe

um den initialen Zustand einer Komponente festzulegen. In CRUISe wird hierbei ein allgemeinerer Ansatz verfolgt. Es ist möglich, über Schlüssel-Wert-Listen, Schlüssel-Wert-Bäume oder durch Kombination beider Arten Komponenten zu konfigurieren. Der Normalfall der Schlüssel-Wert-Paare wird dadurch mit abgedeckt. Die Schlüssel werden vom Entwickler der Komponente festgelegt und können somit flexibel belegt werden. Durch diese Technik ist es möglich, Komponenten mit gewohnten Datenstrukturen von Programmiersprachen beziehungsweise mit verschachtelten Daten zu initialisieren. Dadurch können beispielsweise Listen zur Konfiguration von Komponenten benutzt werden, wie auch in Abbildung 4.2 bei der Komponente *MapViewer* zu sehen.

Ereignisse und Operationen sind im vorgestellten Komponentenmodell die Kernkonzepte für die Kommunikation. Diese verläuft in der CRUISe-Infrastruktur dezentral in Form des Publish/Subscribe-Prinzips. Die Steuerung der Kommunikation übernimmt die Runtime, die als Vermittler zwischen den Komponenten agiert. In der Arbeit [Wal09] wurde die *CRUISe Client Runtime* entwickelt, die den Server entlastet und die Kommunikation auf dem Client ausführt. Neben Ereignissen der Komponenten können in CRUISe auch Ereignisse der Laufzeitumgebung selbst ausgelöst werden. Ein Beispiel dafür wären Fehlerfälle. Komponenten können für solche Ereignisse registriert werden und bei Eintreten darauf reagieren.

Für das abstrakte CRUISe-Komponentenmodell gibt es mehrere Komponenten, die für bestimmte Aufgaben spezialisiert wurden. Diese werden im Folgenden erläutert.

User Interface Components: Die Aufgabe dieser Komponenten ist es, grafische Elemente in der UI darzustellen. Im Allgemeinen sind dies keine Low-Level-

Elemente, wie beispielsweise Bilder oder Buttons, sondern in sich abgegrenzte, eigenständige kleine Web-Applikationen. Als Beispiele seien das Anzeigen von Daten in einer Tabelle oder die Navigation durch einen Baum zu nennen. Aufgrund des serviceorientierten Ansatzes von CRUISe werden *User Interface Components* durch UISs bereitgestellt und durch die Runtime integriert. Dadurch, dass die UISs in eine Registry aufgenommen werden und mehrere gleichartige (beispielsweise unterschiedliche Komponenten für geographische Karten) *User Interface Components* existieren können, ist es in CRUISe möglich, UICs nicht nur über die Adresse ihres Services zu integrieren. So kann eine Klasse angegeben werden, welche die zu integrierende UI-Komponente charakterisiert. Wie schon im Kapitel 2.3 beschrieben, wird dann die am besten passende Komponente aus der angegebenen Klasse bestimmt und integriert. Die entsprechende Klassifikation wird, wie oben schon erwähnt, parallel in der Arbeit [Bau09] entwickelt.

Context Components: Diese Komponenten dienen den Adaptionsprozessen eines UI-Mashups. Sie wirken als Sensoren für UI-Mashups und liefern Kontextdaten, anhand derer Adaptionsprozesse initiiert oder Events ausgelöst werden können. Es können, im Gegensatz zu WebML und UWE, beliebige Kontextdaten erzeugt werden, ohne dass der Autor an bestimmte Schemata gebunden ist. Beispielsweise kann anhand der geographischen Position des Nutzers ermittelt werden, an welchem Ort er sich befindet und, in Abhängigkeit davon, können aktuelle Wetterdaten an eine Klima verarbeitende Komponente gesendet werden.

Logic Components: Wie schon in den Anforderungen beschrieben (siehe Anforderung 9), dienen *Logic Components* der Weiterverarbeitung von Daten anderer Komponenten. Dazu zählen unter anderem die Adaption, Transformation, Filterung oder Aggregation. Beispielsweise kann es erforderlich sein, Daten aus verschiedenen Datenströmen zu vereinen. Dies wäre der Fall, wenn eine Komponente den Vornamen einer Person liefert, eine andere den Nachnamen und eine dritte aber die Kombination beider Daten benötigt. *Logic Components* sind wichtig, da Komponenten von unterschiedlichen Komponentenentwicklern implementiert werden können und demnach kein gemeinsames Datenmodell besitzen müssen. In solchen Fällen sind *Logic Components* in Form von Adaptern sehr hilfreich.

Service Access Components: Um beliebige Backend-Dienste in einem UI-Mashup bereitzustellen, werden *Service Access Components* benutzt. Sie kapseln den Zugriff auf externe Dienste und können demnach als Datenlieferanten oder zur Integration von Anwendungslogik fungieren. Es können nicht nur Webservice, sondern auch BPEL-Prozesse und Nachrichtenfeeds angebunden werden. Demnach kann in *Service Access Components* auch die Adresse zur Beschreibung des jeweiligen Services (wie beispielsweise die WSDL-Datei eines SOAP-Services) angegeben werden. Auf diese Weise können anhand der Servicebeschreibung alle Eigenschaften der Komponente generiert werden, da diese durch ihre Beschreibung eindeutig definiert sind.

Durch diese Komponentenarten ist in UI-Mashups eine Unterteilung in konzeptionelle Schichten möglich. Die Schichten werden *UserInterface*, *Context*, *Logic* und *Service Access* genannt. In Abbildung 4.2 sind diese an einer Beispielkomposition dargestellt.

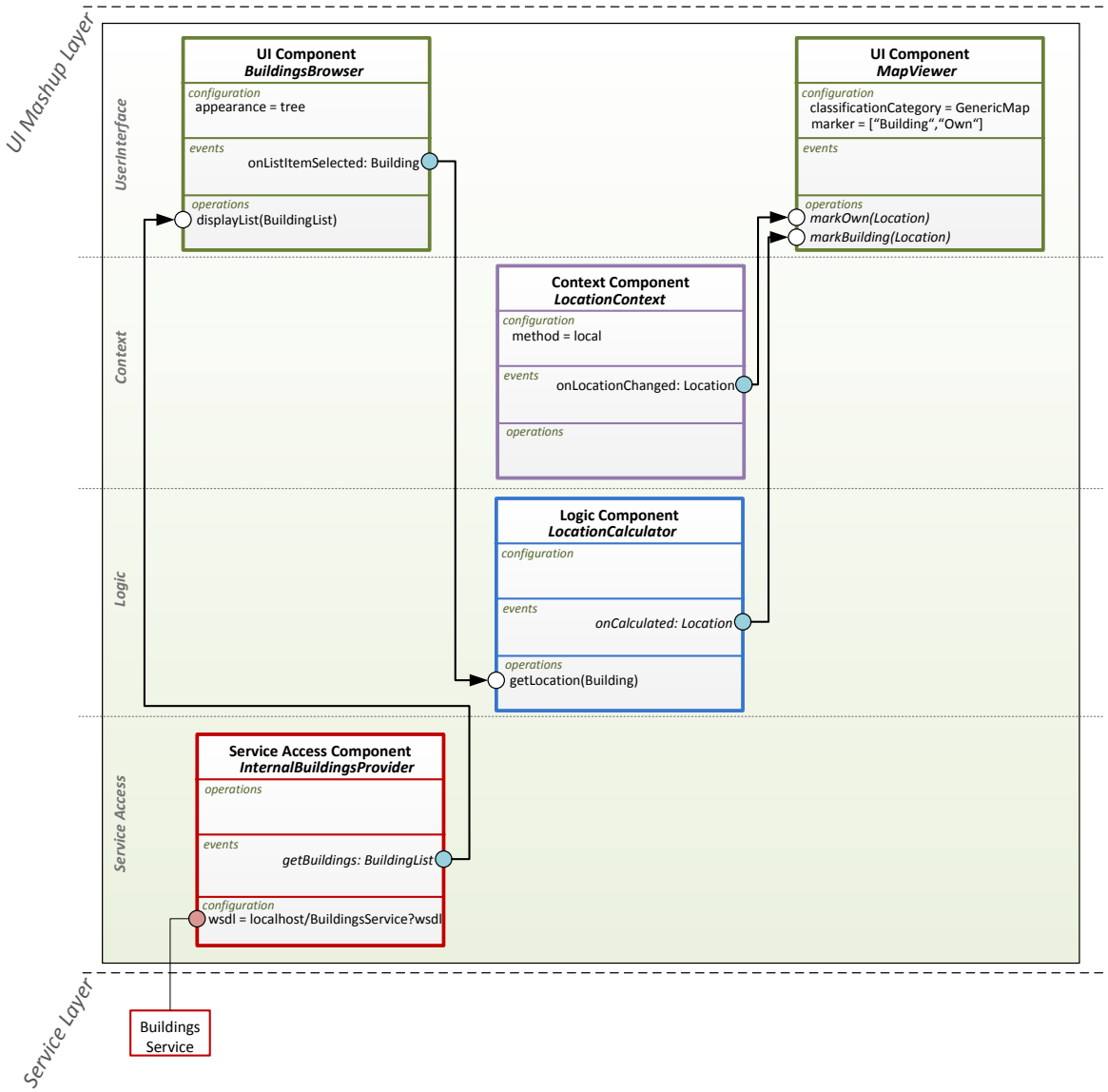


Abbildung 4.2: Schichten und Kommunikation in CRUISe

Darin ist außerdem die Kommunikation dargestellt, wie sie in CRUISe abgebildet wird. Hellblaue Kreise auf den rechten Seiten der Komponenten stehen für Ereignisse und weiße Kreise auf den linken Seiten für dadurch aufgerufene Operationen. Die Pfeile symbolisieren demnach die Kommunikation zwischen den Komponenten. Darüber hinaus ist in der UIC *MapViewer* mit dem Schlüssel `marker` dargestellt, wie die Konfiguration mittels einer Schlüssel-Wert-Liste aussehen könnte.

Ergänzend zu den UISs sei noch erwähnt, dass für diese innherhalb des CRUISe-Projekts ein Beschreibungsformat entwickelt wird. Die Beschreibung wird mit der *User Interface Service Description Language (UISDL)* durchgeführt und beinhaltet

alle Bestandteile des Komponentenmodells. Die UISDL ist für UISs das Pendant zur WSDL für Webservices. Als erster Meilenstein für dieses Beschreibungsformat sei das CRUISe Deliverable 6: »Spezifikation einer semantischen Beschreibungssprache für UIS« genannt. Anhand der UISDL-Beschreibung können alle *Event*s und *Operation*s einer Komponente in einem konkreten Kompositionsmodell automatisch modelliert werden, da Komponenten mit ihrer Beschreibung eindeutig bestimmt sind.

Das in diesem Kapitel vorgestellte Komponentenmodell muss im zu entwickelnden Konzept vollständig unterstützt werden. Im folgenden Kapitel werden weiterführende Überlegungen bezüglich der notwendigen Modelle angestellt.

4.2 Modelle

Im Kapitel 3 wurden drei modellgetriebene Ansätze vorgestellt. WebML und UWE unterteilen ihrerseits den Entwicklungsprozess in mehrere Modelle, während in mashArt nur ein Modell vorgesehen ist. Die ersten beiden Methodiken verfolgen dabei das Prinzip der Trennung der Verantwortlichkeiten (SoC). MashArt hingegen legt den Fokus auf Einfachheit für den Autor in Gestalt eines fortgeschrittenen Web-Nutzers. Wie in der Bewertung von mashArt (siehe Abschnitt 3.4) aber schon erwähnt, fehlen einige wesentliche Anforderungen (Adaptivität, Kontrollfluss). Auf diese Weise enthält das mashArt-Kompositionsmodell lediglich die zu komponierenden Komponenten und zusätzliche *Listener* für die Kommunikation. Weitere Konzepte würden dazu führen, dass unterschiedliche Aspekte ineinander greifen, wenn sie nicht in verschiedene Modelle ausgelagert werden. Im Gegensatz zum Prinzip der Einfachheit auf Modellebene von mashArt [YBCD08] wird hier das Prinzip der Vollständigkeit verfolgt. Das Kompositionsmodell muss alle Anforderungen aus Kapitel 2.6 abdecken, um ein hohes Maß an Ausdrucksstärke zu erlangen. Die Einfachheit der Modellierbarkeit für den Autor muss von der verwendeten IDE sichergestellt werden. Nachfolgend wird erläutert, welche Modelle das Kompositionsmodell enthalten wird und welchen Aspekten von UI-Mashups sie entsprechen.

In Kapitel 2.6 sind elf Anforderungen funktionaler Natur formuliert: Anforderungen 1, 2 und 3, sowie 8 bis 15. Diese beschreiben, was das Kompositionsmodell konkret beinhalten muss. Alle anderen Anforderungen sind nichtfunktionaler Natur und müssen implizit in die funktionalen mit einfließen. In Abbildung 4.3 wird jede funktionale Anforderung anhand des zugehörigen Kriteriums von UI-Mashups in ein Modell eingeordnet. Diese Einordnung wird nachfolgend begründet.

Die zu verwendenden Komponenten stellen den wichtigsten Aspekt dar. Die Komponenten und ihre Eigenschaften werden in einer Komposition in einem ersten Modell spezifiziert. Dieses bekommt den Namen *Conceptual Model* und wird in Kapitel 4.2.2 erläutert. Es enthält die meisten Anforderungen und ist der Kern eines UI-Mashups, da sich alle anderen Modelle darauf beziehen. Im *Conceptual Model* können Klassen für UI-Komponenten angegeben und Komponenten allgemein konfiguriert werden. Damit werden Anforderungen 14 und 3 umgesetzt. Weiterhin können mit den im vorherigen Kapitel beschriebenen Komponentenarten Backend-Dienste in ein UI-Mashup eingebunden, kommunizierte Daten aufbereitet und Kontextdaten gesammelt werden, wodurch die Anforderungen 8, 9 und 10 abgedeckt werden. Des Weiteren können UI-Mashups wieder zu UI-Komponenten abstrahiert werden und in neue Kompositionen einfließen, womit rekursive Komposition (siehe Anforderung

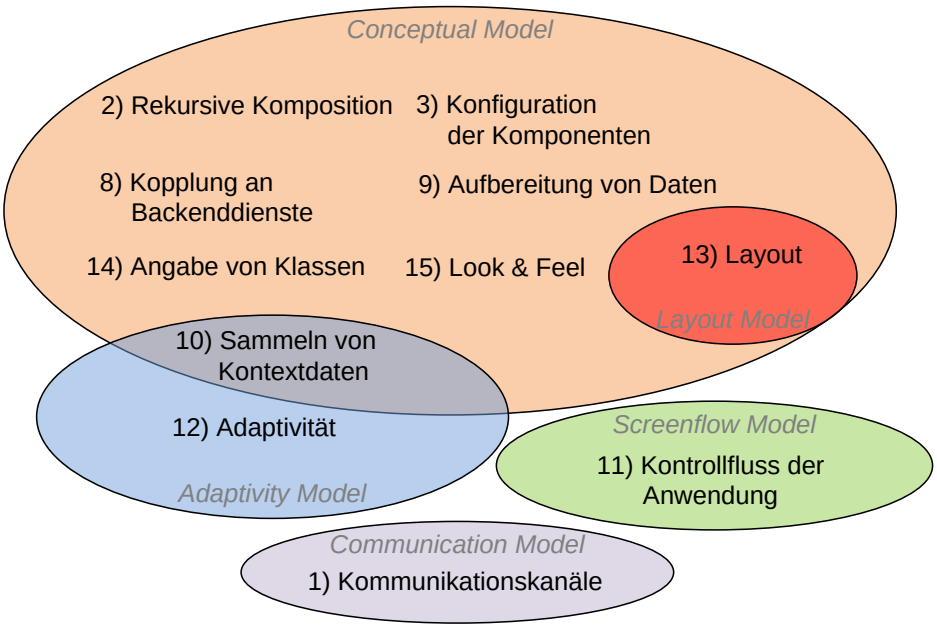


Abbildung 4.3: Modelle des Kompositionsmodells

2) ermöglicht wird. Abgerundet wird dieses Modell mit der Spezifizierbarkeit des Aussehens der UI-Komponenten. Dadurch wird Anforderung 15 erfüllt.

Das Layout von UICs, beispielsweise die Angabe der Abmessungen, wird vom Komponentenentwickler bestimmt und gegebenenfalls über die äußere Schnittstelle konfigurierbar gehalten, weshalb es Teil des *Conceptual Models* ist. Das Layout des gesamten UI-Mashups jedoch benötigt ein eigenes Modell. Dieses wird *Layout Model* genannt und stellt die Unabhängigkeit der Komponenten vom Layout der Anwendung sicher. Im Gegensatz zum mashArt-Ansatz soll der Layout-Aspekt mit ins Metamodell aufgenommen werden, da nur so Unabhängigkeit gegenüber der Zielplattform eines UI-Mashups sichergestellt werden kann. Außerdem soll das *Layout Model* im Gegensatz zum UWE-Ansatz als PIM modelliert werden, da so die Wahl von Zielplattform und Framework erst so spät wie möglich, zum Zeitpunkt der Generierung der finalen Web-Applikation, getroffen werden muss. Damit erfüllt das *Layout Model* Anforderung 13. Es wird im Kapitel 4.2.3 erläutert.

Ein weiteres Modell beschreibt den Kontrollfluss des UI-Mashups und wird *Screenflow Model* genannt. Durch den Kontrollfluss können dem Benutzer in Reaktion auf eintretende Ereignisse neue Sichten auf das UI-Mashup eröffnet werden. Dieser Aspekt wurde in mashArt gar nicht berücksichtigt, soll aber hier unterstütz werden, da auf diesem Weg unterschiedliche Sichten der Web-Applikation dargestellt werden können, beispielsweise als Ergebnis von Adaptionsprozessen. Des Weiteren können sich durch den Kontrollfluss die Möglichkeiten für UI-Mashups, bezüglich ihres Interaktionspotentials mit dem Nutzer, um ein Vielfaches erhöhen, da so nicht nur statische Web-Applikationen entstehen können, die nur aus einer Sicht für den Benutzer bestehen. Das *Screenflow Model* deckt somit Anforderung 11 ab und wird im Kapitel 4.2.4 konzipiert.

Die Kommunikation zwischen den Komponenten wird im *Communication Model* spezifiziert. Im Gegensatz zum mashArt-Ansatz werden Kommunikation und Komponenten in getrennte Modelle gelegt, da dadurch sowohl SoC als auch die Wie-

derverwendung der einzelnen Modelle vorangetrieben wird. Somit können sowohl ein *Conceptual Model*, unabhängig von der Kommunikationsspezifikation, wiederverwendet, als auch neue Kommunikationskanäle deklariert werden. Das *Communication Model* erfüllt Anforderung 1 und wird im Kapitel 4.2.5 ausgearbeitet.

Das fünfte Modell ist das *Adaptivity Model*. Dieses spielt eine wichtige Rolle, denn mit seiner Hilfe erhält ein UI-Mashup Dynamik und Flexibilität, indem auf verschiedene Kontexte reagiert werden und sich die Web-Applikation, in Abhängigkeit davon, anders verhalten kann. Im Gegensatz zu WebML soll jedoch der Autor vollkommen frei in der Wahl der zu adaptierenden Elemente des UI-Mashups sein, weshalb sich das *Adaptivity Model* am UWE-Ansatz orientiert. Wenn der Adaptivitäts-Aspekt in einem eigenen Modell spezifiziert werden kann, bleiben andere Modelle davon unberührt, wodurch die Wiederverwendbarkeit erhöht wird. In diesem Modell werden Adaptionsszenarien modelliert, die in Abhängigkeit von gesammelten Kontextdaten initiiert werden sollen. Neben Anforderung 12 fließt somit auch Anforderung 10 in dieses Modell ein, da Adaptionsprozesse immer mit Kontextdaten kollaborieren. Das *Adaptivity Model* wird im Kapitel 4.2.6 näher vorgestellt.

Das zu entwerfende generische Kompositionsmodell für UI-Mashups besteht aus fünf Teilmodellen. Jedes deckt einen bestimmten Aspekt von Web-Applikationen ab und dient der Wiederverwendbarkeit der einzelnen Modelle und dem SoC. Alle Teilmodelle müssen in ein Hauptmodell aufgenommen werden, damit sie in Relation zueinander gebracht werden können. Das Hauptmodell ist das eigentliche Kompositionsmodell und führt alle anderen zusammen. Es wird im nachfolgenden Kapitel 4.2.1 vorgestellt.

4.2.1 Basismodell

Das Basismodell enthält alle weiteren Modelle. Es wird *CRUISe Composition Model* genannt und ist in Abbildung 4.4 dargestellt.

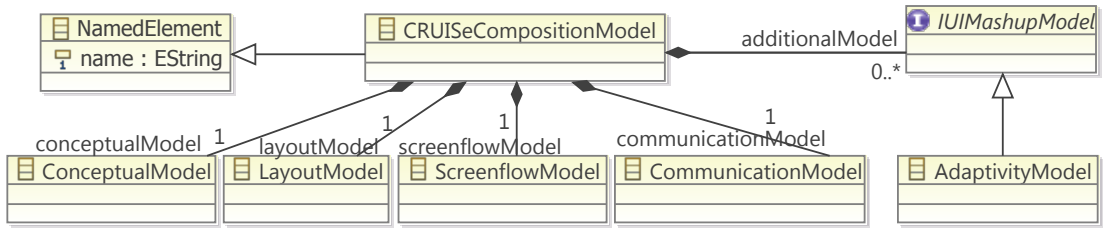


Abbildung 4.4: Basismodell des Kompositionsmodells

Wie man in der Abbildung sehen kann, besitzt das *CRUISe Composition Model* Kompositionen zu den vier Modellen *Conceptual Model*, *Layout Model*, *Screenflow Model* und *Communication Model*. Diese sind im *CRUISe Composition Model* obligatorisch, da sie sicherstellen, dass leistungsfähige UI-Mashups modelliert werden können, die komponentenbasiert sind und die Nachteile der in Kapitel 3 vorgestellten Ansätze kompensieren. Um die Erweiterbarkeit des Kompositionsmodells zu garantieren, kann das *CRUISe Composition Model* mehrere Instanzen des Interfaces *IUIMashupModel* besitzen. In Abbildung 4.4 ist zu sehen, dass dies mit dem optionalen *AdaptivityModel* vorgenommen wurde. Möchten UI-Mashup-Autoren zu-

sätzlich eigene Aspekte in das Kompositionsmodell integrieren, so kann dies, analog zum *AdaptivityModel*, über das Hinzufügen von Unterklassen von *IUIMashupModel* geschehen. Des Weiteren erbt das *CRUISe Composition Model* von der Klasse *NamedElement*. Dadurch wird ein Name festgelegt, der die Komposition charakterisiert und als eindeutiger Schlüssel zur Identifikation dient. Im weiteren Verlauf werden noch viele Klassen von *NamedElement* erben. Aus Gründen der Übersichtlichkeit wird diese Beziehung jedoch nicht immer in den dargestellten Diagrammen abgebildet, sondern nur erwähnt. Weiterhin darf jedes optionale *IUIMashupModel* nur einmal im *CRUISe Composition Model* enthalten sein. Diese Bedingung wird beispielhaft mit dem OCL-Constraint im Listing 4.1 sichergestellt. Die in den folgenden Abschnitten auftretenden Einschränkungen für das Kompositionsmodell werden ebenfalls mit OCL ausgedrückt, da dies ein standardisiertes Mittel zum Formulieren von Bedingungen ist. Außerdem müssen Autorenwerkzeuge, die das entworfene Kompositionsmodell unterstützen, sicherstellen, dass die hier angegebenen OCL-Constraints eingehalten werden.

Listing 4.1: Sicherstellung der Eindeutigkeit der Modelle

```
1 context CRUISeCompositionModel inv:
2         self.additionalModel->isUnique(eClass())
```

Damit ist das Grundgerüst für alle weiteren Modelle entworfen. Als erstes Modell wird im folgenden Kapitel das *Conceptual Model* vorgestellt.

4.2.2 Conceptual Model

In Abbildung 4.3 ist zu sehen, dass in das *Conceptual Model* die meisten Anforderungen eingebracht wurden. Deshalb wird dieses Modell sukzessive entwickelt, beginnend mit der Darstellung der enthaltenen Konzepte, wie sie in Abbildung 4.5 präsentiert werden.

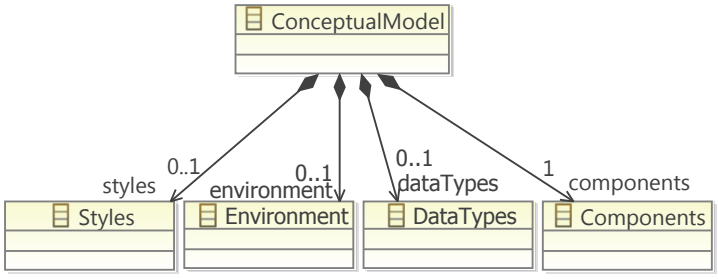


Abbildung 4.5: Bestandteile des *Conceptual Models*

In der obigen Abbildung ist dargestellt, dass das *Conceptual Model* die interagierenden Komponenten (Klasse *Components*) enthält, für deren *Events* und *Operations* Datentypen (Klasse *DataTypes*) deklariert werden können. Darüber hinaus hat man die Möglichkeit, Stile (*Styles*), die der Modellierung des Aussehens der UI-Komponenten dienen, sowie die Umgebung, in der die Komposition ausführbar gemacht werden soll, zu spezifizieren. Die zu verwendenden Datentypen werden in Kapitel 4.2.2.1 beschrieben und mit der Klasse *DataTypes* modelliert. Danach wird im Abschnitt 4.2.2.2 untersucht, welche Möglichkeiten bei der Modellierung der Komponenten existieren. Dort wird auch das CRUISe Komponentenmodell integriert.

Im Abschnitt 4.1 wurde schon erwähnt, dass auch die Laufzeitumgebung *Event*s auslösen kann. Mit der Klasse *Environment* wird dieser Aspekt modelliert und im Kapitel 4.2.2.3 konzipiert. Schließlich muss es möglich sein, gemeinsames *Look & Feel* zu spezifizieren, wofür die Klasse *Styles* eingeführt wird. Diese wird im Abschnitt 4.2.2.4 beschrieben.

4.2.2.1 Data Types

Wie im Komponentenmodell zu sehen (siehe Abbildung 4.1), können sowohl *Operation*s als auch *Event*s Parameter besitzen. Um komplexe Datenkommunikation gewährleisten zu können, sind Parameter getypt. Demzufolge hat jeder Parameter eine Referenz zu einem *Type*. Um nicht gleiche Typen für verschiedene Komponenten mehrmals definieren zu müssen, werden sie im *Conceptual Model* zentral definiert, so dass im Weiteren dann auf diese Definitionen verwiesen werden kann. Abbildung 4.6 zeigt das Konzept der Datentypen.

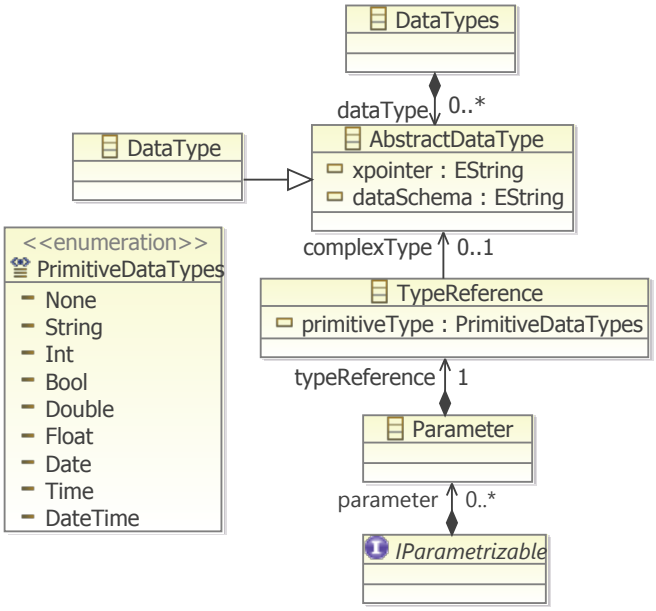


Abbildung 4.6: *DataTypes* im *Conceptual Model*

DataTypes kann aus einer beliebigen Anzahl von Instanzen der Klasse *AbstractDataType* bestehen, welche von *NamedElement* erbt. Diese Klasse wurde als abstrakt modelliert, um die Erweiterbarkeit des Metamodells zu unterstützen. So kann eine neue Unterklasse hinzugefügt werden, wenn zusätzliche Eigenschaften benötigt werden. Derzeit existiert *DataType* als einziger konkreter Datentyp im Metamodell. Zur Beantwortung der Frage, wie ein Datentyp genau zu spezifizieren ist, gibt es zwei Möglichkeiten. Wenn der Autor des UI-Mashups eine Sammlung von Typen besitzt, die in einer XSD[1]-Datei deklariert wurden, wäre es von Nutzen, wenn diese Datei wiederverwendet werden könnte, anstatt Typen redundant anzulegen. Dies ist die erste Möglichkeit. Über das Attribut *xpointer* kann man einen Ausdruck spezifizieren, der auf einen Datentyp in der XSD-Datei verweist. Die XPointer-Spezifikation

[1] eine Empfehlung vom W3C zur Spezifikation von Datentstrukturen – siehe http://www.w3.org/XML/Schema#dev

ist eine Empfehlung des W3C, mit deren Hilfe man über einen eindeutigen Bezeichner (URI) gezielt Elemente eines XML-Dokumentes adressieren kann. Existiert beispielsweise ein XML-Schema in der Datei `DataTypes.xsd` die einen Datentyp `Building` enthält, so kann an dieser Stelle mit dem Ausdruck `xlink:DataTypes.xsd#xpointer(//complexType[@name="Building"])` darauf verwiesen werden. Die zweite Möglichkeit ist, mit dem Attribut *dataSchema* direkt die XSD-Definition des Datentyps anzugeben. In jedem Fall darf jedoch nur eine dieser beiden Möglichkeiten verwendet werden. Um dies sicherzustellen, kann das OCL-Constraint im Listing 4.2 benutzt werden.

Listing 4.2: XOR zwischen *xpointer* und *dataSchema*

```
context AbstractDataType inv:
        not self.xpointer.oclIsUndefined()
        implies self.dataSchema.oclIsUndefined() and
        self.xpointer.oclIsUndefined()
        implies not self.dataSchema.oclIsUndefined()
```

Des Weiteren sind in Abbildung 4.6 zwei weitere Klassen, ein Interface und eine Aufzählung dargestellt. *IParametrizable* ist ein Interface, das von anderen Klassen, die Konzepte der Parametrisierbarkeit erhalten sollen, implementiert werden kann. Beispiele sind die *Event*s und *Operation*s der Komponenten, wie im Abschnitt 4.2.2.2 erläutert. Dadurch können implementierende Klassen beliebig viele *Parameter* erhalten, die jeweils aus einer *TypeReference* bestehen. Die Klasse *TypeReference* wurde eingeführt, um auch die Möglichkeit der Verwendung primitiver Datentypen zu schaffen. Auch hier darf nur entweder das Attribut *primitiveType* oder *complexType* angegeben werden, was durch das OCL-Constraint in Listing 4.3 gewährleistet wird.

Listing 4.3: XOR zwischen *primitiveType* und *complexType*

```
context TypeReference inv:
        not self.primitiveType.oclIsUndefined()
        implies self.complexType.oclIsUndefined() and
        self.primitiveType.oclIsUndefined()
        implies not self.complexType.oclIsUndefined()
```

Um primitive Datentypen nicht selbst erstellen zu müssen, wurde die Aufzählung *PrimitiveDataTypes* eingeführt, deren Elemente an die Grundtypen von Java angelehnt sind. Der nächste Abschnitt behandelt die Modellierung der Komponenten.

4.2.2.2 Components

Um die zu modellierenden Komponenten vollständig beschreiben zu können, muss das CRUISe-Komponentenmodell aus Kapitel 4.1 hier integriert werden. In Abbildung 4.7 wurde dies, neben weiteren Besonderheiten, die nachfolgend erläutert werden, eingearbeitet.

Ganz oben in der Abbildung ist die Umsetzung der Konfiguration der Komponenten dargestellt. Diese wird nicht näher erläutert, da sie äquivalent zum Komponentenmodell entworfen wurde. Wie dort auch gefordert, besitzt eine *MashupComponent* beliebig viele Elemente der Klassen *IOperation* und *Event*. Deren vererbende Interfaces *IOperation* und *IEvent* sind Unterklassen von *IParametrizable*. Dadurch ist deren Parametrisierbarkeit modellierbar. Die Beziehung von *MashupComponent* zu seinen *Event*s ist hier nicht zum Interface *IEvent* realisiert, da im Abschnitt

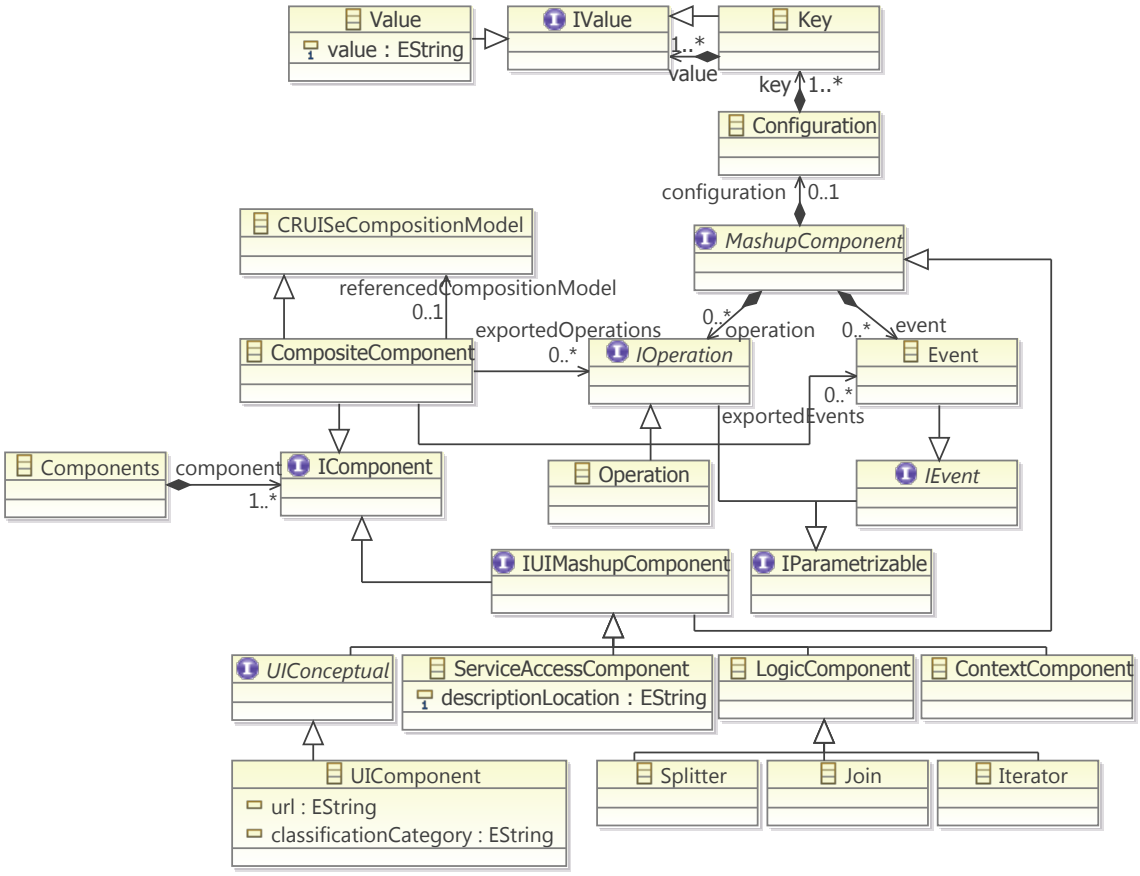


Abbildung 4.7: Komponenten im *Conceptual Model*

4.2.2.3 eine neue Unterklasse hinzukommt, die an dieser Stelle nicht erstellt werden darf. Das Interface *IUIMashupComponent* ist die Oberklasse für die vier Arten von Komponenten, wie sie schon in Abbildung 4.2 zu sehen waren und auch in den Anforderungen in Kapitel 2.6 genannt wurden. *IUIMashupComponent* erbt von *NamedElement*, wodurch jede Komponente einen Namen erhält. Die Vererbungsbeziehung zu *MashupComponent* stattet die Komponentenarten mit den Eigenschaften des Komponentenmodells aus. *IUIMashupComponent* ist ein Interface, das das Hinzufügen neuer Komponentenarten ermöglicht, so dass das Metamodell einfach erweitert werden kann. Aus demselben Grund wurde für die Klasse *UIComponent* das Interface *UIConceptual* eingeführt, so dass auch hier weitere speziellere UI-Komponenten hinzugefügt werden können. Mit dem Attribut *url* der Klasse *UIComponent* wird die Adresse des UIS angegeben, der die konkrete UIC bereitstellt. Das Attribut *classificationCategory* deckt Anforderung 14 ab, wodurch es möglich ist, nur eine Kategorie der in das UI-Mashup zu integrierenden Komponente anzugeben. Dadurch löst man sich vollkommen von konkreten Komponenten, und die Web-Applikation wird sehr flexibel. Die Angabe von *url* und *classificationCategory* gleichzeitig darf nicht möglich sein. Diese Einschränkung wird mit dem OCL-Constraint in Listing 4.4 ausgedrückt.

Listing 4.4: XOR zwischen *url* und *classificationCategory*

```
1 context UIComponent inv:
```

```
2          not self.url.oclIsUndefined()
3          implies self.classificationCategory.oclIsUndefined() and
4          self.url.oclIsUndefined()
5          implies not self.classificationCategory.oclIsUndefined()
```

Die Klasse *ServiceAccessComponent* dient der Kopplung an Backend-Dienste. Sie verfügt über das Attribut *descriptionLocation*, womit die Anforderung 8 erfüllt wird. Damit ist es möglich, eine Referenz zur Beschreibung des jeweiligen Services anzugeben, sei es die WSDL-Datei eines SOAP-basierten oder die WADL-Datei eines REST-basierten Webservices. Sogenannte *RESTful Services* basieren auf reiner Kommunikation über HTTP, und die Adressierung von Ressourcen steht im Vordergrund [@Bay02]. Dadurch können *RESTful Services* einfach über URIs angesprochen werden, wohingegen bei einer Anfrage mit SOAP ein komplettes XML-Dokument gesendet werden muss, welches der WSDL-Beschreibung des Services zu entsprechen hat. In der WADL-Datei kann für *RESTful Services* angegeben werden, welche Parameter an die URI angehängt werden können und welche Funktionalität damit angefordert wird. SACs sind aber allgemein als Datenlieferant gedacht. Eine derzeit populäre Art, Daten zur Verfügung zu stellen, sind Feeds. Dabei wird wie bei *RESTful Services* eine URI aufgerufen, an die Parameter angehängt werden[2]. Für Feeds gibt es jedoch kein Beschreibungsformat wie WADL, wodurch die Gefahr besteht, dass SACs nicht für alle Arten von Backend-Diensten generisch gehalten werden können. Da jedoch alle HTTP-basierten Dienste mit WADL beschrieben werden können, kann dieses Problem durch die Nutzung einer WADL-Beschreibung für Feeds gelöst werden, da sich diese auch auf der gleichen Kommunikationsebene befinden. Dadurch ergibt sich die Möglichkeit, für diese eine Beschreibung anzulegen, womit SACs generisch bleiben können und die *descriptionLocation* ausreicht, um den Backend-Dienst vollständig zu beschreiben. Existieren für Dienste keine Beschreibungen, gibt es zwei Möglichkeiten. Zum einen können diese angelegt werden, wodurch jedoch vorerst ein Mehraufwand entsteht. Dieser macht sich aber bezahlt, wenn Services wiederverwendet werden. Die erstellten Beschreibungen können in diesem Fall erneut benutzt werden, wodurch sich der Nachteil wieder relativiert. Zum anderen können für SACs ohne Beschreibungen die *Event*s und *Operation*s manuell modelliert werden, und der Autor muss deren korrekte Arbeitsweise sicherstellen.

Die Klassen *ContextComponent* und *LogicComponent* erfüllen die Anforderungen 9 beziehungsweise 10. Außerdem befinden sich in Abbildung 4.7 drei Unterklassen der *LogicComponent*. Diese repräsentieren häufig benötigte Datentransformationen. So ist die Klasse *Splitter* als eine LC zu verstehen, die einen Eingang (*Operation*) sowie mehrere Ausgänge (*Event*s) besitzt. Prinzipiell existieren zwei Möglichkeiten. Zum einen, wenn die *Operation* einen Parameter erwartet, der einen zusammengesetzten Datentyp besitzt. Dieser wird dann von dem *Splitter* in seine Bestandteile zerlegt, und in der Folge wird für jeden enthaltenen Datentyp ein *Event* bereitgestellt, das das jeweilige Datum kommuniziert. Man stelle sich am Eingang einen zusammengesetzten Datentyp `Adresse` vor, welcher drei Teiltypen `String` besitzt, die für »Straße«, »Ort« und »Postleitzahl« stehen. Dadurch erhält eine *Splitter*-Komponente drei Ausgänge (jeweils vom Typ `String`), womit `Adresse` zerlegt wird und die dort enthaltenen Teildaten bereitgestellt werden. Die andere Möglichkeit tritt ein, wenn die eingehende *Operation* mehrere Parameter besitzt. In diesem Fall

[2] beispielsweise Flickr-Feeds unter http://www.flickr.com/services/feeds/

wird für jeden Parameter ein *Event* bereitgestellt, dessen Parameter den Typ des jeweiligen Eingangsparameters trägt. Die Klasse *Join* fungiert als Gegenstück zum *Splitter*. Sie besitzt mehrere Eingänge und einen Ausgang. Der Ausgang soll den vereinten Datentyp aller Eingänge kommunizieren. Erwartet beispielsweise eine *Operation* einen Parameter vom Typ `Int` und eine weitere *Operation* einen vom Typ `String`, so produziert das ausgehende *Event* zusammengesetzte Daten vom Typ `(Int, String)` in der Reihenfolge, in der die eingehenden *Operation*s deklariert wurden. Des Weiteren dient die Klasse *Iterator* als Komponente, die einzelne Elemente eines Datentyps, der einer Datensammlung entspricht, zurückgibt. Hier ist das typische Beispiel der Liste zu nennen, die eine Menge von Daten gleichen Typs enthält. Diese Komponente iteriert demnach über die Liste und kommuniziert die enthaltenen Daten elementweise.

Schließlich enthält Abbildung 4.7 noch die Klasse *CompositeComponent*. Diese dient der rekursiven Komposition, wie sie in Anforderung 2 erläutert wurde. Aus diesem Grund erbt diese Klasse von *IComponent* und vom *CRUISe Composition Model*. Damit ist es möglich, ein komplett eigenständiges Kompositionsmodell als Komponente in einem UI-Mashup zu modellieren. So wird jetzt auch verständlich, warum *CompositeComponent* keine direkte oder implizite Unterklasse von *MashupComponent* ist, da in diesem Fall *Event*s und *Operation*s als deren Kinder spezifiziert werden könnten, diese jedoch in ihrem eigenen *Conceptual Model* modelliert werden. Eine weitere Möglichkeit der rekursiven Komposition und Wiederverwendung von Modellen ist mit dem Attribut *referencedCompositionModel* geschaffen worden. Damit wird ein schon vorhandenes Kompositionsmodell referenziert, welches in ein neues UI-Mashup als Komponente eingebunden werden soll. Zu guter Letzt können für eine *CompositeComponent* mit *exportedEvents* und *exportedOperations* innere *Event*s beziehungsweise *Operation*s angegeben werden, die nach außen kommuniziert werden sollen. Dadurch erhält auch diese Komponente die Möglichkeit, mit allen anderen des UI-Mashups zu kommunizieren, und wirkt somit wie ein normales *UIConceptual*. Eine andere Art der Angabe der exportierten *Event*s und *Operation*s wäre am *CRUISeCompositionModel* selbst; also zum Zeitpunkt der Modellierung und nicht zum Zeitpunkt der rekursiven Integration. So würde der Autor des UI-Mashups angeben, welche *Event*s und *Operation*s nach außen sichtbar sein sollen, und nicht der Autor, der dieses UI-Mashup als *CompositeComponent* benutzt. Es wird jedoch hier die zuerst vorgestellte Variante verwendet, da Autoren zum Zeitpunkt der Modellierung des zu integrierenden UI-Mashups im Allgemeinen nicht wissen können, ob die Web-Applikation jemals zu einer *CompositeComponent* abstrahiert werden soll. Zudem ist nicht bekannt, welchem Zweck das UI-Mashup zukünftig in einer rekursiven Komposition dienen soll. Demzufolge können sich die benötigten *Event*s und *Operation*s in unterschiedlichen rekursiven Kompositionen unterscheiden, weshalb deren Autoren selbst entscheiden sollen, was benötigt wird. Aufgrund der zwei möglichen Arten der rekursiven Komposition muss die jeweils andere wieder ausgeschlossen werden. Enthält eine *CompositeComponent* Kindelemente, so darf *referencedCompositionModel* nicht gesetzt sein und umgekehrt. Des Weiteren darf nicht die *CompositeComponent* selbst oder das sie beinhaltende *CRUISe Composition Model* referenziert werden, da sonst eine Endloskomposition entsteht. Schließlich ist sicherzustellen, dass nur eigene innere *Event*s und *Operation*s exportiert werden. Diese Bedingungen drückt das OCL-Constraint in Listing 4.5 aus.

Listing 4.5: Constraints der *CompositeComponent*

```
context CompositeComponent inv:
        not self.referencedCompositionModel.oclIsUndefined()
        implies self.model->isEmpty() = true
        and self.referencedCompositionModel.oclIsUndefined()
        implies self.referencedCompositionModel != self
        and self.referencedCompositionModel.oclIsUndefined()
        implies self.referencedCompositionModel != self.parentModel.
            parentCompositionModel and
        self.exportedEvents->forAll(ev|
                ev->closure(eContainer())->includes(self)
        ) and
        self.exportedOperations->forAll(op|
                op->closure(eContainer())->includes(self)
        )
```

Damit ist die Modellierung der Komponenten abgeschlossen. Wie *Event*s und *Operation*s der Laufzeitumgebung spezifiziert werden können und welche Besonderheiten diese noch auszeichnen, wird im nachfolgenden Abschnitt konzipiert.

4.2.2.3 Environment

Wie schon weiter oben aufgeführt, kann die Laufzeitumgebung des UI-Mashups auch *Event*s produzieren oder deren angebotene *Operation*s können aufgerufen werden. Aus diesem Grund wird, wie in Abbildung 4.8 zu sehen, *Environment* als Unterklasse von *MashupComponent* modelliert.

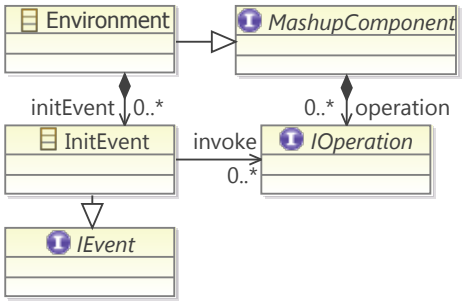


Abbildung 4.8: *Environment* im *Conceptual Model*

Damit ist es möglich, über *Operation*s der *Environment*, einen globalen Datenspeicher zu simulieren, in dem Komponenten Daten speichern können, die andere Komponenten erst später benötigen, zu einem Zeitpunkt, da die Daten an der liefernden Komponente nicht mehr verfügbar sind. Der Abruf der Daten erfolgt über korrespondierende *Event*s. Das in Abschnitt 4.2.2.2 schon vorgestellte Interface *IEvent*, erhält hier eine zweite Unterklasse: *InitEvent*. Sie dient dazu, für die Laufzeitumgebung *Operation*s angeben zu können, die beim initialen Aufruf des UI-Mashups ausgeführt werden. *InitEvents* referenzieren beliebig viele *IOperations* aller anderen Komponenten, wodurch modelliert werden kann, welche *Operation*s der Komponenten initial aufgerufen werden sollen, sobald die Web-Applikation gestartet wird. Jetzt wird auch verständlich, dass *InitEvents* nicht als *Event*s von *MashupComponents* spezifiziert werden dürfen, da sie nur der Laufzeitumgebung vorbehalten sind.

Das letzte Konzept, die Modellierung von *Look & Feel* und Layout der Komponenten, wird im folgenden Abschnitt konzipiert.

4.2.2.4 Styles

Anforderungen 13 und 15 beschreiben, dass im Kompositionsmodell angegeben werden muss, wie Komponenten dargestellt werden sollen. Dazu zählen beispielsweise die Angabe ihrer Abmessungen beziehungsweise die Möglichkeit eines einheitlichen *Look & Feel*. Dieser Teil des *Conceptual Models* wird in Abbildung 4.9 gezeigt.

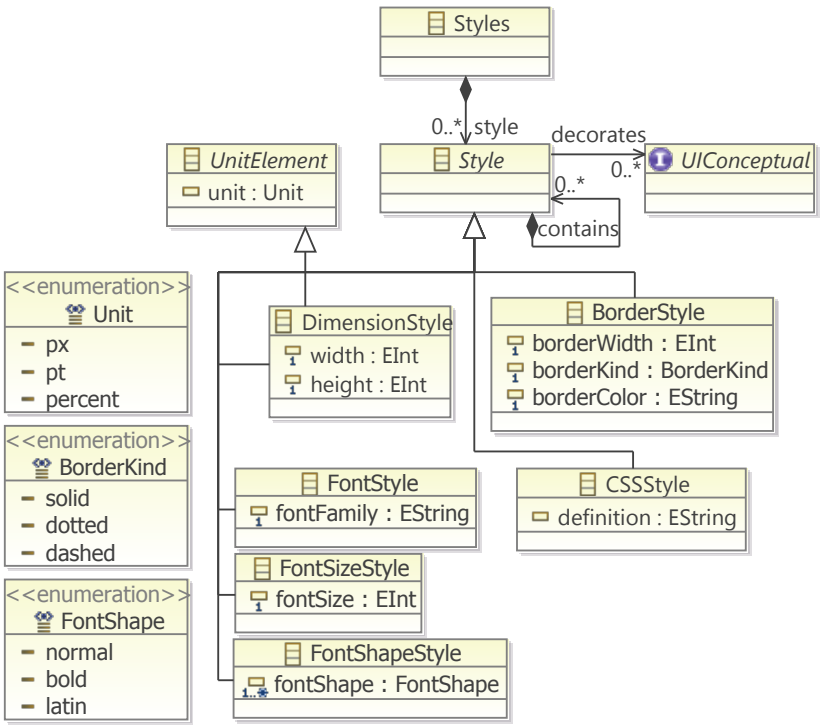


Abbildung 4.9: *Styles* im *Conceptual Model*

Es können mehrere konkrete Instanzen der Klasse *Style* modelliert werden, die jeweils für einen bestimmten Stil stehen. *Style* wurde auch hier wieder abstrakt definiert, um eine nachträgliche Erweiterung weiterer Stile zu ermöglichen. Das Besondere ist, dass *Style* mehrere Unterstile (Komposition *contains*) besitzen und beliebig viele *UIConceptuals* referenzieren kann. Dieses rekursive Vorgehen bedeutet, dass Stile so nicht nur UI-Komponenten dekorieren, sondern auch andere *Styles* umhüllen können. Dadurch wird ein Dekorationsbaum aufgebaut. Jedes *UIConceptual* und jeder *Style* wird somit von dem darüber liegenden *Style* dekoriert. Somit können UI-Komponenten auf unterschiedlichen Ebenen des Dekorationsbaumes referenziert werden. Alle Komponenten auf derselben Ebene haben demnach das gleiche *Look & Feel*. Die Klasse *DimensionStyle* ist für die Modellierung des Layouts der UI-Komponenten zuständig. Mit ihr können Höhe und Breite sowie die Maßeinheit angegeben werden. Dazu erbt *DimensionStyle* von *UnitElement*, womit ein Element der Aufzählung *Unit* angegeben werden muss. Diese stehen für Pixel (*px*), Punkt (*pt*) oder Prozent (*percent*) der verfügbaren Breite im Browser. Mit *CSSStyle* existiert eine weitere Art, das Aussehen der UI-Komponenten zu beeinflussen. Wie der Name schon vermuten lässt, kann damit die CSS[3]-Definition eines Stils angegeben

[3]Formatierungsmechanismus zur Gestaltung von Web-Dokumenten – siehe http://www.w3.org/Style/CSS/

werden, wodurch sich Autoren von Web-Applikationen auf gewohntem Terrain befinden. Mit den anderen *Styles* können das Schriftbild modelliert sowie Rahmen für die UI-Komponenten spezifiziert werden.

Damit wurde das *Conceptual Model* vollständig spezifiziert und alle in Abbildung 4.3 enthaltenen Anforderungen integriert. Im nächsten Kapitel wird das Layout des gesamten UI-Mashups im *Layout Model* vorgestellt.

4.2.3 Layout Model

Dieses Modell spezifiziert das Gesamtlayout der Web-Applikation und ist die Grundlage für deren Kontrollfluss, der im nachfolgenden Abschnitt 4.2.4 konzipiert wird. In Abbildung 4.10 ist zu sehen, dass im *Layout Model* mehrere *Layouts* definiert werden können.

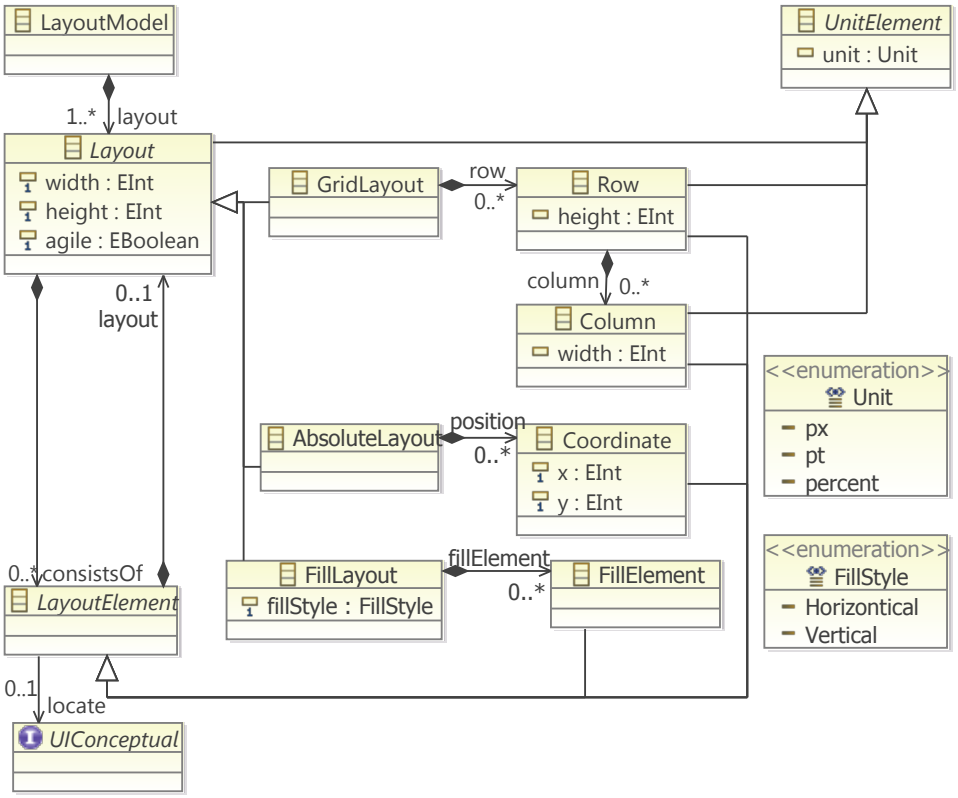


Abbildung 4.10: Das *Layout Model*

Layout erbt von *NamedElement* und ist eine Unterklasse von *UnitElement*, wodurch spezifiziert werden kann, mit welcher Einheit die Höhe und Breite angegeben wird. Die Klasse *Layout* besteht aus beliebig vielen *LayoutElements*. Beide sind als abstrakt definiert, wodurch neue konkrete Arten von *Layouts* im Nachhinein hinzugefügt werden können. Ein *LayoutElement* kann wiederum ein *Layout* besitzen, wodurch es möglich ist, verschiedene *Layouts* ineinander zu schachteln. Außerdem kann ein *LayoutElement* ein *UIConceptual* referenzieren. Damit wird erreicht, dass UI-Komponenten an einer bestimmten Stelle im *Layout* platziert werden können.

Derzeit befinden sich drei verschiedene *Layout*-Arten im Kompositionsmodell, die nachfolgend erläutert werden.

GridLayout: Dieses Layout orientiert sich am schon von Java bekannten GridLayout und soll UI-Komponenten in einem Gitternetz anordnen. Dafür kann es mehrere Zeilen (*Rows*) enthalten, die wiederum mehrere Spaltenelemente (*Columns*) beinhalten. *Row* und *Column* erben von *UnitElement*, womit man mit einer der drei vorgegebenen Einheiten die Höhe der Zeile beziehungsweise die Breite der Spalte spezifizieren kann.

AbsoluteLayout: Möchte man UI-Komponenten mit absoluten Koordinaten im UI-Mashup positionieren, so benutzt man dieses Layout. Dabei können beliebig viele Instanzen der Klasse *Coordinate*, die jeweils eine x- und y-Koordinate besitzen, angegeben werden. Diese Art von Layout ist angelehnt an die absolute Positionierbarkeit mittels CSS[4].

FillLayout: Diese Art von Layout orientiert sich am FillLayout der SWT[5]-Bibliothek von Java. Über das Attribut *fillStyle* wird angegeben, ob die zu positionierenden UI-Komponenten horizontal oder vertikal angeordnet werden sollen. Durch das *FillLayout* wird erreicht, dass alle enthaltenen Komponenten den gesamten verfügbaren Platz einnehmen und zusätzlich alle Komponenten gleich groß sind. Sie orientieren sich in der Breite wie auch in der Höhe an der jeweils größten UI-Komponente. Hierbei spielt es keine Rolle, welche Abmessungen (Klasse *DimensionStyle*) mit den Stilen des *Conceptual Models* modelliert wurden, da sie mit diesem *Layout* automatisch angepasst werden sollen.

Des Weiteren besitzt *Layout* das Attribut *agile*. Es kann nur einen der beiden Werte `true` oder `false` annehmen und soll dazu benutzt werden, das freie Verschieben von UI-Komponenten im Browser des Anwenders zu ermöglichen. Wird dieses Attribut auf `true` gesetzt, so repräsentiert das entsprechende *Layout* den Initialzustand der Anordnung der Komponenten und kann dann frei verschoben werden. Wird beispielsweise ein vertikales *FillLayout* auf *agile* gesetzt, so kann man alle darin enthaltenen UI-Komponenten frei auf ihrer senkrechten Achse positionieren.

Eine Einschränkung muss jedoch definiert werden. Ein *LayoutElement* darf kein *UIConceptual* referenzieren, wenn es ein weiteres *Layout* beinhaltet, da sonst die Eindeutigkeit verloren geht. Diesen Zusammenhang spezifiziert das OCL-Constraint 4.6

Listing 4.6: XOR zwischen *layout* und *locate*

```
1 context LayoutElement inv:
2         not self.layout.oclIsUndefined()
3         implies self.locate.oclIsUndefined()
4         and
5         self.layout.oclIsUndefined()
6         implies not self.locate.oclIsUndefined()
```

Mit dem hier vorgestellten Konzept der Layouts ist es nun möglich, UI-Komponenten im UI-Mashup anzuordnen. Die Komponenten bleiben zudem unabhängig vom Layout, was die Wiederverwendbarkeit des *Conceptual Models* fördert. Auf der anderen Seite sind jedoch die Layouts abhängig von den Komponenten, da sie darin referenziert werden. Dadurch kann ein Layout nicht ohne Anpassungen wiederverwendet werden. Dieser Nachteil relativiert sich jedoch wieder, wenn man überlegt,

[4]siehe Beispiel unter http://www.css4you.de/position.html
[5]siehe `http://www.eclipse.org/swt`

dass es sich hierbei um die Anordnung von UI-Komponenten für ein bestimmtes UI-Mashup handelt. Das Layout ist demnach immer sehr spezifisch auf die Web-Applikation zugeschnitten und wird nur in den seltensten Fällen wiederverwendet werden. Demnach ist die Abhängigkeit, die vom *Layout Model* zum *Conceptual Model* besteht, vertretbar. Eine dritte Variante der Modellierung des Layouts wäre die Definition von Layouts sowie von UI-Komponenten und ein dazwischen spezifiziertes Mapping. Dadurch bleiben *Conceptual Model* und *Layout Model* unabhängig voneinander, es wird aber ein neues, anwendungsspezifisches Mapping-Modell benötigt. Jedoch ergibt sich aus der oben geführten Argumentation, dass es beim Layout nicht notwendig ist, vollkommene Unabhängigkeit zu garantieren, da das Layout einer Web-Applikation von Natur aus sehr spezifisch ist. Ein weiterer Nachteil des Mappings wäre der Mehraufwand, weshalb die hier vorgestellte Variante gewählt wurde.

Mit der Modellierung des Layouts wurde ein wichtiger Aspekt von UI-Mashups spezifiziert. Damit hat der Autor, im Gegensatz zum WebML-Ansatz, mehrere Möglichkeiten zur Positionierung der UI-Komponenten. Mit dem Attribut *agile* wird es außerdem möglich, die initiale Anordnung für Web-Applikationen mit frei positionierbaren Komponenten zu modellieren. Das *Layout Model* fließt grundlegend mit in die Modellierung des Kontrollflusses ein, welcher im kommenden Abschnitt 4.2.4 konzipiert wird.

4.2.4 Screenflow Model

Wie in Anforderung 11 schon erläutert, muss das Kompositionsmodell die Möglichkeit bieten, den Kontrollfluss des UI-Mashups zu modellieren. Dafür ist das *Screenflow Model* verantwortlich. Im Gegensatz zu allen in Kapitel 3 analysierten Ansätzen soll hier der Weg des *Event*-basierten Kontrollflusses eingeschlagen werden, da man auf diese Weise sehr generisch Übergänge zwischen Sichten modellieren kann. Infolge der Tatsache, dass *Event*s nicht nur von Benutzerinteraktionen ausgelöst, sondern auch von der Komponentenlogik propagiert werden können, sind mit dieser Art der Modellierung Übergänge möglich, die nicht nur durch das Einwirken des Nutzers initiiert wurden. Man stelle sich beispielsweise eine Upload-Komponente vor, mit der ein Nutzer eine Datei auf einen Server (beispielsweise einen FTP-Server) laden kann. Wurde der Upload gestartet, kann die Komponente bei Fertigstellung ein *Event* auslösen, wodurch dem Nutzer eine neue Sicht eröffnet wird, beispielsweise die Anzeige der aktualisierten Verzeichnisstruktur. In Abbildung 4.11 ist das Metamodell des *Screenflow Models* dargestellt.

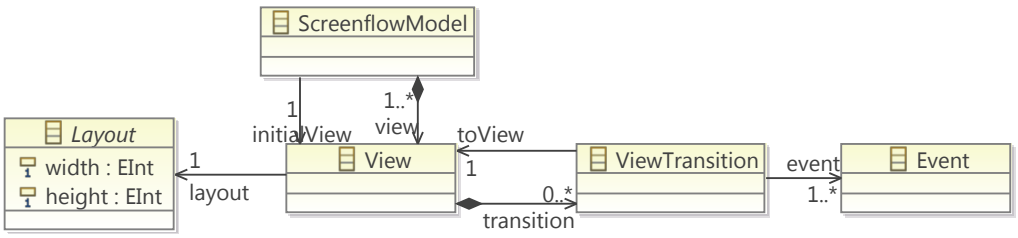


Abbildung 4.11: Das *Screenflow Model*

In der Abbildung ist ersichtlich, dass das *Screenflow Model* mindestenz eine *View* besitzen muss. Man gelangt mittels einer *ViewTransition* von einer *View* zur nächsten. Demnach kann eine *View* mehrere *ViewTransitions* besitzen, wohingegen diese genau eine *View* referenzieren. Das Besondere ist, dass eine Transition mindestens ein *Event* referenzieren muss. Damit wird modelliert, welche *Event*s die Transition auslösen sollen. Doch wie definiert man nun Sichten in diesem Modell? In dem schon bekannten *Layout Model* wurde genau das getan. Die darin spezifizierten *Layouts* sind nichts anderes als Sichten auf das UI-Mashup. Demzufolge muss man in einer *View* genau das *Layout*, welches dann dargestellt werden soll, referenzieren. Durch dieses Vorgehen wird ein endlicher gerichteter, Graph aufgebaut. Mit dieser Eigenschaft des *Screenflow Models* ist es möglich, einfache Modellprüfer heranzuziehen, um die Erreichbarkeit der verschiedenen Sichten zu testen. So kann beispielsweise ermittelt werden, ob es Wege durch den Graphen gibt, die einen Deadlock hervorrufen, also in einer Sicht enden, wo sie nicht mehr herausführen. Bei UI-Mashups kann so ein Verhalten allerdings erwünscht sein. Denkt man zum Beispiel an Workflows, die in Web-Applikationen abgebildet werden, so soll es nicht immer möglich sein, Schritte zurück zu navigieren. Der Autor des UI-Mashups kann allerdings darauf hingewiesen werden und dann selber überprüfen, ob der Deadlock einen Fehlerfall darstellt. Des Weiteren muss im *Screenflow Model* genau eine initiale *View* angegeben werden, um die Sicht des UI-Mashups zu definieren, die dem Nutzer als erste angezeigt wird. Dies geschieht mit der Referenz *initialView*.

Mit dem *Screenflow Model* wird nicht nur die Anforderung 11 erfüllt, sondern auch der Grundstein für die Testbarkeit der Korrektheit des *Screenflows* gelegt. Dadurch kann teilweise automatisiert überprüft werden, ob die modellierten Transitionen frei von Deadlocks sind. Mit dem *Screenflow Model* wurde eine einfache Möglichkeit geschaffen, Übergänge zwischen den Sichten eines UI-Mashups zu modellieren. Das Besondere hierbei ist, dass Transitionen auch von Komponenten ausgelöst werden können, da, im Gegensatz zum nutzergesteuerten Ansatz von WebML, allgemein *Event*s die Übergänge auslösen. An die Konzeption des Kontrollflusses schließt sich die Konzeption des Datenflusses im nachfolgenden Kapitel 4.2.5 an.

4.2.5 Communication Model

Eine der wichtigsten Anforderungen ist die Spezifikation von Kommunikationskanälen (siehe Anforderung 1). Die Kommunikation zwischen Komponenten macht das Wesen eines UI-Mashups aus. Eine Web-Applikation wird damit erst leistungsfähig, denn sonst würde sie nur eine Ansammlung von Komponenten, ohne jegliche Interaktion, darstellen. Als Beispiel solcher Komponentensammlungen sei iGoogle[6] genannt. Es gibt verschiedene Ansätze der Kommunikation. So ist zum Beispiel das Pull-Paradigma anzuführen, mit dem eine Komponente Daten einer anderen anfordert, wenn diese verarbeitet werden sollen. Ein Nachteil davon ist, dass die datenverarbeitende Komponente unentwegt Anfragen an die datenliefernde Komponente stellen muss, um zu prüfen, ob neue Daten bereitstehen. Außerdem muss die verarbeitende Komponente bei vielen Lieferanten ihre Sender selbst verwalten, wodurch eine Abhängigkeit aufgebaut wird. Darüber hinaus gibt es den umgekehrten Ansatz des Push-Paradigmas. Hierbei wirkt die datenliefernde Komponente als kon-

[6] siehe http://www.google.com/ig

trollierende Einheit für die empfangende Komponente. Wenn neue Daten am Sender bereitstehen, werden sie an den Empfänger übertragen. Auch hier entsteht ein hoher Verwaltungsaufwand für den Sender, wenn viele Empfänger existieren. Jedoch hat das Push-Prinzip einen Vorteil: es findet nur Kommunikation statt, wenn der Sender etwas mitzuteilen hat. Auf der anderen Seite ist die Verwaltung der Kommunikationspartner noch nicht optimal, da eine starke Abhängigkeit zwischen ihnen besteht. Aus diesem Grund verläuft die Interaktion sowohl in CRUISe, als auch in mashArt, nach dem eventbasierten Publish/Subscribe-Prinzip. Damit ist es möglich, Kommunikation zwischen Komponenten zu realisieren, ohne dass die Beteiligten wissen, wer zu sendende Daten empfängt, beziehungsweise wer sie liefert. Diese Art der Kommunikation wird, ähnlich dem Mediator-Muster in [GHJV94], mittels einer zentralen Verwaltung durchgeführt. Diese Verwaltung entspricht in CRUISe einem Datenkanal (*DataChannel*). Sowohl Sender als auch Empfänger registrieren sich als *Publisher* beziehungsweise als *Subscriber* am *DataChannel*, welcher die Synchronisation übernimmt. Stehen am *Publisher* neue Daten bereit, wird ein *Event* ausgelöst, welches der *DataChannel* erkennt. Die Daten werden dann an alle am Datenkanal registrierten *Subscriber* übertragen. Somit entsteht eine lose Kopplung zwischen Sender und Empfänger, die durch einen Broker miteinander verbunden werden. Dieses Vorgehen entspricht dem gewählten Prinzip in der parallel verfassten Arbeit [Krü09], in der ein Kommunikationsmodell für CRUISe entworfen werden sollte.

Aufgrund der Vorteile des Publish/Subscribe-Paradigmas und der Verwendung in CRUISe wird es im *Communication Model* eingesetzt. Das Metamodell dessen ist in Abbildung 4.12 dargestellt und wird nachfolgend erläutert.

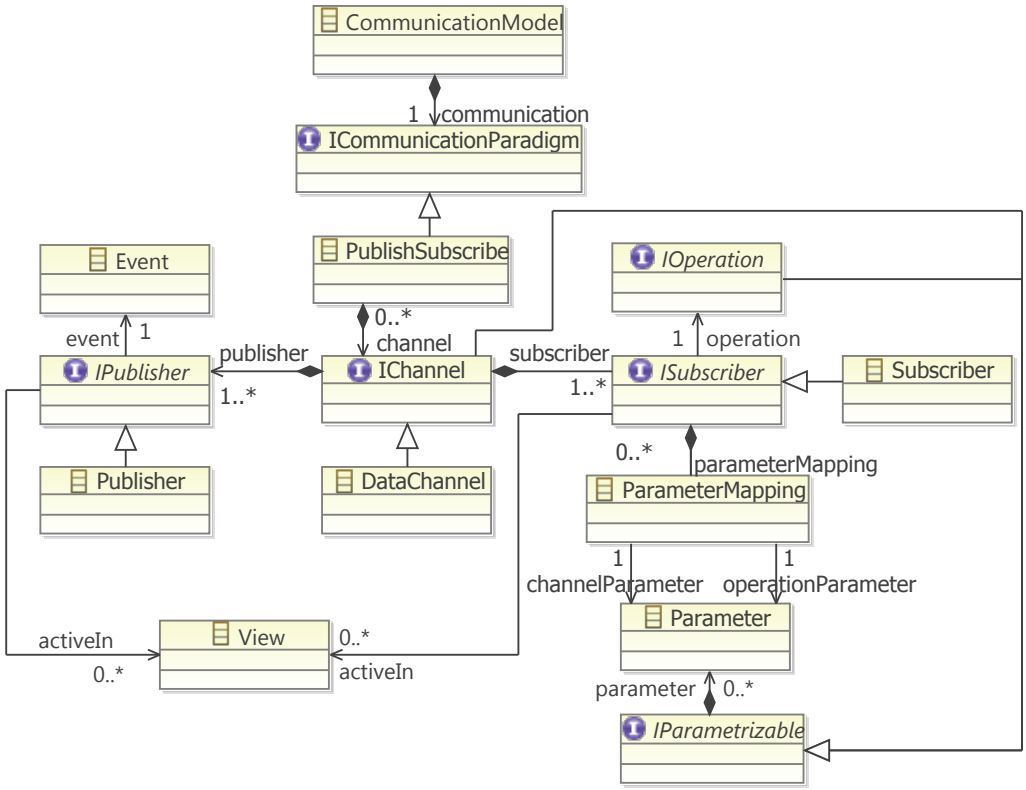


Abbildung 4.12: Das *Communication Model*

Im Ergebnis der zahlreichen Kommunikationsansätze wird Erweiterbarkeit durch die Nutzung des Interfaces *ICommuncationParadigm* sichergestellt. Durch Vererbung können dem Kompositionsmodell so neue Paradigmen hinzugefügt werden. Jedoch darf im *Communication Model* nur genau eines der Paradigmen enthalten sein. In Abbildung 4.12 ist dargestellt, dass *PublishSubscribe* mehrere *Channels* enthalten darf. *IChannel* erbt außerdem von *NamedElement* und *ITypable*. Die Vererbungsbeziehung zu *IParametrizable* stellt sicher, dass ein *DataChannel* getypt ist. Es wird damit angegeben, welche Struktur die darauf transportierten Daten haben müssen. Da *Event*s verschiedener Komponenten Daten der gleichen Struktur kommunizieren, wird es demnach ermöglicht, mehrere *Publisher* an einem *DataChannel* zu registrieren. Des Weiteren referenziert ein *Publisher* genau das *Event*, welches die Kommunikation auslösen soll. Wie schon aus Abschnitt 4.2.2 bekannt und auch in Abbildung 4.12 noch einmal dargestellt, enthält *Event* beliebig viele Parameter. Da unterschiedliche Komponentenentwickler verschiedene Konventionen haben, was die Bezeichnung der *Event*-Parameter angeht, besteht die Möglichkeit unterschiedlicher Namen für semantisch gleiche Parameter. Dazu ist die Namensvergabe der Parameter des *DataChannels* wichtig. Diese erhalten auf dem Datenkanal jeweils verallgemeinernde Bezeichnungen. Kommuniziert beispielsweise ein Publisher ein Datum des Typs `Name:String`, ein weiterer ein Datum des Typs `Nachname:String` und ein Dritter wiederum benutzt `lastname:String`, so haben sie doch alle dieselbe Bedeutung: der Nachname einer Person. So kann der Autor des UI-Mashups eine verallgemeinerte Bezeichnung für den *DataChannel* wählen, wie zum Beispiel einfach `Name:String`. In Abbildung 4.13b ist dieser Zusammenhang vereinfacht dargestellt: die erste Parameter-Bezeichnung von *UIComponent A* lautet `Lastname`, wohingegen für den rot markierten *DataChannel*, im *ParameterMapping*, die Bezeichnung `Name` gewählt wurde. Dieses Beispiel veranschaulicht, dass die transportierten Daten eines Kanals die gleiche Semantik haben sollten, da es sonst zu Komplikationen führen kann und Ergebnisse in den Komponenten nicht vorhersehbar sind. Die Semantik der *DataChannels* kann allerdings nicht mit dem Kompositionsmodell spezifiziert werden, weshalb der Autor die Konsistenz sicherstellen muss. Veröffentlicht beispielsweise eine Komponente ein Datum des Typs `Vorname:String`, so entspricht zwar die Struktur des Typs der des vorher erwähnten Kanals, hat hier jedoch eine andere Bedeutung. In diesem Fall sollte hierfür ein neuer *DataChannel* modelliert werden. Da *IPublisher* eine Referenz auf *Event* besitzt und die *Event*s einer *CompositeComponent* auch davon erben, muss sichergestellt werden, dass keine inneren *Event*s von *CompositeComponents* im *Publisher* referenziert werden. Dies geschieht mit dem folgenden OCL-Constraint.

Listing 4.7: Keine Verweise auf innere *Event*s von *CompositeComponents*

```
context IPublisher inv:
        self.event ->forAll(ev|
        not CRUISeMetaModel::CompositeComponent.allInstances()->forAll(comp|
                comp.eContents()->select(tempEv| tempEv.oclIsTypeOf(CRUISeMetaModel
                    ::Event))-> includes(ev)
                )
        )
```

Des Weiteren verfügt ein *DataChannel* über mindestens einen *Subscriber*. Dieser referenziert genau eine *IOperation*, der die kommunizierten Daten als Parameter übergeben werden. Nun kann aber der Fall eintreten, dass eine *Operation* strukturell nicht alle transportierten Daten des Kanals benötigt, sondern nur eine Teilmenge

davon. Wenn beispielsweise der *DataChannel* Daten der Struktur `(Name:String, Age:Int)` überträgt, eine *Operation* jedoch nur `name:String` erwartet, muss angegeben werden können, welcher Teil der Daten an die *Operation* weitergeleitet werden soll. Für solche Fälle kann am *Subscriber* mit der Klasse *ParameterMapping* festgelegt werden, welche Parameter des *DataChannels* welchen Parametern der *Operation* übergeben werden sollen. Demnach besitzt ein *ParameterMapping* genau zwei Referenzen auf *Parameter*, eine, um den Parameter des *DataChannels* auszuwählen, und die andere, um den Parameter der *Operation* zu bestimmen. Zudem muss sichergestellt werden, dass im *ParameterMapping* auch nur Parameter des *DataChannels* auf Parameter der *Operation* gemappt werden, und dass die aufeinander abgebildeten Typen gleich sind. Dieser Zusammenhang wird mit dem OCL-Constraint im Listing 4.8 gewährleistet.

Listing 4.8: Nur gültige Parameter und gleiche Typen dürfen gemappt werden

```
context ParameterMapping inv only_valid_params:
        self.parentSubscriber.parentChannel.parameter->includes(self.
            channelParameter) and
        self.parentSubscriber.operation.parameter->includes(self.operationParameter
            )

context ParameterMapping inv equal_types:
        self.channelParameter.typeReference = self.operationParameter.typeReference
```

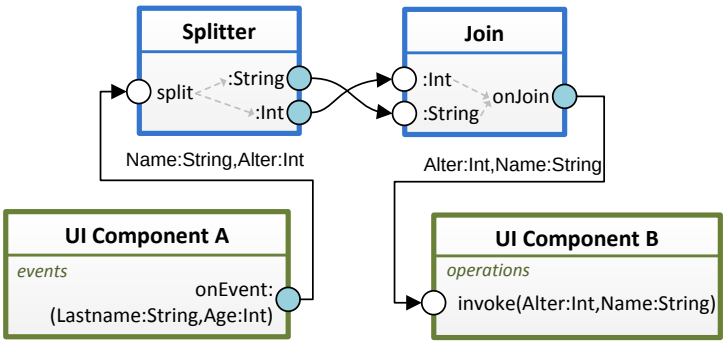
Ein Vorteil des *ParameterMappings* ist, dass die Hinzunahme eines *Splitters* und eines dahintergeschalteten *Joins* vermieden wird, wenn sich die Reihenfolge der Parameter auf dem *DataChannel* von denen der *Operation* unterscheidet. In Abbildung 4.13a ist dieser Sachverhalt unter Verwendung von *Splitter* und *Join* dargestellt, in Abbildung 4.13b mit direktem *ParameterMapping* dargestellt. Die Komponenten sind vereinfacht abgebildet, um nur die relevanten Informationen darzustellen.

Mit einem *Splitter* müssten die Daten des sendenden *Publishers* aufgetrennt und, wie in Abbildung 4.13a dargestellt, über eine *Join*-Komponente in die richtige Reihenfolge gebracht und wieder vereint werden. Diese Vorgehensweise kann mit einem *ParameterMapping* am *Subscriber* leicht vermieden werden, da nur die richtigen Daten aufeinander abgebildet zu werden brauchen. Dies ist in Abbildung 4.13b dargestellt. Die rot markierten Parameter-Namen gehören zum *DataChannel* und die blau markierten zur aufgerufenen *Operation*. Im *ParameterMapping* werden diese aufeinander abgebildet. Außerdem müssen zwei Invarianten für *ISubscriber* gewährleistet werden. Zum einen dürfen, analog zu den *Event*s beim *Publisher*, keine inneren *Operation*s von *CompositeComponents* referenziert werden, zum anderen ist es nur sinnvoll, maximal soviele *ParameterMappings* zu spezifizieren, wie Parameter am *DataChannel* angegeben wurden. Diese Umstände werden durch das Listing 4.9 ausgedrückt.

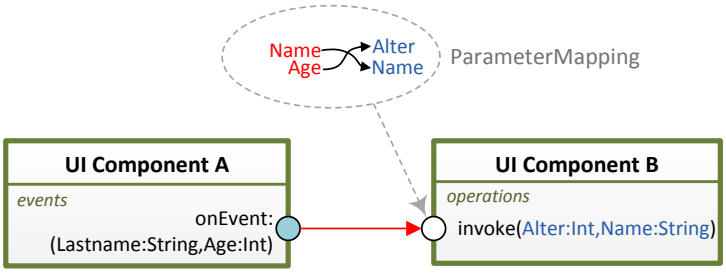
Listing 4.9: Keine Referenzen auf innere *Operation*s von *CompositeComponents* und gültige Anzahl von *ParameterMappings*

```
context ISubscriber inv no_CompositeComponent_operations_allowed:
        self.operation ->forAll(op|
        not CRUISeMetaModel::CompositeComponent.allInstances()->forAll(comp|
                comp.eContents()->select(tempOp| tempOp.oclIsTypeOf(CRUISeMetaModel
                    ::Operation))-> includes(op)
                )
        )

context ISubscriber inv restrict_max_Mappings_to_channel_param_count:
        self.parentChannel.parameter->size() >= self.parameterMapping->size()
```



(a) Indirektes *ParameterMapping* mittels *Splitter* und *Join*



(b) Direktes Parameter-Mapping am *Subscriber*

Abbildung 4.13: Vermeidung der Nutzung von *Splitter* und *Join* durch *ParameterMapping*

Schließlich muss noch eine weitere wichtige Bedingung garantiert werden. Besitzt ein *Subscriber* keine *ParameterMappings*, muss sichergestellt werden, dass der Typ des *DataChannels* dem der *Operation* entspricht. Weiterhin muss dies auch für den Typ des *Events* des *Publishers* zutreffen. Folgendes Constraint gewährleistet dies.

Listing 4.10: Gleiche Typen

```
1 context ISubscriber inv:
2         self.parameter->forAll(param|
3                 self.publisher.event.parameter->at(self.parameter->indexOf(param)).
                      typeReference = param.typeReference
4         ) and
5         self.subscriber.parameterMapping->isEmpty()
6         implies
7         self.parameter->forAll(param|
8                 self.subscriber.operation.parameter->at(self.parameter->indexOf(
                      param)).typeReference = param.typeReference
9         )
```

Ein letzter wichtiger Punkt aus Abbildung 4.12 sind die Referenzen von *IPublisher* und *ISubscriber* zur Klasse *View*. Damit kann modelliert werden, in welchen *Views* die jeweiligen Sender und Empfänger von Daten aktiv sein sollen. Dieses Vorgehen ist relevant, wenn UI-Komponenten in bestimmten Sichten ausgeblendet sind und demnach ein- und ausgehende Kommunikation nicht notwendig ist. Aus Performanzgründen sollte die Interaktion dieser Komponenten für diese *Views* deaktiviert werden.

Mit dem in diesem Abschnitt konzipierten *Communication Model* kann nun die lose gekoppelte Kommunikation der Komponenten nach dem Publish/Subscribe-Paradigma modelliert werden. Dieses Paradigma wurde durch die Möglichkeit der *ParameterMappings* und die Referenzen auf *Views* erweitert, in denen *Publisher*

und *Subscriber* aktiv sein sollen. Auch der Spezialfall der Interaktion von parameterlosen *Event*s und *Operation*s ist möglich, wodurch kein Datentransport, sondern nur die Ausführung von *Operation*s stattfindet. Im Gegensatz zu WebML kann mit dem *Communication Model* nicht nur 1:1-Kommunikation modelliert werden; da sich *Publisher* und *Subscriber* Datenkanäle teilen können, ist sogar n:m-Kommunikation modellierbar. Außerdem kann, im Gegensatz zu den anderen vorgestellten Ansätzen, die Datenübertragung bei Transitionen zwischen Sichten realisiert werden. Dies hängt mit der Tatsache zusammen, dass sowohl Kommunikation als auch Transitionen durch *Event*s ausgelöst werden. Dadurch ist Datentransport applikationsweit und nicht nur innerhalb einer Sicht der Web-Applikation möglich. Mit diesem Modell wurden die Kernkonzepte von UI-Mashups veranschaulicht. Auf dieser Niveaustufe sind schon komplexe Web-Applikationen möglich. Um zusätzlich die Modellierung von UI-Mashups, die auf bestimmte Kontexte reagieren können beziehungsweise sich an den Benutzer anpassen, zu ermöglichen, wurde das entwickelte Kompositionsmodell um das *Adaptivity Model* erweitert. Dieses schließt das Kompositionsmodell ab und wird im folgenden Kapitel 4.2.6 vorgestellt.

4.2.6 Adaptivity Model

Das *Adaptivity Model* zählt nicht zu den Hauptkonzepten von UI-Mashups. Es ist eine Erweiterung des bis hierhin entworfenen Metamodells und dient der Anpassbarkeit der Web-Applikation an den Kontext des Nutzers. Wie schon in Anforderung 12 dargestellt, wird die Adaption der Benutzerschnittstelle zur Laufzeit des UI-Mashups immer wichtiger. Darauf bezogene Regeln, die dann von der Laufzeitumgebung ausgewertet und angewendet werden, müssen schon zum Zeitpunkt des Entwurfs festgelegt werden. So wurden in den Kapiteln 3.2 und 3.3 die Methode von WebML mit *Context Clouds* und die Methode von UWE, der aspektorientierten Adaptivitätsmodellierung, vorgestellt. Wie schon in der Bewertung von WebML (siehe Abschnitt 3.2) dargelegt, ist es nicht erstrebenswert, zu adaptierende Elemente als solche zu deklarieren, da somit eine Abhängigkeit zum Adaptionsmodell entsteht. Im Gegensatz dazu ist der aspektorientierte Modellierungsansatz von UWE sehr vielversprechend. Dabei werden Adaptivitätskonzepte außerhalb der anzupassenden Elemente spezifiziert, so dass eine lose Kopplung entsteht. In Abbildung 4.14 ist dargestellt, wie aspektorientierte Adaptivität grundsätzlich funktioniert.

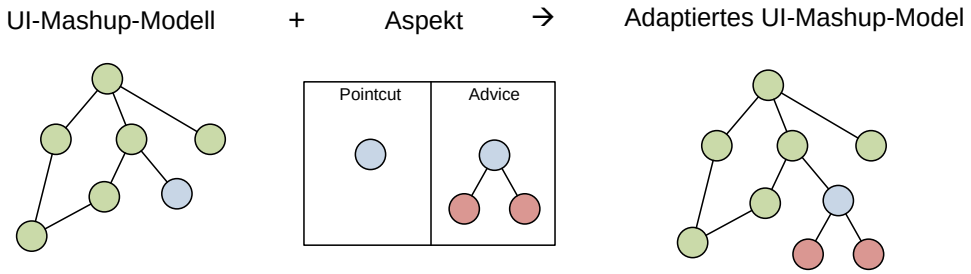


Abbildung 4.14: Funktionsweise von aspektorientierter Adaptivität

Das hellblau markierte Element des Ausgangsmodells ist das anzupassende Element. In der Mitte ist ein Aspekt dargestellt, der aus einem *Pointcut* und einem

Advice besteht. Der *Pointcut* markiert die Stelle des Modells, die adaptiert werden soll und der *Advice* gibt die Änderung an. Der Aspekt wird dann durch einen Aspektweber in das originale Modell eingewoben [Lop02], wodurch das Ergebnis im Zielmodell, mit den neu hinzugekommenen, rot markierten Elementen entsteht. Dies ist das Vorgehen in UWE, hat jedoch auch einen Nachteil. Adaptivitätsmodellierung hängt sehr stark mit dem *User Model* zusammen. Wurde dies einmal angegeben, können nur darauf basierende Adaptivitätskonzepte modelliert werden. So ist es beispielsweise nicht möglich, Adaptivitätsregeln zu formulieren, die auf einer anderen Datenbasis arbeiten.

Aus diesem Grund werden für das hier entwickelte *Adaptivity Model* der aspektorientierte Ansatz von UWE aufgegriffen und noch weitere Konzepte mit eingebracht, welche die oben beschriebenen Probleme lösen. In Abbildung 4.15 wird das Metamodell der Adaptivitätsmodellierung dargestellt und nachfolgend erläutert.

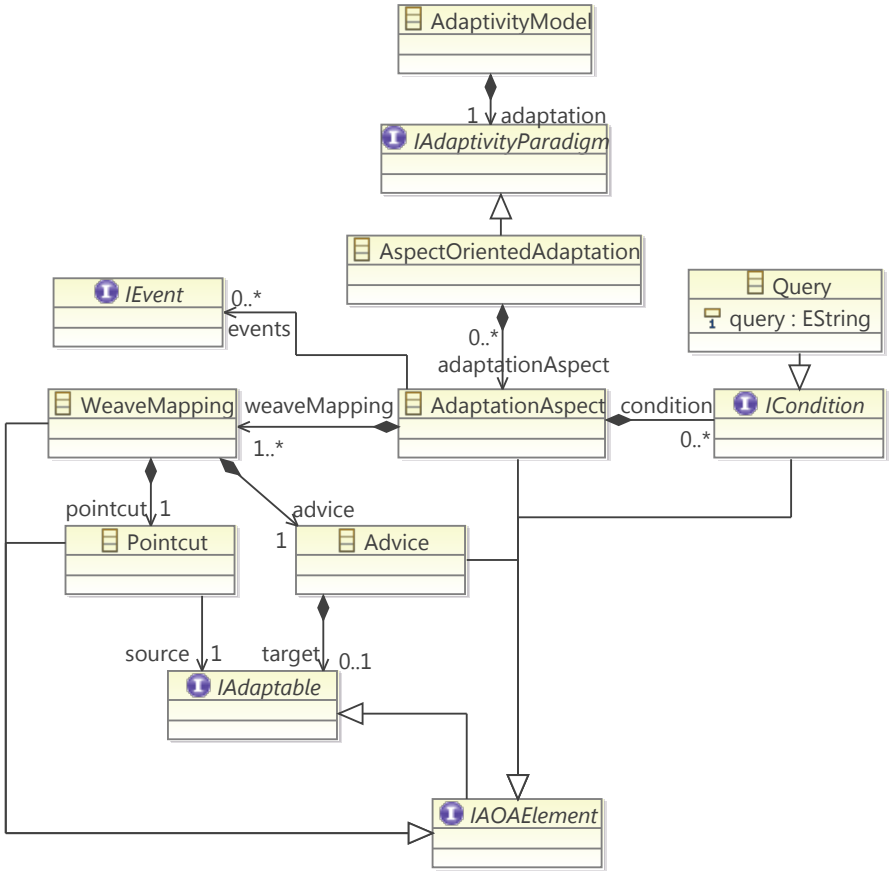


Abbildung 4.15: Das *Adaptivity Model*

Es ist zu erkennen, dass auch hier wieder der Grundstein für andere Adaptivitätsparadigmen gelegt wurde. Dafür existiert das Interface *IAdaptivityParadigm*, welches durch neue Klassen erweitert werden kann. Die hier gewählte Methode beginnt mit der Klasse *AspectOrientedAdaptation*. Diese besitzt beliebig viele *AdaptationAspects*, welche von *NamedElement* erben. An dieser Stelle unterscheidet sich der Ansatz von UWE. In einem *AdaptationAspect* können *Event*s referenziert werden. Diese Beziehung hat zwei Hintergründe, wodurch wiederum zwei unterschiedliche Arten der Adaptivitätsmodellierung entstehen. Zum einen dienen die referenzierten *Event*s der

Auslösung des spezifizierten *AdaptationAspects*. Wird demzufolge ein *Event* an einer Komponente generiert, muss die Laufzeitumgebung die *AdaptationAspects* ausführen, die auf dieses *Event* verweisen. Dies stellt eine einfache Art der aspektorientierten Adaptivität dar. Eine zweite Form wird zum anderen mit der Hinzunahme einer *Condition* realisiert. Möchte der Autor die durch ein *Event* gelieferten Daten analysieren und die Adaption davon abhängig machen, muss nur ein *Query* hinzugefügt werden. Da die transportierten Daten einem XML-Schema entsprechen, kann mit dem *Query* ein XPath-Ausdruck spezifiziert werden, anhand dessen die Laufzeitumgebung entscheiden kann, ob adaptiert werden muss oder nicht. Wie in Abbildung 4.15 zu sehen ist, besteht zwischen *AdaptationAspect* und *ICondition* eine 1:n-Beziehung. Dadurch wird die Spezifizierung erleichtert, da nicht in einer einzigen *Condition* ein langer XPath-Ausdruck angegeben werden muss, sondern dieser in mehrere Bedingungen aufgeteilt werden kann, was einer ODER-Verknüpfung entspricht. Somit kann entweder nur durch *Event*s oder mit der zusätzlichen Analyse der damit transportierten Daten Adaption ausgelöst werden. Außerdem muss auch hier wieder sichergestellt werden, dass keine *Event*s von *CompositeComponents* referenziert werden. Diese Bedingung wird mit dem OCL-Constraint in Listing 4.11 realisiert.

Listing 4.11: Keine *Event*s von *Composite Components*

```
context AdaptationAspect inv:
        self.events ->forAll(ev|
        not CRUISeMetaModel::CompositeComponent.allInstances()->forAll(comp|
                comp.eContents()->select(tempEv| tempEv.oclIsTypeOf(CRUISeMetaModel
                    ::Event))-> includes(ev)
        )
)
```

Eine Besonderheit der hier zu modellierenden Initiierung von Adaptivität mit *Event*s ist, dass es keine Einschränkung hinsichtlich der Frage gibt, durch welche Komponenten diese ausgelöst wird. So können Adaptivitätsprozesse nicht nur als Reaktion auf CCs, sondern ebenso als Reaktion aller Komponenten durchgeführt werden. Dieses Vorgehen ist von Bedeutung, wenn durch CCs gesammelte Kontextdaten erst noch von LCs verarbeitet werden sollen. Durch eine Einschränkung auf bestimmte *Event*s wäre Adaption auf weiterverarbeitete Kontextdaten nicht möglich.

Des Weiteren besitzt *AdaptationAspect* mindestens ein *WeaveMapping*. Mit diesem Mapping werden *Pointcut* und *Advice*, wie sie in Abbildung 4.14 schon dargestellt sind, spezifiziert. Diese Bezeichnungen wurden aus dem HyperAdapt[7]-Projekt [NvdSH+09] übernommen, da dort ein aspektorientierter Adaptivitätsansatz für Web-Applikationen entworfen wird, der den Vorgang des Webens sogar zur Laufzeit der Anwendung durchzuführen verspricht. *Pointcut* besitzt mit dem Parameter *source* eine Referenz zum Interface *IAdaptable*. Damit kann auf genau ein Element der gesamten Komposition verwiesen werden, wodurch angegeben wird, welches Element adaptiert werden soll. Von *IAdaptable* erben alle bisher vorgestellten Klassen, bis auf die im Abschnitt 4.2.2.1 beschriebenen des Datentypkonzeptes. Ein *Advice* beinhaltet hingegen eine neu erstellte Instanz von *IAdaptable*. Dadurch ist es zum einen möglich, ein beliebiges Element des konkreten Kompositionsmodells zu adaptieren, indem dafür ein neues modelliert wird. Wird hingegen keine neue In-

[7] siehe http://www.hyperadapt.net

stanz von *IAdaptable* spezifiziert, dann bedeutet dies, dass das Element aus dem Modell entfernt wird. Zum anderen können zu adaptierende Elemente im *Advice* sehr detailliert adressiert werden, wodurch der Adaption komplexer Konzepte des konkreten Modells vorgebeugt wird, wenn nur kleine Teile dessen angepasst werden sollen. Außerdem kann ein *AdaptationAspect* mehrere *WeaveMappings* besitzen, wodurch mehrere Elemente mit demselben Aspekt angepasst werden können. Mit diesem Konzept ist es beispielsweise möglich, dass aufgrund eines *AdaptationAspects* eine *ViewTransition* nach der Anpassung zu einem anderen *View* führt als vorher, ein *Layout* durch ein anderes ersetzt oder aber eine beliebige Komponente neu konfiguriert wird. Zu erwähnen ist außerdem, dass auch die hier vorgestellten Klassen des *Adaptivity Models* Unterklassen von *IAdaptable* sind. Damit können sogar *AdaptationAspects* angepasst werden, wodurch hochflexible Adaptivität modellierbar ist. Trotzdem muss am *WeaveMapping* eine Invariante gelten, die im Listing 4.12 ausgedrückt wird. Der konkrete Typ des referenzierten *IAdaptable* am *Pointcut* muss derselbe sein wie der am *Advice* neu erstellte, da nach der Adaption nur so die Konformität bezüglich des Metamodells gewahrt wird.

Listing 4.12: Typen im *WeaveMapping* müssen übereinstimmen

```
1 context WeaveMapping inv:
2         self.pointcut.source.oclIsTypeOf(self.advice.target)
```

Mit diesem Vorgehen der aspektorientierten Adaptivitätsmodellierung wird ein hohes Maß an Flexibilität und Anpassbarkeit erreicht. Der Autor kann, im Gegensatz zu UWE, anhand beliebiger Daten Adaptivität modellieren, da die *Context Components* mit allen Arten von Komponenten interagieren können. Zudem sind bis auf die deklarierten Typen alle weiteren konkreten Klassen adaptierbar. Mit diesem Modell ist das Kompositionsmodell abgeschlossen. Um den Autor bei der Modellierung jedoch noch weiter zu unterstützen, wird im kommenden Kapitel 4.3 untersucht, inwieweit einzelne Modelle ineinander überführt beziehungsweise ob Teile zur Generierung benutzt werden können.

4.3 Transformationen

In diesem Kapitel wird analysiert, wie sich das konzipierte Kompositionsmodell in den traditionellen MDA-Ansatz aus Abschnitt 2.5 eingliedert. Dazu wird erläutert, welcher Art die konzipierten Modelle und welche Transformationen möglich und sinnvoll sind. Aus den gewonnenen Erkenntnissen werden sich grundlegende Abhängigkeiten der Modelle ergeben, die ein prinzipielles Vorgehen bei der Modellierung vorgeben werden.

In der Abbildung 2.4 aus Kapitel 2.5.2 ist zu sehen, dass die traditionelle Transformationskette von einem unabhängigen Domänenmodell (CIM) in ein unabhängiges Systemmodell (PIM) überführt wird, welches wiederum plattformspezifisch verfeinert (PSM) und schließlich in den Code der Zielumgebung überführt wird. Das konzipierte Kompositionsmodell rückt jedoch von diesem Prozess dahingehend ab, dass es zum einen kein CIM besitzt und zum anderen den Transformationsprozess konzeptionell trennt. Es wurde bewusst darauf geachtet, dass der Autor des UI-Mashups keine Entscheidungen bezüglich der Zielplattform treffen muss, um das entworfene Modell so generisch wie möglich zu halten. Dadurch entfällt die Generie-

rung eines PSMs und dessen Verfeinerung. Stattdessen werden lediglich Generatoren vom PIM zum jeweiligen Code der Zielumgebung benötigt, wodurch Autoren der Web-Applikation vollkommen unabhängig modellieren können. Demnach ist durch das entworfene Kompositionsmodell die Erzeugung des Codes klar von der Modellierung getrennt, wodurch die Erstellung ähnlicher PSMs, die sich nur in verschiedenen Plattform-Aspekten unterscheiden, vermieden wird. Als Folge der Tatsache, dass das Kompositionsmodell nur aus PIMs besteht, sind nur M2M-Transformationen nötig. Transformationen von *Model to Code (M2C)* werden ausgelagert. Die im Folgenden beschriebenen Transformationen sind in so entworfenen Modellen durchführbar.

4.3.1 Conceptual Model verfeinern

Im *Conceptual Model* gibt es mehrere Komponenten, die Eigenschaften besitzen, aufgrund derer andere Eigenschaften generiert werden können. Aufgrund der Tatsache, dass neue Eigenschaften zum selben Modell hinzukommen, spricht man von einer Verfeinerung.

Zum einen können SACs verfeinert werden. Wie im Abschnitt 4.2.2.2 schon dargestellt, besitzen diese ein Attribut, das *descriptionLocation* genannt wird. Mit diesem wird die Beschreibung referenziert, die angibt, wie ein Backend-Dienst aufgebaut ist und welche Funktionen angesprochen werden können. Anhand dieser Service-Beschreibung kann eindeutig abgeleitet werden, welche *Event*s und *Operation*s eine SAC besitzen muss, um den jeweiligen Service vollständig abbilden zu können. Für jede definierte Funktion kann demnach für die SAC eine *Operation* und ein *Event* generiert werden. Die *Operation* dient der Initiierung der Funktion des Services, und das *Event* dient der Rückmeldung. Da die Service-Beschreibung auch die Typung der Parameter für Anfrage und Antwort enthält, können diese auch generiert werden. Somit ist es möglich, dass einzig durch die Angabe der *descriptionLocation* die korrespondierenden *Event*s und *Operation*s für SACs automatisiert zu erzeugen.

Eine weitere Komponente, für die *Event*s generiert werden können, ist der *Splitter*. In Abschnitt 4.2.2.2 wurde einerseits beschrieben, dass am *Splitter* für jeden Teiltyp eines zusammengesetzten Typs eines *Operation*-Parameters ein *Event* bereitgestellt wird, das die Teildaten kommuniziert. Die Erzeugung der *Event*s kann automatisch erfolgen. Steht einmal fest, wie die Struktur des zusammengesetzten Typs des *Operation*-Paramters definiert ist, kann für jeden Teiltyp ein *Event* mit adäquatem Typ generiert werden. Dies ist möglich, da Typen, beschrieben im Abschnitt 4.2.2.1, mittels XML-Schema definiert werden und diese leicht maschinell verarbeitet werden können. Dadurch braucht nur eine den zu teilenden Typ als Parameter beinhaltende *Operation* definiert zu werden, und die *Event*s werden automatisiert erzeugt. Andererseits wurde in Abschnitt 4.2.2.2 beschrieben, dass aus mehreren eingehenden Parametern jeweils neue *Event*s bereitgestellt werden. Auch dieses Vorgehen kann automatisiert durchgeführt werden, indem für jeden eingehenden Parameter ein *Event* generiert wird. Dieses besitzt jeweils einen Parameter, der denselben Typ referenziert wie sein korrespondierender Eingangsparameter.

Der *Join* funktioniert in der umgekehrten Richtung. Man definiert mehrere *Operation*s mit ihren Parametern, woraufhin ein *Event* generiert werden kann, welches die eingehenden Parameter zusammenfasst und ein Datum auf einem Datenkanal kommuniziert, das dem zusammengesetzten Typ entspricht.

Als letzte Komponente, die generierbare Eigenschaften enthält, ist der *Iterator* zu nennen. Wie im Abschnitt 4.2.2.2 dargestellt, kommuniziert er die einzelnen Elemente einer Datensammlung. Anhand des XML-Schemas kann auch für den *Iterator* analysiert werden, welchem Typ die Elemente der Datensammlung entsprechen, und dafür ein *Event* generiert werden, das einen Paramter des Element-Typs besitzt.

Anhand dieser Transformationen können die einzelnen Komponenten konsistent gehalten werden. Ändert sich beispielsweise der Parameter-Typ einer Service-Funktion, braucht die dazugehörige Komponente nicht manuell angepasst zu werden, wobei Fehler unterlaufen können oder dies gänzlich vergessen wird. Die *Event*s und *Operation*s werden neu generiert, wodurch dieses Vorgehen dem Single-Source-Prinzip gerecht wird. Daraus ergibt sich der Vorteil, dass Änderungen nur an einer Stelle durchgeführt zu werden brauchen und an allen anderen Stellen generiert oder referenziert werden.

4.3.2 Layout Model zu Screenflow Model

Eine weiterere mögliche Transformation kann vom *Layout Model* zum *Screenflow Model* durchgeführt werden. Hierbei handelt es sich jedoch nicht um die komplette Generierung, sondern um die Erzeugung eines Grundgerüstes des *Screenflow Models*. Es wird davon ausgegangen, dass hinter jedem modellierten *Layout* die Absicht steckt, es auch dem Nutzer zu präsentieren. Demnach wird für jedes *Layout* eine *View* benötigt. Derart kann auch die Transformation durchgeführt weden. Das generierte Grundgerüst des *Screenflow Models* erhält genauso viele *Views* wie *Layouts*. Die Bezeichner werden an die Namen der *Layouts* angelehnt, indem sie den Suffix »View« erhalten. Außerdem wird für jede *View* die Referenz auf das korrespondierende *Layout* gesetzt und eine noch leere *ViewTransition* generiert. Somit kann aus dem *Layout Model* ein Teil des *Screenflow Models* erzeugt werden.

4.3.3 Conceptual Model zu Communication Model

Eine weitere sehr hilfreiche Transformation ist die Erzeugung des *Communication Models*. Hierbei werden die *Operation*s und *Event*s der modellierten Komponenten analysiert. Aus der Typung der Parameter der einzelnen *Event*s kann abgeleitet werden, welche *DataChannels* existieren müssen. Besitzt beispielsweise ein *Event* einen Parameter des Typs `String` und einen des Typs `Int`, so wird auch ein *DataChannel* benötigt, um die erzeugten Daten kommunizieren zu können. Demnach wird für jede unterschiedliche *Event*-Typung ein Datenkanal, der dieselben Typen besitzt, generiert und der dazugehörende *Event* als *Publisher* registriert. Existiert für ein *Event* schon ein *DataChannel*, der dessen Typung entspricht, so wird kein neuer Kanal gleichen Typs generiert, sondern nur ein weiterer *Publisher* erstellt. Die Erzeugung der *Subscriber* erfolgt ähnlich. Es werden die Typen aller *Operation*s analysiert, und dabei wird überprüft, ob entsprechende *DataChannels* generiert wurden. Im positiven Falle werden dort neue *Subscriber* generiert, die jeweils eine Referenz auf die zugehörige *Operation* der Komponente erhalten. Somit entsteht auch hier ein Grundgerüst für das *Communication Model*. Wie jedoch in Kapitel 4.2.5 schon erwähnt, können kommunizierte Daten verschiedener *Event*s zwar die gleiche Struktur besitzen, aber einer unterschiedlichen Semantik unterliegen. Diese

Unterscheidung kann nicht in der Transformation berücksichtigt werden. Deshalb muss das generierte *Communication Model* auch hier vom Autor überprüft und gegebenenfalls Anpassungen durchgeführt werden.

Aus den hier beschriebenen PIM-to-PIM-Transformationen ergeben sich prinzipielle Abhängigkeiten für die Modellierung von UI-Mashups. Das *Conceptual Model* ist erwartungsgemäß die Basis für alle anderen. Dieses muss als erstes modelliert werden. Danach ist der Autor nicht zwingend an eine Reihenfolge gebunden. In Abbildung 4.16 ist dargestellt, wie die Modelle zusammenhängen.

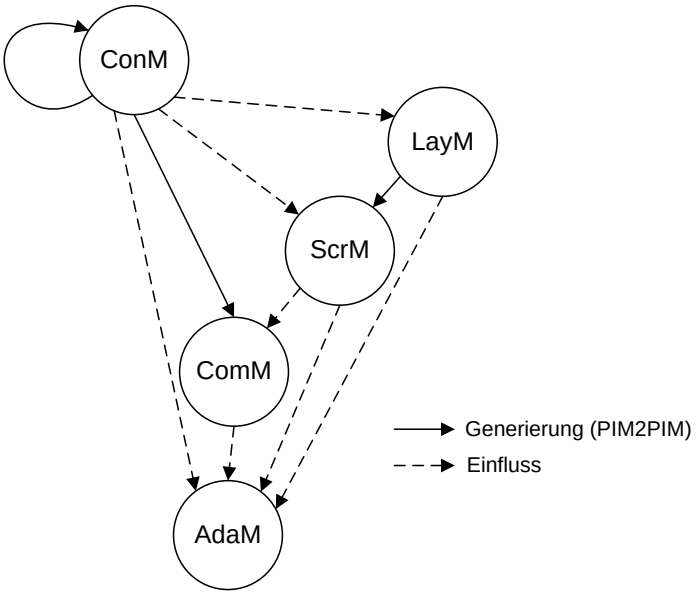


Abbildung 4.16: Abhängigkeiten zwischen den Modellen

Durchgezogene Verbindungen bedeuten, dass Modelle oder Teile davon generiert werden können. Gestrichelte Verbindungen stehen für den Einfluss, den ein Modell auf ein anderes hat, was bedeutet, dass Elemente des Quellmodells im Zielmodell referenziert werden können oder müssen. Abbildung 4.16 muss von oben nach unten gelesen werden. Je weniger Verbindungen man in einem erstellten Modell entsprechend der Abbildung abdeckt, desto weniger Features besitzt es. Um Vollständigkeit der Teilmodelle zu erlangen, müssen demnach alle darüber liegenden Teilmodelle existieren. Somit wäre die übliche Reihenfolge eines Modells, welches alle Teilmodelle enthält, die folgende: das *Conceptual Model* wird erstellt und mit einer ersten Transformation verfeinert; das *Layout Model* wird gestaltet und Referenzen auf das *Conceptual Model* gesetzt; daraus wird das *Screenflow Model* generiert, welches Verweise auf *Event*s des *Conceptual Models* besitzt; ein Grundgerüst des *Communication Models* wird automatisch erzeugt und die Verweise auf *Views* des *Screenflow Models* gesetzt; zum Schluss wird das *Adaptivity Model* erstellt, welches beliebige Elemente aller anderen Modelle referenzieren kann.

Damit wurden die möglichen Transformationen dargestellt und die Abhängigkeiten der Modelle erläutert. Im folgenden Kapitel 4.4 erfolgt nun eine Zusammenfassung der Konzeption und die anschließende Bewertung.

4.4 Zusammenfassung

Ausgehend von den in Kapitel 2.6 formulierten Anforderungen, wurde in diesem Kapitel das generische Kompositionsmodell für UI-Mashups vorgestellt. Nachfolgend wird das vorliegende Konzept zusammengefasst und bewertet.

Mit dem *Conceptual Model* aus Abschnitt 4.2.2 wird die Modellierung der im UI-Mashup enthaltenen Komponenten ermöglicht. Im Fokus dieses Modells lag die Integration des in Abschnitt 4.1 vorgestellten CRUISe-Komponentenmodells, sowie die Modellierbarkeit aller in CRUISe vorkommenden Komponentenarten. Mit der *CompositeComponent* wird die rekursive Komposition unterstützt, wodurch man UI-Mashups auf unterschiedliche Abstraktionsstufen heben kann. Des Weiteren ist mit der Klasse *Environment* modellierbar, welche Komponenten-*Operations* beim Start der Web-Applikation ausgeführt werden sollen, wodurch Komponenten initialisiert werden können. Mit dem in Abschnitt 4.2.2.1 erläuterten Konzept der Typung und Parametrisierung wurde ein Mechanismus geschaffen, der es erlaubt, Typen einfach anhand von XML-Schemata zu definieren. Ein Vorteil ist, dass auch schon vorhandene XSD-Dateien referenziert werden können, wodurch Typen nicht redundant angelegt zu werden brauchen und jeder UI-Mashup-Autor seine eigene Typbasis aufbauen kann. Des Weiteren können im *Conceptual Model* zusätzlich *Styles* definiert werden, mit denen das *Look & Feel* der UI-Komponenten spezifiziert wird. Auch hier kann auf vorhandene CSS-Definitionen zurückgegriffen werden. Dieses Modell dient als Basis für die anderen und kann mit der in Abschnitt 4.3.1 vorgestellten Transformation automatisch verfeinert werden.

Das in Abschnitt 4.2.3 vorgestellte *Layout Model* ermöglicht die Positionierung der UI-Komponenten. Dafür können verschiedene *Layouts* benutzt werden, die ineinander geschachtelt werden können. Die zu positionierenden UI-Komponenten werden dann einfach an der entsprechenden Lokation referenziert. Mit der in Abschnitt 4.3.2 vorgestellten Transformation kann aus diesem Modell das Grundgerüst für das *Screenflow Model* generiert werden, wobei für jedes definierte *Layout* eine *View* erzeugt wird.

Der Kontrollfluss des UI-Mashups wird mit dem in Abschnitt 4.2.4 dargestellten *Screenflow Model* spezifiziert. Damit werden *Views* und Transitionen zwischen ihnen angegeben. Hervorzuheben ist, dass Übergänge von *Event*s ausgelöst werden, die nicht allein auf Nutzerinteraktion beschränkt sind, wodurch sich dieses Modell von anderen Ansätzen abhebt.

Das *Communication Model* aus Abschnitt 4.2.5 ermöglicht die Modellierung der Interaktion aller Komponenten. Es wurde nach dem *Event*-basierten Publish/Subscribe-Prinzip umgesetzt, wobei sich ein Broker um die Synchronisierung sendender und empfangender Komponenten kümmert. Dafür müssen *DataChannels* definiert werden, die anhand der Parameter der modellierten *Event*s und *Operation*s im *Conceptual Model* mit der in Abschnitt 4.3.3 erläuterten Transformation generiert werden können. Außerdem ermöglicht ein Mapping der Kanalparameter auf die Eingangsparameter des *Subscribers*, dass auch nur Teilmengen der transportierten Daten am Empfänger konsumiert werden können.

Mit dem *Adaptivity Model* aus Abschnitt 4.2.6 wurde schließlich die Möglichkeit der aspektorientierten Adaptivitätsmodellierung geschaffen. Damit kann als Reaktion auf *Event*s modelliert werden, welche Modellelemente angepasst werden und wie

sie nach der Adaption aussehen sollen. Durch Spezifizierung eines XPath-Ausdruckes wurde eine erste Art der Angabe von Bedingungen vorgestellt, anhand derer Adaptionsprozesse durchgeführt werden können. Es sollte außerdem in Betracht gezogen werden, solche Bedingungen einfacher angeben zu können, etwa durch Modellierung von Regeln.

Mit diesen fünf Modellen wurden alle funktionalen Anforderungen, sowie die Forderung nach der Trennung von Verantwortlichkeiten auf Modellebene erfüllt. Zusätzlich wird durch die Verwendung von Interfaces an geeigneten Stellen im Kompositionsmodell die nachträgliche Erweiterbarkeit sichergestellt. Da nur PIMs erstellt werden, ist ein konkretes Kompositionsmodell vollkommen unabhängig von der Zielplattform und einzelne Modelle oder Teile davon können leicht wiederverwendet werden. Somit kann abschließend gesagt werden, dass alle Anforderungen, die unabhängig vom Prototyp sind, mit dem hier vorgestellten Kompositionsmodell erfüllt wurden.

5 Implementierung

Im Kapitel 4 wurde ein generisches Kompositionsmodell konzipiert, mit dem leistungsstarke und dynamische UI-Mashups modelliert werden können. In diesem Kapitel werden nun die prototypische Umsetzung vorgestellt und wichtige Implementierungsentscheidungen begründet. Dazu wird zunächst die Wahl der Modellierungsumgebung in Kapitel 5.1 erläutert. Anschließend werden die Implementierung des Kompositionsmodells in Kapitel 5.2, die Implementierung der vorgestellten Transformationen in Kapitel 5.3, sowie die Validierung an einer Beispielanwendung in Kapitel 5.4 vorgestellt. Abschließend wird ein Fazit gezogen.

5.1 Wahl einer geeigneten Modellierungsumgebung

Wie schon in Abbildung 2.3 dargestellt, befindet sich das konzipierte Kompositionsmodell auf der M2-Ebene der MOF-Architektur. Da sich die UML auf derselben Ebene befindet, könnte man das Kompositionsmodell durch Erstellung entsprechender Stereotypen umsetzen. Dieses Vorgehen wird jedoch ausgeschlossen. Ein Teil der Gründe dafür wurden schon im Kapitel 3.3 genannt. Außerdem würde es nicht ausreichen, Stereotypen zu erstellen, sondern alle Assoziationen und Vererbungshierarchien des Metamodells müssten auf die UML abgebildet werden. Dies ist jedoch nicht möglich, da die UML nur leichtgewichtig erweitert werden kann, existierende Beziehungen jedoch nicht aufgelöst werden können. Außerdem würden somit die eigentlichen Bedeutungen der Metaklassen der UML verfälscht werden. Als UML-Erweiterung wären zudem noch alle anderen aus der UML bekannten Konzepte modellierbar, die aber nicht in den Rahmen von UI-Mashups gehören.

In der Konsequenz wird das Kompositionsmodell hier als eigenständiges, autonomes Metamodell implementiert. Damit wird erreicht, dass alle in Kapitel 4 erläuterten Konzepte 1:1 umgesetzt werden können und genau die Bedeutung erhalten, die ihnen zugeschrieben wurde. Dadurch wird das konzipierte Kompositionsmodell einzig für die Modellierung von UI-Mashups implementiert, wodurch sich deren Autoren sofort in ihrer gewohnten Domäne befinden. Deshalb ist es leicht zu erlernen und konkrete Modelle sind leichter lesbar. Damit werden gewissermaßen nebenbei die Anforderungen 4 und 5 abgedeckt.

Als Voraussetzung bei der Wahl des Modellierungs-Frameworks ist es unerlässlich, dass das umgesetzte Kompositionsmodell MOF-konform ist und auch im XMI-Format serialisiert wird. Nur so können konkrete Modelle zwischen unterschiedlichen IDEs ausgetauscht und weiterverarbeitet werden, und Autoren sind nicht an bestimmte Werkzeuge gebunden. Für die Erstellung eines Metamodells wird eine Umgebung benötigt, in der ein Modell zur Beschreibung von Metamodellen verfügbar ist. Ein solches Modell wird, wie in Abbildung 2.3 dargestellt, Meta-Metamodell genannt.

Vom Eclipse Modeling Framework (EMF)[1] wird das Meta-Metamodell namens Ecore zur Verfügung gestellt. Die Erfahrungen, die im EMF mit Ecore gesammelt wurden, trugen wesentlich zur Entwicklung der aktuellen Version von MOF bei. Des Weiteren werden Ecore-Modelle im XMI-Format serialisiert, und die Hauptkonzepte der MDA werden von EMF unterstützt [SBPM08, S. 40]. Außerdem ist EMF ein integraler Bestandteil der quelloffenen Eclipse-Plattform, welche hohe Akzeptanz in der Entwicklergemeinschaft genießt, da sie sehr leistungsstark und einfach zu erweitern ist. Aus diesen Gründen wird das in dieser Arbeit entworfene Konzept mit EMF und dem darin enthaltenen Meta-Metamodell Ecore implementiert und im Folgenden näher vorgestellt.

Mit EMF können Metamodelle auf vier verschiedene Arten erstellt werden. Die erste ist die Modellierung mit Ecore selbst, wodurch ein MOF-konformes Modell im XMI-Format angelegt wird. Bei den drei anderen kann das Metamodell aus folgenden Quellen generiert werden: XML-Schema, annotierter Java-Code und UML-Diagramm. All diese Formen der Repräsentation können mit EMF in die jeweils andere überführt werden. Gemäß der Abbildung 2.3 befinden sich sowohl mit UML erstellte Modelle als auch annotierter Java-Code auf MOF-Ebene M1. Deshalb können diese zwar nicht direkt als Metamodell genutzt, jedoch mit EMF auf die M2-Ebene angehoben werden. Demzufolge wirkt Ecore als Unifikator für die drei anderen Verfahren. Diese Beziehung ist in Abbildung 5.1 dargestellt.

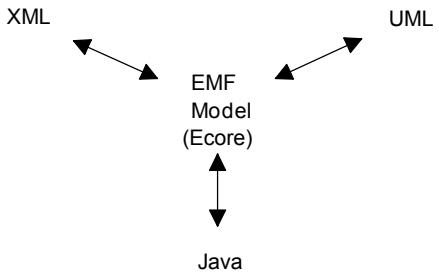


Abbildung 5.1: Ecore als Unifikator [SBPM08, S. 14]

Daraus wird ersichtlich, dass EMF einen weiteren Vorteil bietet. Durch die mögliche Generierung von Java-Code kann ein komplettes API für das Metamodell erzeugt werden. Davon profitiert der *Application Generator* von CRUISe, da so konkrete Kompositionsmodelle einfach programmatisch analysiert und verarbeitet werden können. EMF bringt einen weiteren Vorteil mit. Wurde das Metamodell einmal erstellt, kann daraus ein Eclipse-Plugin generiert werden, welches einen Baum-Editor enthält, mit dem konkrete Modellinstanzen erstellt werden können. Dieser Editor gliedert sich nahtlos in die Entwicklungsumgebung Eclipse ein und dient auch der Erstellung valider Modelle. Somit erhält der Autor umgehend Rückmeldung über modellierte Elemente, die noch nicht konform zum Metamodell sind.

Des Weiteren ist EMF auch das Meta-Metamodell für die im *Eclipse Model To Model Project* enthaltenen Transformationssprachen *ATL*, *QVT Operational* und *QVT Relational*. QVT ist ein Standard der OMG, mit dem Modell-Transformationen spezifiziert werden können [OMG07a]. Die QVT-Spezifikation ist Teil von MOF und beinhaltet einen relationalen und einen operationalen Teil. Ersterer gestattet es,

[1] siehe http://eclipse.org/modeling/emf/

Modelle deklarativ aufeinander abzubilden. Der zweite schreibt die imperative Definition von Abbildungsregeln vor. Dieser besitzt einen wichtigen Vorteil. Die Spezifikation vom operationalen Teil der QVT gestattet es, sogenannte *Black Box Libraries* für Transformationen zu benutzen. Damit können Abbildungen in beliebigen Programmiersprachen spezifiziert werden, solange die interpretierende QVT-Engine damit umgehen kann. *QVT Operational* ist demnach die Eclipse-Implementierung des operationalen Teils und *QVT Relational* die des relationalen Teils. ATL ist eine weit verbreitete QVT-ähnliche Sprache, die auf Initiative der QVT-Ausschreibung der OMG entstanden ist [@ATL04]. Aufgrund der Tatsache, dass QVT so eng auf MOF aufbaut und einen Standard darstellt, wird für die Implementierung der Transformationen QVT gewählt, insbesondere *QVT Operational* (nachfolgend QVTO genannt), da durch den Black-Box-Mechanismus alle erdenklichen Transformationen durchführbar sind, die mit Standard-QVT nicht möglich wären. Dies ist der Fall, wenn Informationen benötigt werden, die selbst nicht im Metamodell liegen. Ein Beispiel dafür wären die aus einer entfernten WSDL-Datei zu generierenden *Event*s und *Operation*s einer SAC.

Somit wurden Eclipse als Modellierungs-Umgebung und das darin integrierte EMF als Modellierungs-Framework gewählt. Als Transformationssprache wurde QVTO ausgewählt. Dadurch können die Transformationen aus Kapitel 4.3 ohne Zwischenschritt umgesetzt werden und direkt auf dem mit Ecore implementierten Kompositionsmodell arbeiten. Im nachfolgenden Abschnitt 5.2 wird die konkrete Umsetzung des konzipierten Kompositionsmodells mit EMF erläutert. Die konkrete Syntax und Semantik von QVT wird anhand der Implementierungen der in Kapitel 4.3 erläuterten Transformationen im Abschnitt 5.3 vorgestellt

5.2 Implementierung des Kompositionsmodells

In Eclipse werden neue Funktionalitäten als Plugins zur Verfügung gestellt. Dabei sollten in sich abgeschlossene Funktionen, auch als allgemeines Programmier-Paradigma bekannt, modularisiert und in ein eigenes Plugin ausgelagert werden, wie in der vorliegenden Arbeit auch praktiziert. In der folgenden Grafik 5.2 werden eine Übersicht sowie die Abhängigkeiten der hier implementierten Eclipse-Plugins dargestellt. Die Plugins für die Modultests und die Logging-Mechanismen sind nicht abgebildet. Alle weiteren werden in diesem und im nächsten Kapitel vorgestellt.

Das Metamodell wurde mit dem *Ecore Diagram Editor*, exakt wie im Kapitel 4 dargestellt, in Form eines *EMF Projects* umgesetzt. Die dort erläuterten Einschränkungen verschiedener Klassen werden hier auch mit OCL umgesetzt, da es, wie schon in Kapitel 3.1.5 dargestellt, den Standard darstellt, Bedingungen für Modelle zu formulieren. Als weiteres Argument zählt, dass OCL-Constraints für MOF-konforme Modelle spezifiziert werden können, sie demnach nicht von konkreten Metamodellen abhängen. Die in Abschnitt 4 dargestellten OCL-Constraints können auf diese Weise bei der Implementierung einfach übernommen werden. Dies geschieht durch Hinzufügen von *Annotations* an die jeweiligen Klassen. Als Folge der Erzeugung dieses Projekts wird im Eclipse-Workspace ein Plugin-Projekt angelegt (*cruise.compositionmodel*), welches das neu erstellte Metamodell über einen Extension-Point an der EMF-Registry anmeldet. Dadurch steht das Kompositionsmodell anschließend in einer zweiten Eclipse-Instanz zur Verfügung. In Abbildung

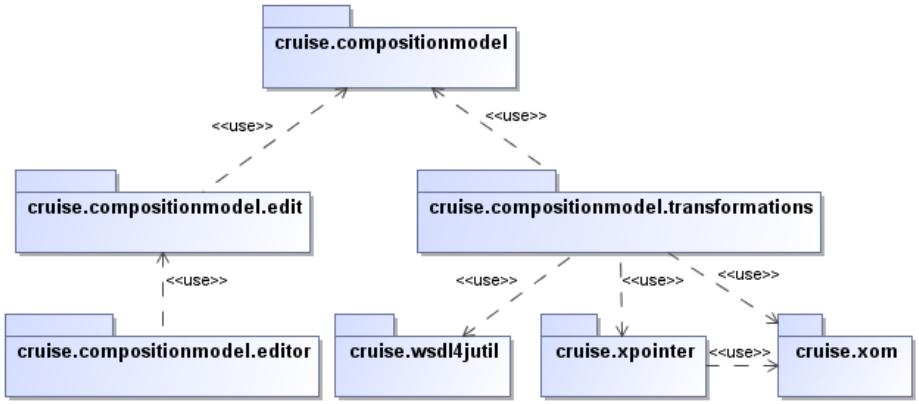


Abbildung 5.2: Abhängigkeiten der implementierten Eclipse-Plugins

5.3 ist dargestellt, wie danach bei der Arbeit mit EMF prinzipiell vorgegangen wird. Darin sind die nötigen Modelle blau hinterlegt und der generierte Java-Code orange. Aus dem *EMF Generator* ausgehende Pfeile stehen für die Generierungen, die von ihm durchgeführt werden.

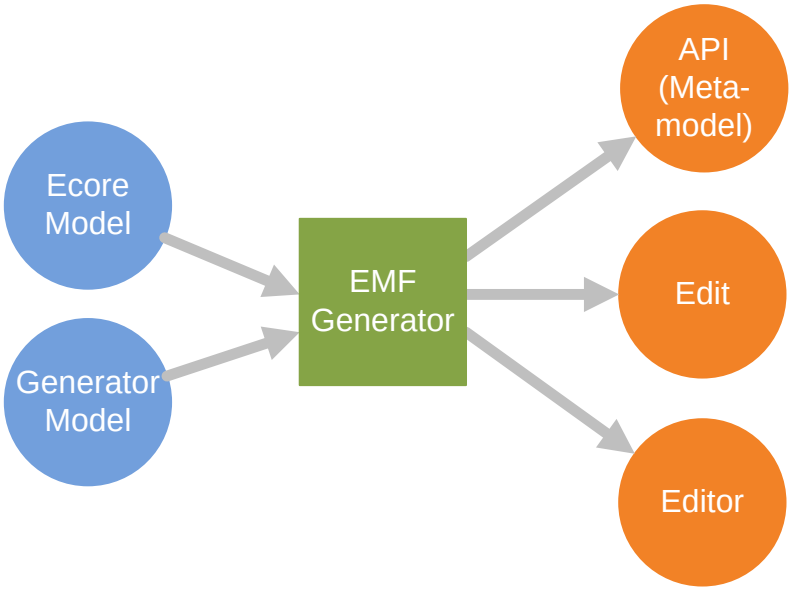


Abbildung 5.3: Generierungs-Prozess in EMF

Für die Generierung des Java-APIs und des Editors muss ein *EMF Generator Model* angelegt werden. Dieses enthält alle modellierten Klassen des Metamodells sowie weitere Informationen zur Code-Generierung, die nicht im eigentlichen Metamodell enthalten sind. Dazu zählen beispielsweise die Bezeichnungen der Packages. Nun führt der *EMF Generator* die Generierung des Codes durch, wonach sich zwei weitere Eclipse-Plugins im Workspace, und das Java-API im Plugin des Metamodells befinden. Das eine trägt den im *Generator Model* angegebenen Namen mit dem Suffix *edit* (hier *cruise.compositionmodel.edit*) und enthält die Zugriffsklassen auf die Klassen des Metamodells, sowie Icons für deren Darstellung im Editor. Das andere trägt den Suffix *editor* (hier *cruise.compositionmodel.editor*) und enthält den generierten Baum-Editor und einen Wizard zur Erzeugung konkreter Modelle. Star-

tet man nun eine zweite Eclipse-Instanz, sind die erzeugten Plugins aktiv und ein Beispielmodell im Editor wird wie in Abbildung 5.4 dargestellt.

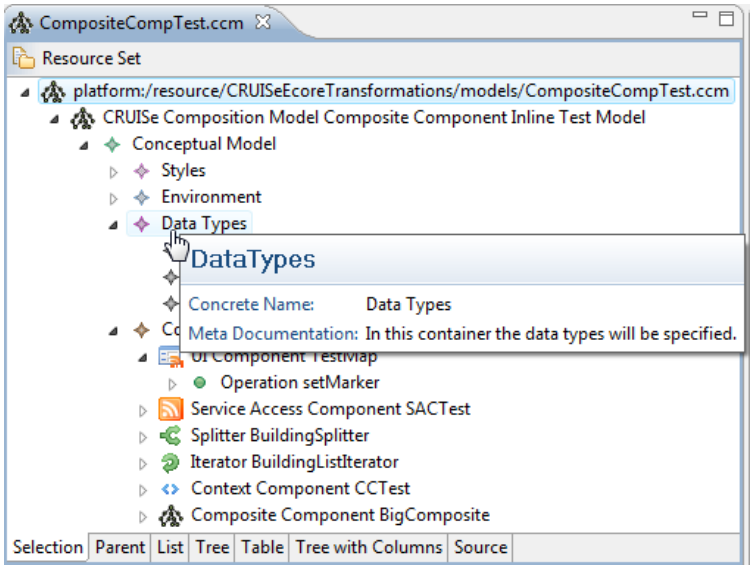


Abbildung 5.4: Beispielmodell im generierten Baum-Editor

Dabei dienen die ersten sechs Tabs im Editor unterschiedlichen Darstellungen des konkreten Modells. Der Tab *Source* allerdings wird nicht generiert. Dieser wurde nachträglich hinzugefügt und ermöglicht die Anzeige der XMI-Repräsentation des Modells. In Listing 5.1 ist die XMI-Serialisierung eines Beispielmodells dargestellt.

Listing 5.1: Serialisierung eines Beispielmodells

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <ccm:CRUISeCompositionModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:ccm="http://
       CRUISeMetaModel/1.0">
3    <conceptualModel>
4      <styles>
5        <style xsi:type="ccm:DimensionStyle" decorates="//@conceptualModel/
             @components/@component[name='TestComponent']" width="300" height="200"/>
6      </styles>
7      <components>
8        <component xsi:type="ccm:UIComponent" name="TestComponent">
9          <event name="testEvent">
10           <parameter name="outParam">
11             <typeReference primitiveType="Int"/>
12           </parameter>
13         </event>
14       </component>
15     </components>
16   </conceptualModel>
17 </ccm:CRUISeCompositionModel>
```

In diesem Listing kann man sehen, dass die Komponente *TestComponent* mit einem Parameter *outParam* modelliert wird. Zudem werden deren Abmessungen über einen Style definiert. Mit dem Attribut *decorates* wird die zu dekorierende Komponente referenziert.

Um den Autor bei der Modellierung unter der Eclipse-Plattform zu unterstützen, wurden zusätzlich Tooltips für den Baum-Editor implementiert. Diese zeigen die Dokumentation der Metaklassen des Elements an, auf das der Mauszeiger im ge-

nerierten Editor gerade zeigt. In Abbildung 5.4 wird beispielsweise der Tooltip der Klasse *DataTypes* mit dargestellt.

Eine weitere wichtige Einstellmöglichkeit, die man im *Generator Model* vornehmen kann, ist anzugeben, dass das Metamodell in der Eclipse-Umgebung erweiterbar ist. Damit kann ein Autor, der zusätzliche Konzepte mit dem Kompositionsmodell umsetzen möchte, sowohl ein neues Ecore-Modell, welches auf das hier erstellte verweist, als auch ein neues *Generator Model*, welches das des Kompositionsmodells referenziert, anlegen. Der *EMF Generator* generiert in diesem Fall nur die zusätzliche API und ein neues Edit-Plugin, wodurch die neu modellierten Klassen zur Verfügung gestellt werden. Ein Editor wird nicht neu generiert. Der schon existierende Baum-Editor integriert die Erweiterungen automatisch, und die neuen Modellelemente können fortan dort erstellt werden. Durch dieses Vorgehen brauchen Autoren, die dieses Metamodell um eigene Konzepte erweitern möchten, das hier zur Verfügung gestellte Kompositionsmodell nicht mehr physisch zu verändern. Diese Tatsache stellt erneut die Leistungsfähigkeit von EMF unter Beweis.

Damit ist die Implementierung des Kompositionsmodells und seines Baum-Editors abgeschlossen. Im nun folgenden Abschnitt wird die Umsetzung der in Kapitel 4.3 erläuterten Transformationen vorgestellt.

5.3 Implementierung der Transformationen

Die Transformationen werden im Eclipse-Plugin *cruise.compositionmodel.transformations* gekapselt (siehe Abbildung 5.2). Diese sind als `qvto`-Dateien hinterlegt und werden in diesem Plugin programmatisch über ein neu definiertes *command* ausgeführt. Mit dem *command* wird durch den Erweiterungsmechanismus von Eclipse sichergestellt, dass Autoren nachträglich eigene Transformationen an der richtigen Stelle einbinden können. Dazu muss nur der Identifikator des *commands* referenziert und mit dem darin definierten Parameter die auszuführende `qvto`-Datei angegeben werden. Mit dem implementierten *DefaultHandler* des *commands* wird dann die jeweilige Transformation mit der QVTO-Engine ausgeführt. In den folgenden drei Abschnitten werden die Implementierungen aller im Kapitel 4.3 definierten Transformationen vorgestellt.

5.3.1 Conceptual Model verfeinern

Im Listing 5.2 ist die Grundstruktur einer QVTO-Transformation abgebildet und stellt gleichzeitig den Einstiegspunkt der in Abschnitt 4.3.1 beschriebenen Generierung dar: die Verfeinerung des *Conceptual Models*.

Listing 5.2: Grundgerüst der Verfeinerung des *Conceptual Models*

```
1 modeltype CMM uses CRUISeMetaModel('http://CRUISeMetaModel/1.0');
2
3 transformation conceptualModelRefinement(inout base:CMM) access
      ConceptsGenerationBlackBox, SACGenerationBlackBox;
4
5 main() {
6   base.rootObjects()[CMM::CRUISeCompositionModel]-> map generateConcepts();
7 }
8
9 mapping inout CMM::CRUISeCompositionModel::generateConcepts(){
```

```
10    self.allSubobjects()[CMM::ConceptualModel].allSubobjects()[CMM::Splitter]->
          generateSplitterEvents();
11    self.allSubobjects()[CMM::ConceptualModel].allSubobjects()[CMM::Iterator]->
          generateIteratorEvent();
12    self.allSubobjects()[CMM::ConceptualModel].allSubobjects()[CMM::Join]-> map
          generateJoin();
13    self.allSubobjects()[CMM::ConceptualModel].allSubobjects()[CMM::
          ServiceAccessComponent]-> map generateSAC();
14  }
```

Als Erstes wird in QVTO immer eine Kurzform für die zu verwendenden Metamodelle angegeben. In diesem Fall ist dies die Bezeichnung `CMM` für das zuvor implementierte Kompositionsmodell. Als nächstes wird wie in Zeile 3 der Name der Transformation festgelegt (`conceptualModelRefinement`). Mit dem Schlüsselwort `inout` wird definiert, dass sowohl das Quellmodell als auch das Zielmodell dem *CRUISe MetaModel* entsprechen. Die nach `access` angegebenen Bezeichner sind in Java implementierte Black-Boxes, auf die gleich näher eingegangen wird. Die `main()`-Methode ist der Einstiegspunkt jeder Transformation. Die Variable `base` nimmt hier Bezug auf das in Zeile 3 angegebene Quellmodell. Methodenzugriffe auf das Metamodell werden mit der Punktnotation angegeben. Ist der Rückgabewert davon eine Liste, so können die enthaltenen Elemente mit eckigen Klammern gefiltert werden. Der Operator `->` ist der Listenoperator und iteriert implizit über alle darin enthaltenen Elemente. Mit `map` werden schließlich die konkreten Transformationen aufgerufen – in diesem Fall *generateConcepts()*. Transformationen beginnen mit dem Schlüsselwort `mapping`, gefolgt von der Angabe der Metaklasse des eingehenden Modellelements. Da auch hier wieder nur auf dem *CRUISe MetaModel* gearbeitet wird, wurde `inout` als Richtung angegeben. Als einfachstes der vier hier aufgerufenen Mappings erweist sich die Generierung der nötigen Eigenschaften von *Join*-Komponenten (vgl. Abschnitt 4.2.2.2), die für die Vereinigung von anliegenden Daten benutzt werden. Diese Transformation wird mit dem folgenden Listing 5.3 umgesetzt.

Listing 5.3: Generierung von *Join*-Komponenten

```
1  mapping inout CMM::Join::generateJoin()
2  {
3    var joinEvent : CMM::Event := object CMM::Event{};
4    joinEvent.name := "onJoin";
5    self.operation->forEach(op){
6      var eventTypeRef : CMM::TypeReference := object CMM::TypeReference{};
7      eventTypeRef.primitiveType:=op.parameter->first().typeReference.primitiveType;
8      eventTypeRef.complexType := op.parameter->first().typeReference.complexType;
9      var eventParam : CMM::Parameter := object CMM::Parameter{};
10     eventParam.name := op.parameter->first().name;
11     eventParam.typeReference := eventTypeRef;
12     joinEvent.parameter += eventParam;
13   };
14   self.event += joinEvent;
15 }
```

In diesem Mapping wird ein *Event* namens *onJoin* erzeugt. Mit dem nachfolgenden Code in der `forEach`-Schleife wird für den Parameter jeder eingehenden *Operation* der korrespondierende Parameter desselben Typs für *onJoin* generiert. Der Name wird dabei übernommen und in der letzten Zeile des Mappings wird der neu erzeugte *Event* den *Event*s des *Joins* hinzugefügt. Am Listenoperator `->` in Listing 5.2 ist zu erkennen, dass für jeden *Splitter* iterativ die Methode *generateSplitterEvents()* aufgerufen wird. Diese Methode wurde in einer Black-Box-Library realisiert und ist im Listing 5.4 dargestellt.

Listing 5.4: Black-Box-Library-Methode für *Splitter*

```
@org.eclipse.m2m.qvt.oml.blackbox.java.Operation(contextual=true,kind=Kind.
    OPERATION)
public void generateSplitterEvents(Splitter splitter){
  generateInternal(splitter);
}
```

In diesem Listing ist zu sehen, wie Methoden einer Black-Box-Library grundsätzlich in QVTO eingebunden werden. Durch die vorangestellte Annotation wird der QVTO-Engine bekanntgegeben, dass die folgende Methode in Mappings aufgerufen und ausgeführt werden kann. Mit `contextual=true` wird spezifiziert, dass der erste Parameter der Methode dem ausführenden Kontext entspricht. Das heißt, dass diese Methode in einer `qvto`-Datei im Kontext eines *Splitters* in Punktnotation aufgerufen werden darf. Mit `kind=Kind.OPERATION` wird definiert, dass diese Methode die Natur eines normalen Mappings besitzt. Die Methode *generateInternal(splitter)* führt schließlich die Auftrennung der Parameter-Typen der eingehenden *Operation* aus und erzeugt ausgehende *Event*s der jeweiligen Komponente. Dazu dient das in Abbildung 5.2 dargestellte Plugin *cruise.xpointer*, welches eine einfache XPointer-Implementierung enthält. Dieses Plugin benutzt wiederum *cruise.xom*, welches Funktionalität bereitstellt, um auf der Baumstruktur von XML-Elementen im Speicher zu arbeiten.

Die komplexeste Black-Box-Methode bei der Verfeinerung des *Conceptual Models* ist die der Generierung der *Event*s und *Operation*s der SACs. Darin werden prototypisch, aufbauend auf der Bibliothek WSDL4J[2], lokale oder entfernte WSDL-Dateien analysiert sowie die dort definierten angebotenen Services in Form von *Event*s und *Operation*s erzeugt. Diese Funktionalität wurde im Plugin *cruise.wsdl4jutil* umgesetzt. Aufgrund der Komplexität wird an dieser Stelle auf die Darstellung des Quellcodes verzichtet.

5.3.2 Layout Model zu Screenflow Model

Die in Kapitel 4.3.2 beschriebene Transformation zur Generierung des Grundgerüsts des *Screenflow Models* wird in einer neuen `qvto`-Datei spezifiziert. Darin wird für jedes *Layout* eine *View*, die eine neue *ViewTransition* besitzt, erzeugt. Existiert jedoch nur ein *Layout*, wird eine *View* ohne Transition generiert. Diese Transformation ist im Listing 5.5 angegeben.

Listing 5.5: *Screenflow Model*-Generierung

```
main() {
  base.rootObjects()[CMM::CRUISeCompositionModel]->map layout2screenflowModel();
}
mapping inout CMM::CRUISeCompositionModel::layout2screenflowModel(){
  self.allSubobjects()[CMM::LayoutModel]-> map addScreenflowModel(self);
}
mapping CMM::LayoutModel::addScreenflowModel(inout rootModel:CMM::
    CRUISeCompositionModel):CMM::ScreenflowModel{
  if(self.layout->size() = 1) then{
    self.allSubobjects()[CMM::Layout]-> map addView(result, true);
  }else{
    self.allSubobjects()[CMM::Layout]-> map addView(result, false);
  }endif;
  rootModel.screenflowModel := result;
```

[2] als Eclipse-Plugin verfügbar und basiert auf der Implementierung unter https://sourceforge.net/projects/wsdl4j/

```
}
mapping CMM::Layout::addView(inout screenModel : CMM::ScreenflowModel, in single:
    Boolean):CMM::View{
  result.name := self.name + "View";
  result.layout := self;
  if(not single) then{
    result.transition += object ViewTransition{};
  }endif;
  screenModel.view += result;
}
```

In den Mappings *addScreenflowModel* und *addView* ist zu sehen, dass Transformationen auch Parameter und einen Rückgabetyp besitzen können. Wenn ein Rückgabetyp angegeben wurde (beispielsweise *CMM::View* in Mapping *addView*), ist dafür implizit immer die Variable `result` verfügbar. Damit wird in der letzten Zeile des Listings dem *Layout* eine neue *View* hinzugefügt.

5.3.3 Conceptual Model zu Communication Model

Für die Implementierung der in Abschnitt 4.3.3 erläuterten Transformation, des *Conceptual Models* zum *Communication Model*, wird wieder eine Black-Box-Library benutzt. Die dementsprechende Methode ist im Listing 5.6 dargestellt.

Listing 5.6: Generierung des *Communication Models*

```
@org.eclipse.m2m.qvt.oml.blackbox.java.Operation(contextual = true,kind = Kind.
    OPERATION)
public CommunicationModel generateCommunicationModel(ConceptualModel model){
  CRUISeMetaModelFactory FACTORY = CRUISeMetaModelFactory.eINSTANCE;
  EList<IComponent> iComponents = model.getComponents().getComponent();
  EList<EObject> components = new BasicEList<EObject>(iComponents);
  if(model.getEnvironment() != null){
    components.add((EObject) model.getEnvironment());
  }
  generatePublishers(components);
  generateSubscribers(components);
  CommunicationModel commModel = FACTORY.createCommunicationModel();
  PublishSubscribe pubsub = FACTORY.createPublishSubscribe();
  commModel.setCommunication(pubsub);
  for (DataChannel channel : dataChannels.values()) {
    channel.setName(generateChannelName(channel.getParameter()));
    secondStepParameterNaming(channel);
    if(!channel.getPublisher().isEmpty() && !channel.getSubscriber().isEmpty()){
      pubsub.getChannel().add(channel);
    }
  }
  return commModel;
}
```

In diesem Listing ist zu sehen, wie neue Objekte des Kompositionsmodells erzeugt werden. Für mit EMF erstellte Metamodelle wird bei der Generierung des APIs auch immer eine Factory (hier *CRUISeMetaModelFactory*) angelegt. Diese besitzt für alle Klassen eine *create*-Methode, wodurch die korrekte Erzeugung sichergestellt wird. Mit der oben dargestellten Black-Box-Operation werden die jeweiligen *DataChannels* erzeugt und nur, wenn diese sowohl *Publisher* als auch *Subscriber* besitzen, dem *Communication Model* hinzugefügt.

Damit ist die Implementierung des Kompositionsmodells mit EMF abgeschlossen. Die vorgestellten Transformationen wurden mit QVTO umgesetzt. Abschließend wird im folgenden Kapitel eine Beispielanwendung modelliert und konkret gezeigt, wie generierte Modellelemente in das Modell eingefügt werden.

5.4 Modellierung einer Beispielanwendung

Für die Validierung der prototypischen Implementierung werden zunächst das Modell einer Beispielanwendung erstellt und anschließend die vorgestellten Transformationen ausgeführt, um die Korrektheit der generierten Elemente zu prüfen. Als Beispielanwendung wird ein UI-Mashup für eine Immobilienfirma gewählt. Die Benutzer dieser Web-Applikation werden die Eigentümer der Immobilien sein. Diese können sich in einer ersten Sicht all ihre Gebäude in einem Baum-Navigator anzeigen lassen und bei Selektion wird die jeweilige Immobilie in einer Karte lokalisiert, sowie Fotos und Kosten werden angezeigt. In Abbildung A.1 ist ein Screenshot der ersten Sicht dargestellt. Wird der Knopf »suche Dienstleister« betätigt, wird die Ansicht wie in Abbildung A.2 dargestellt. Dort werden Dienstleister, die sich in der Nähe des vorher gewählten Gebäudes befinden, aufgelistet. Selektiert man einen Dienstleister in der Tabelle, erscheinen Details wie zum Beispiel Kontaktdaten. In Abbildung A.3 sind die Schichten dieses UI-Mashups mit den nötigen Komponenten dargestellt, die nachfolgend zu modellieren sind.

Als erstes muss mit dem CRUISe-Composition-Model-Wizard eine neue Modell-Datei angelegt werden. Dadurch öffnet sich der generierte Editor, und neue Elemente können stets mit Rechtsklick angelegt werden. Außerdem steht die *Properties View* von Eclipse zur Verfügung, mit der die Attribute des jeweils selektierten Elements festgelegt werden können. Entsprechend der Abbildung 4.16 wird nun zuerst das *Conceptual Model* erzeugt. Begonnen wird mit der Spezifizierung der benötigten *DataTypes*. In der Beispielanwendung sind dies `Address`, `Building`, `URL` und `ServiceProvider`, sowie für die drei letztgenannten zusätzlich jeweils ein Listentyp. Für die SACs *InternalBuildingsProvider* und *YellowPagesProvider* werden manuell keine *Event*s und *Operation*s modelliert, sondern nur beim Attribut *descriptionLocation* Verweise auf WSDL-Dateien gesetzt. Diese wurden vorher auch im Workspace angelegt und repräsentieren WSDL-Beschreibungen allgemein zugänglicher SOAP-Services. In Abbildung 5.5 ist die WSDL-Datei des *InternalBuildingsServices* schematisch dargestellt, um überprüfen zu können, ob die entsprechenden *Event*s und *Operation*s später dementsprechend generiert werden.

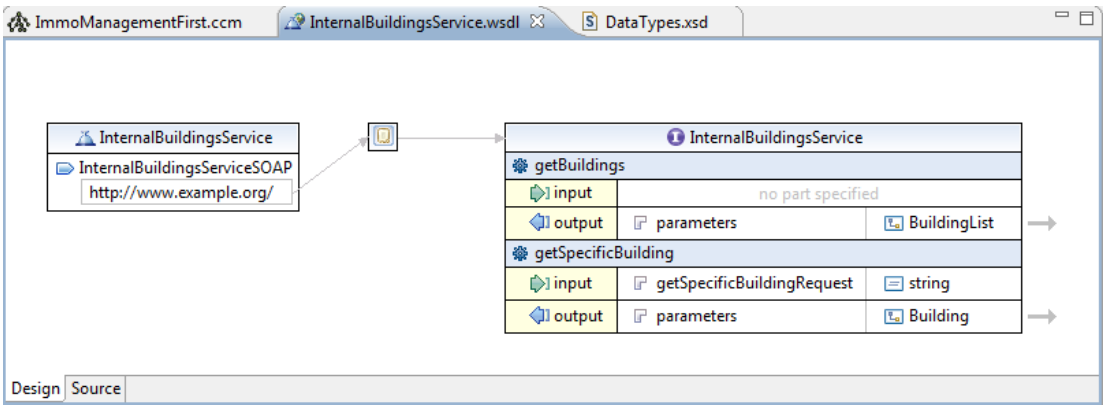


Abbildung 5.5: WSDL-Beispiel vom *InternalBuildingsService*

In dieser Abbildung ist zu sehen, dass die angebotene Service-Operation *getBuildings* keinen Eingangsparameter besitzt und eine `BuildingList` zurückgibt. Die Operation *getSpecificBuilding* erwartet den Identifikator eines Gebäudes als `String`

und gibt dieses dann als Typ `Building` zurück. Nun wird entsprechend der Abbildung A.3 die *Environment* modelliert. In Abbildung 5.6 ist dargestellt, wie das Modell der Immobilienverwaltung bisher im Editor abgebildet wird.

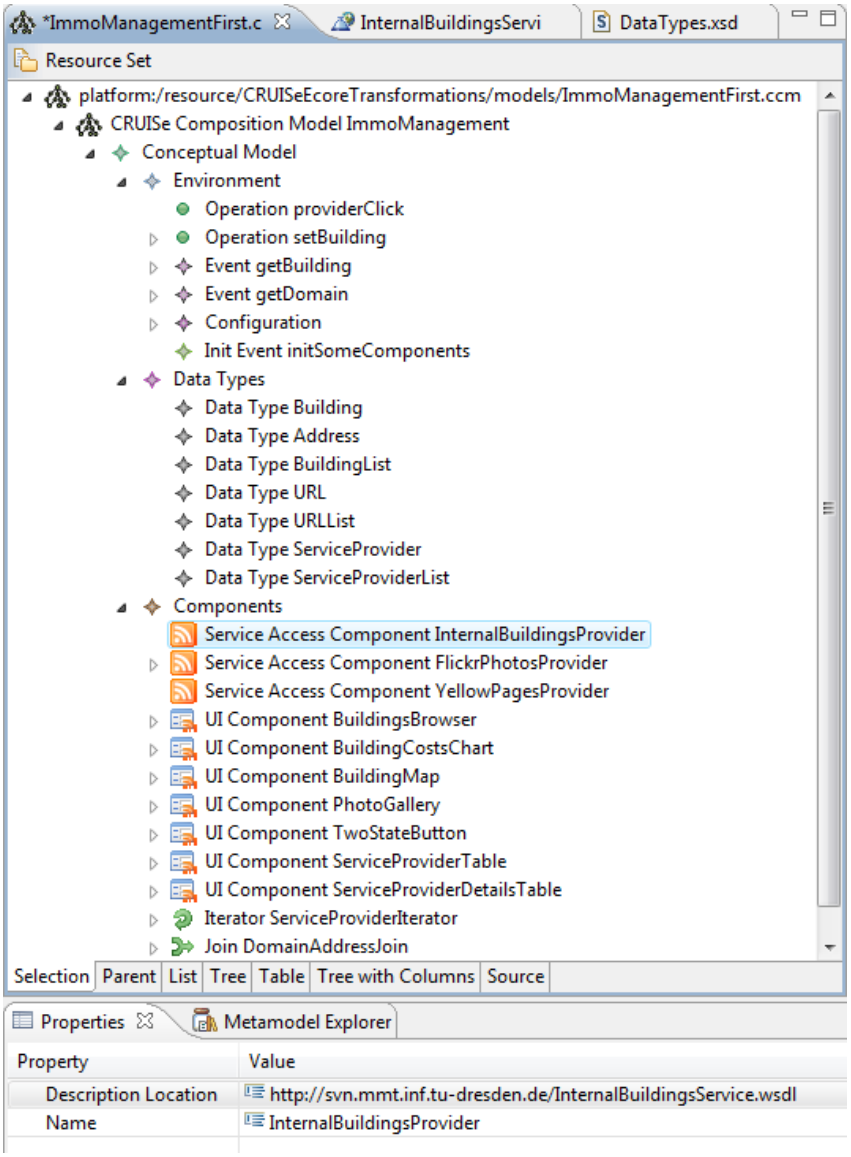


Abbildung 5.6: *Conceptual Model* der Immobilienverwaltung

Im unteren Teil der Abbildung ist die *Properties View* zu sehen, mit der Angabe der WSDL-Beschreibung des *InternalBuildingsProviders*. Nun wird die erste Transformation ausgeführt, um das *Conceptual Model* zu verfeinern. Öffnet man im Editor das Kontextmenü, erreicht man über »CRUISe Transformations« den Befehl »Generate concepts in ConceptualModel«. Damit wird die Generierung der *Event*s und *Operation*s der SACs sowie von *Iterator*, *Splitter* und *Join* gestartet. In Abbildung 5.7 ist dargestellt, dass die SAC *InternalBuildingsProvider* nach der Generierung *Event*s und *Operation*s besitzt. Außerdem ist in der *Properties View* zu sehen, dass der Rückgabewert von *onGetBuildings* den korrekten Typ `BuildingList` innehat.

Als nächstes wird das *Layout Model* erstellt, womit für die Beispielanwendung zwei *GridLayouts* definiert werden; für jede Sicht ein *Layout*.

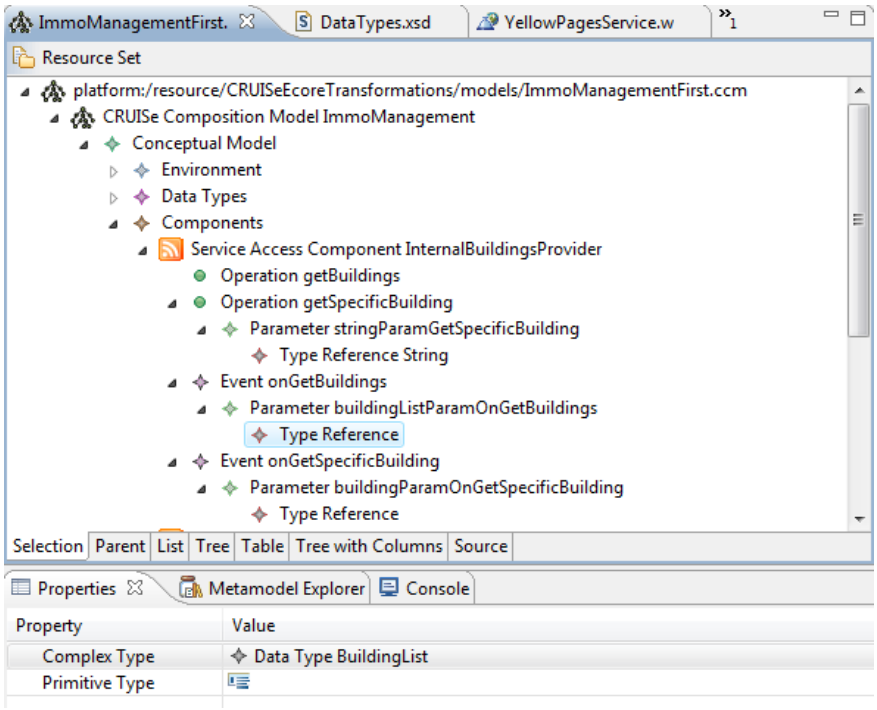


Abbildung 5.7: Verfeinerung des *Conceptual Models*

Danach wird das Grundgerüst des *Screenflow Models* generiert, indem man im schon erwähnten Kontextmenü den Befehl »Create ScreenflowModel from Layout-Model« ausführt. Dadurch entstehen zwei *Views*, die das entsprechende *Layout* referenzieren. Als initiale *View* wird *BuildingsLayoutView* festgelegt. Komplettiert wird dieses Modell, indem die *Events* des *TwoStateButtons* in den Transitionen referenziert werden und an den *Views* die jeweils andere als Ziel deklariert wird. Sind im *Conceptual Model CompositeComponents* enthalten, muss an dieser Stelle der Befehl »Generate all concepts of CompositeComponents« gestartet werden. Damit werden auch alle Transformationen ausgeführt, für den Fall, dass die *CompositeComponent* direkt im übergeordneten Modell spezifiziert wurde. Integrieren *CompositeComponents* andere UI-Mashups über ihr Attribut *referencedCompositionModel* und haben demnach keine Kindelemente, so hat dieser Befehl keine Auswirkungen.

Zuletzt muss das Grundgerüst des *Communication Models* generiert werden. Dazu wird im Kontextmenü »Create CommunicationModel« betätigt. Das Resultat dieser Transformation ist in Abbildung 5.8 dargestellt. Als Beispiel der korrekt generierten Typen der Kanäle ist in der *Properties View* der Kanal *StringAddressChannel* geöffnet. Dieser enthält einen Parameter vom Typ `String` und einen zweiten vom Typ `Address`. Als *Publisher* wurde, entsprechend der Abbildung A.3, korrekt das *Event onJoin* der Komponente *DomainAddressJoin* und als *Subscriber* die *Operation collectProvidersByAddressDomain* der Komponente *YellowPagesProvider* referenziert. Für die generierten *DataChannels* muss nun vom Autor überprüft werden, ob sie der ihnen zugeordneten Semantik entsprechen. Des Weiteren müssen gegebenenfalls *ParameterMappings* definiert und die Sichten, in denen die *Publisher* und *Subscriber* aktiv sein sollen, referenziert werden. Damit ist die Modellierung der Beispielanwendung abgeschlossen. Da die in diesem Kapitel modellierte Beispielanwendung sehr komplex ist, wird im Listing A.1 die XMI-Serialisierung eines einfachen UI-Mash-

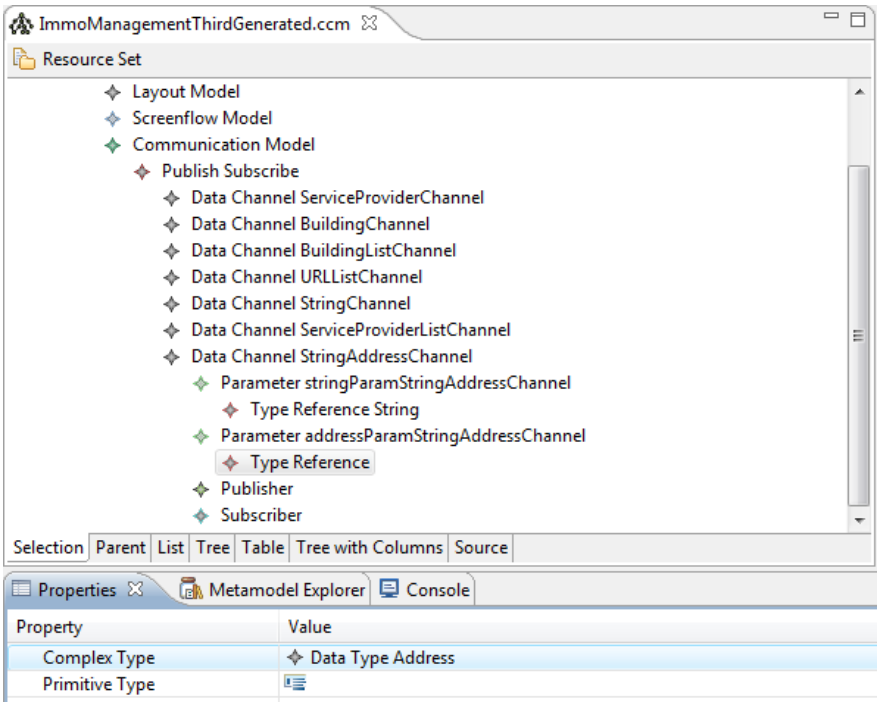


Abbildung 5.8: Generierung des *Communication Models*

ups mit nur zwei Komponenten dargestellt, um einen Eindruck vom Quelltext zu bekommen und dass er gut lesbar ist.

5.5 Fazit

In diesem Kapitel wurde die Umsetzung des entworfenen Kompositionsmodells mit dem *Eclipse Modeling Framework (EMF)* vorgestellt. Die implementierten Transformationen mit QVTO setzten direkt darauf auf. Aufgrund dieser Kooperation traten keine Kompatibilitätsprobleme auf. Nachfolgend werden sowohl der erreichte Stand als auch aufgetretene Probleme und noch nicht umgesetzte Aspekte zusammengefasst.

Die Wahl der Plattform Eclipse mit EMF für die Modellierung und QVTO für die Transformationen erwies sich als sehr gut. Nicht zuletzt ist die Tatsache von Vorteil, dass man im Eclipse-Umfeld auf eine große Community trifft, welche schnell und unkompliziert Hilfe bietet. Mit EMF konnten das Kompositionsmodell wie erwünscht umgesetzt und daraus API und Editor generiert werden. Diese fügten sich nahtlos in Eclipse als Plugins ein. Musste das Metamodell im Laufe der Implementierung angepasst werden, so konnten die Änderungen trotzdem leicht propagiert werden. Der generierte Editor konnte um zusätzliche Features erweitert werden, wodurch ein Werkzeug zur Verfügung steht, mit dem konkrete Modelle erstellt werden können. Als nachteilig erwies sich allerdings dessen Baumstruktur. Bei komplexen UI-Mashups sind die vielen nötigen Interaktionsschritte bis zur Fertigstellung des Modells sehr aufwändig. Zudem werden, wenn Referenzen zu schon existierenden Modellelementen gesetzt werden müssen, nur deren Namen angezeigt, wodurch die Übersicht leidet. Möchte man beispielsweise an einem *DataChannel* mehrere *Event*s als *Publisher* setzen, und wurden diese in unterschiedlichen Komponenten aber gleich

benannt, können sie im Editor beziehungsweise in der *Properties View* nicht unterschieden werden. Im Ausblick in Abschnitt 6.2 wird dazu eine mögliche Lösung vorgeschlagen. Zur Erstellung erster Beispielmodelle und auch zur Überprüfung der korrekten Modellierung der gewünschten Konzepte im Metamodell war der generierte Editor jedoch ein sehr hilfreiches Werkzeug.

Durch Dokumentation und Community war eine Einarbeitung in QVTO schnell möglich. Aufgrund der hier gewählten programmatischen Aktivierung der Transformationen erwies sich anfangs jedoch die Ausgabe von Logging-Anweisungen als problematisch. Nach dem Studium der QVT-Newsgroup[3] von Eclipse konnte allerdings herausgefunden werden, dass für die hier gewählte Technik die Ausgabe des Logs umgeleitet werden muss. Mit dem Mechanismus der Black-Box-Libraries konnten Transformationen auf Basis des generierten APIs in Java implementiert werden. Allerdings wurde die Generierung der *Event*s und *Operation*s prototypisch nur für SOAP-Services durchgeführt. Demzufolge können SACs derzeit nur anhand von WSDL-Dateien verfeinert werden. Eine weitere Implementierung für HTTP-basierte Services (mit WADL-Beschreibung) muss außerhalb dieser Arbeit analog vorgenommen werden. Außerdem wurde mit *cruise.xpointer* ein Plugin implementiert, welches einfache, in diesem Kontext ausreichende, XPointer-Funktionalitäten zur Verfügung stellt. Derzeit sind dabei nur Verweise auf lokale XML-Schema-Dateien möglich, die relativ zum jeweiligen Modell adressiert werden können. Diese Implementierung kann erweitert werden, wenn die Fähigkeit des Umgangs mit entfernten Dateien (beispielsweise in einem Repositorium, wie es bei Verweisen auf WSDL-Beschreibungen schon möglich ist) gewünscht ist. Im aktuellen Stadium der Umsetzungen der Transformationen können jedoch schon sehr flexibel Generierungen durchgeführt werden.

Durch die in dieser Arbeit realisierte Implementierung konnte die Durchführbarkeit des Konzeptes nachgewiesen werden. Mit der Nutzung von EMF werden MOF-konforme Modelle erzeugt, die im XMI-Format serialisiert werden. Verweise mit XPointer-Ausdrücken auf Elemente in XSD-Dateien sowie die Verarbeitung von WSDL-Beschreibungen sind möglich. Außerdem wurde nicht zuletzt mit der Fokussierung auf QVTO in der hier vorgestellten Implementierung darauf geachtet, dass Standards unterstützt werden, wodurch Anforderung 19 zusätzlich erfüllt wird. Zusammenfassend ist zu sagen, dass mit der prototypischen Umsetzung ein Kompositionsmodell entworfen wurde, mit dem UI-Mashups vollständig spezifiziert werden können. Mit den implementierten Transformationen konnte gezeigt werden, dass Teile automatisch generierbar sind.

Mit den Kapiteln 4 und 5 konnten alle funktionalen und die meisten nichtfunktionalen Anforderungen erfüllt werden. Die Werkzeugunterstützung konnte nur für Eclipse als positiv bewertet werden, jedoch wurde mit XMI und der MOF-Konformität allgemein der Grundstein dafür gelegt. Somit sind mit diesem Abschnitt das Konzept und die Implementierung abgeschlossen. Im nachfolgenden Kapitel 6 werden die in dieser Arbeit erreichten Ziele noch einmal zusammengefasst und ein Ausblick gegeben.

[3] siehe `news://eclipse.modeling.m2m`

6 Zusammenfassung und Ausblick

Im letzten Kapitel dieser Arbeit sollen die erreichten Ergebnisse reflektiert und diese abschließend bewertet werden. Zudem wird ein Ausblick und Hinweise auf zukünftige Arbeiten im CRUISe-Projekt gegeben.

6.1 Ergebnisse

Ziel dieser Arbeit war es, ein generisches Kompositionsmodell zu entwickeln, mit dem die modellgetriebene Erstellung von UI-Mashups ermöglicht wird. Das Kompositionsmodell sollte hierbei in die serviceorientierte Architektur des CRUISe-Projekts eingegliedert werden. Dazu musste zunächst in Kapitel 2 definiert werden, was unter einem UI-Mashup zu verstehen ist. Darauf aufbauend, konnte dargestellt werden, was unter einem Kompositionsmodell für UI-Mashups verstanden werden kann und wo sich dieses Metamodell in die Modell-Architektur MOF eingliedert. Des Weiteren wurde das Forschungsprojekt CRUISe vorgestellt und aufgezeigt, welchen Platz das Kompositionsmodell in dessen Architektur einnimmt. Am Ende des zweiten Kapitels konnten die Anforderungen an das zu entwickelnde Kompositionsmodell aufgestellt werden.

In Kapitel 3 erfolgte die Analyse existierender Ansätze zur Modellierung von Web-Applikationen. Das mashArt-Projekt zeigte dabei Stärken in seinem komponentenbasierten Ansatz, weshalb dort verwendete Konzepte der Komponenten-Kommunikation aufgegriffen werden konnten. Beispielsweise konnte mit der Modellierung von *ParameterMappings* die Anzahl der benötigten Komponenten in der Publish/Subscribe-Kommunikation von CRUISe reduziert werden. Eine sehr leistungsfähige Variante der Modellierung von Adaptivität bringt der objektorientierte Ansatz von UWE mit. Dabei wird auf das noch junge Gebiet der aspektorientierten Modellierung gesetzt, mit dem Ziel, Adaptivität lose zu modellieren und später in die betroffenen Modelle zu weben. An diesen Ansatz wird in der vorliegenden Arbeit angeknüpft, und er wird um hier notwendige Konzepte erweitert. Dazu zählt beispielsweise die unabhängige Modellierung von Adaptivität bezüglich bestimmter Datenbasen. Außerdem war die technologieunabhängige Modellierung des Layouts der Web-Applikation von WebML als positiv zu werten, weshalb dieser Ansatz aufgegriffen und erweitert wurde. In diesem Kapitel konnten viele Erkenntnisse und Anregungen gewonnen werden, welche zum Teil der Konzeption als Grundlage dienten.

In Kapitel 4 wird die Konzeption des generischen Kompositionsmodells vorgestellt. Dafür wurde zunächst das Komponentenmodell, wie es in CRUISe Verwendung findet, erläutert, um darzustellen, welche Eigenschaften zu modellierende Komponenten im Kompositionsmodell mindestens aufweisen müssen. Des Weiteren wird gezeigt, welche Komponentenarten in CRUISe existieren und welche konzeptionelle Schichten dadurch in UI-Mashups entstehen. Das Kompositionsmodell zeichnet sich durch die Trennung von Verantwortlichkeiten auf Modellebene aus. Es wurden

demnach fünf Charakteristika von UI-Mashups herausgestellt, die in unterschiedliche Teilmodelle ausgelagert werden konnten: *Conceptual Model* (beinhaltet Komponenten, Typen und *Styles*), *Layout Model* (spezifiziert die Anordnung der Komponenten), *Screenflow Model* (definiert den Kontrollfluss), *Communication Model* (modelliert Kommunikationskanäle) und *Adaptivity Model* (dient der aspektorientierten Adaptivitätsmodellierung). Dabei sind die ersten vier Modelle obligatorisch, und mit dem optionalen *Adaptivity Model* wurde die Erweiterbarkeit des Metamodells veranschaulicht. Das *Conceptual Model* setzt das Komponentenmodell von CRUISe um und ermöglicht die Modellierung von Datentypen, um *Event*s, *Operation*s und Datenkanäle zu parametrisieren. Ganz im Sinne des Single-Source-Prinzips kann hier mit XPointer-Ausdrücken auf schon vorhandene XML-Schemata verwiesen werden, wodurch Typen nicht neu spezifiziert werden müssen. Außerdem wird mit den schachtelbaren *Styles*, die auf UI-Komponenten angewendet werden, die Modellierung von einem einheitlichen *Look & Feel* ermöglicht. Das *Layout Model* spezifiziert die Anordnung der UI-Komponenten. Drei Layout-Arten erlauben es, Komponenten zu positionieren oder aber Unter-Layouts zu definieren. Das *Screenflow Model* zeichnet sich durch die Event-basierte Modellierung des Kontrollflusses der Web-Applikation aus, wodurch Übergänge zwischen Sichten nicht nur durch Nutzerinteraktion initiiert werden können, sondern durch Komponenten-Events im Allgemeinen. Um den Datenfluss und die Kommunikation der Komponenten zu spezifizieren, wurde das *Communication Model* entworfen. Seine Stärken sind es, dass die modellierte Interaktion zwischen den Komponenten auf das Publish/Subscribe-Prinzip von CRUISe übertragen werden kann, in der Broker als Vermittler zwischen den Komponenten fungieren. Sender und Empfänger von Nachrichten sind unabhängig voneinander und können in unterschiedlichen *Views* aktiviert werden, wodurch ein UI-Mashup an Performanz gewinnen kann. Mit dem *Adaptivity Model* wurde ein Ansatz der aspektorientierte Modellierung von Adaptivität vorgestellt. Die Aspektorientierung selbst ist sehr vielversprechend, sollte jedoch noch um die Modellierbarkeit der Bedingungen ausgebaut werden. Hier können derzeit nur XPath-Ausdrücke definiert werden, die angeben, unter welchen Voraussetzungen adaptiert werden soll. Dieses Modell zeichnet sich zudem durch die eventbasierte Adaptivitätsmodellierung aus. Durch Definition von Aspekten können Modellelemente durch ausgelöste *Event*s adaptiert und durch neue ersetzt werden. Hier kann der Autor für jedes im konkreten Modell enthaltene Element Adaption modellieren.

Die Umsetzung des Kompositionsmodells wurde in Kapitel 5 vorgestellt. Mit der Wahl von EMF als Modellierungs-Framework konnten weitere nichtfunktionale Anforderungen erfüllt werden. Das Zusammenspiel zwischen EMF und QVTO verläuft problemlos. Somit konnte die IDE Eclipse um das hier entworfene Kompositionsmodell und den definierten Transformationen bereichert werden. Außerdem wurden alle Vorkehrungen getroffen, um sowohl Metamodell als auch Transformationen erweiterbar zu machen. Eine beispielhafte Immobilienverwaltung konnte erfolgreich modelliert, und die Generierungsschritte konnten durchgeführt werden.

Damit konnten die in Kapitel 1.2 formulierten Ziele umgesetzt und die in Kapitel 2.6 aufgestellten Anforderungen erfüllt werden. Die modellgetriebene Entwicklung von UI-Mashups kann mit dem hier entwickelten Kompositionsmodell durchgeführt werden, und Transformationen erleichtern diese Aufgabe zusätzlich. Somit ergibt sich schließlich ein Rahmen für den Autorenprozess. Im Kompositionsmodell wurde

außerdem stets darauf geachtet, an den geeigneten Stellen Interfaces beziehungsweise abstrakte Klassen zu verwenden, um Erweiterbarkeit zu garantieren. Darüber hinaus lag der Fokus auf der vollkommenen Technologieunabhängigkeit, wodurch für unterschiedliche Plattformen keine semantisch redundanten Modelle entstehen. Dies konnte erfolgreich umgesetzt werden, und die Generierung des finalen UI-Mashups kann, wie in CRUISe vorgesehen, vom *Application Generator* durchgeführt werden. Diesbezüglich wird im folgenden letzten Abschnitt noch ein Ausblick gegeben.

6.2 Ausblick

Nachfolgend werden Hinweise gegeben, die als Ausgangspunkt für weiterführende Arbeiten dienen können. Sie basieren auf identifizierten Problemen bei der Implementierung beziehungsweise auf herausgestellten Möglichkeiten bestimmter Technologien oder Modellierungen.

Mit dem entworfenen *Adaptivity Model* wurde eine leistungsfähige Art der Adaptivitätsmodellierung geschaffen. Jedoch birgt sie noch Potential zur Verbesserung, insbesondere bezogen auf die Modellierung von Bedingungen. Dieses Thema wurde hier soweit bearbeitet, wie es die gewünschte Vollständigkeit erfordert hat. Aufgrund der Komplexität des Aspektes der Adaptivität sollte diese Gegenstand einer zukünftigen Arbeit sein. Außerdem sind aufgrund der Wahl der aspektorientierten Adaption Anforderungen an den *Adaptation Manager* von CRUISe entstanden. Dieser muss die Natur eines Aspekt-Webers besitzen, der zur Laufzeit des UI-Mashups Adaptions-Prozesse durchführen kann. An dieser Stelle können beispielsweise Erkenntnisse aus dem HyperAdapt-Projekt herangezogen werden.

Aufgrund der Verwendung von EMF würde auch der in CRUISe verwendete *Application Generator* zur Erzeugung des finalen UI-Mashups von EMF profitieren. Durch das bereitgestellte API könnten konkrete Modelle mit EMF analysiert und in entsprechenden Ziel-Code überführt werden. Dies wäre sowohl programmatisch als auch über M2C-Transformationen, die auf EMF aufsetzen, möglich. Um diverse Zielplattformen und -umgebungen zu ermöglichen, wäre die Umsetzung geeigneter M2C-Tranformationen am sinnvollsten, da somit die Neu-Kompilierung programmatischer Ziel-Code-Erzeugung (beispielsweise mit Java) wegfällt.

Des Weiteren könnte die rekursive Komposition noch ausgebaut werden. Es wäre denkbar, zu untersuchen, ob für fertiggestellte Modelle von UI-Mashups UISDL-Beschreibungen generiert werden könnten. Dadurch wäre es zum Beispiel möglich, zu exportierende *Event*s und *Operation*s schon dort zu definieren und die Integration als *CompositeComponent* weiter zu vereinfachen. Dazu müsste analysiert werden, inwiefern die UISDL erweitert werden muss und ob das hier entworfene Kompositionsmodell zusätzliche Konzepte benötigt.

Wie schon in Kapitel 5.5 erwähnt, birgt der generierte Editor einige Nachteile. Deshalb sollte in der weiterführenden Arbeit von CRUISe ein grafischer Editor entworfen werden, der den hier generierten und erweiterten Baum-Editor ablöst. Dadurch kann der Entwicklungsaufwand noch weiter reduziert werden. Als Anregung sei an dieser Stelle das *Graphical Editing Framework (GEF)*[1] genannt, welches die Erstellung grafischer Editoren auf Basis eines EMF-Modells unterstützt. Damit

[1] siehe http://www.eclipse.org/gef

können einerseits ein speziell auf das Kompositionsmodell zugeschnittener Editor entwickelt und andererseits die Nachteile auf ein Minimum reduziert werden. Außerdem setzt das Projekt GEF3D[2] darauf auf und erweitert Editoren um eine dritte Dimension. Auf diese Weise könnten Verbindungen zwischen Modellen visualisiert werden.

Im Abschnitt 4.2.2.2 wurden die drei speziellen *Logic Components Join*, *Splitter* und *Iterator* vorgestellt. Hier wäre eine Untersuchung noch weiterer LCs denkbar, um die Modellierung der Verarbeitung von Daten weiter zu vereinfachen oder oft auftretende Funktionalitäten ins Metamodell aufzunehmen, die sonst selbst als Komponente implementiert werden müssten. Beispielsweise könnte ein *Timer* nützlich sein, der nach einem festgelegten Intervall kontinuierlich *Event*s auslöst und Daten kommuniziert.

Ein weiterer Aspekt, der von EMF profitieren kann, ist die Persistenz der Modelle. Mit dem Projekt *Connected Data Objects (CDO)*[3] existiert ein Modell-Repository, das für die Persistierung von EMF-Modellen in eine Datenbank verwendet werden kann. Wenn demnach konkrete Kompositionsmodelle in einer Datenbank abgelegt werden würden, ergäben sich ganz neue Möglichkeiten für die Wiederverwendung von UI-Mashups in CRUISe. Derart könnten alle Techniken zur Integration benutzt werden, die die jeweilige Datenbank mit sich bringt. Insbesondere können somit Stärken von Abfragen ausgespielt werden, um ähnliche Web-Applikationen zu finden. Eine Anfrage an die Datenbank, die MUSS-Kriterien an das wiederzuverwendende UI-Mashup enthält, kann formuliert werden und die passendsten werden zurückgegeben. Somit könnte der Ranking-Mechanismus aus Kapitel 2.3 erweitert werden.

[2]siehe http://gef3d.org
[3]siehe http://wiki.eclipse.org/CDO

A Anhang

A.1 Screenshots

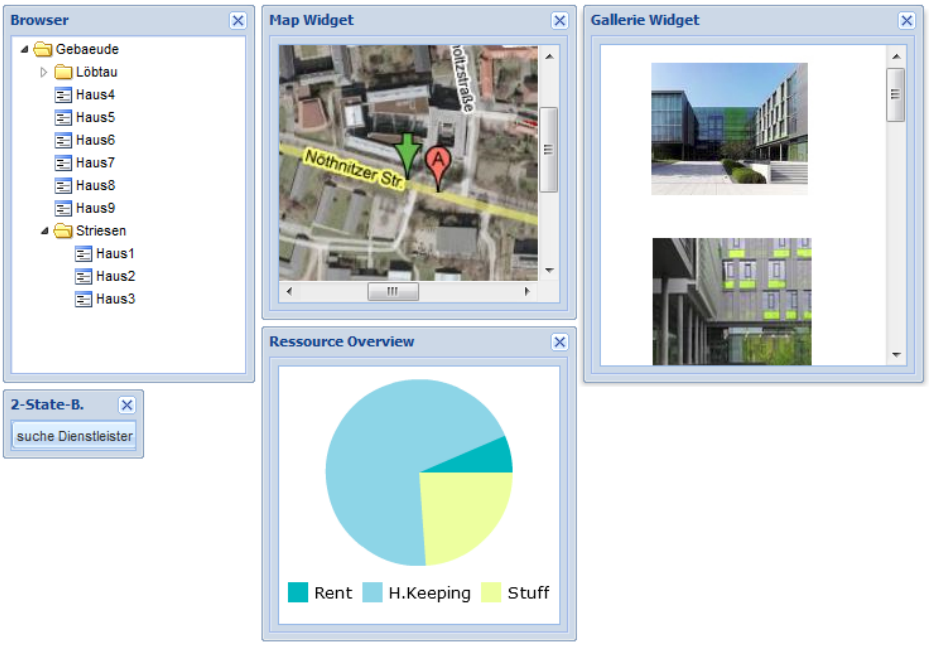


Abbildung A.1: Erste Sicht der Immobilienverwaltung

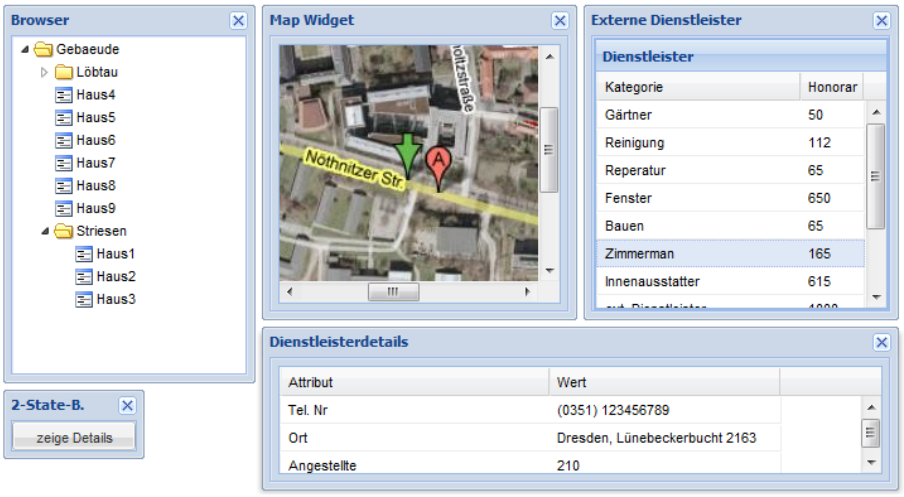


Abbildung A.2: Zweite Sicht der Immobilienverwaltung

A.2 Beispiel-Listing

Listing A.1: XMI-Serialisierung eines einfachen UI-Mashups

```
<?xml version="1.0" encoding="UTF-8"?>
<ccm:CRUISeCompositionModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:ccm="http://
    CRUISeMetaModel/1.0" name="SmallExample">
  <conceptualModel>
    <styles>
      <style xsi:type="ccm:DimensionStyle" decorates="//@conceptualModel/
          @components/@component[name='TestWidgetA'] //@conceptualModel/@components
          /@component[name='TestWidgetB']" width="200" height="200"/>
    </styles>
    <components>
      <component xsi:type="ccm:UIComponent" name="TestWidgetA" url="localhost/
          testWidget" classificationCategory="">
        <event name="onButtonClick">
          <parameter name="outString">
            <typeReference primitiveType="String"/>
          </parameter>
        </event>
      </component>
      <component xsi:type="ccm:UIComponent" name="TestWidgetB"
          classificationCategory="genericTest">
        <operation xsi:type="ccm:Operation" name="setString">
          <parameter name="inString">
            <typeReference primitiveType="String"/>
          </parameter>
        </operation>
      </component>
    </components>
  </conceptualModel>
  <layoutModel>
    <layout xsi:type="ccm:AbsoluteLayout" name="AbsolutTest" unit="percent" width="
        100" height="100" agile="true">
      <position locate="//@conceptualModel/@components/@component[name='TestWidgetA
          ']"/>
      <position locate="//@conceptualModel/@components/@component[name='TestWidgetB
          ']" x="300" y="300"/>
    </layout>
  </layoutModel>
  <screenflowModel initialView="//@screenflowModel/@view[name='AbsolutTestView']">
    <view name="AbsolutTestView" layout="//@layoutModel/@layout[name='AbsolutTest']
        "/>
  </screenflowModel>
  <communicationModel>
    <communication xsi:type="ccm:PublishSubscribe">
      <channel xsi:type="ccm:DataChannel" name="StringChannel">
        <parameter name="stringParamStringChannel">
          <typeReference primitiveType="String"/>
        </parameter>
        <publisher xsi:type="ccm:Publisher" event="//@conceptualModel/@components/
            @component[name='TestWidgetA']/@event[name='onButtonClick']"/>
        <subscriber xsi:type="ccm:Subscriber" operation="//@conceptualModel/
            @components/@component[name='TestWidgetB']/@operation[name='setString']
            "/>
      </channel>
    </communication>
  </communicationModel>
</ccm:CRUISeCompositionModel>
```

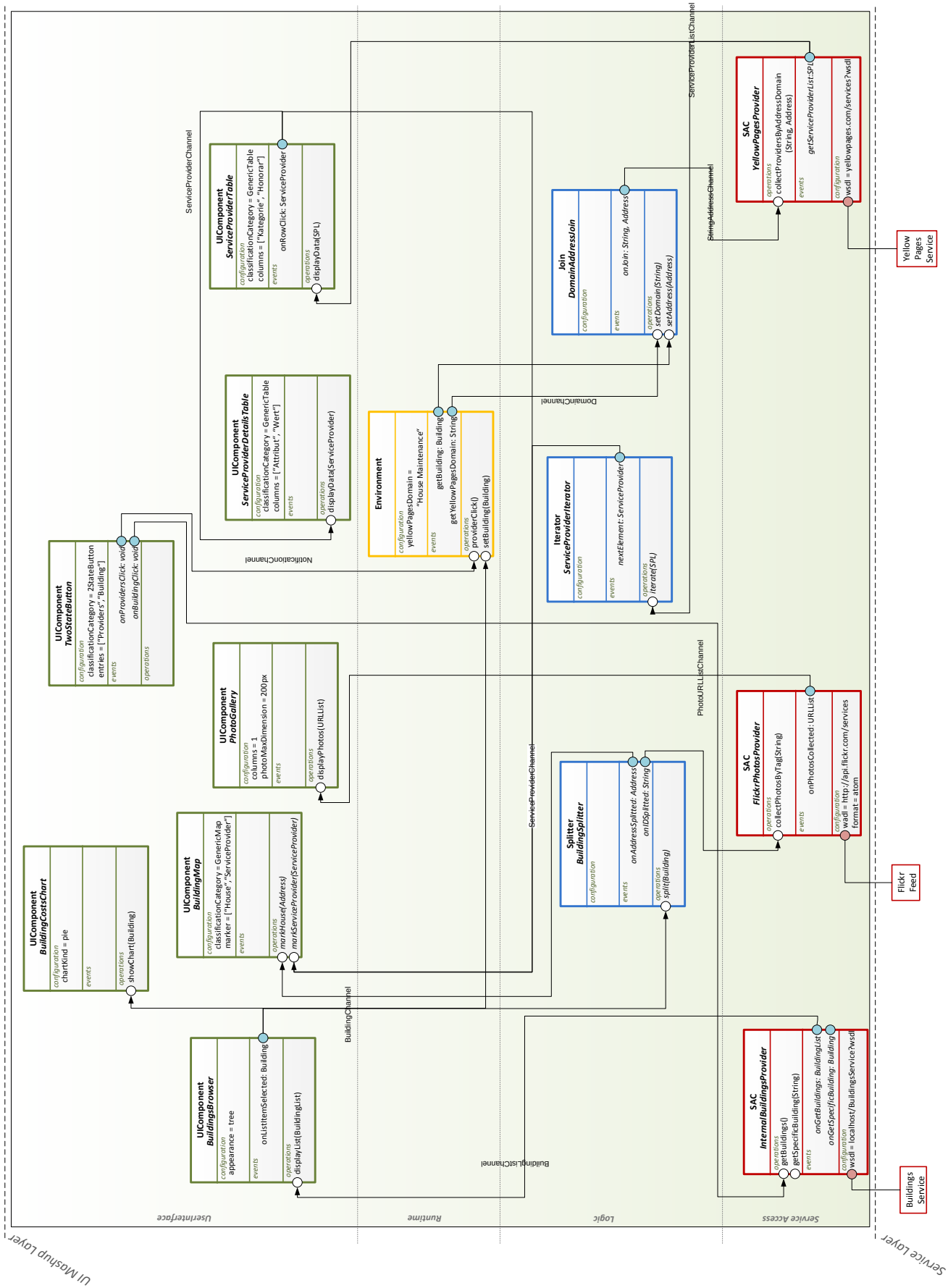


Abbildung A.3: Schichten und Komponenten der Immobilienverwaltung

Literaturverzeichnis

[Aß03] Assmann, Uwe: *Invasive Software Composition*. Springer, 2003.

[AH04] Abrams, Marc und Jim Helms: *Retrospective on UI Description Languages, Based on 7 Years Experience with the User Interface Markup Language (UIML)*. In: *Developing User Interfaces with XML: Advances on User Interface Description Languages, Satellite Workshop of Advanced Visual Interfaces 2004*, Seiten 1–8, Gallipoli, Italien, Mai 2004.

[@ATL04] INRIA: *ATL Language definition*. `http://atlanmod.emn.fr/AMMA/languageDefinition/`, Juli 2004. Besucht am 22. Oktober 2009.

[Bau09] Baumgart, Matthias: *Klassifikation von User Interface Services*. Belegarbeit Technische Universität Dresden, Fakultät Informatik, Institut für Software- und Multimediatechnik, Fachgebiet Multimediatechnik, 2009.

[@Bay02] Bayer, Thomas: *REST Web Services - Eine Einführung*. `http://www.oio.de/public/xml/rest-webservices.htm`, November 2002. Besucht am 04. Oktober 2009.

[BCFM08] Brambilla, Marco, Sara Comai, Piero Fraternali und Maristella Matera: *Designing Web Applications with WebML and WebRatio*, Band *Web Engineering: Modelling and Implementing Web Applications* der Reihe *Human-Computer Interaction*, Kapitel 9, Seiten 221–261. Springer London, 2008.

[BKN07] Brusilovsky, Peter, Alfred Kobsa und Wolfgang Nejdl (Herausgeber): *The Adaptive Web - Methods and Strategies of Web Personalization*, Band 4321/2007. Springer Berlin / Heidelberg, 2007.

[CDMF07] Ceri, Stefano, Florian Daniel, Maristella Matera und Federico Facca: *Model-driven Development of Context-Aware Web Applications*. In: *ACM Transactions on Internet Technology (TOIT)*, Band 7. ACM Press, Februar 2007.

[Con99] Conallen, Jim: *Modeling Web Application Architectures with UML*. Commun. ACM, 42(10):63–70, 1999.

[CWD00] Curbera, Francisco, Sanjiva Weerawarana und Matthew J. Duftler: *On Component Composition Languages*. In: *Proceedings of the Fifth International Workshop on Component-Oriented Programming (WCOP 2000)*, Hawthorne, NY 10532, USA, April 2000.

[DCBS09] Daniel, Florian, Fabio Casati, Boualem Benatalla und Ming-Chien Shan: *Hosted Universal Composition: Models, Languages and Infrastructure in mashArt.* In: *Proceedings of ER'09*, November 2009.

[DCS+09] Daniel, Florian, Fabio Casati, Stefano Soi, Jonny Fox, David Zancarli und Ming-Chien Shan: *Hosted Universal Integration on the Web: the mashArt Platform.* In: *Proceedings of ICSOC/ServiceWave 2009.* Springer, November 2009.

[DYB+07] Daniel, Florian, Jin Yu, Boualem Benatallah, Fabio Casati, Maristella Matera und Regis Saint-Paul: *Understanding UI Integration: A Survey of Problems, Technologies, and Opportunities.* Internet Computing, IEEE, 11(3):59–66, 2007.

[FBN08] Fischer, Thomas, Fedor Bakalov und Andreas Nauerz: *Towards an Automatic Service Composition for Generation of User-Sensitive Mashups.* In: *Proceedings of the 16th Workshop on Adaptivity and User Modeling in Interactive Systems*, Würzburg, Deutschland, Oktober 2008.

[FBN09] Fischer, Thomas, Fedor Bakalov und Andreas Nauerz: *An Overview of Current Approaches to Mashup Generation.* In: *Proceedings of the International Workshop on Knowledge Services and Mashups (KSM09)*, Solothurn, Schweiz, März 2009.

[Fla02] Flatscher, Rony G.: *Metamodeling in EIA/CDIF—Meta-Metamodel and Metamodels.* ACM Transactions on Modeling and Computer Simulation (TOMACS), 12(4):322–342, 2002.

[GCP01] Gómez, Jaime, Cristina Cachero und Oscar Pastor: *Conceptual Modeling of Device-Independent Web Applications.* Multimedia, IEEE, 8(2):26–39, Juni 2001.

[GHJV94] Gamma, Erich, Richard Helm, Ralph E. Johnson und John Vlissides: *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley Professional, Oktober 1994.

[HvdSB+08] Houben, Geert-Jan, Kees van der Sluijs, Peter Barna, Jeen Broekstra, Sven Casteleyn, Zoltán Fiala und Flavius Frasincar: *Hera*, Band *Web Engineering: Modelling and Implementing Web Applications* der Reihe *Human-Computer Interaction*, Kapitel 10, Seiten 263–301. Springer London, 2008.

[KK02] Karagiannis, Dimitris und Harald Kühn: *Metamodelling Platforms.* In: *EC-WEB '02: Proceedings of the Third International Conference on E-Commerce and Web Technologies*, Seite 182, London, UK, 2002. Springer-Verlag.

[KKK07] Kraus, Andreas, Alexander Knapp und Nora Koch: *Model-Driven Generation of Web Applications in UWE.* In: CEUR-WS (Herausgeber): *MDWE 2007 - 3rd International Workshop on Model-Driven Web Engineering*, Band 261, 2007.

[KKWZ07] Knapp, Alexander, Nora Koch, Martin Wirsing und Gefei Zhang: *UWE - Ein Ansatz zur modellgetriebenen Entwicklung von Webanwendungen.* i-com, 6(3):5–12, 2007.

[KKZB08] Koch, Nora, Alexander Knapp, Gefei Zhang und Hubert Baumeister: *UML-Based Web Engineering: An Approach based on Standards*, Band *Web Engineering: Modelling and Implementing Web Applications* der Reihe *Human-Computer Interaction*, Kapitel 7, Seiten 157–191. Springer London, 2008.

[Kro08] Kroiss, Christian: *Modellbasierte Generierung von Web-Anwendungen mit UWE (UML-based Web Engineering).* Diplomarbeit, Institut für Informatik, Ludwig-Maximilians-Universität München, Juni 2008.

[Krü09] Krüger, Robert: *Kompositions- und Kommunikationsmodell für Web-Widgets.* Diplomarbeit, Technische Universität Dresden, 2009.

[LdS04] Leite, Jair C. und Lirisnei Gomes de Sousa: *Extensibility and Reusability of Web User Interface Components using XICL.* In: *Developing User Interfaces with XML: Advances on User Interface Description Languages, Satellite Workshop of Advanced Visual Interfaces 2004*, Seiten 31–38, Gallipoli, Italien, Mai 2004.

[Lop02] Lopes, Cristina Videira: *Aspect-Oriented Programming: An Historical Perspective (What's in a Name?).* Technischer Bericht UCI-ISR-02-5, Institute for Software Research, University of California, Irvine, Dezember 2002.

[LW07] Lu, Xudong und Jiancheng Wan: *Model Driven Development of Complex User Interface.* In: *MDDAUI*, 2007.

[@Mer06] Merrill, Duane: *Mashups: The new breed of Web app.* `http://www.ibm.com/developerworks/library/x-mashups.html`, August 2006. Besucht am 13. Mai 2009.

[@MLM+06] MacKenzie, C. Matthew, Ken Laskey, Francis McCabe, Peter F. Brown und Rebekah Metz: *Reference Model for Service Oriented Architecture 1.0.* `http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.html`, Oktober 2006. Besucht am 13. Mai 2009.

[Mur08] Murugesan, San: *Web Application Development: Challenges and the Role of Web Engineering*, Band *Web Engineering: Modelling and Implementing Web Applications* der Reihe *Human-Computer Interaction*, Kapitel 2, Seiten 7–32. Springer London, 2008.

[@Neu09] Neumann, Alexander: *Forrester-Studie: SOA ist sehr lebendig.* `http://www.heise.de/newsticker/Forrester-Studie-SOA-ist-sehr-lebendig--/meldung/137721`, Mai 2009. Besucht am 21. Mai 2009.

[NM95a] Nierstrasz, Oscar und Theo Dirk Meijler: *Requirements for a Composition Language.* In: *Proceedings of the ECOOP 94 workshop on Models and Languages for Coordination of Parallelism and Distribution, Bologna, Italy, July 5, 1994, Selected Papers*, Band 924 der Reihe *Lecture Notes in Computer Science*, Seiten 147–161, Bologna, Italien, Juli 1995. Springer.

[NM95b] Nierstrasz, Oscar und Theo Dirk Meijler: *Research directions in software composition.* ACM Comput. Surv., 27(2):262–264, 1995.

[NvdSH+09] Niederhausen, Matthias, Kees van der Sluijs, Jan Hidders, Erwin Leonardi, Geert-Jan Houben und Klaus Meissner: *Harnessing the Power of Semantics-based, Aspect-Oriented Adaptation for AMACONT.* In: Gaedke, M., M. Grossniklaus und O. Díaz (Herausgeber): *Proceedings of the 9th International Conference on Web Engineering (ICWE 2009)*, Edition 5648, San Sebastian, Spain, Juni 2009. Springer.

[OMG02] Object Management Group: *Meta Object Facility (MOF) Specification*, April 2002.

[OMG03] Object Management Group: *MDA Guide Version 1.0.1*, Juni 2003.

[OMG06a] Object Management Group: *Meta Object Facility (MOF) Core Specification*, Januar 2006.

[OMG06b] Object Management Group: *Object Constraint Language, OMG Available Specification Version 2.0*, Mai 2006.

[OMG07a] Object Management Group: *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*, Juli 2007.

[OMG07b] Object Management Group: *MOF 2.0/XMI Mapping, Version 2.1.1*, Dezember 2007.

[OMG09] Object Management Group: *OMG Unified Modeling Language (OMG UML), Superstructure Version 2.2*, Februar 2009.

[PFPA06] Pastor, Oscar, Joan Fons, Vicente Pelechano und Silvia Abrahão: *Conceptual Modelling of Web Applications: The OOWS Approach*, Band *Web Engineering*, Kapitel 9, Seiten 277–302. Springer Berlin Heidelberg, 2006.

[PVM09] Pietschmann, Stefan, Martin Voigt und Klaus Meissner: *Dynamic Composition of Service-Oriented Web User Interfaces*. In: *Proceedings of the 4th International Conference on Internet and Web Applications and Services (ICIW 2009)*, Seiten 217–222, Mestre/Venice, Italien, Mai 2009. IEEE CPS.

[PVRM09] Pietschmann, Stefan, Martin Voigt, Andreas Rümpel und Klaus Meissner: *CRUISe: Composition of Rich User Interface Services*. In: Gaedke, M., M. Grossniklaus und O. Díaz (Herausgeber): *Proceedings of the 9th International Conference on Web Engineering (ICWE 2009)*, Edition 5648, Seiten 473–476, San Sebastian, Spanien, Juni 2009. Springer.

[RR06] Robertson, Suzanne und James Robertson: *Mastering the requirements process*. Addison-Wesley, 2 Auflage, 2006.

[SBPM08] Steinberg, Dave, Frank Budinsky, Marcelo Paternostro und Ed Merks: *EMF Eclipse Modeling Framework*. Eclipse Series. Addison-Wesley Professional, 2 Auflage, Dezember 2008.

[SK03] Schwinger, Wieland und Nora Koch: *Modellierung von Web-Anwendungen*, Band *Web Engineering: Systematische Entwicklung von Web-Anwendungen*, Kapitel 3, Seiten 49–75. dpunkt.verlag, Oktober 2003.

[@UWE09] UWE-Group: *UWE - Publications ordered by year*. `http://uwe.pst.ifi.lmu.de/publicationsByYear.html`, Juli 2009. Besucht am 25. August 2009.

[Voc09] Vock, Dominik: *Analyse und Vergleich bestehender Mashup-Ansätze*. Belegarbeit Technische Universität Dresden, Fakultät Informatik, Institut für Software- und Multimediatechnik, Fachgebiet Multimediatechnik, April 2009.

[Voi08] Voigt, Martin: *Entwicklung einer Service-Architektur für User Interfaces*. Belegarbeit Technische Universität Dresden, Fakultät Informatik, Institut für Software- und Multimediatechnik, Fachgebiet Multimediatechnik, Juli 2008.

[Wal09] Waltsgott, Johannes: *Clientseitige Integration von User Interface Services*. Diplomarbeit, Technische Universität Dresden, 2009.

[@WebR07] Kompetenzzentrum, WebRatio: *WebRatio Flyer Cebit*. `http://www.isearch-it-solutions.de/documents/webratio-flyer-cebit.pdf`, März 2007. Besucht am 23. August 2009.

[WH07] Wong, Jeffrey und Jason I. Hong: *Making Mashups with Marmite: Towards End-User Programming for the Web*. In: *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, Seiten 1435–1444, New York, NY, USA, 2007. ACM.

[YBC+07] Yu, Jin, Boualem Benatallah, Fabio Casati, Florian Daniel, Maristella Matera und Regis Saint-Paul: *Mixup: a Development and Runtime Environment for Integration at the Presentation Layer.* In: *Proceedings of the Seventh International Conference on Web Engineering (ICWE'07), LNCS*, Band 4607, Seiten 479–484, Como, Italien, Juli 2007. Springer.

[YBCD08] Yu, Jin, Boualem Benatallah, Fabio Casati und Florian Daniel: *Understanding Mashup Development and its Differences with Traditional Integration.* Internet Computing, IEEE, 12(5):44–52, September–Oktober 2008.

[YBSP+07] Yu, Jin, Boualem Benatallah, Regis Saint-Paul, Fabio Casati, Florian Daniel und Maristella Matera: *A Framework for Rapid Integration of Presentation Components.* In: *Proceedings of the 16th International Conference on World Wide Web*, Seiten 923–932, Banff, Alberta, Canada, Mai 2007. ACM Press.

[ZRN08] Zang, Nan, Mary Beth Rosson und Vincent Nasser: *Mashups: Who? What? Why?* In: *CHI '08: CHI '08 extended abstracts on Human factors in computing systems*, Seiten 3171–3176, New York, NY, USA, 2008. ACM.