Nutzung einer interaktorbasierten UI-Beschreibung zur dynamischen Generierung multimodaler Web2.0-Applikationen

Diplomarbeit von Janó Borrmann
Matrikelnummer: 3122722
Juli 2009

Institut für Systemarchitektur
Fakultät Informatik
TU Dresden

Betreuer:
Dipl.-Inf. Marius Feldmann

verantwortlicher Hochschullehrer:
Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Lehrstuhl Rechnernetze

Aufgabenstellung für die Diplomarbeit

Name, Vorname: Borrmann, Jano

Studiengang: Medieninformatik ***Matr. Nr.:*** 3122722

Thema: ***„Nutzung einer interaktorbasierten UI-Beschreibung zur dynamischen Generierung multimodaler Web2.0-Appliaktionen"***

Beschreibung:

Bereits seit einiger Zeit werden interaktorbasierte Abstrakte User Interfaces (AUI) als Werkzeug zur Kopplung mehrerer Modalitäten in einer Applikation verwendet.

Ziel dieser Diplomarbeit ist, diesen Ansatz auf Charakteristika von "Web 2.0"-Applikationen auszudehnen. Konkret bedeutet dies, dass unterschiedliche UI-Repräsentationen, die die Dynamik einer Ajax-Applikation aufweisen (Ausblenden/Deaktivieren einzelner Interaktoren, Nachladen von Interaktoren etc.) über ein gemeinsames AUI gekoppelt werden sollen. Die AUI-Ebene soll dabei nicht nur das Applikationsmodel, sondern ebenfalls den strukturellen Zustand verwalten. In der Arbeit ist zunächst ein Überblick zu interaktorbasierten Systemen und zu generativen Ansätzen von Web2.0-Applikationen zu geben. Daraufhin ist ein System zu konzipieren, das ausgehend von einer interaktorbasierten UI-Beschreibung konkrete UIs erzeugen kann, die oben genannte Dynamik aufweisen und bei Änderungen sowohl das Datenmodell, als auch die Struktur mit dem zugrundeliegenden AUI synchronisieren (getriggert sowohl durch Änderungen in den konkreten Repräsentationen, als auch im AUI). Mehrere konkrete UI Repräsentationen sind parallel an ein AUI zu koppeln. Zentrale Entität in diesem System ist dabei eine AUI-Repräsentation, die im Laufe der Arbeit ermittelten Anforderungen entspricht. Dazu ist ausgehend von einer in Entwicklung befindlichen AUI-Sprache (Maria XML) eine den Bedingungen gerecht werdende Sprache zu spezifizieren. Das konzipierte System ist in prototypischer Weise zu implementieren und dabei auf geeignete Weise zu evaluieren. Bei der Implementierung sollen möglichst unterschiedliche konkrete UI-Repräsentationen generativ erzeugt werden.

Zentrale Ergebnisse:

- Übersicht zu Ansätzen modellgetriebener Entwicklung von Web2.0-Applikationen
- Anforderungsspezifikation für interaktorbasierte UI-Beschreibung
- Konzeption eines Systems zur dynamischen Generierung von Web2.0-Applikationen
- Prototypische Umsetzung des Systems

Betreuer: Dipl.-Inf. Marius Feldmann

Verantwortlicher Hochschullehrer: Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill

Institut: Systemarchitektur

Beginn am: 01. Februar 2009

Einzureichen am: 31. Juli 2009

Unterschrift des verantwortlichen Hochschullehrers

Inhalt

1 Einleitung

Aktuell sind verschiedene Ansätze zur modellgetriebenen Entwicklung von Webapplikationen Gegenstand der Forschung. Eine Möglichkeit zur Modellierung derartiger Applikationen ist die Verwendung einer Beschreibungssprache zum abstrakten Spezifizieren der Nutzerschnittstelle. Mit MARIA XML (*Model-bAsed descRiption of Interactive Applications*) wurde eine solche Sprache entwickelt, mit der interaktorbasierte User Interfaces (UIs) abstrakt definiert werden können. Bisher wurden keine Analysen der Eignung einer solchen Sprache zur abstrakten Beschreibung von multimodalen Web2.0-Applikationen durchgeführt.

1.1 Motivation und Ziele

Um die Eignung der abstrakten UI-Beschreibungssprache MARIA XML zu analysieren, werden im Rahmen der vorliegenden Arbeit die vier folgenden Kernfragen untersucht:

Kernfrage 1: Reicht die Mächtigkeit der Sprache MARIA XML aus, um multimodale Web2.0-Applikationen abstrakt zu beschrieben?

Es stellt sich die Frage, ob die Definition eines Abstract User Interfaces (AUI) unter Verwendung der in MARIA XML definierten Elemente zur Generierung von Final User Interfaces (FUI) für Web2.0-Applikationen ausreicht oder um weitere Interaktoren erweitert werden muss und damit den Charakter eines Concrete User Interfaces (CUI) annimmt.

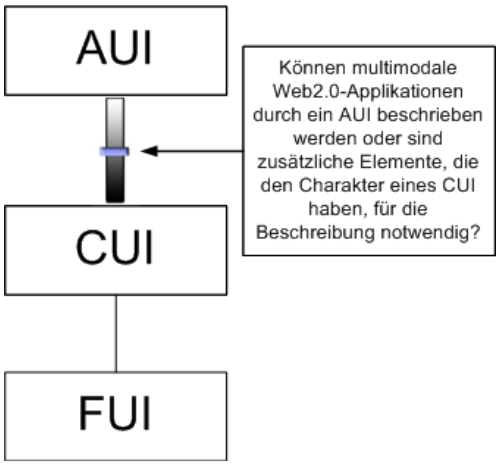


Abbildung 1: Schema: AUI, CUI, FUI

Dazu gilt es im ersten Schritt die Mächtigkeit von MARIA XML zur Beschreibung von multimodalen Web2.0-Applikationen zu analysieren und daraufhin gegebenenfalls notwendige Erweiterungen zu definieren, die zwar einen direkten Bezug zu den aus ihnen generierten Interaktoren in dem FUI aufweisen können, jedoch eine implementierungsunabhängige Beschreibung gewährleisten.

Kernfrage 2: Führt eine Zunahme der Funktionalität zur Abnahme der Generik?
Aus der abstrakten Beschreibung einer Web2.0-Applikation sollen möglichst viele Funktionalitäten abgeleitet werden. Es muss untersucht werden, inwieweit die für die Transformation zur Erzeugung der FUIs zuständige Komponente einerseits so gestaltet wird, dass aus einer kompakten Beschreibung einer Applikation viele Funktionalitäten abgeleitet werden können, ohne dass diese explizit definiert werden müssen und andererseits der Transformator generisch aufgebaut wird, so dass er für verschiedene Anwendungsszenarien wiederverwendbar ist.

Kernfrage 3: Kann die Anwendung zur Laufzeit durch Änderung von Parametern konfiguriert werden?
Ein Untersuchungsschwerpunkt ist zudem die Konfigurierbarkeit der aus den abstrakten Beschreibungen erzeugten Applikationen. Welche Kontextinformationen wie beispielsweise der eingesetzte Client oder die Präferenzen bezüglich verwendeter Frameworks können das Transformationsergebnis beeinflussen?

Kernfrage 4: Kann die erforderliche Systemarchitektur so konzipiert werden, dass ein flexibler und effizienter Transformationsablauf unter Verwendung kompakter Transformatoren gewährleistet ist?
Für die Durchführung der Transformationen zum direkten Erzeugen der finalen Nutzerschnittstellen aus den abstrakten Beschreibungen ist ein geeignetes System zu entwickeln. Das Transfervolumen zum Übertragen der Daten vom Server zum Client soll möglichst gering ausfallen, um die Anwendung effizient zu gestalten. Die Transformationen sollen ad-hoc durchgeführt werden, damit einzelne Interaktoren in der finalen Nutzerschnittstelle zur Laufzeit ausgetauscht werden können. Die Systemarchitektur soll unabhängig von der eingesetzten Modalität konzipiert werden, jedoch einen Modalitätswechsel zur Laufzeit ermöglichen. Der Ablauf einer erzeugten Applikation soll nicht nur linear, sondern in Abhängigkeit der Nutzerinteraktionen erfolgen. Das System soll eine flexible Auswahl der zu generierenden Präsentationen erlauben.

1.2 Aufbau der Arbeit

Ausgehend von den in der Motivation gestellten Fragen soll im Rahmen der vorliegenden Arbeit die Eignung von MARIA XML zur abstrakten Beschreibung Ajax-basierter Webszenarien untersucht werden. Für eine tiefgründige Analyse ist es erforderlich, klare Anforderungen zu definieren, die während der Konzeption aufgegriffen werden. Aus diesem Grund werden im ersten Schritt vier Beispielszenarien mit typischen Web2.0-Interaktionsmöglichkeiten entwickelt, aus denen die Anforderungen abgeleitet und allgemein formuliert werden. Im nächsten Schritt werden bedeutende verwandte Arbeiten vorgestellt, um einen Überblick über den Stand der Technik und Forschung zu geben. Anhand der aufgestellten Anforderungen erfolgt anschließend die Abgrenzung von diesen Arbeiten, indem den Anforderungen entsprechende Differenzierungskriterien gegenübergestellt werden. Die folgende Konzeption gliedert sich in zwei Schritte. Im ersten Schritt wird ein System zur Durchführung der Transformation einer abstrakt beschriebenen UI in die dazugehörige finale Nutzerschnittstelle entwickelt. Mit dem nächsten Schritt folgt eine detaillierte Analyse der Beispielszenarien, um die Eignung der in MARIA XML definierten Elemente und Attribute festzustellen. Es wird untersucht, welche Funktionalitäten abgeleitet werden können und notwendige Erweiterungen von MARIA XML definiert. Anschließend wird der generische Transformator mit den zu den abstrakt beschriebenen Interaktoren gehörigen Templates konzipiert. Damit ist die Konzeption abgeschlossen und es können entsprechende AUI-Beschreibungen für die vorgestellten Szenarien entwickelt werden, um aus diesen nach Umsetzung des konzipierten Systems die finalen Nutzerschnittstellen zu generieren. Zum Abschluss erfolgt die Evaluierung des Vorgehens. Es wird untersucht, inwieweit die zu Beginn der Arbeit definierten Anforderungen erreicht wurden und auf die Eignung von MARIA XML zur abstrakten Beschreibung von Web2.0-Nutzerschnittstellen eingegangen. Im letzten Kapitel wird die Arbeit zusammengefasst und ein Ausblick gegeben.

2 Anforderungsanalyse

Um dem Ziel, eine möglichst große Anzahl typischer Web2.0-Interaktionsmöglichkeiten umzusetzen, gerecht zu werden, erfolgt eine Zusammenstellung vier verschiedener Ajax-basierter Szenarien, die aus im Web oft anzutreffenden Anwendungsfällen bestehen. Außerdem gilt es neben der Interaktion per Maus und Tastatur eine weitere Modalität mit einzubeziehen. Ein Szenario wird deshalb die Möglichkeit zur Sprachein- und ausgabe beinhalten.

2.1 Szenarienbeschreibung

Bei dem ersten Szenario werden statt einfacher textbasierter Interaktoren Widgets eingesetzt, um die Eingabe von Informationen entsprechend der Nutzergewohnheiten umzusetzen. Ein solches Widget soll gegen ein anderes mit gleicher Funktionalität zur Laufzeit ausgetauscht werden können oder nach Ermittlung des User Agents gegebenenfalls durch eine platzsparende Eingabemöglichkeit ersetzt werden. Zudem ist ein Mechanismus zu implementieren, der es erlaubt, die Daten für einzelne Interaktoren entsprechend der Nutzereingaben vom Server zu laden und in die bestehende Seite einzubinden. Die Eingaben des Anwenders sollen getriggert werden, um Ereignisse auszulösen, die zusätzliche Funktionalitäten aktivieren. Vor einem Darstellungswechsel werden die Angaben des Nutzers validiert und daraus der Ablauf des Szenarios bestimmt.

Einige vorhandene Web2.0-Applikationen sind so gestaltet, dass sie sich mit anderen zu einem Mashup kombinieren lassen. Beim nächsten Szenario soll durch die Einbindung einer vorhandenen Web2.0-Applikation ein Mashup erzeugt werden. Die eingebundenen Applikationen sollen zur Laufzeit durch andere, welche die gleiche Funktionalität bieten, austauschbar sein. Zudem wird in diesem Szenario das Aktivieren von vorhandenen, aber nicht eingeblendeten Interaktoren nach Eintreten eines Ereignisses umgesetzt.

Das dritte Szenario dient der Präsentation periodisch wechselnder Informationen. Als eine typische Web2.0-Präsentationsmöglichkeit wird dazu ein Newsticker eingebunden. Die Interaktion zur Selektion einzelner Objekte soll mit Drag&Drop erfolgen. Zur Ablage der ausgewählten Objekte wird die Warenkorbmetapher verwendet.

Beim vierten Szenario soll neben der visuellen Nutzerschnittstelle die Interaktion per Sprachein- und ausgabe angeboten werden. Der Anwender soll jederzeit zwischen den beiden Modalitäten wechseln können. Das Szenario ist ein Dialog, in welchem dem Anwender Entscheidungsfragen gestellt werden.

2.1.1 Szenario 1: Flugbuchung

Im ersten Szenario soll der Nutzer eine Flugreise samt Unterkunft buchen können. Nachdem aus einer Liste das gewünschte Reiseziel gewählt wurde, steht für die Auswahl des Abflugtermins ein Kalenderwidget zur Verfügung, in dem mögliche Flugtermine markiert sind. Auf der nachfolgenden Seite kann der Nutzer einen der angegebenen Termine für den Rückflug auswählen. Ist dies geschehen, muss er sich für eine Kategorie seiner Unterkunft entscheiden. Daraufhin wird dynamisch eine Liste mit möglichen Unterkünften angezeigt. Nach Auswahl einer Unterkunft werden die notwendigen persönlichen Daten des Buchenden abgefragt. Das Textfeld für die Straße zeigt, nach Eingabe eines oder mehrerer Buchstaben, mögliche Straßennamen als Vorschläge, welche übernommen werden können, an. Die Eingabe der Postleitzahl wird auf korrektes Format überprüft. Wurden alle Informationen korrekt eingetragen, erfolgt eine Zusammenfassung aller getätigten Angaben und die Buchung kann bestätigt werden. Daraufhin ist auf der letzten Seite eine Bestätigungsmeldung zu sehen.

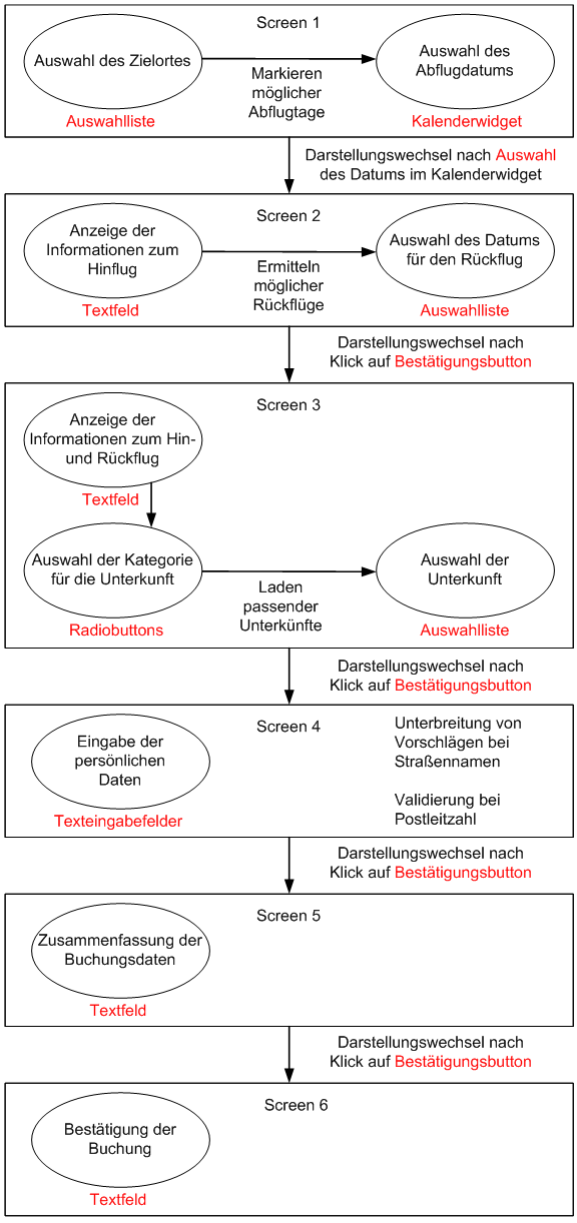


Abbildung 2: Ablauf und Interaktoren des Flugbuchungsszenarios

2.1.2 Szenario 2: Positionsermittlung

Bei diesem Szenario soll ein Mashup durch Kombination einer externen mit der eigenen Web2.0-Anwendung umgesetzt werden. Dazu werden die von Google beziehungsweise Microsoft angebotenen Kartendienste Google Maps [1] und Virtual Earth [2] eingebunden, um die Koordinaten einer Sehenswürdigkeit oder eines Ortes zu finden und auf der Karte zu lokalisieren. Als Ausgangspunkt wird die Auswahlmöglichkeit zwischen Sehenswürdigkeiten und Orten angeboten. Hat sich der Nutzer für eine Variante entschieden, wird dynamisch eine Liste mit entsprechenden Örtlichkeiten gefüllt. Wählt der Buchende davon eine per Maus aus, erfolgt auf der nächsten Seite die Anzeige der Adresse dazu, nachdem diese asynchron vom Server geladen wurde. Durch Anklicken eines Buttons kann sich der Nutzer daraufhin die Koordinaten anzeigen lassen. Möchte er die Stelle des Ortes auf der Karte sehen, ist ein Klick auf den entsprechenden Button notwendig. Daraufhin öffnet sich im nächsten Fenster die Kartenansicht mit dem korrekt platzierten Marker auf der Karte.

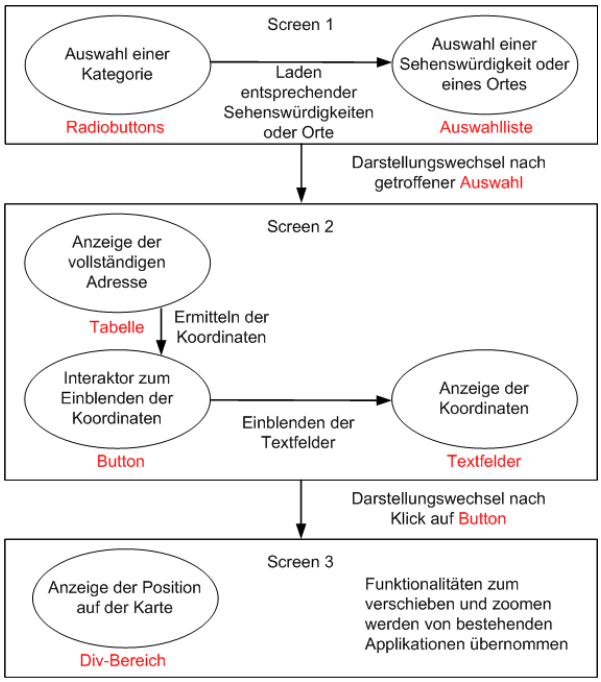


Abbildung 3: Ablauf und Interaktoren des Szenarios zur Positionsermittlung

2.1.3 Szenario 3: Newsticker

Im dritten Szenario wird als eine typische Interaktionsmöglichkeit von Ajax ein Newsticker umgesetzt. Dafür wird die Pollingfunktionalität benötigt. Es werden fortwährend in einem festgelegten Intervall Anfragen an den Server gestellt, um verschiedene Listen mit Produkten und deren Preisen abzurufen. Entscheidet sich der Nutzer für eine der Produktlisten, kann er dies durch Anklicken eines Buttons der Anwendung mitteilen. In der nachfolgenden Ansicht wird die Metapher eines Warenkorbs umgesetzt. Dieser kann durch Nutzerinteraktion per Drag&Drop gefüllt werden. Dazu werden die Produkte aus der in der vorherigen Ansicht gewählten Liste mit der Maus in eine freie Fläche, welche den Warenkorb darstellt, verschoben. Sind die gewünschten Produkte an dieser Stelle abgelegt, kann mittels Klick auf einen Button der Kauf durchgeführt werden. In der sich daraufhin öffnenden Ansicht werden die Produkte aufgelistet und der berechnete Gesamtpreis angezeigt. Hier kann durch Anklicken eines Buttons der Kauf bestätigt werden, woraufhin sich die letzte Ansicht mit einer Bestätigungsmeldung öffnet.

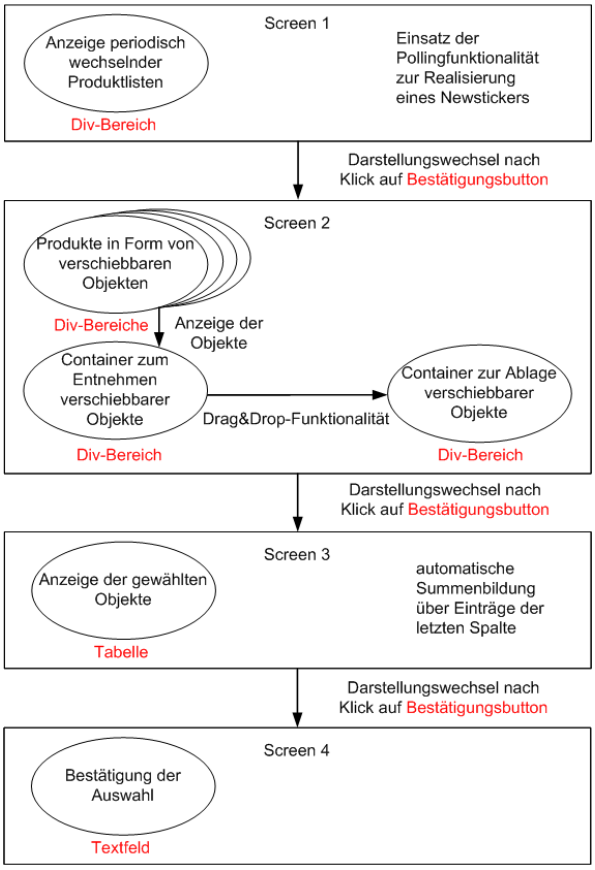


Abbildung 4: Ablauf und Interaktoren des Newstickerszenarios

2.1.4 Szenario 4: Ticketkauf mit multimodaler Interaktion

Um die Realisierbarkeit eines multimodalen Szenarios nicht von der Qualität des verwendeten Spracherkenners abhängig zu machen, wird auf die Verwendung eines Vokabulars, welches beliebige Worte enthält, verzichtet. Die Interaktion beschränkt sich bei diesem Szenario auf *Yes*- und *No*-Eingaben. Aus einer abstrakten Nutzerschnittstellenbeschreibung sollen beide Modalitäten erzeugt werden. Für die Generierung der konkreten Nutzerschnittstelle zur Sprachein- und ausgabe wird ein separater Transformator erstellt. Während der Interaktion kann von der visuellen Modalität

in die akustische gewechselt werden. Das Szenario wird eine Anwendung zum Erwerb von Bahntickets. Im ersten Schritt muss der Nutzer die gewünschte Tarifzone festlegen, anschließend angeben, ob er ein ermäßigtes Ticket kaufen möchte und danach der Anwendung mitteilen, wann er abreisen möchte. Zum Abschluss wird der Anwender gefragt, ob er weitere Tickets zu gleichen Konditionen kaufen möchte. Da sich die Interaktion auf Bestätigung beziehungsweise Verneinung beschränkt, werden die möglichen Tarifzonen und Abreisetage von der Anwendung aufgezählt, bis der Nutzer eine beziehungsweise einen auswählt. In Abbildung 5 ist der Ablauf des Szenarios dargestellt.

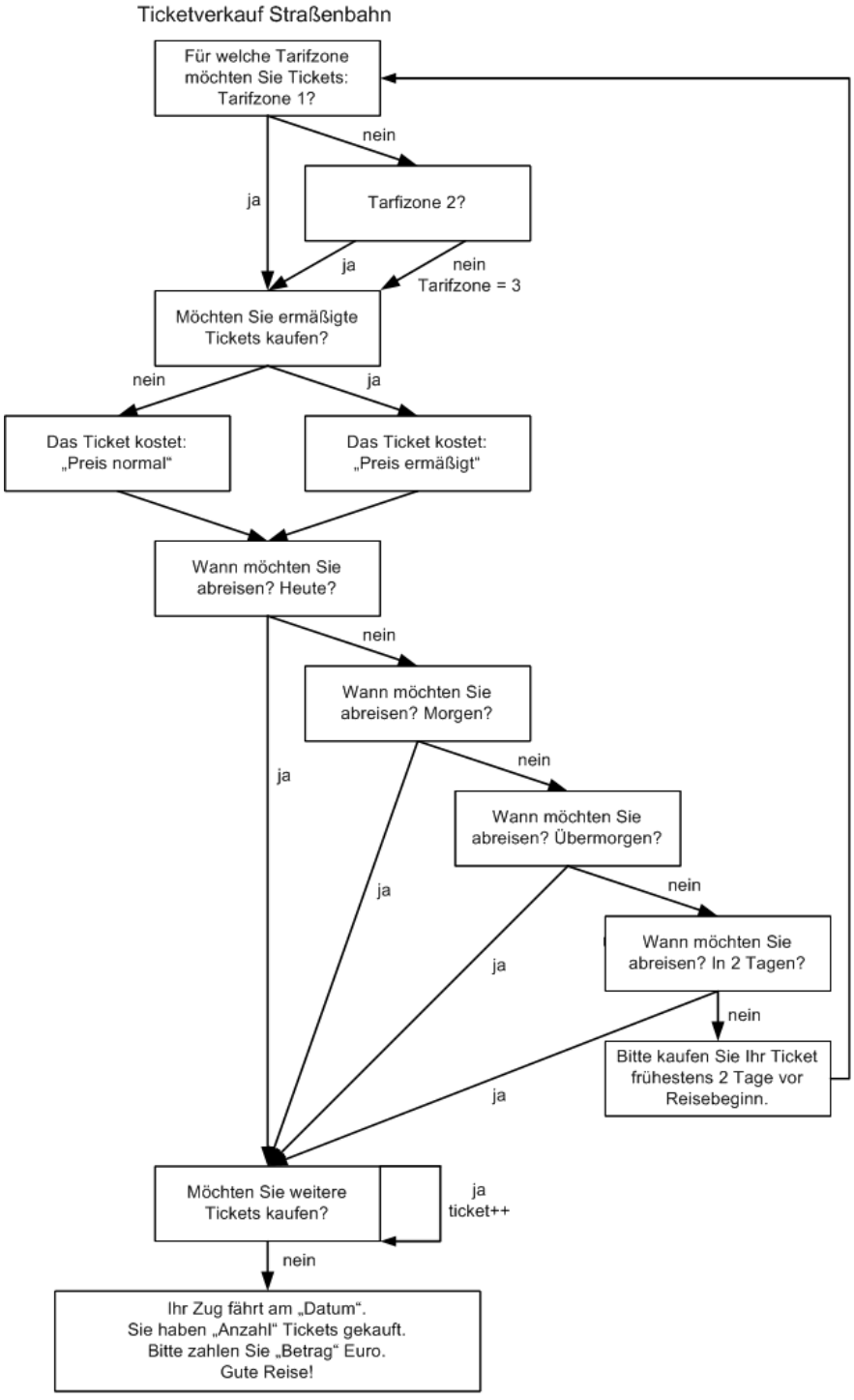


Abbildung 5: Ablauf des multimodalen Szenarios zum Ticketkauf

2.2 Identifikation der Anforderungen

Für die Umsetzung der genannten Szenarien sind verschiedene Anforderungen zu erfüllen, denen insbesondere bei der Definition der abstrakten Beschreibung der Nutzerschnittstelle und Entwicklung des Systems zur effizienten Durchführung der Transformationen Beachtung geschenkt werden muss. Um ein multimodales Szenario zu erzeugen, bei welchem für die Interaktion neben einer Ajax-basierten Weboberfläche die Ein- und Ausgabe von Informationen per Sprache möglich ist, muss die abstrakte Beschreibung der Nutzerschnittstelle passende Möglichkeiten zur gegenseitigen Synchronisation bereithalten. Abweichungen bei der Implementierung zwischen der visuellen Umsetzung mit Ajax und der akustischen müssen durch die abstrakte Beschreibung mit Hilfe entsprechender Attribute kompensiert werden, um den Ablauf bei der Bedienung der Anwendung identisch gestalten zu können. Die Möglichkeit Daten vom Server anzufordern, nachdem die eigentliche Seite schon geladen wurde, muss bei der Entwicklung des Systems und bei der Definition des abstrakten User Interfaces (AUI) durch entsprechende Attribute berücksichtigt werden. Dies kann entweder als Ereignis nach einer Nutzerinteraktion oder automatisiert durch Polling geschehen. Die Einbindung von Widgets, wie im Flugbuchungsszenario erfolgt, muss durch das AUI derart unterstützt werden, dass das Ergebnis, im konkreten Fall das Datum, welches der Nutzer mit Hilfe des Kalenders auswählt, korrekt für die weitere Verwendung bereitgestellt wird. Die durch den Einsatz von Ajax geschaffenen Möglichkeiten der Nutzerinteraktion, wie beispielsweise Drag&Drop, sollen bei der abstrakten Beschreibung des UI bereits Beachtung finden. Das AUI soll es ermöglichen, eigene Funktionen einzubinden und auch eigene Datentypen, beispielsweise die Rückgabewerte von Operationen, zu definieren. Damit auf diese Rückgabewerte über mehrere Seiten hinweg zugegriffen werden kann, ist die Unterstützung von Sessions notwendig. Durch den Einsatz von Sessions soll es möglich sein, die Anwendung zur Laufzeit durch Änderung einzelner Parametern zu konfigurieren. Zudem soll die Möglichkeit bestehen, eine externe Anwendung wie beispielsweise Google Maps [1] oder Microsoft Virtual Earth [2] einzubinden, um ein für Web2.0-Anwendungen typisches Mashup-Szenario umzusetzen. Die modellbasierte UI-Beschreibung soll weiterhin von den verwendeten Frameworks abstrahieren. An einigen Stellen werden in den oben beschriebenen Szenarien die Javascriptbibliotheken Prototype [3] und Scriptaculous [4] eingesetzt. Das AUI soll jedoch nicht speziell auf diese zugeschnitten sein, sondern es ermöglichen, sie durch andere, welche die gleiche Funktionalität bieten, zur Laufzeit zu ersetzen. Weiterhin sollen die aus einem einmal definierten AUI durch Transformation erzeugten finalen UIs nicht nur auf einer speziellen Plattform ausgeführt werden, sondern allgemein auf Geräten mit Browsern, die AJAX unterstützen, lauffähig sein. Nachdem das AUI für die oben genannten Szenarien erstellt ist, soll es möglich sein, aus diesem beispielsweise die letztendliche Anwendung sowohl für Desktopcomputer als auch für Mobiltelefone, welche Ajax unterstützen, zu generieren. Das Erstellen des AUI soll nicht nur mit Hilfe eines proprietären Werkzeugs möglich sein, sondern dem Entwickler die Wahl eines geeigneten Werkzeugs überlassen.

Die genannten Anforderungen an das zu konzipierende System und an das AUI werden in Tabelle 1 beziehungsweise Tabelle 2 im Überblick dargestellt.

Anforderungen an das System	
Multimodalität	• akustisch: Sprachein- und ausgabe • visuell: Ajax-basierte Webseiten • Wechsel der Modalität zur Laufzeit und Synchronisation zwischen beiden Modalitäten
asynchroner Datentransfer	• Übertragung von Daten und dynamische Änderungen an einer Seite nachdem diese schon geladen wurde
Einsatz von Sessions	• Erzeugen der einzelnen Seiten bei Aufruf durch ad-hoc Transformation • flexible Gestaltung des Ablaufs innerhalb der Szenarien • Konfigurierbarkeit der Anwendung zur Laufzeit
Verwendung von Standards	• Verzicht auf ein proprietäres Entwicklungswerkzeug • Einsatz einer standardisierten Sprache zur Erstellung der Transformationsregeln
Effizienz	• effiziente Durchführung der Transformationen • Ermitteln der für die Transformation benötigten Zeit durch Vergleich mit dem Zeitaufwand beim Aufruf statisch erzeugter Seiten

Tabelle 1: Anforderungen an das System

Anforderungen an das AUI	
Einsatz von Widgets	• Einbindung externer UI-Komponenten
Web2.0-Nutzerinteraktionen	• Identifizierung wiederkehrender Codefragmente • Generalisierung • Wiederverwendbarkeit
Erzeugen von Mashups	• Einbindung externer Web2.0-Anwendungen
Abstraktion von Zielplattform	• AUI-Beschreibung unabhängig von verwendeten Frameworks zur konkreten Umsetzung und der Zielplattform
Verwendung einer gegebenen Sprache zur abstrakten Beschreibung	• Analyse der Mächtigkeit von MARIA XML mit Fokus auf Web2.0-Applikationen • Identifizierung ableitbarer Funktionalitäten • gegebenenfalls Erweiterung von MARIA XML

Tabelle 2: Anforderungen an das AUI

3 Grundlagen

3.1 Allgemeines

Zur Umsetzung der vorgestellten Web2.0-Applikationen wird AJAX verwendet. Unter dem Marketingbegriff AJAX = *Asynchronous Javascript And XML* [5] sind die Möglichkeit, Daten asynchron vom Server zum Client zu übertragen, die Skriptsprache *Javascript* [6] und die Auszeichnungssprache *Extensible Markup Language* (XML) [7] zusammengefasst. Das Zusammenspiel der Technologien erlaubt es, Inhalte auf Webseiten aufgrund von durch Nutzerinteraktion ausgelösten Ereignissen dynamisch anzupassen. Mit Hilfe von Javascript können Änderungen an einer Seite lokal durchgeführt werden, ohne nach jeder Interaktion eine neue HTTP-Anfrage an den Server zu stellen. Das asynchrone Laden benötigter Daten erfolgt im Hintergrund transparent für den Nutzer. Dadurch sind die Nutzerinteraktion und die serverseitige Verarbeitung voneinander entkoppelt. Zur Speicherung der Daten, die asynchron geladen und in eine Seite eingebunden werden können, wird das Format XML verwendet.

3.2 MARIA XML

In diesem Kapitel wird die zur abstrakten Beschreibung von multimodalen Web2.0-Applikationen verwendete Sprache vorgestellt. Nach einer Einführung erfolgen die Darstellung der hierarchischen Struktur der Sprache und die Vorstellung der Elemente, die zur abstrakten Beschreibung von Interaktoren verwendet werden können. Anschließend wird an einem Beispiel gezeigt, wie eine abstrakte Beschreibung erweitert werden kann, um ein plattformabhängiges CUI auf Basis von MARIA XML zu entwickeln. Zur Generierung von Modellen zur UI-Erzeugung für bestehende Webservices kann ebenfalls MARIA XML verwendet werden. In diesem Unterkapitel wird der Ablauf zum Erstellen derartiger FUIs beschrieben. Nachfolgend wird ein Werkzeug vorgestellt, welches es erlaubt, die Transformationen zwischen den einzelnen Abstraktionsebenen und die Generierung der FUIs semi-automatisch durchzuführen. Abschließend erfolgt die Vorstellung eines Konzeptes zur Migration einer Nutzerschnittstelle von einem Endgerät auf ein anderes.

Einführung

Das Ziel bei der Erstellung von MARIA XML ist die Entwicklung einer modellbasierten Sprache, welche es erlaubt, Multi-Device-Interfaces abstrakt zu definieren [8]. Dabei soll auf die Daten und Funktionalitäten verschiedener Dienste zugegriffen und die Generierung migrierbarer Nutzerschnittstellen für Applikationen, die auf Webservices basieren, unterstützt werden. Neben der Definition der Sprache wird ein Werkzeug zur semi-automatischen Durchführung der Transformationen entwickelt. Es soll möglich sein, die Transformatoren anzupassen, ohne die Werkzeugimplementierung zu ändern. Deshalb sollen die Transformatoren nicht hart-codiert in das Werkzeug eingebunden werden, sondern extern spezifiziert werden. In der vorliegenden Arbeit werden die Transformatoren, wie in Kapitel 2.2 bereits beschrieben, unabhängig von einem Werkzeug, welches zur Durchführung der Transformationen dient, entwickelt. Aus diesem Grund ist es möglich, die Transformatoren jederzeit zu modifizieren und zu erweitern. Der Ausgangspunkt für die Transformationen sind nach dem durch die Sprache MARIA XML gegebenen Schema erstellte abstrakte Interfacebeschreibungen.

MARIA XML basiert auf dem Cameleon Framework [14]. Durch dieses Framework wird definiert, dass eine abstrakte Beschreibung einer Nutzerschnittstelle unabhängig von den verfügbaren Interaktionsressourcen des Endgerätes definiert werden soll und eine konkrete Beschreibung in Abhängigkeit von der Interaktionsmodalität, aber unabhängig von der Implementierungssprache, erfolgt. Bei der Definition von MARIA wurde dieser modulare Ansatz verwendet. Neben der Möglichkeit, abstrakte Beschreibungen zu erstellen, gibt es mehrere plattformabhängige Sprachen, durch welche die abstrakte Definition in Abhängigkeit der betrachteten Interaktionsressourcen erweitert werden kann. Neben visuellen Interfaces können damit beispielsweise akustische Ein- und Ausgabemöglichkeiten beschrieben werden.

Durch MARIA XML werden ein *Datenmodell* und ein *Ereignismodell* definiert. Die verwendeten Interaktoren können an Elemente eines Datentyps im Datenmodell gebunden werden. Die Beschreibung des Datenmodells erfolgt mit der Sprache XSD. Mit Hilfe des Datenmodells kann beispielsweise das Format von Eingabewerten spezifiziert oder bedingte Verbindungen zwischen Präsentationen definiert werden, wobei sich die Überprüfung in dem Fall auf den Datentyp bezieht. Durch das Ereignismodell wird definiert, wie das User Interface auf getriggerte Ereignisse antwortet. In MARIA XML sind zwei Arten von Ereignissen definiert:

property change events: Ereignisse, die den Status einiger UI Eigenschaften verändern

activation events: Ereignisse, die durch Interaktoren ausgelöst werden, um Applikationsfunktionalitäten zu aktivieren

Der zu den *activation events* gehörende *script*-Handler dient zur Einbindung von AJAX-Skripten, die ein kontinuierliches Aktualisieren von Feldern bereits auf abstrakter Ebene unterstützen. Damit ist es möglich, einen Teil einer Präsentation neu zu erstellen, ohne die gesamte Präsentation neu zu laden.

abstrakte Beschreibung von UIs

Die Elemente zur abstrakten Beschreibung einer Nutzerschnittstelle sind in MARIA XML entsprechend der in Abbildung 6 dargestellten Struktur hierarchisch angeordnet:

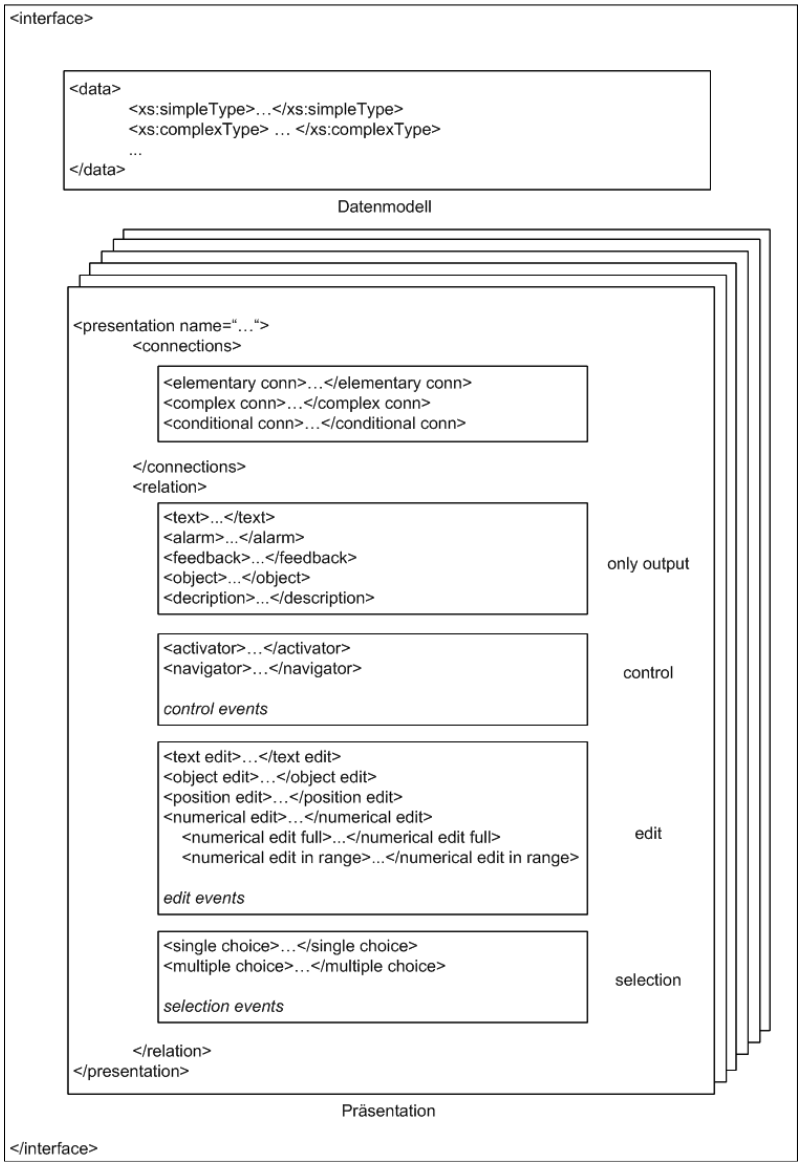


Abbildung 6: Aufbau von MARIA XML

Ein abstrakt beschriebenes *interface* besteht aus einem Datenmodell und einer oder mehreren Präsentationen. Jeder dieser mit *presentation* bezeichneten Präsentationsbereiche ist anhand des Wertes seines *name*-Attributes identifizierbar. Die einzelnen Präsentationen beinhalten jeweils das *connections*-Element, mit welchem die Verbindung zur nächsten aktiven Präsentation hergestellt wird. Es gibt drei verschiedene Möglichkeiten eine solche Verbindung zu definieren. Wird das *elementary-conn*-Element eingesetzt, kann der Wechsel zur nächsten Präsentation ohne Überprüfung einer Bedingung durchgeführt werden. Beim Einsatz des *complex-conn*-Elementes wird vor dem Wechsel überprüft, ob ein boolescher Operator mit dem korrekten Wert belegt ist. Bei Verwendung des *conditional-conn*-Elementes können beliebige Bedingungen spezifiziert werden, die vor der Aktivierung der nachfolgenden Präsentation überprüft werden. Nach der abstrakten Definition der Verbindung wird mit Hilfe des *relation*-Elementes eine Interaktorkomposition

angelegt. Die Interaktoren sind in Gruppen zusammengefasst. Die *only-output*-Gruppe dient der abstrakten Beschreibung von Interaktoren zur Anzeige von Inhalten. Die Elemente der *control*-Gruppe können zum Aktivieren von Funktionalitäten und zur Navigation innerhalb der Anwendung verwendet werden. Zum Editieren von Informationen sind in der *edit*-Gruppe entsprechende abstrakte Interaktoren vorhanden. Soll eine Interaktionsmöglichkeit zur Auswahl von Inhalten beschrieben werden, finden die in der *selection*-Gruppe vorhandenen Interaktoren Anwendung. Ausgelöst werden die Aktionen durch entsprechende Ereignisse, die den einzelnen Gruppen zugeordnet sind.

konkrete Beschreibung von UIs

Die konkrete Beschreibung von Nutzerschnittstellen erfolgt plattformabhängig, aber unabhängig von der für das FUI verwendeten Implementierungssprache. Ein CUI soll zwischen einer abstrakten Beschreibung und den für eine Plattform verfügbaren Implementierungssprachen vermitteln. Um Redundanzen zwischen den Modellen für AUIs und CUIs zu vermeiden, enthält die konkrete Beschreibung keine Elemente, die in der abstrakten Beschreibung vorkommen. Ein CUI kann beispielsweise zum Beschreiben von Gesten für ein Touchscreen-Interface, welches gleichzeitig mehrere Berührungen erkennen kann, erstellt werden. Dazu werden verschiedene Gesten wie beispielsweise das Rotieren oder Zoomen als vordefinierte Interaktionen, die aus einem Array von Berührungspunkten bestehen, abgelegt. Die abstrakten Beschreibungen von Interaktoren werden im CUI durch die Spezifikation von derartigen konkreten Ereignissen erweitert. In der vorliegenden Arbeit wird auf Verwendung der konkreten Abstraktionsebene verzichtet. Das Erkennen von Gesten ist nicht vorgesehen. Mit der Maus oder Tastatur sowie per Sprachbefehl durchgeführte Interaktionen lösen entsprechende Ereignisse aus, die bei der abstrakten Beschreibung der Interaktoren definiert werden.

UI-Generierung für Webservices

Webservices werden durch WSDL-Dateien [50] beschrieben. Mit WSDL wird zur Beschreibung der Dienste wie zum Beschreiben von UIs mit MARIA XML eine XML-basierte [7] Sprache verwendet. Die Dienste zur Bereitstellung von Funktionalitäten werden unabhängig von den Applikationen, in welche sie eingebunden werden, erstellt. Dadurch wird die Wiederverwendbarkeit der Dienste für verschiedene Anwendungen gewährleistet. Durch Annotation der Dienstbeschreibungen kann eine Verbindung zu den UI-Spezifikationen hergestellt und daraus eine entsprechende Nutzerschnittstelle für den Zugriff auf einen Webservice erstellt werden. Zum Beispiel kann die Definition eines Datentyps, der in einer WSDL-Datei angelegt wurde, annotiert werden, um Validierungsrestriktionen hinzuzufügen. Die manipulierte Typdefinition, welche in das AUI eingebunden wird, kann einfach auf den ursprünglichen Dienstdatentyp abgebildet werden. Die in der WSDL-Datei definierten Operationen werden Interaktoren zugewiesen. Soll beispielsweise durch eine Operation ein Objekt angelegt werden, wird diese mit einem Interaktor verknüpft, der es erlaubt, einen Namen und weitere Informationen einzugeben. Zur Erstellung einer auf Webservices basierenden Anwendung können diese kombiniert werden. Dies kann entweder dynamisch zur Laufzeit oder statisch zur Designzeit erfolgen. In der vorliegenden Arbeit soll zur Erzeugung eines Mashups unter Einbindung bestehender Applikationen der Runtime-Ansatz verwendet werden.

MARIA Tool

Das Werkzeug soll den Entwickler beim Design und der Erzeugung von UIs für Applikationen, die auf Webservices basieren, unterstützen. Mit Hilfe des Werkzeugs sollen mehrere Transformationen semi-automatisch durchgeführt werden, um die WSDL-Dateien von Webservices auf logische UIs abzubilden. Um verschiedene Ansätze zu unterstützen, soll das Werkzeug flexibel gestaltet werden. Meist handelt es sich um modellbasierte Lösungen, bei denen nach dem Top-Down-Prinzip zur Designzeit mit Hilfe des MARIA Tools die Dienstbeschreibungen der Webservices annotiert werden, um entsprechende Ergänzungen für die UI-Erzeugung vorzunehmen. Bei der Generierung von Applikationen nach dem Bottom-Up-Prinzip wird im ersten Schritt eine Aufgabenanalyse durchgeführt. Anschließend wird ein Modell entwickelt, um mit Hilfe von bestehenden Diensten die Aufgaben zu lösen. Werden beide Ansätze kombiniert, kann dieses Modell durch das Hinzufügen von Annotationen mit dem MARIA Tool verfeinert werden, um eine finale Nutzerschnittstelle zu erzeugen. Mit Hilfe des Werkzeugs sollen Transformationen, die es erlauben zwischen verschiedenen Abstraktionsebenen zu wechseln, möglich sein. Die dazu notwendigen Transformatoren werden nicht hart-codiert in das Werkzeug eingebunden, sondern extern definiert. Dadurch können sie ohne Änderungen an der Werkzeugimplementierung modifiziert werden. Es werden Modell-zu-Modell- und Modell-zu-FUI-Transformationen unterstützt. Das Werkzeug kann zur Generierung von FUIs unter Verwendung vorhandener Transformatoren genutzt werden. Ein solcher Transformator zur direkten Erzeugung von finalen Nutzerschnittstellen aus einer abstrakten Beschreibung wird im Rahmen der vorliegenden Arbeit entwickelt.

Migrierbare Nutzerschnittstellen

Durch den Runtime-Ansatz ist die Unterstützung von migrierbaren UIs möglich. Die interaktiven Applikationen können zwischen mehreren Geräten transferiert werden und behalten dabei ihren Zustand. Dadurch ist für den Anwender eine gefühlte unterbrechungsfreie Interaktion mit der Anwendung möglich. Der Zustand wird durch Skripte, die mit Javascript [6] erstellt werden, ermittelt und an den Server übertragen. Die für die Migration notwendige Architektur muss die Geräte, auf welche die Anwendung migriert werden kann, erkennen und identifizieren, um relevante Informationen für die Migration zur Anpassung der Anwendung an das Endgerät zu ermitteln. Dazu wird ein Migrationsserver eingesetzt. Im nächsten Schritt wird durch einen *reverse-engineering*-Prozess aus dem bestehenden FUI für einen Desktop-PC eine logische konkrete UI-Beschreibung (CUI) für die Desktopplattform durch den Migrationsserver generiert. Anschließend erfolgt im Rahmen des *semantic redesign*-Prozesses die Überarbeitung dieses CUI, um daraus ein neues CUI, welches an das Zielendgerät adaptiert wird, zu generieren. Durch das *semantic resdign* werden die Interaktoren an die Möglichkeiten des Endgerätes unter Beibehaltung der Semantik der eingesetzten Interaktoren angepasst. Dadurch können für mobile Endgeräte mit kleinem Display beispielsweise platzsparerende Interaktionsmöglichkeiten bereitgestellt werden. Die Möglichkeit zur dynamischen Anpassung des FUI an ein Endgerät soll im Rahmen dieser Arbeit ebenfalls umgesetzt werden. Der Verzicht auf den Zwischenschritt durch Verwendung eines CUI erfordert die Definition der Anpassungsoptionen direkt im AUI.

In diesem Kapitel wurden verschiedene Möglichkeiten zum Einsatz von MARIA XML vorgestellt. In der vorliegenden Arbeit wird MARIA XML zum Erstellen von AUI-Beschreibungen verwendet.

3.3 Verwandte Arbeiten

Nachdem im Kapitel 2.2 die Definition der Anforderungen mit Hilfe der Beispielszenarien erfolgt ist, wird im nächsten Kapitel der Stand der Technik und Forschung beschrieben. Es werden verwandte Arbeiten vorgestellt, bei denen modellbasierte abstrakte Beschreibungen von Webanwendungen zum Generieren entsprechender finaler Nutzerschnittstellen für diese Anwendungen verwendet werden. Neben der modellbasierten Beschreibung des Frontends ist in einigen dieser Veröffentlichungen die Modellierung von beispielsweise Workflows, der Geschäftslogik oder der Datenbankanbindung Gegenstand der Untersuchung. Diese für das Backend relevanten Aspekte werden in der vorliegenden Arbeit nicht behandelt. Die Veröffentlichungen sind für die Untersuchung relevant, da sie auch Ansätze zur Erzeugung von Nutzerschnittstellen aus abstrakten Beschreibungen beinhalten. Der Fokus wird bei der Analyse der Arbeiten auf die Ansätze zur Beschreibung des Frontends gelegt.

Zum Erstellen von Ajax-Anwendungen mit Hilfe eines modellgetriebenen Ansatzes [9] wird eine Modellierungssprache entwickelt, die es ermöglicht den Code für eine konkrete Webanwendung automatisch durch Transformation zu erzeugen. Die Entwickler streben dabei an, die Anwendung nur einmal zu definieren, um auf Basis verschiedener Webframeworks den Code für die eigentliche Webanwendung generieren zu lassen. Für die Modellierung wird das plattformunabhängige Open Source Werkzeug ANDROMDA [10], mit welchem auf Basis von UML Ajax-Benutzerschnittstellen erzeugt werden können, in modifizierter Form verwendet. Das dabei genutzte Metamodell besteht aus einer Baumstruktur, in der die einzelnen Benutzerschnittstellenkomponenten abgelegt werden. Dazu gehören Container zur Aufnahme der Inhalte, die Navigationselemente, die einzelnen Views, Interaktionskomponenten, wie beispielsweise Buttons und entsprechende Listener und Ereignisse. Um aus diesem Modell den Code zu erzeugen, werden Apache Velocity [11] Templates verwendet. Die Entwicklung der als Ajax-Cartridges bezeichneten, in ANDROMDA [10] verwendeten abstrakten Komponenten, ist zum Zeitpunkt der Veröffentlichung noch nicht abgeschlossen. Ebenso steht die Integration von Back-End-Komponenten, wie die Datenbankanbindung mit beispielsweise Hibernate [12], noch aus. Bei diesem Designtime-Ansatz wird zum Modellieren von Ajax-Anwendungen mit dem Werkzeug ANDROMDA eine proprietäre Technologie verwendet. Bei den für die Transformation eingesetzten Apache Velocity Templates wird ebenfalls nicht auf eine Standardtechnologie zurückgegriffen. Für die Ajax-basierten finalen Nutzerschnittstellen ist zudem lediglich die visuelle Modalität vorgesehen.

Bei einem anderen Ansatz zur modellgetriebenen Entwicklung dynamischer Webanwendungen [13] werden die grafischen Benutzerschnittstellen einer Rich Internet Application (RIA) aus einer abstrakten Beschreibung mit Hilfe von XSL-Transformationen [16] automatisch erzeugt. Die Modellierung erfolgt dabei nach den Richtlinien des Cameleon Frameworks [14], indem vier verschiedene Abstraktionslevel, angefangen bei der Aufgabendefinition, über die abstrakte und konkrete bis zur finalen Benutzerschnittstelle, bei den Transformationen durchlaufen werden. Für die Beschreibung wird die User Interface eXtensible Markup Language (UsiXML) [15] verwendet. Der Entwicklungszyklus durchläuft gemäß den genannten Abstraktionsebenen vier Schritte. Zuerst wird ein von der späteren grafischen Benutzerschnittstelle unabhängiges Aufgabenmodell entwickelt, und anschließend wird die Benutzerschnittstelle ohne Beachtung der später verwendeten Interaktionsmodalität erzeugt. Aus diesem abstrakten User Interface (AUI) werden

nachfolgend die sogenannten Concrete Individual Objects (CIOs) mit XSLT [16] erstellt. Das sind native Widgetsets von entsprechenden Toolkits wie Adobe Flex [37] oder Microsoft XAML [38]. Im vierten und letzten Schritt werden die einzelnen Komponenten auf Plattformen wie Microsoft .NET [17] kompiliert, interpretiert und ausgeführt. Als Vorhaben wird in der Arbeit die Erzeugung eines kompletten mit XSL erstellten Stylesheets, welches alle in XAML definierten UI-Komponenten erzeugen kann, genannt. Durch die Verwendung von vier Abstraktionsebenen ist bei diesem Ansatz keine direkte ad-hoc Transformation aus der abstrakten Beschreibung in eine finales Nutzerschnittstelle vorgesehen. Zudem wird für die Interpretation dieser mit Microsoft .NET beispielsweise eine proprietäre Plattform vorausgesetzt und keine Skriptsprache verwendet, die ohne zusätzliche Erweiterungen in aktuellen Browsern ausgeführt werden kann.

Bei der Entwicklung vieler Webanwendungen werden für verschiedene Aufgaben wie Eingabebehandlung, UI-Design und Navigation separate Sprachen verwendet. Dabei können Redundanzen sowie Inkonsistenzen entstehen, und es sind verschiedene Werkzeuge zum Kompilieren notwendig. Abhilfe schafft die Sprache WebDSL (Web Domain-Specific Language) [18]. Diese Sprache zur Implementierung dynamischer Webanwendungen unter Verwendung des *rich data model* liefert eine integrierte Webentwicklungsplattform für heterogene Umgebungen. Neben Erweiterungen für Abstraktionen auf höhere Ebene zur Modellierung von beispielsweise des Workflows und der Zugriffskontrolle sind grundsätzlich Konstrukte zum Definieren von Entitäten, Seiten und der Geschäftslogik vorhanden. Dieser Teil der Sprache wird als *core language* bezeichnet. Zur Workflow-Abstraktion wird die objektorientierte Modellierungssprache WebWorkFlow [19] verwendet. Die Codeerzeugung erfolgt durch Modelltransformationen unter Verwendung von Stratego/XT [20], einem System, das Modell-zu-Code, Modell-zu-Modell und Code-zu-Code Transformationen beinhaltet.

Zur Entwicklung von RIAs können bereits vorhandene Webentwicklungsmethoden verwendet werden [21]. Für die automatisierte Codeerzeugung wird die Webmodellierungssprache WebML [22] zusammen mit dem Rich User eXperience Model (RUX-Model) [23] verwendet. Dabei sind ebenfalls vier Phasen für die Entwicklung von RIAs notwendig. Im ersten Schritt erfolgt die Datenmodellierung. Dabei gilt es zwischen client- und serverseitiger sowie persistenter und temporärer Datenspeicherung zu unterscheiden. Um die Konsistenz der Daten sicherzustellen, werden Trigger verwendet. Im nächsten Schritt wird die Geschäftslogik mit dem Ziel, eine Single-Page-Applikation zu erstellen, modelliert. Dadurch soll die Anwendung innerhalb einer Internetseite ausführbar sein. Auch hier muss zwischen der Verarbeitung auf dem Server oder Client unterschieden werden. Allgemein erfolgt die Ausführung an der Stelle, an der auch die Daten abgelegt sind. Anschließend muss die eigentliche Präsentation umgesetzt werden. Da die Anwendungen für unterschiedliche Gerätetypen renderbar sein sollen, werden verschiedene Sets von Präsentationselementen definiert. Im letzten Schritt gilt es zwischen synchroner und asynchroner Kommunikation zu unterscheiden und insbesondere den Ladevorgang über die Laufzeit der Anwendung zu verteilen, um für die nach dem Single-Page-Paradigma erzeugte Anwendung eine zu lange Wartezeit beim initialen Laden zu vermeiden. Nach diesen allgemein gehaltenen Betrachtungen wird die Umsetzung mit WebML und dem RUX-Model durch die Autoren vorgestellt. Die WebML Datenmodellierung erfolgt mittels Entity Relation- (ER-) und UML-Diagrammen. Für die Modellierung der Geschäftslogik wird das Hypertextmodell von WebML durch die explizite Spezifikation der Verteilung der Verarbeitung zwischen Server

und Client erweitert. Das RUX-Model findet bei der Präsentationsmodellierung Verwendung. Dabei erfolgt die UI-Spezifikation in drei Ebenen: abstrakte, konkrete und finale Nutzerschnittstelle. Auf der ersten Ebene werden die einzelnen Komponenten und deren Beziehungen festgelegt. Danach erfolgt die Definition des Verhaltens, beispielsweise die Festlegung von Synchronisationspunkten und letztendlich die Codeerzeugung mit XSLT [16] für verschiedene Plattformen wie Adobe Flex [37] und Microsoft XAML [38]. Die Arbeit schließt mit der Feststellung, dass bisher mit der Kombination aus RUX-Model und WebML noch nicht alle vier Phasen der RIA-Entwicklung komplett abgedeckt werden. Durch die Verwendung von drei Abstraktionsebenen ist bei diesem Ansatz keine direkte ad-hoc Transformation zur Erzeugung der finalen Nutzerschnittstelle aus einem AUI vorgesehen. Der Fokus liegt auf der Generierung von Single-Page-Applikationen. Die Erzeugung von RIAs, welche auf mehrere Seiten aufgeteilt werden, wird nicht betrachtet. Mit Adobe Flex [37] und Microsoft XAML [38] werden proprietäre Technologien für das FUI eingesetzt. Die Erzeugung multimodaler RIAs wird bei diesem Designtime-Ansatz nicht untersucht.

Zur Erzeugung von RIAs kann die oben erwähnte RUX-Methode statt mit WebML ebenso mit UML-based Web Engineering (UWE) [24] kombiniert werden [25]. Das Ziel ist dabei eine modellgetriebene Lösung zur RIA-Entwicklung, wobei UWE für die Daten-, Navigations- und Geschäftslogikmodellierung und die RUX-Methode zum Modellieren der Benutzerschnittstelle verwendet wird. Zur Erstellung kompletter RIAs ist es notwendig, beide Lösungen miteinander zu verknüpfen, indem entsprechende Regeln für die Transformationen zwischen deren Metamodellen erstellt werden. Die Neuerung in dieser Arbeit ist dabei, die Erzeugungsregeln der RUX-Methode für eine automatische Verbindung zu UWE zu erweitern. Zudem werden Geschäftsprozesse bei der Erzeugung von RIAs mit einbezogen. Mit UWE ist es möglich, Modelltransformationen mit UML für verschiedene spezifische Plattformen zu definieren. Zur eigentlichen Transformation werden die Sprachen Query View Transformation (QVT) [34] und ATLAS Transformation Language (ATL) [35] verwendet. Die abstrakte Beschreibung der Benutzerschnittstelle besteht bei der RUX-Methode aus sogenannten *Connectors* für die Beziehung zwischen den UI-Komponenten, den Daten und den mit *Media* bezeichneten Elementen, die geräteunabhängige atomare UI-Informationen repräsentieren sowie den *Views*, welche die in dem UI angezeigten Informationen gruppieren. Für die systematische Spezifikation von geräteunabhängigen und interaktiven Webbenutzerschnittstellen mit der RUX-Methode werden drei verschiedene Ebenen von AUI, über CUI, bis FUI durchlaufen. Dementsprechend gibt es drei Transformationsphasen. In der ersten werden die Daten gesammelt sowie adaptiert und die Geschäftslogik spezifiziert. Das Ergebnis wird mit *Connection Rules* (CR) bezeichnet. Diese Regeln stellen die Verbindung zwischen der RUX-Methode und UWE her, indem sie die von UWE repräsentierten Informationen filtern und die Informationen extrahieren, die zur Erzeugung des AUI notwendig sind. Im nächsten, *Transformation Rules 1* (TR1) genannten Schritt wird das AUI für ein oder mehrere bestimmte Geräte adaptiert und die Möglichkeit für den Zugriff auf die Geschäftslogik geschaffen. Der MDA-Zyklus wird letztendlich mit der Codeerzeugung mittels der *Transformation Rules 2* (TR2) abgeschlossen. Um das Look&Feel einer Web2.0-Applikation zu erhalten, muss das RUX-Modell allerdings noch erweitert werden. Features wie Animationen, asynchrone Client-/Serverkommunikation und clientseitige Berechnungen fehlen derzeit. Zudem wird Umsetzung der plattformspezifischen Transformation und damit die Erzeugung eines FUI nicht betrachtet. Der Fokus der vorliegenden Arbeit liegt hingegen auf diesem, in der Veröffentlichung mit *Transformation*

Rules 2 bezeichneten, Schritt. Außerdem ist bei der Verwendung der RUX-Methode zur Präsentationsmodellierung keine Multimodalität vorgesehen.

Ein modellbasierter Ansatz wird auch eingesetzt, wenn vorhandene Webanwendungen, die sich über mehrere Seiten erstrecken, entsprechend dem Single-Page-Paradigma auf eine Seite reduziert werden sollen [26]. Bei mehreren Seiten muss nach jeder Anfrage an den Server die gesamte Nutzerschnittstelle neu geladen werden, um erfolgte Änderungen darzustellen. Dank Ajax besteht dagegen eine Single-Page-Anwendung aus einer Komposition von einzelnen Komponenten, die unabhängig voneinander aktualisiert und ersetzt werden können. Um eine solche Migration mehrerer Seiten auf eine durchführen zu können, wird eine schemabasierte Clusteringtechnik verwendet, um das Navigationsmodell einer vorhandenen Webanwendung zu extrahieren und durch weitere Analyse der einzelnen Cluster diese als UI-Komponenten zu identifizieren. Die Autoren der Arbeit sind zum Zeitpunkt der Veröffentlichung damit beschäftigt, ein abstraktes Single-Page-UI-Metamodell für Ajax-Anwendungen zu entwickeln, welches abstrakte UI-Komponenten, die Spezifikation von Navigationspfaden und die Umsetzung von dynamischen Änderungen der Nutzerschnittstelle enthalten soll. Dieses Metamodell beinhaltet präzisere Informationen für die Transformation in eine plattformspezifische Sprache eines beliebigen Ajax-Frameworks als der allgemeine HTML Syntaxbaum, der durch die Extraktion der bestehenden Webanwendung erstellt wird. Nachdem im weiteren Verlauf der Arbeit die Analyse, strukturelle Klassifikation und Identifizierung der UI-Komponenten detailliert beschrieben werden, stellen die Autoren eine erste Werkzeugimplementierung vor. Der mit Retjax (Reverse Engineer To Ajax) bezeichnete, Java-basierte Prototyp zur Identifizierung der Cluster und Festlegung passender UI-Komponenten wurde an einer Fallstudie getestet und arbeitete korrekt. Die Autoren gehen zum Zeitpunkt der Veröffentlichung davon aus, dass das Tool auch allgemein für beliebige Webanwendungen einsetzbar ist. Der Fokus liegt auf der Erzeugung von Single-Page-Applikationen. Die UI-Komponenten vorhandener Web-Applikationen werden analysiert, identifiziert und klassifiziert, um daraus ein Modell abzuleiten. In der vorliegenden Arbeit ist ein solches Modell zur Generierung Ajax-basierter Applikationen durch MARIA XML bereits gegeben.

Zur Verbesserung der Benutzbarkeit vorhandener Webanwendungen wird eine modellbasierte Lösung für die systematische Transformation solcher in RIAs verwendet [27]. Dies wird durch kleine Designänderungen, sogenannte RIA refactorings, umgesetzt. Das Konzept für die Formalisierung der Migration nennt sich Web Model Refactoring (WMR) und bezieht sich auf die Änderung der Navigation und des Interfacemodells. Dadurch soll die externe Qualität einer Webanwendung bei unverändertem Anwendungsverhalten verbessert werden. Ein Beispiel dafür ist die Möglichkeit im Warenkorb eines Webshops befindliche Artikelbezeichnungen als Link umzusetzen, um Details zu dem gewählten Artikel aufrufen zu können. Die notwendigen RIA refactorings werden dazu formal mit Abstract Data Views (ADVs) spezifiziert. Nachdem mögliche Änderungen an einer Webseite identifiziert und in einen Refactoringkatalog aufgenommen wurden, werden den zugehörigen Refactoringmechanismen die eingebundenen ADVs zugewiesen. Für jedes Anliegen kann dazu ein Set von ADVs definiert werden. Mittels XSLT [16] können die ADVs anschließend automatisiert oder manuell auf beliebige lauffähige Interfaceelemente abgebildet werden. Die ADVs legen fest, wie die Interaktion erfolgen soll und welche Effekte nach dem Auftreten eines Ereignisses eintreten. Es besteht die Möglichkeit vorhandene Interfaceobjekte beizubehalten und nur deren Verhalten zu verändern, zum Beispiel Teile

einer Seite scrollbar zu machen, vorhandene Objekte in andere zu transformieren und neue hinzuzufügen. Dazu werden sogenannte ADV-Templates eingesetzt. Diese können zum Beispiel genutzt werden, um eine einfache vertikale Navigation in eine kreisförmige zu transformieren. Eingesetzt wird dies beispielsweise bei der Präsentation von Produktlisten beim Onlineversandhandel amazon [28]. Die Autoren nennen als Vorteile die Wiederverwendbarkeit einmal definierter ADV-Templates und das Umgehen der Neuimplementierung einer Legacy-Anwendung durch die Durchführung inkrementeller refactorings auf Modellebene. Als ausstehende Aufgaben werden die Erweiterung und Organisation des Refactoringkataloges, die Entwicklung eines Tools für ADV-Repräsentationen und Komposition derselben und die Adaption des gesamten Prozesses an eine Standardnotation wie UML, aufgeführt. Durch die Verwendung der ADVs wird zur Erzeugung der abstrakten Nutzerschnittstellenbeschreibungen eine proprietäre Technik eingesetzt. Die Notwendigkeit der Analyse der vorhandenen Interaktoren und Durchführung inkrementeller Refactorings erfordert einen Designtime-Ansatz. Eine Konfigurationsmöglichkeit zum dynamischen Hinzufügen neuer beziehungsweise alternativer Interaktoren zur Laufzeit ist nicht vorgesehen.

Zum modellgetriebenen Erstellen von neuen RIAs [29] wird die weiter oben genannte Sprache WebML [22] erweitert, um die Lücke zwischen vorhandenen Webentwicklungsmethoden und dem RIA-Paradigma zu schließen. Eine Zielstellung bei der Modellierung ist hierbei, die letztendlich aus dem Modell entstehende Anwendung für verschiedene Nutzertypen und Geräte erzeugen zu können. Da bei RIAs die Speicherung der Daten auf Client- und Serverseite erfolgen kann, soll das konzeptuelle Datenmodell vom physischen Ort des Inhalts und von der Persistenz abstrahieren. Im Folgenden werden in der Arbeit einige Anwendungsfälle diskutiert und daraus Richtlinien zur Datenspeicherung abgeleitet. Im Rahmen der Hypertextmodellierung wird anschließend die generelle Struktur des Frontends spezifiziert, wobei die einzelnen Views mit Nutzergruppen verbunden und zu homogenen Seiten aggregiert werden. Die Berechnungen der Seiteninhalte werden zwischen Server und Client aufgeteilt. Dabei gilt es zu beachten, dass bei diesem Ansatz alle Berechnungen, die vom Server ausgeführt werden, auf den Daten und Operationen des Servers basieren, da aufgrund der asymmetrischen Natur des Webs davon ausgegangen wird, dass der Client Anfragen an den Server stellen kann und nicht andersherum. Die Geschäftslogik wird mit WebML-Operationen modelliert. Im RIA Kontext wird dabei zwischen Server-, Client- und gemischten Operationen unterschieden. Bei letzteren wird eine *operation chain* gebildet, um clientseitige und serverseitige Operationen zu mischen. Nachdem die genannten Konzepte vorgestellt wurden, schildern die Autoren eine mögliche Umsetzung derselben mit dem Werkzeug WebRatio [30] für den serverseitigen Code und der objektorientierten, tagbasierten Sprache Laszlo LZX [36], die XML und Javascript verwendet, um die Präsentationsschicht zu erzeugen. Dabei erwies sich der große Hauptspeicherbedarf für die XML-Repräsentation in Form eines DOM-Baums als problematisch. Es wird zum Abschluss ein umfangreiches Forschungsprogramm vorgestellt, welches notwendig ist, um RIAs, ausgehend von einem modellgetriebenen Ansatz, komplett automatisiert erzeugen zu lassen. Die erzeugten Applikationen sind zwar für verschiedene Nutzertypen und Geräte einsetzbar, jedoch wird der neben der visuellen Nutzerschnittstelle keine weitere Modalität unterstützt. Während im Rahmen der Hypertextmodellierung die Definition der generellen Struktur des Frontends allgemein erfolgt, wird zur Erzeugung der Präsentationsschicht die proprietäre Sprache Laszlo LZX [36] verwendet. Die Konfiguration der erzeugten Anwendung zur Laufzeit ist bei diesem Designtime-Ansatz nicht vorgesehen.

Beim Erstellen von abstrakten Beschreibungen und der Durchführung von Transformationen kann der Entwickler durch den Einsatz eines Werkzeugs unterstützt werden. TERESA (Transformation Environment for interRactivE Systems representAtions) ist ein solches Werkzeug, mit dem Benutzerschnittstellen für verschiedene Geräte aus einer logischen Beschreibung erzeugt werden können [31]. Mit TERESA erhält der Entwickler eine semi-automatische Umgebung, die es erlaubt Designs auf unterschiedlichen Abstraktionsniveaus zu analysieren und UIs für verschiedene Plattformen zu erzeugen. Es werden unterschiedliche Automatisierungsstufen von automatisch bis interaktiv unterstützt. Die logische Beschreibung dient der Transformation in weborientierte konkrete Präsentationen. Allerdings soll das Werkzeug auch für weitere Plattformen wie Java- und Microsoftumgebungen erweiterbar sein. Basierend auf XHTML sind Benutzerschnittstellen für den Desktop und für Mobiltelefone sowie VoiceXML [32] UIs angedacht. Der Ablauf bei der Arbeit mit dem Werkzeug unterteilt sich in vier Schritte. Begonnen wird mit der Festlegung der von der Anwendung durchzuführenden Aufgaben und der Spezifikation von Bedingungen für Transitionen zwischen den Aufgaben. Anschließend wird aus der aufgabenbasierten Spezifikation des Systems eine interaktorbasierte Beschreibung erstellt. Danach erfolgt die Transformation des in XML erstellten AUIs in ein CUI für eine spezifische Plattform. Bei diesem Schritt können die Interaktoren durch die Veränderung entsprechender Parameter angepasst werden. Abschließend erzeugt das Werkzeug automatisch die FUI für die Zielplattform. Unterschiede bei der Definition für verschiedene grafische Benutzerschnittstellen, zum Beispiel Plattformen auf Mobiltelefonen und dem Desktop, sind nur beim letzten Schritt notwendig. Es handelt sich um ein proprietäres Werkzeug. Eine Unterstützung des Imports von Cameleon [14] XML-Repräsentationen und der Austausch mit anderen Werkzeugen sind geplant. Durch die Unterteilung des Ablaufs der Transformation in vier Schritte ist beim Einsatz dieses Werkzeugs keine direkte Erzeugung eines FUI aus einer abstrakten Beschreibung möglich. Auf Basis des AUI wird zuerst ein plattformspezifisches CUI erzeugt, um daraus das FUI automatisch zu generieren.

Die Modellierung von Dialogen kann auf Basis von abstrakten Interaktoren und UML Statecharts erfolgen [33]. Die Interaktoren vermitteln die Informationen zwischen dem Nutzer und dem interaktiven System über sogenannte Gates. Netzwerke von Interaktoren werden ebenfalls über Gates miteinander verbunden. Ein Gate ist als eine Funktion zu verstehen, die ein von ihr berechnetes Ergebnis zum nächsten Gate weitergibt. Durch dieses Vorgehen können generische Interaktoren, die durch Parameter veränderbar sind, wiederverwendet werden. Zudem abstrahiert die Interaktorbeschreibung von einer konkreten Modalitätsbeschreibung, um sich nicht von vornherein an eine bestimmte Eingabemethode zu binden. Die Modellierung von Interfaces zur direkten Manipulation, für die Interaktion per Drag&Drop zum Beispiel, wird ebenfalls betrachtet. Dies erfolgt hier durch triggern von sich potentiell überlappenden Objekten. Dazu werden die Zustände einzelner Gesten verwendet, um eine abstrakte Beschreibung eines entsprechenden Interaktors zu erstellen. Obwohl die modellbasierte Beschreibung von der später verwendeten Modalität abstrahiert, wird nur die Erzeugung einer visuellen Nutzerschnittstelle beschrieben. Zudem erfolgt die Identifizierung einer Interaktion wie Drag&Drop durch Analyse entsprechender Gestern und nicht durch die Zuweisung von Ereignissen zu Interaktoren.

Die vorgestellten Arbeiten zeigen, dass bereits verschiedene Ansätze zur modellbasierten abstrakten Beschreibung von Applikationen existieren. Bei den beschriebenen Ansätzen wurden jedoch keine ad-hoc Transformationen zur direkten Generierung von multimodalen Ajax-Applikationen aus AUI-Beschreibungen eingesetzt. In Tabelle 3 werden die vorgestellten Arbeiten zusammengefasst und insbesondere die verwendeten proprietären Sprachen, Komponenten und Frameworks genannt. Dies soll aufzeigen, dass diese Ansätze nicht ausschließlich standardisierte Techniken und Sprachen wie XML verwenden. Wenn den für die vorliegende Arbeit genannten Anforderungen, wie beispielsweise der Multimodalität bei den vorgestellten Veröffentlichungen Beachtung geschenkt wurde, dann ist dies in der Auflistung mit erwähnt.

[9]	modellgetriebener Ansatz zur Entwicklung von Ajax-Anwendungen • Modellierung mit speziellen Framework: ANDROMDA [10] • Entwicklung der notwendigen abstrakten Komponenten nicht vollständig • Codeerzeugung mit Apache Velocity Templates [11]
[13]	modellgetriebener Ansatz zur Entwicklung dynamischer Webanwendungen • Modellierung nach Cameleon Framework [14] • abstrakte Beschreibung der UI mit UsiXML [15] • Vorhaben: Erzeugung eines kompletten XSLT-Sheets zur Transformation nach Microsoft XAML [38]
[18]	Vorstellung von WebDSL • Modellierung von Entitäten, Seiten, Geschäftlogik • Erweiterungen zur Umsetzung von Workflows und Zugriffkontrolle • Codeerzeugung mit Stratego/XT [20] • keine Umsetzung eines Prototypen
[21]	Verwendung von WebML und RUX für Entwicklung von RIAs • Modellierung der Daten und Geschäftslogik für eine Single-Page-Applikation • Verwendung von Entity Relation- und UML-Diagrammen für Datenmodellierung • Verwendung des Hypertextmodells von WebML für Geschäftslogik • Präsentation für verschiedene Gerätetypen renderbar • mit WebML und RUX noch nicht alle Phasen der RIA-Entwicklung abgedeckt
[25]	Verwendung von UWE und RUX für Entwicklung von RIAs • Daten-, Navigations-, Geschäftslogikmodellierung mit UWE • Transformationen mit ATL [35] und QVT [34] umgesetzt • Spezifikation geräteunabhängiger Webbenutzerschnittstellen • Umsetzung des Look&Feel einer Web2.0-Applikation nur geplant
[26]	Transformation einer Webanwendung zu einer Single-Page-Applikation • Beschränkung auf ein Single-Page-UI-Metamodell • Transformation in beliebige Ajax-Frameworks möglich • Prototypwerkzeug Retjax zur Analyse und Clustererstellung aus vorhandenen Webseiten • Fokus liegt nicht auf der Neuentwicklung von Webanwendungen
[27]	modellbasierte Lösung für die Transformation einer bestehenden Webanwendung in eine RIA • Konzept: Web Model Refactoring zur Änderung der Navigation und des Interfacemodells

	• abstrakte Definition der RIA refactorings in Abstract Data Views (ADVs) • ADVs werden mit XSLT auf lauffähige Interfaceelemente abgebildet • Adaption des Gesamtprozesses an eine Standardnotation wie UML nur vorgesehen
[29]	modellgetriebene Erstellung von RIAs • Erweiterung von WebML entsprechend dem RIA-Paradigma • Anwendung für verschiedene Nutzer- und Gerätetypen aus dem Modell erzeugbar • asynchrone Kommunikation zwischen Server und Client • für serverseitigen Code wird proprietäres Tool WebRatio verwendet • für clientseitigen Code wird objektorientierte, tagbasierte Sprache Lazlo LZX [36] verwendet
[31]	TERESA • Werkzeug zur Erstellung von Benutzerschnittstellen aus logischer Beschreibung für verschiedene Geräte und Plattformen • Transformation in weborientierte konkrete Präsentationen • AUI wird in XML definiert • proprietäres Tool, Austausch mit anderen Tools ist geplant
[33]	Modellierung von Dialogen auf Basis von abstrakten Interaktoren • Interaktorbeschreibung abstrahiert von konkreter Modalität • Betrachtung von Interfaces mit direkten Manipulationsmöglichkeiten wie Drag&Drop

Tabelle 3: Zusammenfassung der verwandten Arbeiten

3.4 Abgrenzung

Die beschriebenen Arbeiten stellen verschiedene Ansätze der modellgetriebenen und automatisierten Erzeugung moderner Webanwendungen vor. Das Ergebnis der Arbeiten ist meist ein Prototyp, der nur einen geringen Teil der gewünschten Möglichkeiten der vorgestellten Konzepte umsetzt. Ein Bezug zu real existierenden Szenarien ist dabei nicht zu erkennen. Vielmehr werden nur einige Versuche unternommen, aus der abstrakten Beschreibung einzelne Komponenten für eine Nutzerschnittstelle umzusetzen und daraus dann auf die Tauglichkeit der Entwicklung für größere Anwendungsbereiche geschlossen. Einige der Arbeiten verwenden dafür proprietäre Tools wie beispielsweise TERESA und Methoden, die später abgeändert werden sollen, um eine standardkonforme Definition zu ermöglichen. In der vorliegenden Arbeit wird mit MARIA XML eine durch ein XML Schema [39] definierte Sprache verwendet, die mit jedem beliebigen Editor, der XML unterstützt, erstellt und bearbeitet werden kann. Es werden keine proprietären Konzepte, wie die in [27] entwickelten ADVs, zur abstrakten Beschreibung des UI verwendet. Die Definition eigener Datentypen ist gemäß XML Schema [39] auf standardkonforme Weise möglich. In den vorgestellten Arbeiten wird zumeist von bis zu vier Schritten ausgegangen, die durchlaufen werden müssen, um aus einer abstrakten Beschreibung automatisiert eine fertige Benutzerschnittstelle zu erstellen. Diese vier Ebenen werden auch durch das Cameleon Framework, welches in [13] verwendet wird, vorgegeben. Beginnend mit einer allgemeinen Aufgabenanalyse werden daraus die weiteren Schritte abgeleitet. Die Sprache MARIA XML basiert ebenfalls auf dieser Methodologie. Da die modellbasierte Beschreibung von Ajax-Szenarien im Fokus steht und diese auf wiederverwendbaren Komponenten zur

Autovervollständigung oder Validierung von Eingaben beispielsweise sowie Interaktionsmöglichkeiten wie Drag&Drop basieren, wird die abstrakte Beschreibung nicht auf die zu erledigenden Aufgaben zugeschnitten, sondern direkt auf die für verschiedene Aufgaben einsetzbaren Technologien. Dadurch ist es zum Beispiel möglich, die Validierungsfunktionalität in dem AUI durch ein Attribut einzubinden, anstatt wiederholt den gesamten dafür notwendigen Code abzulegen. In der vorliegenden Arbeit wird auf den Zwischenschritt zur Generierung eines CUI verzichtet. Stattdessen wird direkt aus der abstrakten Beschreibung eine finale Nutzerschnittstelle (FUI) durch ad-hoc Transformation erzeugt. In [27] wird ein Ablauf vorgeschlagen, um aus bestehenden Webanwendungen durch Detailänderungen einfacher zu bedienende RIAs zu erzeugen. Betrachtet man den Aufwand allein für die komplette Analyse der bestehenden Anwendung, erscheint eine Neuimplementierung, bei der einmalig die notwendigen Ajax-Funktionalitäten abstrakt definiert und danach immer wieder verwendet werden können, als die effizientere Lösung. Durch die oben beschriebenen Szenarien wird eine Vielzahl typischer Nutzungsmuster von Ajax-Anwendungen abgedeckt. Ein weiterer Bestandteil der Arbeit ist es, die Interaktion in verschiedenen Modalitäten zu realisieren. An einem Szenario wird gezeigt, dass der Nutzer die Anwendung sowohl akustisch als auch visuell bedienen kann. Diese Möglichkeit ist bei den vorgestellten Veröffentlichungen nicht gegeben. Weiterhin soll bei der Entwicklung des AUI mit MARIA XML den später eingesetzten Plattformen und Geräten keine Beachtung geschenkt werden müssen. Die abstrakte Beschreibung erfolgt plattformunabhängig. Lediglich die Standardsprachen HTML, CSS und Javascript müssen vom Browser unterstützt werden. Damit ist die Transformation nicht an ein externes Toolkit, wie das in [9] verwendete Apache Velocity gebunden. Die finale Nutzerschnittstelle wird aus der AUI-Beschreibung unabhängig von speziellen Toolkits ad-hoc erzeugt. Zudem ist durch die Durchführung der ad-hoc Transformationen beim Aufruf einer Seite die Konfigurierbarkeit der Anwendung durch Anpassung einzelner Parameter zur Laufzeit gegeben. Mit diesem Runtime-Ansatz können multimodale Szenarien, bei denen ein Wechsel der Modalität zur Laufzeit möglich ist, erzeugt werden. Eingebundene Widgets und externe Applikationen können zur Laufzeit ausgetauscht werden. Bei den vorgestellten Veröffentlichungen werden keine Untersuchungen zur Effizienz der entwickelten Ansätze vorgenommen. In [29] wird lediglich erwähnt, dass sich der große Hauptspeicherbedarf für die XML-Repräsentation in Form eines DOM-Baums bei der Verwendung der tagbasierten Sprache Laszlo LZX [36] zur Generierung der Präsentationsschicht als problematisch erwiesen hat. Eine Zielstellung der vorliegenden Arbeit ist die effiziente Generierung des FUI. Dies wird durch die Erzeugung der finalen Nutzerschnittstelle ohne weitere Zwischenschritte direkt aus der abstrakten Beschreibung erreicht. Zudem soll der zur Generierung eines Szenarios eingesetzte Transformator modular aufgebaut sein und somit möglichst keine Transformationsregeln und Skripte enthalten, die für das zu erzeugende Szenario nicht relevant sind.

In Tabelle 4 werden zu jeder der genannten Anforderungen entsprechende Differenzierungskriterien aufgelistet, um die Abgrenzung der vorliegenden von den verwandten Arbeiten zu verdeutlichen. Die aufgestellten Differenzierungskriterien sind aus den Beschreibungen der verwandten Arbeiten entnommen und werden durch solche ergänzt, die Punkte aufzeigen, denen in den Arbeiten keine Beachtung geschenkt wird.

Anforderung	Differenzierung
Multimodalität	• Beschränkung auf eine Modalität • teilweise zweite Modalität vorgesehen, aber nicht umgesetzt • keine Abstraktion der Interaktorbeschreibung von der Modalität
asynchroner Datentransfer	• nur Datenmodellierung wird beschrieben • keine Angaben zu verwendeten Übertragungsmechanismen
Einsatz von Sessions	• keine ad-hoc Transformation direkt vom AUI in konkrete Nutzerschnittstelle • vier Abstraktionslevel bei Cameleon Framework • Beschränkung auf Single-Page-Applikationen • Konfigurierbarkeit der Anwendung zur Laufzeit nicht vorgesehen
Verwendung von Standards	• Verwendung proprietärer Werkzeuge zur AUI-Entwicklung • Einsatz proprietärerer Sprachen wie Query View Transformation (QVT) [34] oder ATLAS Transformation Language (ATL) [35] • Nutzung proprietärer Sprachen wie Laszlo LZX [36] um Präsentationsschicht zu erzeugen
Effizienz	• keine Untersuchungen zur Effizienz der vorgestellten Ansätze
Einsatz von Widgets	• Verwendung nativer Widgetsets wie Adobe Flex [37] Microsoft XAML [38] • kein Austausch von Widgets zur Laufzeit vorgesehen
Web2.0-Nutzerinteraktionen	• Umsetzung von Web2.0 Look&Feel lediglich geplant
Erzeugen von Mashups	• Kombination von Web2.0-Applikationen nicht vorgesehen
Abstraktion von Zielplattform	• Verwendung vorgefertigter Templates für finale Nutzerschnittstelle, zum Beispiel Apache Velocity [11]
Verwendung einer gegebenen Sprache zu abstrakten Beschreibung	• Entwicklung neuer Sprachen zur AUI-Beschreibung • Verwendung bestehender Webanwendungen als Ausgangspunkt für die Erzeugung von Modellen

Tabelle 4: Anforderungen und Differenzierungskriterien

4 Konzeption

4.1 Systemübersicht

Die nachfolgenden Abbildungen veranschaulichen die verwendeten Komponenten des konzipierten Systems, welches zur Durchführung der Transformation der durch MARIA XML vorgegebenen AUI-Beschreibung [Abbildung 8/ Komponente 6] mit Hilfe eines generischen Transformators, bestehend aus Generatortemplates [Abbildung 7/ Komponente 3], notwendig ist. Ziele der Transformation sind dabei einerseits eine visuelle Benutzerschnittstelle in Form einer Web2.0-Anwendung und andererseits eine weitere, bei der die Interaktion auf Sprachein- und ausgaben basiert [Abbildung 8/ Komponente 9]. Die detaillierte Beschreibung des Transformators und die Evaluierung der entstandenen Zieldokumente erfolgt in späteren Kapiteln. An dieser Stelle soll das zugrundeliegende System, bestehend aus einer serverseitigen Komponente zur ad-hoc Transformation, einem Webproxy und einem Webservice vorgestellt werden.

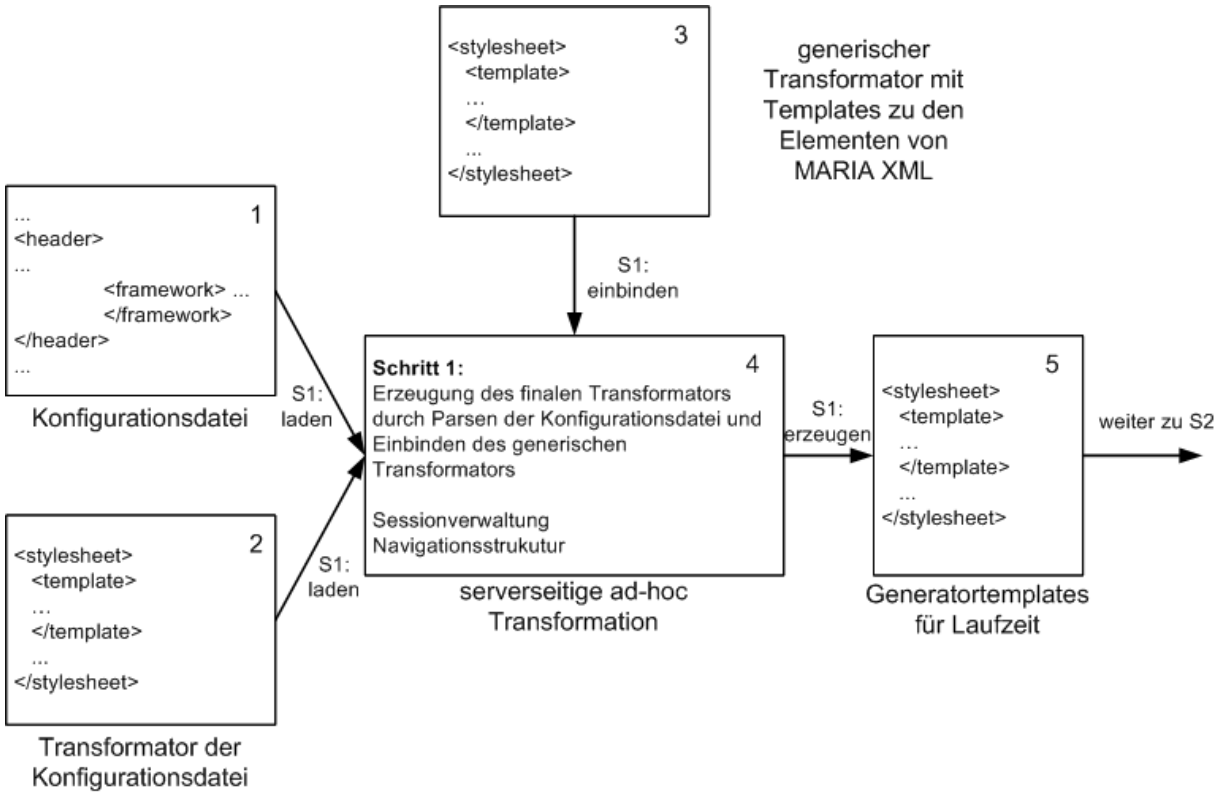


Abbildung 7: Systemübersicht - Schritt 1

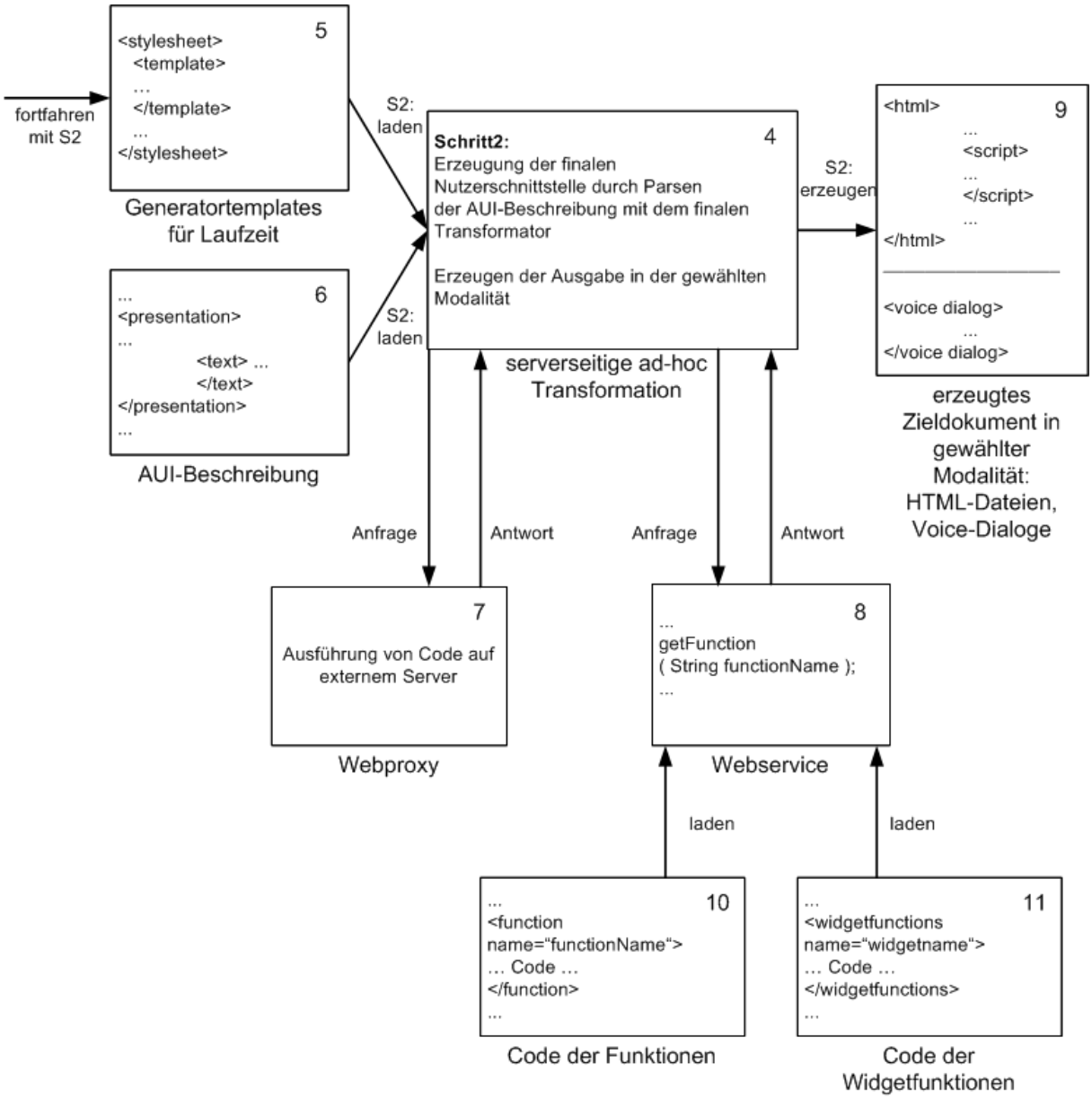


Abbildung 8: Systemübersicht - Schritt 2

4.1.1 Serverseitige Komponente zur ad-hoc Transformation [Abbildung 8/ Komponente 4]

Die zentrale, auf einem Webserver platzierte Komponente [Abbildung 8/ Komponente 4] des Systems soll Anfragen vom Client entgegennehmen und diesem als Antwort die dynamisch generierten Seiten beziehungsweise Sprachdialoge liefern. Das Erstellen der Zieldokumente durch ad-hoc Transformation wird für jede einzelne Anfrage durchgeführt. Dadurch ist die Änderungsmöglichkeit von Parametern in der Konfigurationsdatei zur Laufzeit gewährleistet. Durch die Aufteilung der Grafik zur Systemübersicht in zwei übersichtliche Abbildungen ist die serverseitige Komponente [Abbildung 7/ Komponente 4 und Abbildung 8/ Komponente 4] doppelt dargestellt. Die Durchführung der beiden aufeinanderfolgenden Transformationsschritte soll bei der Umsetzung durch eine Komponente abgedeckt werden.
Ausgangspunkt ist die AUI-Beschreibung [Abbildung 8/ Komponente 6], in der in Maria XML enthaltene Interaktoren, die für ein Szenario benötigt werden, abstrakt definiert sind. Für

diese Interaktoren gibt es einen generischen Transformator [Abbildung 7/ Komponente 3], in dem für jedes mögliche Element von MARIA XML ein Template für die Transformation der abstrakten Beschreibung des jeweiligen Elementes in die finale Nutzerschnittstelle existiert. Dieser Transformator ist für jedes Szenario einsetzbar. Es ist davon auszugehen, dass insbesondere in Bezug auf Web2.0-Applikationen einige Erweiterungen von MARIA XML notwendig sind und damit in die AUI-Beschreibung Elemente aufgenommen werden, für die keine Templates im generischen Transformator [Abbildung 7/ Komponente 3] existieren. Zudem sollen beispielsweise für die Ausführung der Anwendungen benötigte Bibliotheken und Styleinformationen individuell eingebunden werden. Für die Definition dieser Anpassungen wird eine Konfigurationsdatei [Abbildung 7/ Komponente 1] eingeführt. Im ersten Schritt [Abbildung 7/ Schritt 1] wird eine Transformation dieser Konfigurationsdatei mit dem dazugehörigen Transformator [Abbildung 7/ Komponente 2] durchgeführt. Das Ergebnis dieser Transformation ist wieder ein Transformator [Abbildung 8/ Komponente 5]. In dem erzeugten temporären Transformator werden die Templates des schon beschriebenen generischen Transformators [Abbildung 7/ Komponente 3] eingebunden. Damit enthält dieser die Templates für die durch MARIA XML definierten Interaktoren sowie die für ein spezielles Szenario benötigten Templates, die aus der Konfigurationsdatei abgeleitet werden. Im zweiten Schritt [Abbildung 8/ Schritt 2] erfolgt durch Transformation der AUI-Beschreibung [Abbildung 8/ Komponente 6] mit dem temporären Transformator [Abbildung 8/ Komponente 5] die Erzeugung der Seite der finalen Nutzerschnittstelle [Abbildung 8/ Komponente 9] im HTML-Format beziehungsweise in Form von Sprachdialogen.

Beim Aufruf der Komponente 4 müssen der Name der AUI-Beschreibung und somit der Name der eigentlichen Anwendung sowie die Bezeichnung der Konfigurationsdatei und die gewünschte *Modalität* angegeben werden. Da eine Anwendung zumeist aus mehreren Seiten oder Dialogen besteht, ist die Voraussetzung für eine korrekte Reihenfolge der Seiten bei der erzeugten Anwendung eine an dieser Stelle vorliegende *Navigationstruktur*. Der Nutzer soll allerdings nicht nur einen linearen Ablauf einzelner Seiten präsentiert bekommen. Stattdessen sollen die in der abstrakten Beschreibung definierten Seiten mit ihren Interaktoren entsprechend der Interaktion mit dem System dynamisch aus dem AUI ausgewählt und daraus die Nachfolgeseite generiert werden.
Durch die Einführung von *Sessions*, deren Zustand auf dem Server zwischengespeichert wird, lassen sich Verzweigungen innerhalb der Navigationsstruktur und damit beliebige Abläufe in Abhängigkeit von Nutzereingaben realisieren. Um dies umzusetzen, muss bei der später folgenden Implementierung die Möglichkeit geschaffen werden, den Namen der aktuell angezeigten Seite abzulegen und davon ausgehend, die möglichen Transformationen für die zu erstellende Nachfolgeseite zu definieren. Wird eine Anfrage an die Serverkomponente gestellt, erfolgt demnach zuerst eine Überprüfung, ob bereits eine Session für die Anwendung existiert. Wenn das nicht der Fall ist, wird diese erzeugt und der Name der aktuellen Seite gespeichert. Ist beim Aufruf schon eine entsprechende Session vorhanden, wird der Name der Seite, von welcher der Aufruf aus erfolgte, ausgelesen und daraus die anzuzeigende Nachfolgeseite abgeleitet. Durch den Einsatz dieses *Sessionkonzeptes* sollte es bei der Implementierung möglich sein, sämtlichen Interaktionselementen, die der Navigation dienen, die gleiche Adresse zuzuweisen. Dadurch können derartige Elemente in dem AUI generisch beschrieben und wiederverwendet werden. Änderungen der Parameter sind nur vorzunehmen, wenn der serverseitigen Komponente der Wechsel der Modalität mitgeteilt werden soll.

4.1.2 Webproxy zur Ausführung von Code [Abbildung 8/ Komponente 7]

Um externe Funktionalitäten nutzen zu können, wird die Anbindung von Servern unterstützt. Für die Weiterleitung der Anfragen des Client an diese wird ein Webproxy [Abbildung 8/ Komponente 7] verwendet. Er nimmt die Anfragen vom Client, für deren Verarbeitung eine externe Funktionalität notwendig ist, entgegen und leitet sie anschließend an den passenden Server weiter. Die Antwort gerät auf gleichem Weg zurück zum Client.

4.1.3 Webservice zum Laden von externen Funktionen [Abbildung 8/ Komponente 8]

Da der Transformator möglichst generisch gestaltet werden soll und gleichzeitig nur die für eine Anwendung notwendigen Codefragmente in Form von Funktionen bereitzustellen sind, können diese nicht direkt in den Generatortemplates abgelegt werden, denn die für verschiedene Anwendungen benötigen Funktionen variieren, so dass der Transformator, wenn er sämtlichen Code enthält, sehr groß wird. Aus diesem Grund werden die Funktionen ausgelagert und bei Bedarf entsprechend den Vorgaben in der AUI-Beschreibung geladen und eingebunden. Das Anlegen eines solchen Kataloges mit Funktionen ermöglicht zudem die einfache Wiederverwendbarkeit derselben.
Für die Auslagerung des Codes wird ein *Webservice* [Abbildung 8/ Komponente 8] verwendet. Der Code der einzelnen Funktionen wird unter deren Namen [Abbildung 8/ Komponente 10] abgelegt und kann über den Webservice geladen und anschließend in die erzeugte HTML-Seite beziehungsweise den Sprachdialog [Abbildung 8/ Komponente 9] eingebunden werden. Beim späteren Erstellen der AUI-Beschreibung [Abbildung 8/ Komponente 6] für einzelne Szenarien muss die Möglichkeit zur Definition der zu ladenden Funktionen für eine bestimmte Seite eines Szenarios geschaffen werden. Durch dieses Vorgehen soll sich der Inhalt des Transformators auf Templates zur Erzeugung der UI-Komponenten für die visuelle Nutzerschnittstelle beziehungsweise Sprachein- und ausgaben und der Skriptteil auf Funktionsaufrufe beschränken. Die benötigte Funktionalität kann über den Webservice nach Bedarf geladen und eingebunden werden.
Zudem ist es in Bezug auf die Einbindung von Web2.0-Funktionalitäten vorteilhaft, wenn verwendete Widgets einfach austauschbar sind. Deshalb ist eine Trennung der eigentlichen Funktionen von solchen, die für die Ausführung eines Widgets notwendig sind von Vorteil. Alle benötigten Komponenten eines Widgets können dann durch einen Aufruf geladen und eingebunden werden. Mit einer Änderung dieses einzelnen Aufrufs kann somit das komplette Widget gegen eines mit gleicher Funktionalität einfach ausgetauscht werden. Um die Austauschbarkeit von Widgets zu gewährleisten, werden auch deren Funktionen nicht in den Generatortemplates, sondern extern abgelegt [Abbildung 8/ Komponente 11] und gruppiert.

Die Konzeption des in Abbildung 7 und Abbildung 8 dargestellten Systems ist durch umfangreiche Vorüberlegungen entstanden. Die nachfolgend beschriebenen Möglichkeiten zur Umsetzung der Transformationen sind ebenfalls betrachtet worden. Aufgrund der beschriebenen Nachteile ist eine Umsetzung mit diesen Alternativen allerdings ausgeschlossen.

4.2 Systemalternativen

4.2.1 toolbasierter Ansatz

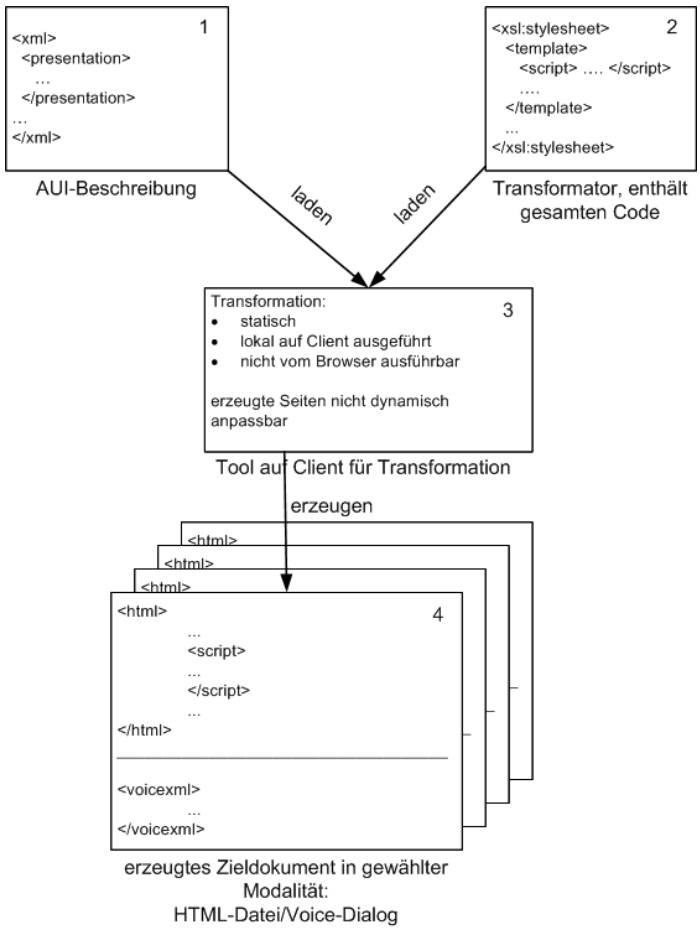


Abbildung 9: toolbasierter Ansatz

Eine einfache Art der Umsetzung des Systems für die Transformation, ausgehend von der AUI-Beschreibung, zur Erzeugung der Zieldokumente ergibt sich, wenn die gesamte Funktionalität des Transformators sowie alle für das zu erzeugende Szenario notwendigen Funktionen zentral abgelegt und daraus die einzelnen Seiten und Dialoge jedes Szenarios zu Beginn der Transformation erzeugt werden. Die Transformation selber erfolgt dabei statisch mit einem toolbasierten Ansatz [Abbildung 9/ Komponente 3]. Anschließend können die erzeugten HTML-Seiten [Abbildung 9/ Komponente 4] im Browser aufgerufen und das Szenario durchlaufen werden. Durch den toolbasierten Ansatz für die Transformation ist es bei dieser Variante nicht möglich, diese vom Browser aus zu starten. Bei diesem Designtime-Ansatz werden die Seiten statisch erzeugt. Es sind keine Anpassungen einzelner Interaktoren zur Laufzeit möglich. Es ist davon auszugehen, dass das Werkzeug zur Durchführung der Transformationen auf Clientseite installiert ist und aus diesem Grund die lokal erzeugten Seiten vor ihrer Verwendung auf den Server übertragen werden müssen.

4.2.2 serverbasierter Ansatz

Wird, im Gegensatz zum toolbasierten Ansatz, die Transformation auf dem Server durchgeführt, kann die Anwendung vom Browser aus gestartet werden, wobei die

Ausführung sämtlicher Transformationen unmittelbar erfolgt und alle für ein Szenario notwendigen HTML-Seiten erzeugt werden. Da in diesem Fall von nur einer serverseitigen Komponente ausgegangen wird, müssen alle Funktionen im Transformator abgelegt werden. Wenn der Transformator, wie vorgegeben, für mehrere Szenarien verwendet werden soll, liegt dadurch viel ungenutzter Code vor. Zudem verhindert die Erstellung aller Seiten eines Szenarios beim ersten Aufruf das dynamische Anpassen einzelner UI-Komponenten entsprechend der Nutzerinteraktion, was bei der Durchführung von ad-hoc Transformationen möglich ist. Die Wahl der Modalität durch den Anwender während der Benutzung der Webanwendung kann ebenfalls nicht angeboten werden. Ohne eine Konfigurationsdatei ist keine Trennung der Elemente, die aus MARIA XML abgeleitet werden können und solchen, die für ein spezifisches Szenario hinzugefügt werden müssen, möglich. Insbesondere enthält der generische Transformator dann Templates für Elemente, die nicht in MARIA XML definiert sind. Die Einführung eines weiteren Transformators, der die Einstellungen der Konfigurationsdatei eines Szenarios einliest und die dafür notwendigen Templates während der Transformation erstellt, ermöglicht es, den generischen Transformator nur auf Elemente von MARIA XML zu beschränken und damit universell einsetzbar zu realisieren. Der Verzicht auf Webproxys verringert die Flexibilität beim Einsatz von Widgets und vorhandenen Skriptbibliotheken. Ist eine zur Ausführung der Applikation notwendige Funktionalität auf einem externen Server abgelegt, dann kann ohne den Einsatz eines Proxys nicht darauf zugegriffen werden. Durch die Weiterleitung der Anfragen zu einem Proxy können die eingesetzten Widgets und Bibliotheken frei gewählt werden.

4.3 Detailanalyse der Szenarien

Nachdem die Vorstellung des konzipierten Systems zur Durchführung der ad-hoc Transformation abgeschlossen ist, erfolgt ausgehend von den beschriebenen Szenarien eine detaillierte Analyse der für sie notwendigen Interaktoren und Funktionalitäten. Dieses Vorgehen hat zum Ziel, im nächsten Abschnitt festzustellen, welche der benötigten Elemente durch die gegebene Version von MARIA XML abstrakt beschrieben werden können. Dadurch sollen die Stellen aufgezeigt werden, an denen Erweiterungen von MARIA XML notwendig sind, um Web2.0-Anwendungen aus der abstrakten Beschreibung abzuleiten und zu realisieren.

4.3.1 Aufbau der Anwendung

Die beispielhaft umgesetzten Szenarien bestehen aus mehreren HTML-Seiten. Es erfolgt eine Gliederung der Webanwendung in mehrere Seiten. Neben den eigentlichen Seiten zur Präsentation der Inhalte müssen weitere definierbar sein, die den Nutzer beispielsweise auf fehlerhafte Eingaben hinweisen. Entsprechend dieser Vorgabe ist ein dynamischer Navigationsfluss zu realisieren. Durch die Möglichkeit, dynamisch auf Nutzerinteraktionen zu reagieren, reicht es nicht aus eine lineare Navigation zwischen den Seiten vorzusehen. Das Element, welches den Ablauf der Anwendung definiert, muss so mit den entsprechenden Interaktionselementen verbunden werden, dass der Navigationsfluss durch Eingaben des Nutzers beeinflussbar ist.

4.3.2 Skriptcode

Neben den statischen Inhalten werden bei dynamischen Webanwendungen viele Elemente, die dem Nutzer präsentiert werden, während des Ablaufs erst erzeugt und in die Seiten

eingefügt. Dies ist durch den Einsatz von Skriptcode möglich. Da sowohl innerhalb einer Anwendung, als auch bei verschiedenen Webanwendungen an einigen Stellen die gleichen Abläufe stattfinden, ist es von Vorteil den Code global bereitzustellen, so dass er wiederverwendet werden kann. Zudem erfolgt die Ausführung des gesamten Codes nicht schon beim Laden einer Seite, sondern teilweise auch erst nach dem Eintreten bestimmter Ereignisse, wie beispielsweise nach dem Anklicken eines Buttons. Dafür ist eine Verknüpfung zwischen den die Ereignisse auslösenden Interaktionselementen und den entsprechenden Codefragmenten herzustellen.

4.3.3 Anzeige von Inhalten

Ein großer Teil der im Web präsentierten Inhalte besteht aus vorher erstellten Texten, die direkt in eine Seite eingebunden werden. Unterscheidungen sind dabei nur bei der Formatierung zu treffen. Neben der einfachen Anzeige von Fließtext gibt es auch noch die Möglichkeit zur Definition von Überschriften oder der Anzeige der Elemente in Tabellenform. In einem der Szenarien werden beispielsweise die in den Warenkorb gelegten Produkte auf der Folgeseite in einer Tabelle ausgegeben. In solchen Fällen sind die Inhalte allerdings nicht vordefiniert. Nachdem der Nutzer die Produkte ausgewählt hat, wird die Tabelle mit deren Namen und Preisen dynamisch erzeugt und die Namen der Produkte eingetragen. Verantwortlich ist wiederum Skriptcode. Im ersten Schritt wird im Fall der Ausgabe der Inhalte in einer Tabelle diese erzeugt und nachfolgend durch den Code darauf zugegriffen. Diese Verbindung zwischen definiertem Ausgabebereich und dafür dynamisch erzeugten Inhalten muss ebenfalls hergestellt werden.

4.3.4 Eingabe von Inhalten

Neben der Präsentation von Informationen muss es dem Nutzer auch möglich sein, diese einzugeben. Ein typisches Beispiel, was auch in einem der Szenarien verwendet wird, ist die Eingabe von Adressdaten in entsprechende Eingabefelder. Zur weiteren Verarbeitung der Inhalte mit Skriptcode müssen diese Felder eindeutig identifizierbar sein, um die Eingaben des Anwenders auszulesen. Ein wichtiges Merkmal von Web2.0-Anwendungen ist die Unterstützung des Nutzers bei der Interaktion. Bei Texteingaben können eine sofortige Validierung derselben, das Unterbreiten von Vorschlägen nach Eingabe der ersten Buchstaben oder eine Autovervollständigung angeboten werden. Bei der Adresseingabe kann beispielsweise beim Feld für die Postleitzahl sofort eine Rückmeldung an den Nutzer erfolgen, ob er ausreichend Zeichen eingegeben hat und diese nur aus Ziffern bestehen. Die Eingabe der Straße kann vereinfacht werden, indem Vorschläge für mögliche Namen angezeigt werden. Für diese Features ist es notwendig, die benötigten Daten im Hintergrund zu laden und in die schon geöffnete Seite an passender Stelle einzubinden.

4.3.5 Asynchrones Laden von Inhalten

Das erwähnte Laden der Daten im Hintergrund soll asynchron erfolgen. Dadurch kann der Anwender mit der Interaktion fortfahren, während neue Inhalte in eine Seite eingebunden werden. Da die für den Ladevorgang notwendige Funktion an verschiedenen Stellen universell einsetzbar ist, sollte eine Möglichkeit geschaffen werden, diese, nachdem sie einmalig in den globalen Skriptteil einer Seite abgelegt wurde, auf einfache Weise auszuführen. Eine Anforderung an das AUI ist deshalb, bei den entsprechenden Elementen

ein Attribut bereitzustellen, durch welches der Anwendung mitgeteilt wird, dass ein solcher Ladevorgang erfolgen soll.

4.3.6 Auswahl von Inhalten

Der asynchrone Ladevorgang wird auch bei Interaktionelementen, in denen dem Nutzer Inhalte zum Auswählen angeboten werden, eingesetzt. Solche Elemente sind Auswahllisten oder Radiobuttons. In dem Flugbuchungsszenario entscheidet sich der Nutzer bei der Wahl seiner Unterkunft mit Hilfe von Radiobuttons für eine Kategorie. Daraufhin werden asynchron die Daten der möglichen Unterkünfte aus dieser Kategorie vom Server geladen und diese innerhalb der Seite in Form einer Auswahlliste präsentiert. Daraus kann der Anwender anschließend die gewünschte Unterkunft per Mausklick wählen.

4.3.7 Mehrfachauswahl von Inhalten

In einigen Fällen soll dem Nutzer die Möglichkeit geboten werden, mehrere Inhalte gleichzeitig auszuwählen. Ein typisches Beispiel dafür ist ein Warenkorb in Onlineshops, in dem per Drag&Drop mehrere Produkte auf einmal abgelegt werden können, um diese zu erwerben. Im Beispielszenario wird diese Art der Interaktion zusammen mit einem Newsticker, der die Produkte präsentiert, eingesetzt. Bei der Umsetzung müssen im ersten Schritt zwei Bereiche definiert werden. Aus einem werden die Produkte mit der Maus entnommen und in dem anderen abgelegt. Der erste Bereich muss mit einer Funktionalität verknüpft werden, die das Laden der Produktbezeichnungen übernimmt und diese anzeigt. Beim zweiten Bereich müssen die abgelegten Produkte erfasst und in einer geeigneten Datenstruktur zur Weiterverarbeitung abgelegt werden.

4.3.8 Einbinden von externen Widgets

Für die komfortable Bedienung einer Web2.0-Anwendung bietet es sich an, bereits vorhandene Widgets einzubinden. Bei der Wahl geeigneter Widgets kann die Oberfläche benutzerfreundlicher gestaltet werden, indem dem Nutzer Informationen, statt rein textbasiert, in einer grafisch aufbereiteten Form präsentiert werden. So kann neben der Eingabe eines Datums über Texteingabefelder auch ein Kalenderwidget eingesetzt werden, um diese Informationen in das System einzugeben. Da es zu einer Aufgabe oft verschiedene Widgets gibt, sollen diese einfach austauschbar sein. Zudem kann es, vor allem bei mobilen Geräten mit einem kleinen Display, von Nachteil sein, ein Widget einzusetzen. In solchen Fällen soll, in Abhängigkeit vom verwendeten Endgerät, auf die Verwendung des Widgets verzichtet und stattdessen eine platzsparende Möglichkeit zur Eingabe von Informationen geschaffen werden. Beim Flugbuchungsszenario wird der Kalender zur Angabe des Abflugdatums in Abhängigkeit vom verwendeten User Agent beispielweise für Geräte mit kleinem Display durch eine Auswahlliste ersetzt.

4.3.9 Einbinden externer Funktionalität

Im Web2.0 entstehen einige Anwendungen durch Kombination mehrerer bereits vorhandener. Zwei Beispiele für vorhandene Applikationen, die für eine solche Kombination verwendet werden können, sind die Kartendienste Google Maps [1] und Virtual Earth [2]. Sie werden im Szenario zur Bestimmung von Orten eingesetzt. Durch die Einbindung der Kartendienste können die Koordinaten von Orten und deren Position auf der Weltkarte

ermittelt werden. Diese und andere Applikationen sollen nach Bedarf in erstellte Anwendungen eingebunden werden und einfach austauschbar sein.

4.3.10 Navigationselemente

Nachdem auf einer Seite sämtliche Eingaben durch den Nutzer erfolgt sind, kann er zur nächsten Seite wechseln. Typische Elemente für die Navigation zwischen den Seiten sind Buttons und Links. Da allerdings, wie im ersten Punkt dieser Analyse beschrieben, die Anwendung nicht nur linear durchlaufen werden kann, reicht es nicht, eine Adresse statisch mit dem Button zu verknüpfen. Stattdessen muss nach dem Klick auf das Navigationselement überprüft werden, ob definierte Bedingungen erfüllt sind. Daraufhin wird die passende Folgeseite geladen und angezeigt. Zudem soll es möglich sein, zurück zur vorhergehenden Seite zu navigieren.

4.4 Anforderungen und vorhandene Elemente in MARIA XML

Betrachtet man, ausgehend von den beschriebenen technischen Anforderungen vorerst nur die in MARIA XML definierten Elemente, ohne bereits auf deren Attribute und die Eignung derselben für die spätere Umsetzung einzugehen, ergibt sich folgende intuitiv zusammengestellte Zuordnung.

Anforderung	**Elemente in Maria XML**
1. Aufbau der Anwendung	presentation
2. Skriptcode	activator
3. Ausgabe von Inhalten	text
4. Eingabe von Inhalten	text-edit
5. Asynchrones Laden von Inhalten	*partiell vorhanden durch Attribut*
6. Auswahl von Inhalten	single choice
7. Multiselect von Inhalten	multiple choice
8. Einbinden von externen Widgets	*bedingt vorhanden*
9. Einbinden externer Funktionalität	*bedingt vorhanden*
10. Navigationselemente	navigator

Tabelle 5: Anforderungen und Elemente in MARIA XML

Bereits an dieser Stelle fällt auf, dass insbesondere die typischerweise bei Web2.0-Anwendungen eingesetzten Funktionalitäten nicht direkt aus MARIA XML ableitbar sind. Elemente für das asynchrone Laden von Daten im Hintergrund, die Einbindung vorhandener Widgets und das Erstellen von Mashups durch die Kombination mehrerer Web2.0-Anwendungen können nicht unmittelbar abgeleitet werden. Die Zuweisung der Elemente zu den Anforderungen, die durch Maria XML abgedeckt werden, erfolgt an dieser Stelle der Arbeit nach ihrer Terminologie.

4.5 Attribute in MARIA XML

Im nächsten Schritt sollen die Attribute der vorhandenen Elemente genauer betrachtet und ihre Eignung entsprechend der aufgeführten Anforderungen untersucht werden. Auch hier erfolgen vorerst nur Überlegungen, welche Attribute für die geforderten Funktionalitäten geeignet sind. Eine umfangreiche Beschreibung des Einsatzes derselben folgt im nächsten Kapitel der Arbeit. Insbesondere bleiben an dieser Stelle die konkreten Wertzuweisungen außen vor.

Für den **Aufbau der Webanwendungen in Form von einzelnen Seiten** sollte es möglich sein, diesen einen Namen zu geben. Ein entsprechendes Attribut ist beim Element *presentation* vorgesehen.

Diesem untergeordnet, gibt es das mit *connections* bezeichnete Element, welches wiederum im XML Schema [39] von MARIA XML *elementary conn*, *complex conn* und *conditional conn* als Kindelemente hat. Für die **Definition einfacher Verbindungen zwischen zwei Seiten** sollten die bei *elementary conn* angegebenen Attribute *interactor id* und *presentation name* ausreichen, um ausgehend von der aktuellen Seite die vorhergehende und nachfolgende festzulegen. Entsprechend den unter dem Punkt „Aufbau der Anwendung“ ausgestellten Anforderungen sollen ebenfalls **bedingte Übergänge zwischen zwei Seiten** definierbar sein. Mit seinen Attributen *presentation to load* und *target presentation* zur Angabe einer Seite mit der Beschreibung des Problems bei fehlerhaften beziehungsweise der Nachfolgeseite bei korrekten Eingaben des Nutzers sollte das Element *conditional conn* dafür geeignet sein. Die ebenfalls vorhandenen Attribute *parameter name* und *parameter value* sind zudem zur Angabe einer Variable und eines Wertes, der ihr zugewiesen sein muss, damit die Bedingung erfüllt ist, vorstellbar. Offen bleibt an dieser Stelle allerdings, wie man Funktionen angeben kann, die einerseits diese Wertzuweisung vornehmen und andererseits bei erfüllter Bedingung die Angaben des Nutzers zur weiteren Verarbeitung vor dem Seitenwechsel abspeichern.

Bei der **Einbindung von Code** muss die *Bezeichnung* der auszuführenden *Funktion* dem *activator*-Element zugewiesen werden. Soll die Ausführung derselben, wie in den Anforderungen beschrieben, nach dem Eintreten eines *Ereignisses* beginnen, sollte auch dieses definierbar sein. Die Angabe des *Ereignisses* ist in MARIA XML durch das Attribut *event name* möglich. Auf das in MARIA XML vorhandene Attribut *function reference* zur Angabe der zu ladenden *Funktion* kann durch das *activator*-Element nicht zugegriffen werden. Um dieses Attribut zur Verfügung zu stellen, muss eine Änderung im XML Schema [39] von MARIA XML erfolgen.

Möchte man **Daten asynchron im Hintergrund laden**, sollte dafür ebenfalls das Element *activator* verwendbar sein, denn der Ablauf ist ähnlich dem beim Aufruf beliebiger Funktionen. Statt ein neues Element einzuführen, muss nur eine geeignete Art der Beschreibung gefunden werden, um nach dem Eintreten des definierten *Ereignisses* die Funktion, durch welche das Laden von Daten realisiert wird, aufzurufen. Ein entsprechendes, mit einem booleschen Wert belegtes Attribut, anhand dessen diese Unterscheidung vorgenommen werden kann, ist in MARIA XML mit *continuous update* vorhanden. Da die vom Server geladenen Daten nach der Übertragung verarbeitet werden müssen, sollte auch dafür ein Attribut zur Angabe des Namens einer entsprechenden Funktion vorhanden sein. In Form von *continuous update function* ist dieses Attribut in MARIA XML definiert.

Zur **Anzeige von statischen Inhalten in Textform** reicht es, wenn diese direkt in der AUI-Beschreibung abgelegt werden können. Allerdings sollen an einigen Stellen die Inhalte nicht nur als Fließtext präsentiert werden. Ein Unterscheidungsmerkmal, anhand dessen auf die Formatierung des Textes geschlossen werden kann, sollte durch ein Attribut definierbar sein. Die Unterscheidung kann durch die Vergabe geeigneter Werte für das *id*-Attribut erfolgen. Statische Textbausteine können unter *data reference* abgelegt werden. Für das dynamische Erstellen von Texten oder das Einfügen dieser in Tabellen ist Skriptcode in Form einer Funktion notwendig. Das Attribut *function reference* steht auch für das Element *text* zur Verfügung.

Die **Eingabe von Information** erfolgt im Web typischerweise über Formulare, in denen Texteingabefelder definiert sind. Zur Identifizierung und Beschriftung der Felder sollte die Angabe einer aussagekräftigen *id* ausreichen. Für die Angabe zusätzlicher Funktionalität wie der sofortigen Validierung der eingegebenen Information oder der Unterbreitung von Vorschlägen nach Eingabe der ersten Buchstaben sind keine geeigneten Attribute vorhanden.

Wie in den Anforderungen beschrieben, kann die **Auswahl von Inhalten** auf verschiedene Weise erfolgen. Ähnlich der Unterscheidung der Formatierung bei der Textausgabe sollte auch beim Element *single choice* anhand des vorhandenen Attributes *id* in der AUI-Beschreibung die Art der Umsetzung definierbar sein. Für die Festlegung von *Ereignissen* entsprechend der Auswahl, nach deren Eintreten eine festgelegte Funktion ausgeführt wird, steht für das Element *single choice* das Attribut *event name* zur Verfügung. Das Attribut *function reference* ist dem *single-choice*-Element nicht zugewiesen. Es muss in die entsprechende Attributgruppe im XML Schema [39], wie schon beim *activator*-Element beschrieben, eingefügt werden. Mit den in MARIA XML, unter den Attributgruppen *change interactor attributes* und *change data attributes* angegebenen Attributen *interactor id*, *property name*, *property value*, beziehungsweise *data reference* und *data value* sollten ausreichend Möglichkeiten vorhanden sein, um Interaktoren wie eine Auswahlliste oder Radiobuttons und ihre Eigenschaften abstrakt zu beschreiben. Für die Umsetzung des Newstickers oder anderer Präsentationsmöglichkeiten, die periodisch Daten vom Server anfordern, ist die Verwendung der ebenfalls vorhandenen Attribute *continuous update* und *continuous update function* vorstellbar.

Für das zur Realisierung von Interaktionselementen zum **Multiselect von Objekten** selektierte Element *multiple choice* stehen ebenfalls die unter *single choice* beschriebenen Attribute zur Verfügung. Sie sollten auch in diesem Fall ausreichen, um Interaktionsmöglichkeiten wie Drag&Drop abstrakt zu beschreiben.

Betrachtet man die Definition des Elementes *navigator* zur Realisierung von **Interaktoren für die Navigation** innerhalb des XML Schemas [39] von MARIA XML sind auch hier die schon weiter oben erwähnten Attribute *id* und *event name* vorhanden. Nach dem Hinzufügen des *function-reference*-Attributes sollten die Beschreibungsmöglichkeiten genügen, um die Art des Navigationselementes, beispielsweise *Button*, das *Ereignis* und die nach dessen Eintreten auszuführende *Funktion* anzugeben.

4.5.1 Auflistung fehlender Attribute

Entsprechend der durchgeführten Analyse von MARIA XML und dem Abgleich mit den aufgestellten Anforderungen ergeben sich an dieser Stelle der Arbeit folgende Funktionalitäten, für die bisher keine geeigneten Attribute existieren.

1.	Validierungsfunktion vor einem bedingten Seitenwechsel
2.	Funktion zum Speichern von Werten vor einem bedingten Seitenwechsel
3.	Validierung von Eingaben bei Texteingabefeldern
4.	Autovervollständigung / Vorschlagen von Begriffen bei Texteingabefeldern

Tabelle 6: fehlende Attribute

4.5.2 Definition neuer Attribute

Passend zu der in Maria XML verwendeten Terminologie, sollen für die fehlenden Attribute folgende Bezeichnungen verwendet werden.

zu 1.	validation function
zu 2.	vontinue function
zu 3.	needs validation
zu 4.	event triggered update

Tabelle 7: neu hinzugefügte Attribute

4.6 Erweiterung von MARIA XML

Einige Elemente zur abstrakten Beschreibung von Interaktoren werden durch das Hinzufügen weiterer Attribute ergänzt. Verwendung finden in MARIA XML bereits vorhandene Attribute, die den, in der Tabelle dargestellten, Elementen hinzugefügt werden. Außerdem werden die im vorangegangen Kapitel vorgestellten Attribute neu eingeführt. Sie sind in der Tabelle kursiv hervorgehoben. In der ersten Spalte stehen die Namen der Elemente, welche um weitere Attribute ergänzt werden. In MARIA XML sind die Attribute zu Gruppen zusammengefasst. Möchte man die Attribute einer Gruppe bei einem Element ergänzen, wird der Name der Gruppe in die Elementdefinition im XML Schema [39] eingefügt. Dem Element *edit group* werden die Elemente *external widget type* und *external application type* zur Einbindung von Widgets und Erstellung von Mashups neu hinzugefügt. Bei der Definition des Typs werden diesen beiden Elementen die vorhandenen Attributgruppen *edit attributes* und *interactor attributes* zugewiesen. Die Attribute *function reference*, *needs validation* und *event triggered update* werden in der Attributgruppe *interactor attributes* ergänzt.

Element	Attributgruppe/Typ	Attribute
navigator type	edit attributes	value
cond type		*validation function* *continue function*
single choice type	change interactor attributes	interactor id property name property value
edit group	*external widget type* *external application type*	
	interactor attributes	function reference *needs validation* *event triggered Update*

Tabelle 8: Erweiterungen von MARIA XML

Im Folgenden wird analysiert, aus welchen Gründen die gegebene Version 1.3 von MARIA XML für die Erstellung von Benutzerschnittstellen für Web2.0-Szenarien um die in der Tabelle dargestellten Attribute ergänzt werden muss.

Beim ***navigator***-Element wird das Attribut *value* ergänzt, um in der visuellen Nutzerschnittstelle eine Beschriftung der aus diesem Element abgeleiteten Navigationselemente vornehmen zu können. Bei der Erzeugung von Sprachein- und ausgaben wird durch den Wert dieses Attributes festgelegt, welche Antwort das System vom Nutzer erwartet.

Das Tag ***cond type*** dient der abstrakten Beschreibung von bedingten Seitenübergängen. Die Überprüfung einer Bedingung ist abhängig vom Inhalt. Aus diesem Grund eignet sich dafür keine generische Funktion, die allgemein einsetzbar ist und deren Aufruf abgeleitet werden kann. Für eine Überprüfung eines speziellen Sachverhaltes muss eine neue Funktion in den Katalog aufgenommen werden. Deren Name wird in der abstrakten Beschreibung unter dem Attribut *validation function* abgelegt. Wenn die Bedingung nicht erfüllt ist, wird anschließend eine entsprechende Seite aufgerufen, die auf den Fehler hinweist. War die Überprüfung der Bedingung erfolgreich, kann mit der Folgeseite fortgefahren werden. Bevor der Seitenwechsel erfolgt, müssen die vom Anwender durchgeführten Eingaben aus den Interaktoren ausgelesen und zur weiteren Verwendung auf den Folgeseiten zwischengespeichert werden. Dafür ist wiederum eine auf den Inhalt zugeschnittene Funktion notwendig, deren Name dem Attribut *continue function* zugewiesen wird.

Die Ergänzung des ***single-choice***-Elementes um die Attribute *property name* und *property value* erlaubt es, dieses Element zur Umsetzung einer Auswahlliste zu verwenden, ohne den Namen der Funktion, die für das Eintragen der Inhalte in die Liste verantwortlich ist, in die AUI-Beschreibung aufzunehmen. Wird in der abstrakten Beschreibung das Attribut *id* mit dem Wert *select* belegt, wird daraus die Erzeugung einer Auswahlliste abgeleitet und die entsprechende generische Funktion zum Übernehmen der Daten in die Liste ausgeführt. Durch die Attribute *property name* und *property value* wird in der abstrakten Beschreibung festgelegt, welcher Teil der Daten relevant ist und durch diese Funktion ausgelesen werden soll.

Die neu eingeführten Elemente ***external widget*** und ***external application*** werden in die hierarchische Struktur von MARIA XML aufgenommen, indem ihre Typbezeichnungen der Elementgruppe *edit group* hinzugefügt werden.

Die in der Attributgruppe ***interactor attributes*** definierten Attribute können in MARIA XML beispielsweise bei den Elementen *activator*, *text edit*, *multiple choice* und *navigator* eingesetzt werden. Zu den vorhandenen Attributen wird diese Gruppe um das Tag *function reference* ergänzt, damit zum Beispiel bei Verwendung des *activator*-Elementes, welches unter anderem zum Ausführen von Funktionen eingesetzt wird, der Name einer solchen Funktion angegeben werden kann. Für das Element *text edit,* welches zur Erzeugung von Texteingabefeldern Einsatz findet, werden dieser Gruppe die neu eingeführten Attribute *needs Validation* und *event triggered update* hinzugefügt. Wird diesen in dem AUI der Wert *true* zugewiesen, erfolgt eine Überprüfung der Eingabe beziehungsweise die Unterbreitung von Vorschlägen oder Autovervollständigung von weiteren Textfeldern. Beide Attribute werden zusammen mit dem Attribut *function reference* verwendet. Bei diesem Attribut wird als Wert der Name der Funktion zur Durchführung der Validierung oder Autovervollständigung angegeben. An dieser Stelle können keine generischen Funktionen eingesetzt werden, da die Funktionalität vom konkreten Inhalt abhängig ist.

4.7 AUI-Beschreibung und Konzeption des generischen Transformators

Nachdem in den vorangegangen Kapiteln analysiert wurde, welche Elemente von MARIA XML für welchen Zweck verwendet werden können und notwendige Erweiterungen definiert wurden, erfolgt in diesem Abschnitt der Arbeit die Konzeption der AUI-Beschreibung, aus welcher die Templates des Transformators abgeleitet werden. Die Zielstellung des Kapitels ist es im Detail zu zeigen, wie aus der abstrakten Beschreibung durch die Templates des vorgestellten generischen Transformators direkt das FUI ohne weitere Zwischenschritte erzeugt wird. Für die verwendeten Elemente und Attribute werden im Folgenden entsprechende Wertzuweisungen genannt, die für spätere Umsetzungen relevant sind. Begonnen wird mit der Konzeption der abstrakten Beschreibung des Aufbaus der zu erzeugenden Anwendungen.

Die Umsetzung der Szenarien für das Web kann sowohl als Single-Page-Applikation als auch aufgeteilt auf mehrere Seiten, die nacheinander durchlaufen werden, erfolgen. Innerhalb des AUI wird für jede zu erzeugende Seite unter Verwendung des *presentation*-Tags eine Präsentation angelegt. Ausgehend von diesem Element wird durch den Transformator das HTML-Grundgerüst erstellt. Eine Präsentation kann auch die abstrakte Beschreibung von Interaktoren eines Bereiches einer einzelnen Seite beinhalten. Für alle Interaktoren, die innerhalb einer Präsentation angezeigt werden können, sind im Transformator entsprechende Templates definiert. Sie werden bei der Transformation aufgerufen, wenn das dazugehörige Element in dem AUI vorhanden ist. Dies hat den Vorteil, dass für jeden abstrakt beschriebenen Interaktor nur ein Template angelegt werden muss, welches mehrfach mit unterschiedlichen Attributwerten aufgerufen werden kann. Diese Templates sind nicht nur innerhalb der unterschiedlichen Seiten eines einzelnen Szenarios wiederverwendbar, sondern können unverändert für alle Szenarien eingesetzt werden.

In diesem Kapitel werden nur die in MARIA XML vorhandenen Elemente und die dazugehörigen Templates des generischen Transformators beschrieben. Die Templates für

zusätzlich eingeführte Elemente werden in dem Transformator für die Konfigurationsdateien abgelegt. Die folgenden Elemente von MARIA XML werden für die Entwicklung von Web2.0-Szenarien verwendet:

4.7.1 connection

Innerhalb der AUI-Beschreibung der einzelnen Szenarien ist es notwendig, die Übergänge zwischen den Seiten zu definieren. Neben der Angabe vorheriger und nachfolgender Seiten, im AUI als *presentation* bezeichnet, wird damit die Möglichkeit geschaffen, aufzurufende Seiten außerhalb der linearen Navigationsstruktur direkt zu definieren. Dies ist beispielsweise nach fehlerhaften Eingaben durch den Nutzer notwendig.

4.7.1.1 elementary connection

Dieses Element findet Anwendung, wenn vor dem Übergang von einer zur nächsten Seite keine Bedingung geprüft werden muss und somit eine *elementare Verbindung* der Seiten besteht. Mit dem Attribut *interactor id* und der Zuweisung der Werte *forward* beziehungsweise *back* wird festgelegt, ob die vorherige oder nachfolgende Seite aufgerufen wird. Der eigentliche Seitenwechsel wird durch eine Nutzerinteraktion ausgelöst. Dies kann beispielsweise durch Auswahl eines bestimmten Wertes aus einer Liste oder durch den Klick auf entsprechende Buttons geschehen.

4.7.1.2 conditional connection

Wird statt *elementary connection* dieser Interaktor zur Definition des Übergangs zwischen zwei Seiten verwendet, ist der Wechsel zur nächsten Seite an eine Bedingung geknüpft. Es gilt, diese zu überprüfen und danach die passende Seite auszuwählen. Dies kann beispielsweise dann der Fall sein, wenn Eingaben des Nutzers einem bestimmten Schema entsprechen müssen. Unter dem Attribut *validation function* [Zeile 1] wird der Name der Funktion, welche die eigentliche Validierung durchführt, angegeben. Diese Funktion setzt den unter dem Attribut *parameter name* [Zeile 3] abgelegten Wert auf *true* oder *false*. Entsprechend dem Ergebnis der anschließend durchgeführten Überprüfung wird als die nächste anzuzeigende Seite entweder der unter *target presentation name* [Zeile 6] oder der unter *presentation to load* [Zeile 11] abgelegte Wert ausgewählt, um diese zu laden. Das folgende Codefragment ist ein Auszug aus dem Template für *conditional connection*.

```
…
1 <xsl:value-of select="ns:cond/@validation_function" />;
2
3 if(<xsl:value-of select="@parameter_name" /> ==
4          <xsl:value-of select="ns:cond/@parameter_value" />){
5          var nextScreen =
6          "<xsl:value-of select="ns:cond/@target_presentation_name" />";
7          document.cookie = "SessionCookie=" + nextScreen;
8 }
9 else {
10         var nextScreen =
11         "<xsl:value-of select="ns:cond/@presentation_to_load" />";
12         document.cookie = "SessionCookie=" + nextScreen;
13 }
14 <xsl:value-of select="ns:cond/@continue_function" />;
…
```

Listing 1: Auszug aus dem generischen Transformator, Template für *conditional conn*

Nachdem die Zuweisung des Wertes erfolgt ist, wird mit der unter *continue function* [Zeile 14] angegeben Funktion fortgefahren. Diese dient zum Auslesen der Nutzereingaben sowie der Speicherung derselben, um Sie auf den nachfolgenden Seiten verfügbar zu machen. Der folgende Codeauszug zeigt beispielhaft, wie eine solche bedingte Verbindung in der AUI-Beschreibung definiert werden kann.

```
1 <ns:connections>
2          <ns:conditional_conn
3                    interactor_id="screen4"
4                    id="screen4"
5                    parameter_name="correct">
6          <ns:cond
7                    parameter_value="true"
8                    validation_function="validate()"
9                    continue_function="collectPersonalData()"
10                   presentation_to_load="errorPage"
11                   target_presentation_name="screen5" />
12         </ns:conditional_conn>
13         …
14 </ns:connections>
```

Listing 2: Auszug aus der AUI-Beschreibung, Element *conditional conn*

Der unter *interactor id* [Zeile 3] und *id* [Zeile 4] abgelegte Wert bezeichnet die Seite innerhalb eines Szenarios, die aktuell angezeigt wird. In diesem Beispiel ist dies die vierte Seite. Durch die Ausführung der hier mit *validate* bezeichneten Funktion [Zeile 8] wird die Variable *correct* [Zeile 5] auf *true* oder *false* gesetzt. Die Bedingung ist erfüllt, wenn der Wert der Variable mit dem unter dem Attribut *parameter value* [Zeile 7] abgelegten Wert übereinstimmt. Wenn die Variable *correct* auf *true* gesetzt wurde, dann wird die mit *screen5* [Zeile 11] bezeichnete Folgeseite aufgerufen. Liegt ein Fehler vor, beispielsweise durch unvollständige Nutzereingaben, liefert die Valdierungsfunktion *false* zurück und daraufhin wird, entsprechend der Definition im Template, die mit *errorPage* [Zeile 10] bezeichnete Seite geladen, um den Nutzer auf den Fehler hinzuweisen. Durch die Möglichkeit, die notwendigen Funktionen und die Namen der zu ladenden Seiten für jedes Szenario in der AUI-Beschreibung individuell festzulegen, kann das im Transformator definierte Template für

alle Anwendungsfälle, bei denen eine Validierung vor dem Seitenwechsel notwendig ist, wiederverwendet werden.

4.7.2 description

Dieses Element wird verwendet, um Wertzuweisungen durchzuführen. Unter dem Attribut *id* wird der Name der Variable abgelegt und unter *data reference* der ihr zuzuweisende Wert. Auf diese Weise kann beispielsweise beschrieben werden, welche AUI-Beschreibung für die Transformation oder welche Modalität für den nachfolgenden Dialog verwendet wird.

4.7.3 text

Das Template für das Textelement dient der Anzeige von Zeichenketten. Die Zeichenkette wird unter dem Attribut *data reference* in der AUI-Beschreibung abgelegt. Über das Attribut *id* erfolgt eine Unterscheidung zwischen verschiedenen Möglichkeiten der Ausgabe. Betrachtet werden neben der einfachen Textausgabe für die visuelle Umsetzung auch Überschriften und die Ausgabe in Tabellenform. Für den Fall, dass die Zeichenkette nicht schon beim Laden der Seite vorliegt, sondern entsprechend der Nutzerinteraktion dynamisch erzeugt wird, kann unter *function reference* der Name einer Funktion angegeben werden, die das Erstellen des Inhaltes übernimmt. Durch die Möglichkeit, die Funktion individuell im AUI festzulegen, ist das Template für die Textausgabe universell einsetzbar. Die Funktion ist insbesondere dann notwendig, wenn die Ausgabe in Tabellenform erfolgen soll und die Inhalte dynamisch in die Tabelle eingefügt werden.

4.7.4 text edit

Bei Verwendung dieses Elementes in der AUI-Beschreibung wird der Nutzer zur Eingabe von Informationen aufgefordert. Das Attribut *id* dient zur Identifikation des Interaktors, um die getätigten Eingaben für die weitere Verarbeitung auslesen zu können. Die Anzeige dieses Attributwertes in der finalen Nutzerschnittstelle ist gleichzeitig die Beschriftung des Texteingabefeldes. Um den Anwender gegebenenfalls bei der Eingabe der Informationen zu unterstützen, kann über die entsprechenden Attribute *event triggered update* und *needs validation* die Unterbreitung von Vorschlägen, die Autovervollständigung von Eingaben oder die sofortige Validierung der Eingaben gestartet werden. Auf die Angabe des Namens der Funktion, die diese Funktionalitäten ausführt, kann verzichtet werden, da sie generisch gestaltet ist und wiederverwendet werden kann. Die *id* des Feldes wird beim Funktionsaufruf übergeben und daraus abgeleitet, an welcher Stelle Vorschläge angezeigt oder die Validierung der Eingabe erfolgen soll.

4.7.5 activator

Dieses Element wird für drei verschiedene Aufgaben verwendet. Es dient dem Laden des Codes für Funktionen vom Server und der Ausführung derselben nach dem Eintreten eines definierten Ereignisses. Zudem wird mit dem *activator*-Element das asynchrone Laden notwendiger Daten im Hintergrund gesteuert. Für die beiden erstgenannten Aufgaben wird im Template überprüft, ob in dem AUI das Attribut *function reference* [Zeile 2] mit einem Wert belegt ist. Ist dies der Fall, wird zudem aus dem Attribut *event name* [Zeile 4] der Name des Ereignisses ausgelesen, nach dessen Eintreten die definierte Funktion aufgerufen wird. Dabei kann es sich einerseits um die Funktion *getFunction(String Funktionsname)* zum Laden

von extern abgelegtem Code handeln und andererseits um den Namen einer bereits geladenen und in den Skriptteil der Seite eingebundenen Funktion, welche ausgeführt werden soll. Wenn eine Funktion erst geladen werden muss, wird in der AUI-Beschreibung ihr Name als Übergabewert der Funktion *getFunction* [Zeile 2] angegeben.

```
1 <ns:activator id="onLoadFunction"
2         function_reference="getFunction('forwardElements')" >
3         …
4         <ns:raise event_name="window.onload" />
5         …
6 </ns:activator>
```

Listing 3: Auszug aus der AUI-Beschreibung, Element für *activator*

Für die dritte der genannten Aufgaben, das asynchrone Laden von Daten, muss in dem AUI das Attribut *continuous update* mit dem Wert *true* belegt werden. Der Transformator prüft dies und liest anschließend die Werte der Attribute *data reference* und *continuous update function* aus. Durch ersteres wird der Name der zu ladenden Datei definiert, in welcher die angeforderten Daten abgelegt sind. Der Wert von *continuous update function* ist die Bezeichnung der Funktion, die nach erfolgtem Laden der Daten ausgeführt wird, um diese zu verarbeiten und in die Seite einzubinden.

4.7.6 navigator

Dieses Element wird zur Umsetzung der Navigation verwendet. Zuerst erfolgt die Überprüfung anhand des Attributes *id* [Zeile 1], um welche Art von Navigationselement es sich handelt. Im unten dargestellten Beispiel ist der Wert mit *button* bezeichnet. Denkbar wäre beispielsweise auch ein *Link*, wodurch beim Erzeugen des HTML-Codes statt des *input*-Tags, das *a*-Tag eingesetzt werden muss. Im nächsten Schritt wird der Name des Interaktionsereignisses ausgelesen und in eine Variable geschrieben. Hier ist dies *onclick* [Zeile 6]. Jenes ist für einen Button zwar das gängigste Ereignis, wird aber trotzdem nicht direkt in das Stylesheet geschrieben, sondern aus der AUI-Beschreibung ausgelesen, um den Transformator auch an dieser Stelle generisch zu gestalten.

```
1 <ns:navigator id="button"
2         function_reference="forwardElements()" value="Select">
3         <ns:events>
4                 <ns:activation>
5                         <ns:handler>
6                                 <ns:raise event_name="onclick" />
7           </ns:handler>
8       </ns:activation>
9   </ns:events>
10 </ns:navigator>
```

Listing 4: Auszug aus der AUI-Beschreibung, Element für *navigator*

Die Attribute des *input*-Tags [Zeile 9] setzen sich wie folgt zusammen: als *type* und *id* [Zeile 10, Zeile 16] wird der Wert der *id*, in dem Fall *button*, als *value* [Zeile 19] der in dem AUI unter gleichem Namen abgelegte Wert für die Beschriftung des Buttons zugewiesen. Dem unter *event name* abgespeicherten Ereignisnamen wird der Wert des Attributes *function reference* [Zeile 3, Zeile 14] übergeben. Dass heißt, wenn das Ereignis *onclick* eintritt, wird die im AUI definierte Funktion aufgerufen. Die Durchführung von Seitenwechseln wird in

vielen Fällen ebenfalls durch Interaktion mit einem aus dem *navigator*-Element erzeugten Interaktor durchgeführt. Zum Wechsel zur vorhergehenden beziehungsweise nachfolgenden Seite gibt es wiederverwendbare Funktionen, deren Namen ebenfalls unter dem Attribut *function reference* abgelegt werden. Diese Funktionen lesen die Bezeichnungen der aufzurufenden Seiten, welche durch das *connections*-Element in entsprechenden Variablen abgelegt wurden, aus und rufen die nächste Seite, die daraufhin durch Transformation erzeugt wird, auf.

```
1 <xsl:template match="ns:navigator">
2       <xsl:if test="@id= 'button'">
3             <xsl:variable name="eventName">
4                   <xsl:value-of
5                    select="ns:events/ns:activation/ns:handler/ns:raise/
6                   @event_name"/>
7             </xsl:variable>
8             <div>
9             <input>
10                  <xsl:attribute name="type">
11                        <xsl:value-of select="@id"/>
12                  </xsl:attribute>
13                  <xsl:attribute name="{$eventName}">
14                        <xsl:value-of select="@function_reference"/>
15                  </xsl:attribute>
16                  <xsl:attribute name="id">
17                        <xsl:value-of select="@id"/>
18                  </xsl:attribute>
19                  <xsl:attribute name="value">
20                        <xsl:value-of select="@value"/>
21                  </xsl:attribute>
22            </input>
23            </div>
24      </xsl:if>
25 </xsl:template>
```

Listing 5: Auszug aus dem generischen Transformator, Template für *navigator*

Das *navigator*-Element wird zudem zum Einblenden von Interaktoren, die für eine Seite bei der Transformation erzeugt werden, jedoch beim Laden dieser noch nicht sichtbar sind, verwendet. Zur Umsetzung dieser Funktionalität wird das *id*-Attribut in der abstrakten Beschreibung mit dem Wert *activate* belegt. Beim Kindelement *handler* steht ein Attribut mit der Bezeichnung *name* zur Verfügung, dessen Wert mit der *id* des Bereiches, der eingeblendet werden soll, übereinstimmen muss.

4.7.7 single choice

Das Element *single choice* wird für die Umsetzung verschiedener Möglichkeiten zur Auswahl von Inhalten verwendet. Im Folgenden sollen beispielhaft drei dieser Varianten vorgestellt werden. Für die Unterscheidung der Varianten werden MARIA XML keine neuen Elemente hinzugefügt, sondern vorhandene Attribute mit passenden Wertzuweisungen verwendet.

4.7.7.1 single choice mit continuous update

Ist das Attribut *continuous update* [Zeile 3] auf *true* gesetzt und unter *continuous update function* [Zeile 5] der Name einer Funktion angegeben, können damit dem Nutzer periodisch

wechselnde Inhalte präsentiert werden. Eine denkbare Anwendung ist beispielsweise ein Newsticker, in welchem Produktneuigkeiten vorgestellt werden. Der Nutzer kann dann durch Auswahl eines der Produkte erwerben.

```
1 <xsl:template match="ns:single_choice">
2         …
3         <xsl:if test="@continuous_update='true'">
4                 <script>
5                         <xsl:value-of select="@continuous_update_function"/>
6                 </script>
7         </xsl:if>
8 …
```

Listing 6: Auszug aus dem generischen Transformator, Template für *single choice*

4.7.7.2 single choice mit id *select*

Identifiziert wird diese Variante der *single-choice*-Umsetzung zur Generierung einer Auswahlliste anhand des mit dem Wert *select* belegten *id*-Attributes in der AUI-Beschreibung. Der Transformator soll bei der Realisierung einer Auswahlliste für eine Webseite zwischen solchen unterscheiden, bei denen nach der Auswahl unmittelbar ein Ereignis ausgelöst wird, um beispielsweise einen sich anschließenden Wechsel zur Nachfolgeseite durchzuführen und solchen, bei denen nur die Auswahl relevant ist. Beim letztgenannten Fall gibt es auf der Seite weitere Interaktoren, durch die zum Beispiel ein Seitenwechsel ausgelöst werden kann.
Unterschieden wird zwischen den beiden Fällen im Template anhand der Attribute *property name* [Zeile 1] und *event name* [Zeile 13]. Ist letzteres in der AUI-Beschreibung mit einem Wert belegt, dann ist ein Ereignis definiert, was unmittelbar nach erfolgter Auswahl eintritt und die unter *function reference* [Zeile 25] angegebene Funktion aufruft. Der Name und die *id* der Auswahlliste sind in dem AUI unter *events/selection_change/handler/name* [Zeile 18] abgelegt und werden bei der Transformation zugewiesen. Ist hingegen statt *event name* das Attribut *property name* mit einem Wert belegt, dann wird eine Auswahlliste erzeugt, die den durch Nutzerinteraktion gewählten Eintrag als den Aktiven anzeigt. Erst nach Betätigung eines weiteren Bedienelementes wird ein Ereignis ausgelöst und der ausgewählte Wert für die weitere Verarbeitung ausgelesen. Zudem wird in der finalen Nutzerschnittstelle den beiden Attributen *name* [Zeile 4] und *id* [Zeile 7] durch Übernahme der in dem AUI unter *property name* abgelegten Zeichenkette ein Wert zugewiesen.

```
1 <xsl:if test="not ($propertyName = '')">
2       <form>
3       <select>
4               <xsl:attribute name="name">
5                       <xsl:value-of select="$propertyName" />
6               </xsl:attribute>
7               <xsl:attribute name="id">
8                       <xsl:value-of select="$propertyName" />
9               </xsl:attribute>
10              </select>
11              </form>
12      </xsl:if>
13      <xsl:if test="not ($eventName = '')">
14      <form>
15      <select>
16              <xsl:attribute name="name">
17              <xsl:value-of
18              select="ns:events/ns:selection_change/ns:handler/@name" />
19              </xsl:attribute>
20              <xsl:attribute name="id">
21              <xsl:value-of
22               select="ns:events/ns:selection_change/ns:handler/@name" />
23              </xsl:attribute>
24              <xsl:attribute name="{$eventName}">
25                      <xsl:value-of select="@function_reference" />
26              </xsl:attribute>
27      </select>
28      </form>
29      </xsl:if>
30      …
```

Listing 7: Auszug aus dem generischen Transformator, Template für *single choice*

4.7.7.3 single choice mit id *radio*

Eine weitere Möglichkeit, eine Auswahl innerhalb einer Weboberfläche zu treffen, sind Radiobuttons. Dem Nutzer werden alle wählbaren Elemente präsentiert, und er entscheidet sich durch Setzen einer Marke für eines. Für die Umsetzung dieser Variante wird beim Transformator geprüft, ob in dem AUI das Attribut *id* mit dem Wert *radio* belegt wurde. Dieser Wert wird bei der Transformation in das HTML-Format den Radiobuttons als Typ zugewiesen. Die anzuzeigenden Elemente müssen in dem AUI innerhalb des Tags *change property* definiert werden. Die Attribute *type*, *name* sowie *value* für dieses Element und das Ereignis, durch welches nach Auswahl der Aufruf einer Funktion erfolgt, werden in dem AUI ähnlich der obigen Auswahllistenvariante durch die Attribute *type name*, *interactor id*, *data value* und *event name* beschrieben.

4.7.8 multiple choice

Eine mögliche Anwendung für das *multiple-choice*-Element in einer Web2.0-Applikation ist die Umsetzung der *Drag&Drop*-Funktionalität. Sie ermöglicht, im Gegensatz zu den bei *single choice* genannten Möglichkeiten wie der Auswahlliste oder den Radiobuttons, die Auswahl mehrerer Elemente gleichzeitig. Durch den Transformator wird im ersten Schritt ein Bereich definiert, in dem die verschiebbaren Elemente abgelegt werden können. Zur Identifikation dieses Bereiches wird ihm als *id* der unter dem gleichen Attributnamen in dem AUI abgelegte Wert zugewiesen. Für die Übertragung der Daten vom Server zum Client wird die schon weiter oben beim *activator*-Interaktor vorgestellte Variante zum asynchronen Laden

derselben verwendet. Die Funktionalität zum Erstellen der verschiebbaren Elemente aus den Bezeichnungen der geladenen Daten liefert der generische Transformator. Ihr Einsatz wird aus der Belegung des *id*-Attributes mit dem Wert *dragDiv* abgeleitet. Das Objekt, beispielsweise ein Warenkorb für Produkte, in dem die Elemente abgelegt werden können, wird im nächsten Punkt beschrieben.

4.7.9 object edit

Wird dieses Element zusammen mit *multiple choice* verwendet, muss in der finalen Nutzerschnittstelle ein Bereich definiert werden, welcher durch Übernahme des in der AUI abgelegten *id*-Wertes identifizierbar ist. Das mit *onDrop* bezeichnete Ereignis tritt ein, sobald der Nutzer ein Objekt in dem definierten Bereich ablegt. In diesem Fall erfolgt durch den Interaktor *object edit* nur die Speicherung der Daten in einer geeigneten Datenstruktur. Diese Funktionalität stellt der Transformator bereit. Sie wird immer dann ausgeführt, wenn das Element *object edit* in der abstrakten Beschreibung die *id dropDiv* hat. Die weitere Verarbeitung beginnt erst nach dem Eintreten weiterer Ereignisse, beispielsweise durch den Klick auf einen Button zur Bestätigung der Auswahl. Durch dieses Vorgehen bleibt der Transformator auch an dieser Stelle generisch, denn die nachfolgenden Aktionen werden nicht von vornherein festlegt, sondern können für jeden Anwendungsfall individuell in dem AUI beschrieben werden.

In der Tabelle 9 werden die in diesem Kapitel beschriebenen Elemente von MARIA XML und die dazugehörigen Einsatzmöglichkeiten in der finalen Nutzerschnittstelle zusammengefasst.

Element der AUI	Einsatz in finaler Nutzerschnittstelle
connection *elementary connection* *conditional connection*	• unbedingter Seitenwechsel • bedingter Seitenwechsel
description	• Wertzuweisungen für definierte Variablen
text	• Überschriften • Textabschnitte • Tabellen
text edit	• Texteingabefelder mit Validierung mit Unterbreitung von Vorschlägen
activator	• Laden von Funktionscode • Ausführen von Funktionen • asynchrones Laden benötigter Daten
navigator	• Durchführen von Seitenwechseln • Auslösen von Funktionalitäten • Einblenden von Bereichen auf einer Seite
single choice	• periodisches Laden von Inhalten (Polling) • Auswahllisten mit/ ohne Ereignis nach Auswahl
mulitple choice	• Anzeige von verschiebbaren Objekten bei Drag&Drop
object edit	• Aufnahme der Objekte bei Drag&Drop

Tabelle 9: Einsatz abstrakt beschriebener Elemente in der finalen Nutzerschnittstelle

4.8 Konfigurationsdateien

Nachdem die Konzeption für die durch MARIA XML definierten Elemente abgeschlossen ist, erfolgt in diesem Kapitel die Beschreibung solcher, die durch Maria XML nicht definiert werden können. An dieser Stelle werden die Grenzen von MARIA XML zur Beschreibung von Web2.0-Applikationen deutlich. Elemente zum Importieren von Styleinformationen und im Web2.0 oft benötigter Bibliotheken für die Validierung von Texteingaben sind beispielsweise nicht vorhanden. Zudem ist eine individuelle Konfiguration mit Angaben zur gewünschten Modalität, des Clienttyps, die Auswahl des präferierten Frameworks zur Einbindung von Widgets sowie der Verwaltung von Sessions nicht vorgesehen. Um diese Informationen bereitzustellen und zudem letztendlich die Mächtigkeit von MARIA XML festzustellen, wird

für jedes Szenario eine Konfigurationsdatei angelegt, die alle zusätzlich benötigten Elemente zum Definieren der oben genannten Angaben und Funktionalitäten enthält, die nicht mit den in MARIA XML enthaltenen Elementen beschrieben werden können.

4.8.1 Aufbau der Konfigurationsdatei

In der Konfigurationsdatei werden die für ein Szenario benötigten Parameter wie die Namen der Variablen, deren Werte zwischengespeichert werden müssen, damit sie auf Folgeseiten zur Verfügung stehen, sowie Importanweisungen, abgelegt. Mit Hilfe dieser Datei kann der Transformator generisch gehalten und die spezifischen Angaben für ein Szenario unabhängig davon definiert werden. In der Konfigurationsdatei werden die für das finale Stylesheet benötigten Parameter als String abgelegt. Im Folgenden werden die Tags der Konfigurationsdatei aufgelistet und deren Funktionen angegeben.

param	Angabe der global im Stylesheet benötigten Parameterbezeichnungen wie den Namen der aktuell angezeigten Seite oder der Modalität	
device	Angabe des verwendeten Gerätetyps, um bei der Transformation passende Interaktoren verwenden zu können	
modalitiy	Angabe der zu verwendenden Modalität beim Aufruf der Anwendung, kann im weiteren Verlauf gewechselt werden	
header	*scriptimports*	Angabe der Bibliotheken, die importiert werden müssen, um das Szenario auszuführen
	styleimports	Angabe der Dateien mit Informationen zum Aussehen der einzelnen Seiten
	cookies	Angabe aller Parameter, die innerhalb einer Session zwischengespeichert werden
	functions	Angabe der Funktionen, die auf jeder Seite eines Szenarios benötigt werden, beispielsweise zum Laden des Codes weiterer Funktionen vom Server
	variables	Angabe der Variablen, die als ein bestimmter Datentyp angelegt werden müssen wie beispielsweise Arrays, alle weiteren Variablen werden in der AUI-Beschreibung abgelegt
	frameworks	Angabe des Namens des zu verwendenden Frameworks für Widgets
	ext apps	Angabe der einzubindenden Anwendung zur Erstellung eines Mashups
data	dient zur Erzeugung eines Templates, welches die in der AUI-Beschreibung abgelegten Parameter bei der finalen Transformation in Variable in der finalen Nutzerschnittstelle überführt	
widget	Einbindung der für die Widgets benötigten spezifischen Funktionsaufrufe, welches der hier definierten Widgets verwendet wird, ist durch den Parameter *frameworks* definiert	
external app	Einbindung der für die externen Webanwendungen benötigten spezifischen Funktionsaufrufe, Auswahl der entsprechenden Anwendung über den Parameter *ext apps*	

Tabelle 10: Parameter der Konfigurationsdatei

Für diese Dateien existiert wie auch für MARIA XML ein entsprechendes Schema, welches es dem Entwickler ermöglicht, eine solche Konfigurationsdatei für ein Szenario korrekt zu erstellen. Listing 8 zeigt einen Ausschnitt aus diesem Schema. Das gesamte Schema befindet sich im Anhang. Neben der Definition des Wurzelelementes *config type* sind die Angaben zur abstrakten Beschreibung eines Widgets mit Hilfe der *widgets-type*-Definition im Codeauszug enthalten.

```
<xs:complexType name="config_type">
    <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded" name="params" type="params_type" />
        <xs:element minOccurs="0" maxOccurs="unbounded" name="devices" type="devices_type" />
        <xs:element minOccurs="0" maxOccurs="unbounded" name="modalities"
        type="modalities_type" />
        <xs:element minOccurs="0" maxOccurs="unbounded" name="data" type="data_type" />
        <xs:element minOccurs="0" maxOccurs="unbounded" name="header" type="header_type" />
        <xs:element minOccurs="0" maxOccurs="unbounded" name="widgets" type="widgets_type" />
        <xs:element minOccurs="0" maxOccurs="unbounded" name="external_apps"
        type="external_apps_type" />
    </xs:sequence>
</xs:complexType>
...
<xs:complexType name="widgets_type">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="widget" type="widget_type" />
    </xs:sequence>
    <xs:attributeGroup ref="name_attributes" />
</xs:complexType>

<xs:complexType name="widget_type">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="code" type="xs:string" />
    </xs:sequence>
    <xs:attributeGroup ref="name_attributes" />
</xs:complexType>
...
<xs:attributeGroup name="name_attributes">
    <xs:attribute name="name" type="xs:string" use="required" />
</xs:attributeGroup>
```

Listing 8: Auszug aus dem Schema zur Erstellung der Konfigurationsdatei

4.8.2 Transformator für die Konfigurationsdatei

In einem der vorherigen Kapitel der Arbeit wird bereits der generische Transformator beschrieben. In diesem sind alle Templates abgelegt, die aus den in der AUI-Beschreibung eines Szenarios definierten Elementen von MARIA XML die entsprechenden Interaktoren für die finale Nutzerschnittstelle generieren. Da die durch MARIA XML vorgegebenen Elemente allerdings nicht ausreichen, um die in der beschriebenen Konfigurationsdatei definierten Angaben abzulegen, wird diese Datei dazu eingeführt. Dadurch erfolgt eine Trennung der Elemente, die durch MARIA XML abgedeckt werden und solcher, die zusätzlich definiert werden müssen.

Um aus den Konfigurationsdateien unter Einbindung des generischen Transformators den letztendlich verwendeten Transformator zu erzeugen, ist ein weiteres Stylesheets notwendig. Damit wird aus der Konfigurationsdatei mit Hilfe des in diesem Kapitel beschriebenen Transformators der eigentliche Transformator erzeugt. Dieser Ablauf ist im Kapitel zur Systemübersicht beschrieben. Das für alle Szenarien verwendbare Stylesheet beinhaltet folgende Templates:

4.8.2.1 Root-Template

Durch dieses erste Template werden folgende Elemente im finalen Transformator erzeugt:

Element	Funktion
stylesheet	Angabe der Namensräume
output	Angabe des Ausgabeformates mittels *doctype*
include	Einbinden des generischen Transformators mit den Templates für die in MARIA XML enthaltenen Elemente
template	Anlegen eines Templates, welches das HTML-Grundgerüst für die finale Nutzerschnittstelle erzeugt

Tabelle 11: Elemente des Root-Templates der Konfigurationsdatei

4.8.2.2 globale Parameter

Die Templates für die Elemente *param*, *device* und *modality* legen im finalen Transformator entsprechende Parameter sowie Variable an und weisen diesen die in der Konfigurationsdatei abgelegten Angaben für Parameter, Clienttyp und Modalität als Werte zu.

4.8.2.3 Headerinformationen

Im Template zum Element *header* müssen die Erzeugungsregeln für die im Kapitel zum Aufbau der Konfigurationsdatei beschriebenen Elemente *scriptimports*, *styleimports*, *cookies*, *functions*, *variables, framework* und *external app* definiert werden. Innerhalb der finalen Nutzerschnittstelle werden diese Informationen im Kopfteil jeder einzelnen Seite abgelegt. Bei Webseiten entspricht dies dem *head*-Element von HTML.

scriptimports

In der Konfigurationsdatei werden unter *scriptimports* die URLs der zu importierenden Bibliotheken in Form von Strings abgelegt. Im Transformator werden dazu entsprechende *script*-Elemente erzeugt und diesen als *src*-Attribut die gespeicherten URLs zugewiesen.

```
<ms:scriptimports>
        <ms:JSscript src="lib/prototype/prototype.js" />
        <ms:JSscript src="lib/scriptaculous/scriptaculous.js" />
        <ms:JSscript src="lib/livevalidation/livevalidation.js" />
        <ms:JSscript src="dojo/dojo.js" djConfig="parseOnLoad:true, isDebug:true"/>
    </ms:scriptimports>
```

Listing 9: Auszug aus der Konfigurationsdatei

```
<xsl:for-each select="ms:scriptimports/ms:JSscript">
                <xsl:element name="script">
                        <xsl:attribute name="src">
                                <xsl:value-of select="@src" />
                        </xsl:attribute>
        </xsl:element>
</xsl:for-each>
```

Listing 10: Auszug aus dem Transformator der Konfigurationsdatei

styleimports

Für das Importieren von Dateien mit Informationen zur grafischen Gestaltung der Elemente einer Seite ist das Vorgehen ähnlich. Die in der Konfigurationsdatei ablegten Strings liefern die URL zu den Dateien mit Styleinformationen. Es wird für jede URL ein *Link*-Element

erzeugt, dessen Attribut *rel* statisch mit dem Wert *stylesheet* belegt wird. Dem Attribut *href* wird jeweils die genannte URL dynamisch zugewiesen.

cookies
Im *header*-Template findet ebenfalls die Definition der Funktion statt, welche die Werte sämtlicher Parameter, die innerhalb einer Session erzeugt werden, ausliest und für die aktuell angezeigte Seite zur Verfügung stellt. Da das Stylesheet für alle Szenarien einsetzbar sein soll, werden die Namen der für die Session benötigten Parameter in der Konfigurationsdatei abgelegt. Dies erfolgt mit Hilfe des Tags *cookies*. Die Namen werden bei der Transformation ausgelesen und in die Funktion eingefügt, so dass die Funktion selber universell einsetzbar ist.

functions
Zudem gibt es im *header*-Template das Element *functions* für weitere Funktionen, die für verschiedene Szenarien benötigt und aus diesem Grund direkt im Stylesheet abgelegt werden. Eine dieser Funktionen dient zum Laden des Codes weiterer Funktionen vom Server. Beim Aufruf wird dieser Funktion der Name der zu ladenden Funktion als Parameter übergeben. Nachfolgend stellt sie eine Anfrage an den Webservice, der als Antwort den entsprechenden Code liefert. Eine weitere Funktion extrahiert aus der Serverantwort den eigentlichen Code, indem sie alle zusätzlichen Informationen, die bei der Erzeugung der Antwort angehangen wurden, entfernt. In der Konfigurationsdatei kann durch Angabe der Funktionsnamen festgelegt werden, ob eine Übernahme dieser beiden Funktionen in den finalen Transformator erfolgen soll.

framework
Das Template des Elementes *framework* erzeugt eine Variable mit gleichem Namen und weist dieser den in der Konfigurationsdatei gespeicherten Parameter zu. Damit kann festgelegt werden, welches Framework verwendet werden soll, um beispielsweise ein Widget einzubinden.

variables
Nachdem die unter *variables* abgelegten Strings in Variable des Scriptcodes unter Verwendung des ebenfalls angegeben Datentyps erzeugt wurden, ist der Headerteil vollständig.

Damit sind alle Tags, die innerhalb des *header*-Elementes in der Konfigurationsdatei verwendet werden, beschrieben. Abschließend wird das in HTML notwendige *body*-Tag angelegt, in dem die anzuzeigenden Elemente abgelegt werden. In den folgenden Kapiteln erfolgt die Beschreibung der Elemente *data*, *widget* und *external app* und ihrer dazugehörigen Templates.

4.8.2.4 Parameter der AUI-Beschreibung
Das Template, welches durch Definition des Elementes *data* in der Konfigurationsdatei bei der Transformation verwendet wird, erzeugt im finalen Transformator wiederum ein Template. Dieses liest alle in der AUI-Beschreibung innerhalb des *data*-Elementes mit Hilfe von *complex-type*-Tags definierten Bezeichner aus und erzeugt daraus entsprechende Skriptvariable. Diese Variablen werden an dieser Stelle bei der Transformation in der finalen

Nutzerschnittstelle mit dem Wert *null* initialisiert. Konkrete Wertzuweisungen erfolgen während der Interaktion des Nutzers mit den entsprechenden Seiten. Das Anlegen der Variablen am Anfang einer Seite dient dem globalen Bereitstellen dieser für die gesamte Seite.

4.8.2.5 Einbinden von Widgets

In MARIA XML sind keine Elemente zum Einbinden von Widgets und zum Erzeugen von Mashups vorhanden. Über entsprechende Abfragen im dazugehörigen Template des generischen Transformators ist es zwar möglich, ein vorhandenes Element wie beispielsweise *object_edit* dafür zu verwenden, allerdings fehlt dann der Bezug zur Terminologie des Elementes. Zudem kann in der Konfigurationsdatei unter *framework* angegeben werden, welche Bibliothek für ein Widget verwendet oder welche Applikation bei der Erzeugung eines Mashups eingebunden werden soll. Aus diesem Grund ist es im Hinblick auf die Umsetzung von Vorteil, auch die Funktionsaufrufe zum Aktivieren möglicher Widgets und Applikationen in der gleichen Datei abzulegen. Die beiden dazugehörigen Templates werden dementsprechend im Transformator der Konfigurationsdatei definiert und nachfolgend beschrieben. Durch sie wird wiederum jeweils ein Template erzeugt und in den finalen Transformator eingefügt. In der AUI-Beschreibung müssen die Elemente *external widget* beziehungsweise *external application* eingetragen werden, um ein Widget oder die Funktionalität einer vorhandenen Web2.0-Applikation an der richtigen Stelle in die finale Nutzerschnittstelle einzubinden. Die entsprechenden Templates sind nicht im generischen Transformator vorhanden, sondern werden dynamisch entsprechend der Angaben in der Konfigurationsdatei erzeugt und in den finalen Transformator eingefügt.

Das Template zum Einbinden externer Widgets fügt dem zu erzeugenden Template Code hinzu, mit dem eine Adaption beispielsweise an das verwendete Endgerät vorgenommen werden kann. Durch diese Überprüfung wird gegebenenfalls bei Clients mit kleinem Display kein Widget angezeigt, sondern ein in der AUI-Beschreibung definierter Interaktor verwendet, der weniger Platz benötigt. Das anschließend erzeugte Element dient der Überprüfung des in dem AUI definierten Datentyps. Ist jener beispielsweise als Datum angegeben, dann wird an dieser Stelle ein Kalenderwidget geladen und angezeigt. Im nächsten Schritt wird überprüft, welcher Wert unter *framework* in der Konfigurationsdatei abgelegt wurde und dann der passende Funktionsaufruf zum Laden des entsprechenden Widgets aus dieser ausgelesen und in den finalen Transformator eingefügt. Der folgende Auszug aus dem entsprechenden Template des Transformators für die Konfigurationsdatei zeigt den Ablauf bei der Erzeugung des Templates für den finalen Transformator, mit dem die oben beschriebenen Überprüfungen durchgeführt werden.

```
<xsl:element name="xsl:if">
	<xsl:attribute name="test">$uA = 'Firefox'</xsl:attribute>
	<xsl:element name="script">
		<xsl:attribute name="type">text/javascript</xsl:attribute>
			function loadWidget(){
				if(	<xsl:element name="xsl:value-of">
						<xsl:attribute name="select">
							@data_reference
						</xsl:attribute>
					</xsl:element>
					== "date")
				{
					<xsl:for-each select="ms:widget">
					if(framework == "<xsl:value-of select="@name" />"){
						<xsl:value-of select="ms:code" />
					}
					</xsl:for-each>
				}
			}
	</xsl:element>
</xsl:element>
```

Listing 11: Auszug aus Transformator für die Konfigurationsdatei, Template für Element *widget*

Das auf diese Weise erzeugte Template bindet das definierte Widget an der Stelle in die konkrete Nutzerschnittstelle ein, an der in der AUI-Beschreibung das Element *external widget* definiert wurde.

4.8.2.6 Erzeugen von Mashups

Das Vorgehen zum Erzeugen des Templates zur Einbindung von Funktionalitäten bereits vorhandener Web2.0-Applikationen ist ähnlich. Es wird im finalen Transformator ein Template mit dem Namen *external application* erzeugt. Dieser Name entspricht auch der Bezeichnung des Elementes, welches in der AUI-Beschreibung angelegt werden muss, um ein Mashup zu erstellen. Dabei wird unter dem Attribut *id* in dem AUI definiert, welche Art der Anwendung eingebunden werden soll. Bei den Widgets wird diese Auswahl wie oben beschrieben durch den angegeben Datentyp getroffen. Handelt es sich bei der externen Applikationen beispielsweise um einen Kartendienst, dann kann als *id geoService* zur Bestimmung der Koordinaten eines Ortes oder *mapService* zur Anzeige des Ortes auf einer Karte zugewiesen werden. Eine entsprechende Überprüfung wird in das Template des finalen Transformators eingefügt. Wie auch bei den Widgets gibt es bei den vorhandenen Applikationen oft verschiedene Alternativen, die eingebunden werden können. Welche davon Anwendung finden, wird in der Konfigurationsdatei unter dem Punkt *ext apps* festgelegt. Im Transformator findet eine Überprüfung dieses Wertes statt. Anschließend wird der Funktionsaufruf, welcher in der Konfigurationsdatei unter dem Wert abgelegt ist, in den finalen Transformator eingefügt.

4.9 Ableitung von Funktionalitäten

Nachdem in den vorangegangen Kapiteln die aus Maria XML verwendeten und neu definierten Elemente zum Erstellen von AUI-Beschreibungen für Web2.0-Szenarien und die Konzeption für die Entwicklung der Transformatoren zur Generierung konkreter Nutzerschnittstellen vorgestellt wurden, liegt in diesem Kapitel der Fokus auf den Funktionalitäten, welche aus der abstrakten Definition der Interaktoren abgeleitet werden können. Im ersten Schritt wird in tabellarischer Form dargestellt, welche Funktionalitäten durch die Templates des generischen Transformators den in der AUI-Beschreibung definierten Elementen hinzugefügt werden. Erwähnt werden insbesondere die Funktionalitäten, welche sich aus den Attributwerten der einzelnen Elemente ergeben und nicht explizit im AUI definiert werden müssen. Dies ist für die sich anschließende Umsetzung bedeutsam, da die Bereitstellung einer bestimmten Funktionalität allein durch Angabe eines Attributwertes die Implementierung notwendiger Funktionen unabhängig von den verwendeten Daten eines speziellen Szenarios erfordert, so dass deren Wiederverwendbarkeit gegeben ist. Die Zielstellung ist es, den Aufwand beim Erstellen des AUI für ein Szenario möglichst gering zu halten. Aus einer kompakten abstrakten Beschreibung eines Interaktors sollen alle notwendigen Funktionalitäten abgeleitet werden.

Interaktionsmöglichkeit	**Element in AUI-Beschreibung**	**abgeleitete Funktionalitäten**
Seitenwechsel	Elementary connection / conditional conn	• Notwendigkeit der Überprüfung aus verwendeten Elementen • Durchführung der Überprüfung • Zwischenspeichern des Namens der Nachfolgeseite, dieser wird ausgelesen, wenn beispielweise der *navigator*-Interaktor die Durchführung des Seitenwechsels auslöst
Anzeige von Inhalten	text	• Format: Textabsatz, Überschrift oder Tabelle aus *id* • dynamisches Erstellen einer Tabelle • Eintragen der Werte in die Tabelle • Summenbildung
Eingabe von Inhalten	text edit	• Durchführung der Validierung • Unterbreitung von Vorschlägen
Laden von Code über Webservice	activator	• Anfrage an Webservice • Extrahieren des Codes aus der Antwort • Einbinden in aktuelle Seite
Navigation	navigator	• Typ des Navigationselemente aus *id* • einzublendender Bereich aus *name*-Attribut der *handler*-Kindelementes
Auswahl von Inhalten	single choice	• asynchrones Laden der Inhalte • Auslesen der benötigten Informationen • Eintragen der Begriffe in die Liste • Wertzuweisung in Variable • Zwischenspeicherung für Zugriff auf Folgeseiten
Multiselect	multiple choice + object edit	• Erstellen der Bereiche zum Entnehmen und Ablegen von Objekten • Laden und Einfügen der verschiebbaren Objekte • Auslösen eines Ereignisses bei Ablage eines Objektes und Speicherung
Einbinden von Widgets	external widget	• Wahl des Widgets aus Datentyp • Anpassung an User Agent • Hervorhebung bestimmter Interaktoren innerhalb eines Widgets
Einbinden bestehender Webanwendungen	external application	• Art der Anwendung aus *id* • bei mehreren Alternativen Auswahl durch Wert in Konfigurationsdatei

Tabelle 12: abgeleitete Funktionalitäten

Das *connections*-Element wird zur abstrakten Definition der **Seitenwechsel** innerhalb einer zu erzeugenden Anwendung verwendet. Für einen Seitenwechsel, vor dessen Ausführung keine Bedingung überprüft werden muss, wird im AUI das Element *elementary connection* eingesetzt. Aus dessen *id*-Attribut kann abgeleitet werden, ob der Wechsel eine der möglichen Nachfolgeseiten oder die vorhergehende Seite zum Ziel hat. Es reicht aus, innerhalb der einzelnen Präsentationen dieses Element zu verwenden und die Namen der Seiten, die von der aktuellen aus erreichbar sind, anzugeben. Die eigentliche Durchführung der Seitenwechsel, zu der das Bestimmen der entsprechenden Zielseite und die Zwischenspeicherung des Namens der aktuellen Seite gehören, übernimmt der generische Transformator. Wird im AUI als Kindelement von *connections* das *conditional-conn*-Element verwendet, erfolgt vor dem Seitenwechsel die Überprüfung einer Bedingung. Durch die Attribute dieses Elementes wird festgelegt, welchen Wert ein Parameter haben muss, damit die Bedingung erfüllt ist und die möglichen Folgeseiten werden angegeben. Die Funktionalitäten zur Durchführung der Überprüfung und der daraus resultierenden Wahl der als nächstes anzuzeigenden Seite sind durch den generischen Transformator gegeben. Dieser Ablauf muss beim Erstellen einer Anwendung nicht definiert werden. Die Verwendung der *elementary-connection-* und *conditional-conn*-Tags und die Angabe der notwendigen Attribute reichen aus, um die für die Durchführung von Seitenwechseln notwendigen Schritte daraus abzuleiten.

Die **Anzeige von textbasierten Inhalten** kann als Überschrift, Textabsatz oder Tabelle erfolgen. Welche Art der Darstellung in der konkreten Nutzerschnittstelle Verwendung findet, kann aus der *id* abgeleitet werden. Der Text für Überschriften wird aus der AUI-Beschreibung übernommen. Bei Textabschnitten können zusätzlich zu den statisch im AUI definierten Texten die Werte von Variablen ergänzt werden. Die Funktionalität zum Auslesen dieser Werte ist im generischen Transformator vorhanden und wird über das *function-reference*-Attribut eingebunden. Für die tabellarische Darstellung von Informationen müssen diese in einem Array abgelegt sein. Entsprechend der Anzahl der Elemente in diesem Array wird eine Tabelle passender Größe dynamisch erzeugt. Wird zudem beim Erzeugen einer Seite eine Variable mit der Bezeichnung *sum* initialisiert und liegen die Informationen als Name-/Wert-Paare im Array vor, erfolgt automatisch die Summenbildung aller Werte, die in dem Array abgelegt sind und die Eintragung der berechneten Summe am Ende der Tabelle.

Bei der **Eingabe von Inhalten** kann der Nutzer durch die Unterbreitung von Vorschlägen und sofortige Validierung der Eingabe unterstützt werden. Durch das Setzen der Attribute *event triggered update* beziehungsweise *needs validation* auf *true*, beginnt die Ausführung dieser Funktionalitäten, sobald der Nutzer das erste Zeichen eingegeben hat. Die Adressierung der Quelle, welche die notwendigen Informationen liefert, und die Darstellung dieser an den entsprechenden Stellen im Frontend erfolgt durch den Transformator.

Der zur Ausführung von Web2.0-Anwendungen notwendige **Skriptcode wird über einen Webservice geladen** und in die Seiten eines Szenarios an den entsprechenden Stellen eingebunden. Im AUI wird das *activator*-Element eingesetzt, um anzugeben, welcher Code benötigt wird. Durch Definition des Wertes *getFunction(„Funktionsname“)* beim Attribut *function reference* erfolgt die Auswahl des passenden Codefragmentes. Das *event*-Attribut wird mit dem Ereignis belegt, nach dessen Eintreten der Ladevorgang beginnt. Der notwendige Code zur Durchführung desselben ist durch den Transformator für die Konfigurationsdatei vorgegeben und steht auf jeder Seite eines Szenarios zur Verfügung.

Dadurch ist die Angabe weiterer Attribute, wie beispielsweise der URL des zu adressierenden Webservices im AUI nicht notwendig. Diese Informationen sind im Transformator hinterlegt. Wenn das Attribut *function reference* im AUI mit dem Wert *getFunction(„…“)* belegt ist, wird daraus abgeleitet, dass entsprechender Skriptcode benötigt wird und der Ladevorgang beginnt.

Die Elemente zur **Navigation** zwischen den Seiten eines Szenarios werden im AUI mit dem *navigator*-Element angelegt. Aus dem *id*-Attribut wird der Typ dieses Elementes abgeleitet. Mögliche Interaktionselemente für die konkrete Nutzerschnittstelle sind beispielsweise Buttons, Links oder die Spracheingabe. Neben der Funktion zur Durchführung von Seitenwechseln kann dieses Element ebenfalls zum Einblenden von Informationen einer Seite verwendet werden. Aus dem *name*-Attribut des Kindelementes *handler* wird dabei der Bereich abgeleitet, der sichtbar gemacht werden soll.

Sollen Radiobuttons oder eine Liste **zur Auswahl von Inhalten** in die konkrete Nutzerschnittstelle eingebunden werden, erfolgt die abstrakte Definition mit dem Element *single choice*. Bei einer Auswahlliste wird, wenn das Attribut *continuous update* den Wert *true* hat, das Starten des Vorgangs zum asynchronen Laden der Inhalte abgeleitet. Die universell einsetzbare Funktion zum Eintragen der geladenen Inhalte in die Liste wird durch den Transformator automatisch aufgerufen. Wird beim Element *single choice* der *id*-Wert auf *radio* gesetzt, erfolgt die Auswahl in der konkreten Nutzerschnittstelle mit Hilfe von Radiobuttons. In der abstrakten Definition muss neben der Beschriftung der einzelnen Optionen die Bezeichnung der Werte, die ausgewählt werden können, definiert werden. Die nach Auswahl durchzuführende Speicherung des Wertes in einer Variable beziehungsweise das Zwischenspeichern desselben, um ihn für die Folgeseiten zu Verfügung zu stellen, übernimmt der Transformator.

Die Möglichkeit mittels **Multiselect** mehrere Objekte auszuwählen wird abstrakt durch die Elemente *multiple choice* und *object edit* beschrieben. Der Bereich, in dem in der konkreten Nutzerschnittstelle die wählbaren Objekte angezeigt werden, wird durch Definition des *multiple-choice*-Elementes angelegt. Ist der Wert des *id*-Attributes dieses Elementes *dragDiv*, dann übernimmt der Transformator das Laden der Daten sowie das Extrahieren der daraus benötigten Informationen und fügt diese als verschiebbare Objekte in den vorgesehenen Bereich ein. Diese Funktionalität ist für verschiedene Szenarien mit unterschiedlichen Inhalten verwendbar. Die Definition des Attributwertes *dragDiv* reicht aus, um die Durchführung der genannten Schritte abzuleiten. Zum Erstellen des Bereiches, in dem die zur Auswahl stehenden Objekte abgelegt werden können, findet im AUI das Element *object edit* Anwendung. Hat die *id* dieses Elementes den Wert *dropDiv*, wird daraus abgeleitet, dass bei jedem Objekt, das der Nutzer im definierten Bereich ablegt, ein Ereignis eintritt, welches das Objekt einem Array hinzufügt, um es für den weiteren Verlauf des Szenarios zu speichern.

Das **Einbinden von Widgets** erfolgt durch Definition des Elementes *external widget* in der AUI-Beschreibung. Die Auswahl eines passenden Widgets zur Eingabe von Informationen wird aus dem unter dem *id*-Attribut definierten Datentyp abgeleitet. Die Hervorhebung einzelner Interaktoren innerhalb eines Widgets wird entsprechend der vorher getätigten Eingaben eines Nutzers automatisch vorgenommen. Die Entscheidung, ob der Client für die Anzeige eines Widgets geeignet ist, wird durch Ermittlung des verwendeten User Agents

getroffen. Diese Funktionalität ist bei Verwendung des *external-widget*-Elementes vorhanden und muss nicht explizit in der AUI-Beschreibung definiert werden. Stehen für einen Datentyp mehrere Widgets mit gleicher Funktionalität aus unterschiedlichen Frameworks zur Verfügung, kann in der Konfigurationsdatei der entsprechende Eintrag angepasst werden. Das Laden des korrekten Codes und die Anzeige des Widgets aus dem gewählten Framework übernimmt der Transformator.

Sollen Funktionalitäten **bestehender Webanwendungen eingebunden** werden, wird dazu das neu eingeführte Element *external application* in die AUI-Beschreibung aufgenommen. Aus dem Wert des *id*-Attributes wird die Art der Anwendung gewählt. Im Beispielszenario zur Positionsbestimmung wird ein Dienst zu Ermittlung der Koordinaten durch den Attributwert *geoService* und ein weiterer zur Anzeige der Position auf der Karte durch Zuweisung des Attributwertes *mapService* abstrakt beschrieben. Stehen für diese Aufgaben Dienste verschiedener Anbieter zu Verfügung, sollen diese durch Angabe ihrer Bezeichnung in der Konfigurationsdatei ausgetauscht werden können. Aus der Belegung des *enabled*-Attributes mit dem Wert *false* wird abgeleitet, dass ein Dienst nach dem Laden der Seite im Hintergrund ausgeführt wird, die Anzeige des Ergebnisses jedoch erst nach Nutzerinteraktion erfolgt.

5 Umsetzung

Nachdem die Beschreibung der Konzeption abgeschlossen ist, soll im Folgenden dargestellt werden, wie die Beispielszenarien konkret umgesetzt wurden. Die bei AJAX [5] verwendete clientseitige Skriptsprache zur Umsetzung der Funktionalität ist Javascript. Die gängigen Browser können Javascriptcode interpretieren, sofern dies nicht durch den Nutzer explizit deaktiviert wurde. Sämtliche Funktionen sind in Javascript abgelegt. Sie werden, wie in der Konzeption beschrieben, nicht direkt in die Transformatoren aufgenommen. Stattdessen wird ein Katalog erstellt, der den Code aller Funktionen enthält, um diese bei Bedarf in die Seite eines Szenarios einzufügen. Die eingebundenen Widgets und Dienste nutzen ebenfalls Javascript zur Umsetzung ihrer Funktionalität und können direkt verwendet werden.

Im Folgenden werden die **verwendeten Technologien** zur Beschreibung der abstrakten und konkreten Benutzerschnittstellen sowie die zur Erstellung der Transformatoren genutzte Sprache vorgestellt. Die Definition der abstrakten Nutzerschnittstellenbeschreibung ist durch MARIA XML in Form eines XML Schemas vorgegeben. Deshalb wird als Format zur Erstellung der AUI-Beschreibungen XML verwendet. Die Umsetzung der Transformatoren erfolgt mit XSLT. Für die Umsetzung der konkreten visuellen Benutzerschnittstelle wird, in Ergänzung zu AJAX, HTML verwendet. Anschließend wird das bei Web2.0-Applikationen zum asynchronen Laden der Daten vom Server verwendete XMLHttpRequest-Objekt [43] beschrieben. Im nächsten Schritt erfolgt die Beschreibung der für die Umsetzung des in Kapitel 4.1. konzipierten Systems getroffenen **Technologieentscheidungen**. Dabei werden die verwendeten Webserver, Servlets, Webservices und Proxies vorgestellt. Im sich anschließenden Unterkapitel wird die **Realisierung ausgewählter Funktionalitäten** beschrieben. Es werden die eingesetzten Bibliotheken, Web-Applikationen und Widgets zur Umsetzung typischer Web2.0-Anwendungen vorgestellt. Das abschließende Beispiel veranschaulicht den Ablauf der ad-hoc Transformation.

5.1 Verwendete Technologien

XML & XSLT

Die gegebene Sprache MARIA XML ist durch ein XML Schema [39] definiert. Die nach diesem Schema erstellten AUI-Beschreibungen für die einzelnen Szenarien werden entsprechend im XML-Format (Extensible Markup Language) [7] abgelegt. Die Transformation der abstrakten UI-Beschreibungen in konkrete Nutzerschnittstellen erfolgt mit Extensible Stylesheet Language Transformations (XSLT) [16]. Die Sprache zur Erstellung des generischen Transformators ist die Extensible Stylesheet Language (XSL) [40]. Ein mit dieser Sprache realisiertes Stylesheet besteht aus mehreren Templates. Für jedes in MARIA XML vorhandene und für die Szenarien verwendete Element wird ein solches Template erstellt. Da, wie im Kapitel zur Systemübersicht der Konzeption beschrieben, weitere Elemente für Interaktoren, die nicht in Maria XML definiert sind, benötigt werden, gibt es zusätzlich zur AUI-Beschreibung zu jedem Szenario eine Konfigurationsdatei. Für diese Datei wird ebenfalls XML und für den dazugehörigen Transformator XSL verwendet.

HTML & CSS

Die visuelle Umsetzung der konkreten Nutzerschnittstelle erfolgt mit der Hypertext Markup Language (HTML) [41] in der Version 4.01. Die erzeugten Seiten können in gängigen Browsern angezeigt werden. Zur Anpassung des Layouts und des Styles einzelner Elemente innerhalb der Seite werden Cascading Stylesheets (CSS) [42] verwendet. Diese

Styleinformationen sind in einer separaten Datei abgelegt, auf welche bei Bedarf im Headerteil der erzeugten Seiten verwiesen wird. Zusätzliche CSS-Definitionen für einzelne Widgets sind ebenfalls in CSS-Dateien abgelegt und werden zusammen mit den für die Widgets benötigten Bibliotheken in eine Seite eingebunden.

Daten
Sämtliche verwendeten Daten wie die Zielorte für die Flugbuchung, Unterkünfte, Adressen zu den Orten für die Positionsbestimmung und die Listen mit Produkten für das Newstickerszenario werden als XML-Dateien abgelegt. Mit dem XMLHttpRequest-Objekt [43] werden diese Daten asynchron vom Server geladen und auf Clientseite eingebunden. Dadurch ist es möglich, auf Nutzerinteraktionen zu reagieren und die entsprechenden Daten erst dann vom Server zu laden, wenn sie benötigt werden. Beim Flugbuchungsszenario wählt der Nutzer beispielsweise zuerst eine Kategorie aus, worauf dann die passende XML-Datei mit den Unterkünften geladen und diese in die bestehende Seite in eine Auswahlliste eingebunden werden. In der AUI-Beschreibung ist das Attribut *continuous update* immer dann auf *true* gesetzt, wenn bei der Erzeugung von Seiten einer Web2.0-Anwendung das XMLHttpRequest-Objekt zum Laden von Daten verwendet werden soll.

5.2 Technologieentscheidungen

Transformator
Der generische Transformator und der Transformator für die Konfigurationsdatei werden mit XSLT [16] umgesetzt. Verwendet wird dabei XSLT in der Version 2.0. Der Aufbau der Transformatoren mit den einzelnen Templates und deren Inhalt ist in Kapitel 4.7 und Kapitel 4.8.2 im Rahmen der Konzeption beschrieben.

Java und Tomcat
Auf Serverseite wird die objektorientierte Programmiersprache Java [44] verwendet. Als Webserver zur Ausführung von Javacode wird Apache Tomcat [45] in der Version 6.0 eingesetzt. Die Anfragen vom Browser werden an den lokal eingerichteten Tomcat gestellt. Die Bearbeitung der Clientanfragen mittels XSLT-Parser wird im folgenden Kapitel beschrieben.

Java Servlet mit XSLT-Parser
Die Umsetzung des serverseitigen Kontextes erfolgt mit Java Servlets [46]. Als Parser für die ad-hoc Transformation der Ausgangsdateien im XML-Format nach HTML und Javascript mittels XSLT wird Saxon in der Version 9.1.0.1 [47] verwendet. Zudem wird durch das Servlet der Navigationsfluss gesteuert, indem beim ersten Aufruf eine Session angelegt und anschließend verwaltet wird. Für die Speicherung des Zustands dieser Session findet dabei ein Cookie Verwendung. In ihm wird jeweils der Name der aktuell angezeigten Seite abgelegt. Beim nächsten Aufruf des Servlets wird, ausgehend von diesem Namen, die Nachfolgeseite bestimmt und durch Transformation ad-hoc erstellt. Die möglichen Nachfolgeseiten sind jeweils in der AUI-Beschreibung definiert. Es können mehrere Seiten angegeben werden, von denen eine bei einem Seitenwechsel in Abhängigkeit von durchgeführten Nutzereingaben erzeugt und angezeigt wird. Dadurch ist es möglich, ein Szenario nicht nur linear zu durchlaufen, sondern Verzweigungen in die Navigationsstruktur aufzunehmen. Zudem findet innerhalb des Servlets eine Überprüfung des die Anfrage

stellenden User Agents statt. Bei der anschließenden Transformation kann so auf Einschränkungen von beispielsweise mobilen Geräten mit kleinen Displays reagiert und die anzuzeigende Seite in angepasster Form generiert werden, sofern dies durch die AUI-Beschreibung vorgesehen ist.

Cookies

Wenn ein durch Nutzereingaben erzeugter Wert auf einer Seite im weiteren Verlauf des Szenarios benötigt wird, erfolgt die Zwischenspeicherung desselben vor dem Wechsel zur Folgeseite in einem Cookie unter Verwendung eines passenden Bezeichners. Beim Erstellen der Konfigurationsdatei für ein Szenario müssen alle Bezeichner, die zum Speichern verwendet werden sollen, innerhalb von *cookie*-Tags abgelegt werden. Der Transformator für die Konfigurationsdatei enthält im entsprechenden Template eine Funktion zum Auslesen der unter den definierten Bezeichnern abgelegten Werte:

```
<xsl:for-each select="ms:cookies">
…
        function readCookies(){
                if(document.cookie){
                        a = document.cookie;
                        var array = a.split(";");
                        for (var i = 0; i &lt; array.length; i++){
                        var cookie = array[i];
                        cookieName = cookie.substring(0, cookie.indexOf('='));
                        cookieValue = cookie.substring(cookie.indexOf('=')+1,cookie.length);
                        <xsl:for-each select="ms:cookie">
                                if(cookieName.match("<xsl:value-of select="@name" />")){
                                        <xsl:value-of select="@name" /> = cookieValue;
                                }
                        </xsl:for-each>
                        }
                }
        }
…
</xsl:for-each>
```

Listing 12: Auszug aus dem Transformator der Konfigurationsdatei, Template für *header*, Element *cookies*

Die Bezeichner-/Wertpaare sind in einem Array gespeichert, welches durch die Funktion ausgelesen wird. Wenn der Bezeichner eines Elementes aus dem Array mit einem unter dem *cookie*-Tag abgelegten String übereinstimmt, wird der zugehörige Wert (*cookieValue*) aus dem Array unter dem Namen des Bezeichners in einer Variablen abgelegt. Diese Überprüfung dient der Filterung von Werten, die ebenfalls mit Hilfe von Cookies zwischengespeichert werden wie beispielsweise der Name der aktuellen Präsentation, aber keine relevanten Daten für eine Seite liefern.

Proxy

Zur Umsetzung des verwendeten Newstickers und der Unterbreitung von Vorschlägen bei Texteingabefeldern ist auf Serverseite die Ausführung von PHP-Code notwendig. Wie in der Konzeption der Systemübersicht beschrieben, sollen an dieser Stelle Proxies eingesetzt werden, welche die Anfragen an einen Server übermitteln, der PHP ausführen kann. Die notwendigen Proxies sind als Java Servlets auf dem Tomcat Webserver realisiert. Diese nehmen die Anfragen vom Client entgegen, extrahieren die gegebenenfalls übermittelten Parameter und generieren anschließend eine neue Anfrage an den Apache HTTP Server [48], auf welchem PHP interpretiert werden kann. Die Antwort wird auf dem gleichen Weg zurück zum Client übertragen.

Webservice

Entsprechend der Beschreibung in der Systemübersicht wird der Code für verwendete Funktionen und Widgets nicht innerhalb des Transformators abgelegt, sondern nach Bedarf geladen und in das Zieldokument eingebunden. Um dies umzusetzen, wird ein Java Webservice erstellt, der über eine URL, welche die Namen der zu ladenden Funktionen oder des Widgets als Parameter enthält, aufgerufen wird. Die Antwort im SOAP-Format [49] enthält den Javascriptcode als String, welcher auf Clientseite extrahiert und eingebunden wird. Der nachfolgende Code ist ein Auszug aus der WSDL-Datei (Webservice Description Language) [50] des Webservices. Er beinhaltet die abstrakte Definition der zu übertragenden Daten in Form von Nachrichten (*messages*). Durch das Element *portType* wird der Ablauf der Kommunikation beschrieben. Im vorliegenden Fall ist dies der *request-response*-Typ. Der Webservice erhält eine Anfrage (*input message*) vom Client und liefert diesem eine Antwort (*output message*). Mit dem *binding*-Element wird das Datenformat für die Nachrichten festgelegt. Im Beispiel handelt es sich um SOAP [49].

```
…
<wsdl:message name="fillTableRequest">
      <wsdl:part element="impl:fillTable" name="parameters"/>
 </wsdl:message>

<wsdl:message name="fillTableResponse">
      <wsdl:part element="impl:fillTableResponse" name="parameters"/>
</wsdl:message>
…
<wsdl:portType name="WS">
      <wsdlsoap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http"/>
      <wsdl:operation name="fillTable">
               <wsdl:input message="impl:fillTableRequest"
               name="fillTableRequest"/>
               <wsdl:output message="impl:fillTableResponse"
               name="fillTableResponse"/>
      </wsdl:operation>
</wsdl:portType>
…
<wsdl:binding name="WSSoapBinding" type="impl:WS">
      <wsdl:operation name="fillTable">
          <wsdlsoap:operation soapAction=""/>
          <wsdl:input name="fillTableRequest">
              <wsdlsoap:body use="literal"/>
          </wsdl:input>
          <wsdl:output name="fillTableResponse">
              <wsdlsoap:body use="literal"/>
          </wsdl:output>
      </wsdl:operation>
</wsdl:binding>
…
```

Listing 13: Auszug aus der WSDL-Datei, Funktion *fillTable*

Die Ablage der Funktionen, die mit Hilfe des Webservices angefordert werden können, erfolgt strukturiert in einer XML-Datei. Im Folgenden ist beispielhaft der Aufruf innerhalb der Webservice-Klasse zum Laden einer Funktion *fillTable* dargestellt. In dem Fall wird anhand des Paramaters *fillTable* die Funktion identifiziert und anschließend dieser als Übergabewert an die Funktion *readXML* übergeben. Diese wiederum sucht in der XML-Datei, in der sämtliche Funktionen abgelegt sind, den passenden Knoten und liefert dessen Textinhalt und damit den eigentlichen Code.

```
public String fillTable(){
                return readXML("fillTable");
        }
```
Listing 14: Auszug aus Webservice-Klasse

```
<function name="fillTable">
        …Code…
</function>
```
Listing 15: Auszug aus XML-Datei mit Funktionen

Das Vorgehen beim Laden von Widgets ist ähnlich. Meist bestehen die Widgets jedoch nicht aus einer einzelnen Funktion, sondern aus mehreren. Um den gesamten Code dieser Funktionen durch einen Aufruf zu erhalten, wird eine zusätzliche Klasse eingeführt, die den Code aller für ein Widget notwendigen Funktionen lädt und anschließend an den Webservice zur Weiterleitung zum Client liefert. Der beschriebene Ablauf ist in Abbildung 10 dargestellt.

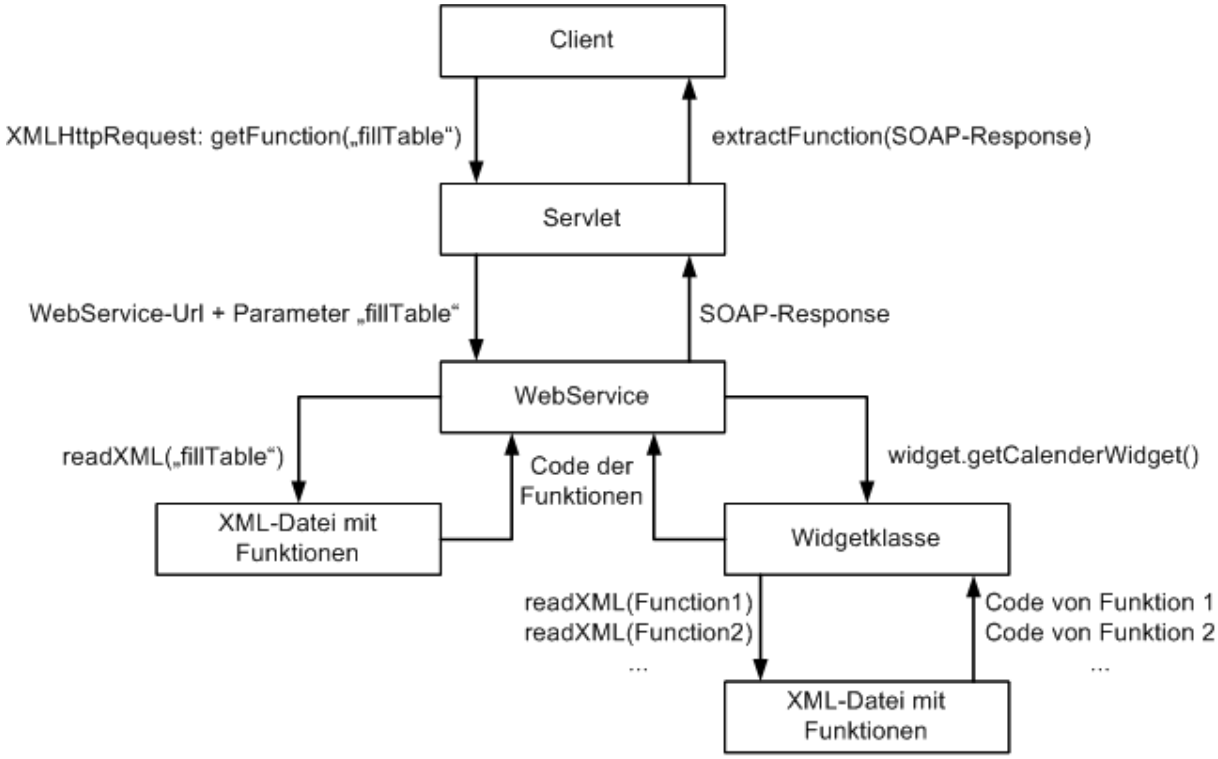


Abbildung 10: Umsetzung des Ladevorgangs für den benötigten Javascriptcode

Listing 16 zeigt beispielhaft den Aufruf eines Kalenderwigdets.

```
public String getDateWidget(){
                Widget widget = new Widget();
                return widget.getCalendarWidget();

}
```
Listing 16: Auszug aus der Webservice-Klasse

Dieses wird durch Angabe des Parameters *getDateWidget* in der Konfigurationsdatei geladen. Wenn stattdessen andere Kalenderwidgets eingebunden werden sollen, müssen

diese in der Konfigurationsdatei definiert und beim jeweiligen Aufruf der Funktion zum Laden des Codes der Übergabewert angepasst werden. Anschließend erfolgt die Auswahl des anzuzeigenden Widgets nur durch Änderungen einer Einstellung in der Konfigurationsdatei. Wenn nach Ermitteln des User Agents statt des Widgets eine vereinfachte Variante zur Auswahl eines Datums präsentiert werden soll, wird der Name einer Funktion an den Webservice übergeben, um den notwendigen Code aus dem Funktionskatalog zu laden.

Das Komponentendiagramm in Abbildung 11 zeigt die verwendeten Server, Klassen und Dateien im Überblick:

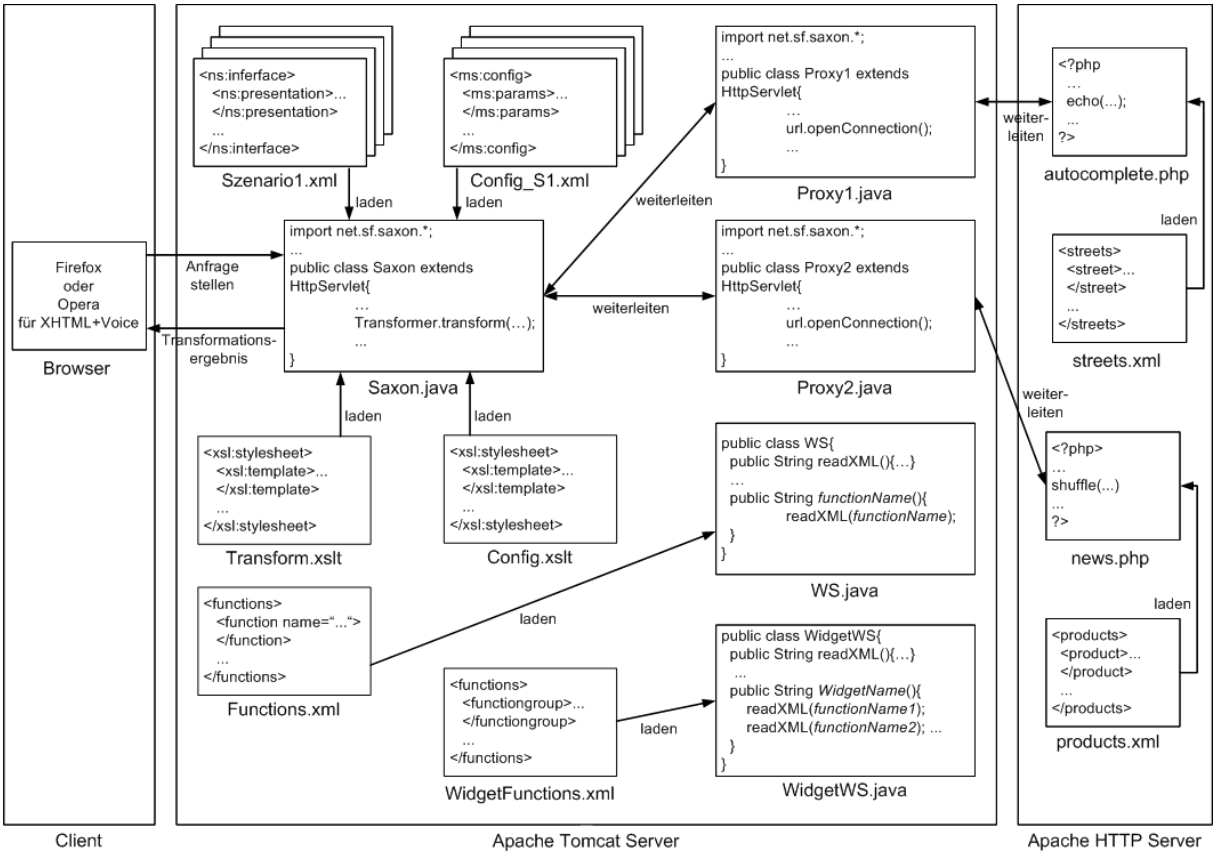


Abbildung 11: Komponentendiagramm

Name	Funktion
Szenario1.xml	AUI-Beschreibung für das erste Beispielszenario
Config_S1.xml	Konfigurationsdatei für das erste Beispielszenario
Saxon.java	Java Servlet mit Parser Saxon zur Durchführung der XSL-Transformationen
Transform.xslt	generischer Transformator
Config.xslt	Transformator für die Konfigurationsdatei
Proxy1.java	Java Servlet mit Proxyfunktionalität zur Weiterleitung der Anfragen für die Unterbreitung von Vorschlägen
autocomplete.php	PHP-Skript zum Laden der Straßennamen
streets.xml	XML-Datei mit Straßennamen
Proxy2.java	Java Servlet mit Proxyfunktionalität zur Weiterleitung der Anfragen für die Umsetzung des Newstickers
news.php	PHP-Skript zu zufälligen Auswahl von Produktlisten für den Newsticker
products.xml	XML-Datei mit Produktlisten
WS.java	Webservice zum Laden von Funktionscode
Functions.xml	XML-Datei, in welcher der Code sämtlicher Javascript-Funktionen abgelegt ist
WidgetWS.java	Webservice zum Laden von Code für die eingebundenen Widgets
WidgetFunctions.xml	XML-Datei, in welcher der Code für die Widgets abgelegt ist

Tabelle 13: verwendete Komponenten

Zusätzlich zu den dargestellten Komponenten sind für jedes weitere Szenario die Konfigurationsdatei und AUI-Beschreibung sowie CSS-Dateien mit Styleinformationen und die im folgenden Kapitel beschriebenen Javascript-Bibliotheken auf dem Tomcatserver abgelegt.

5.3 Realisierung ausgewählter Funktionalitäten

Widgets

Für das Beispielszenario zur Flugbuchung wird ein Kalenderwidget zur Auswahl des Abflugtermins eingesetzt. Verwendung finden dabei der JSCalendar [51] und der im Dojo Javascript Toolkit [52] enthaltene Dijit Calendar. Bei ersterem ist es möglich, bestimmte Tage zu kennzeichnen, wodurch diese mit einer anderen Hintergrundfarbe dargestellt werden. Dadurch können die Tage hervorgehoben werden, an denen Flüge zum vorher ausgewählten Ziel stattfinden. Die Flugziele sind in einer XML-Datei zusammen mit ihren möglichen Terminen abgelegt. Wird ein solcher im Kalender hervorgehobener Termin durch den Nutzer ausgewählt, wird dieser als Abflugtag gespeichert und mit der nächsten Seite des Szenarios fortgefahren. Klickt der Nutzer hingegen auf einen Termin, an dem kein Flug zu dem vorher gewählten Zielort stattfindet, erhält er einen Hinweis, dass an diesem Tag kein Flug möglich ist. Das Hervorheben bestimmter Termine ist beim Dijit Kalender nicht vorgesehen. Trotzdem kann man mit diesem ebenso einen dann frei wählbaren Termin als Abflugtag festlegen.

Einbindung externer Applikationen

Um ein Mashup erzeugen zu können, wurde MARIA XML um das Element *external application* erweitert. Im Beispielszenario zur Positionsbestimmung werden als externe Applikationen die Kartendienste Google Maps [1] und Microsoft Virtual Earth [2] verwendet. Mit beiden Applikationen ist es möglich, ausgehend von einer Adresse deren Koordinaten zu bestimmen und sich anschließend die Position dieser auf der Karte anzeigen zu lassen. Welche der Applikationen Anwendung findet, wird in der Konfigurationsdatei festgelegt. Auch hier ist es durch Änderung dieses einzelnen Parameters möglich, die Applikationen auszutauschen. Der Ablauf des Szenarios und die Präsentation der Koordinaten sind dabei identisch, lediglich die Karte wird je nach Applikation etwas anders dargestellt.

Validierung

Zur Validierung von Texteingabefeldern wird die Open Source Javascript-Bibliothek Livevalidation [53] eingesetzt. Nach dem Einbinden dieser Javascript-Datei in eine Webseite kann die Validierung von Texteingaben auf Clientseite in Echtzeit erfolgen. Dabei werden der Klasse Livevalidation die Id des zu validierenden Texteingabefeldes, die Meldung, welche angezeigt wird, wenn die Eingabe korrekt ist, und das Intervall, nach welchem die Anzeige dieser Meldung erfolgt, als Parameter übergeben. Bei der Überprüfung der Postleitzahl im Flugbuchungsszenario werden als zu überprüfende Kriterien der Datentyp und die Länge der einzutragenden Zeichenkette angegeben. Entsprechende Fehlermeldungen bei falschen Eingaben werden von der Bibliothek geliefert und müssen nicht definiert werden.

Autosuggestion

Für die Unterbreitung von Vorschlägen bei Texteingabefeldern wird die Klasse *AutoCompleter* der Javascript-Bibliothek Scriptaculous [4] verwendet. Als Übergabewerte erhält sie die Id des Texteingabefeldes und die Id eines unter dem Eingabefeld platzierten Div-Bereiches, der beim Laden der Seite vorerst nicht angezeigt wird. Weiterhin wird die URL des Servers angegeben, welcher die Anfrage entgegennimmt und eine Liste mit Textvorschlägen liefert, sowie die Anzahl der Zeichen, die mindestens eingegeben werden müssen, damit die Anzeige von Vorschlägen beginnt. Die Vorschläge sind als Strings in einer XML-Datei abgelegt und werden mit Hilfe eines PHP-Skriptes [54] ausgelesen, nachdem ein Buchstabe eingegeben und anschließend als Parameter übermittelt wurde. Da der Tomcatserver, auf dem das Servlet für die Transformation läuft, kein PHP ausführen kann, wird als URL die Adresse des in Kapitel 5.2 beschriebenen Proxies angegeben. Dieser leitet die Anfrage mit den bereits eingegeben Zeichen als Parameter weiter, nimmt die Antwort vom Apache Server entgegen und liefert die zu der eingegebenen Zeichenkette passenden Strings an den Client.

Newsticker

Zur Präsentation sich periodisch verändernder Informationen wird ein Newsticker verwendet. Für die Umsetzung der notwendigen Polling-Funktionalität eines solchen Newstickers muss dieser nicht neu programmiert werden. Stattdessen wird auf die Javascript-Bibliothek Prototype [3] zurückgegriffen. Diese beinhaltet eine Klasse *PeriodicalUpdater* [55], die eine Anfrage an den Server stellt und anschließend den Inhalt eines Containers, im konkreten Fall eines Div-Bereiches, mit dem Antworttext des Servers periodisch aktualisiert. Als Parameter werden die Id des Div-Bereiches, die URL, an welche die Anfrage gestellt wird und die Frequenz, mit der die Antworten zum Client gesendet werden sollen, angegeben. Das dazugehörige Skript auf Serverseite ist mit PHP [54]

realisiert. Es liest die in XML-Dateien abgelegten Strings ein und liefert in zufälliger Reihenfolge deren Inhalt als Antwort aus. Wie im vorherigen Abschnitt beschrieben wird als URL die Adresse eines Proxies angegeben. Dieser leitet die Anfrage weiter, nimmt die Antworten vom Apache Server entgegen und liefert deren Text an den Client.

Drag&Drop

Für die Umsetzung von Drag&Drop werden die Klassen *Draggable* und *Droppables* der Bibliothek Scriptaculous [4] verwendet. Für jedes mit der Maus verschiebbare Element wird eine Instanz der Klasse *Draggable* erzeugt, wobei als Übergabewert die Id der Elemente angegeben wird. Im konkreten Fall bestehen die Elemente aus Div-Bereichen. Zuvor erstellt werden diese einzelnen Div-Bereiche aus den in Form von Strings in einer XML-Datei abgelegten Inhalten, welche mit Hilfe des XMLHttpRequest-Objektes [43] geladen werden. Für die Definition eines Objekts in Form eines Bereiches innerhalb einer HTML-Seite, in dem die verschiebbaren Elemente durch Ablegen ausgewählt werden können, findet die Klasse *Droppables* Anwendung. Nachdem ein Element mit der Maus in den definierten Bereich gelegt wurde, erfolgt die Abspeicherung desselben in einem Array, welches anschließend für die weitere Verwendung der gewählten Elemente ausgelesen werden kann.

5.4 Beispiel zum Ablauf der ad-hoc Transformation

Abbildung 12 zeigt den Ablauf der ad-hoc Transformation zum Erzeugen einer Seite des ersten Szenarios. In Kapitel 4.1 ist diese Abbildung im Rahmen der Konzeption des Systems bereits ohne konkrete Dateinamen und -formate zu sehen. Ausgangspunkt der Transformation ist die Konfigurationsdatei [Abbildung 12/ Komponente 1] und die AUI-Beschreibung [Abbildung 12/ Komponente 6] für dieses Szenario. Der Transformationsvorgang wird für jede im AUI definierte Präsentation durchgeführt.

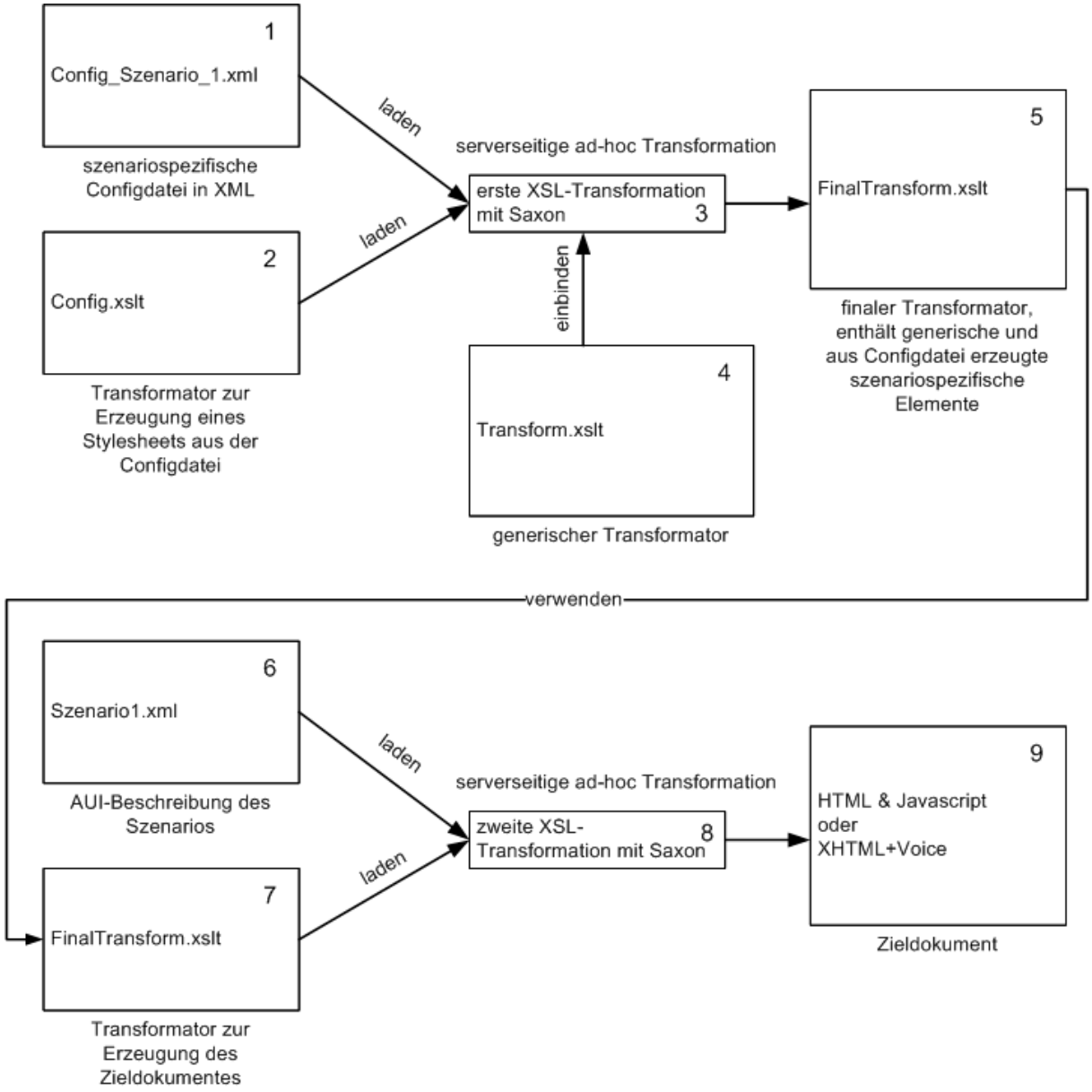


Abbildung 12: Umsetzung der Transformation innerhalb des Java Servlets

Damit die korrekten Quelldateien für die Transformation Verwendung finden, werden dem Servlet beim Aufruf als Parameter jeweils die zum Szenario 1 gehörende Konfigurationsdatei im XML-Format [Abbildung 12/ Komponente 1], der Name des Stylesheets [Abbildung 12/ Komponente 2], welches aus der Konfigurationsdatei und unter Einbindung der Templates des generischen Transformators [Abbildung 12/ Komponente 4] das letztendlich zu verwendende Stylesheets [Abbildung 12/ Komponente 5] erzeugt, übermittelt. Dieses wird nachfolgend genutzt [Abbildung 12/ Komponente 7], um mit Hilfe der AUI-Beschreibung im

XML-Format [Abbildung 12/ Komponente 6] die finalen Seiten der Anwendung zu generieren. Dieser Ablauf findet innerhalb des Servlets automatisch statt und wird für jede Seite einzeln und ad-hoc durchgeführt. Dadurch können die Parameter in der Konfigurationsdatei zur Laufzeit geändert werden. Nur die Konfigurationsdatei und AUI-Beschreibung im XML-Format sind szenariospezifisch. Die beiden Ausgangstransformatoren, *Config.xslt* und *Transform.xslt* zur Erzeugung des Stylesheets für die finale Transformation in das Format des Zieldokumentes können für alle Szenarien verwendet werden.

5.5 Multimodales Szenario

Nachdem in den vorangegangen Kapiteln die Umsetzung der visuellen Benutzerschnittstelle für Web2.0-Applikationen beschrieben wurde, erfolgt nun die Entwicklung der Transformatoren, die eine finale Benutzerschnittstelle zur akustischen Interaktion erzeugen. Der Anwender soll, nachdem er die Anwendung mit der visuellen Nutzerschnittstelle gestartet hat, die Modalität für jede Folgeseite frei wählen können.

Zur Umsetzung der Sprachein- und ausgaben wird mit XHTML+Voice [56] ein Subset von VoiceXML [32] verwendet. Im XHTML+Voice-Format erstellte Dokumente können im Opera Browser [57] interpretiert werden. Nachdem in den Einstellungen die sprachgesteuerte Bedienung aktiviert wurde, gibt Opera Sprachausgaben über die Lautsprecher wieder und nimmt Eingaben über ein Mikrofon entgegen. Für die Umsetzung des in Kapitel 2.4. vorgestellten multimodalen Szenarios ist die Teilmenge der Elemente des VoiceXML-Standards, die in XHTML+Voice definiert ist, ausreichend.

5.5.1 Definition des AUI

Der Ausgangspunkt für eine solche multimodale Applikation ist wie bei den anderen Szenarien eine abstrakte UI-Beschreibung. Für die visuelle Umsetzung werden für dieses Szenario die vorhandenen Transformatoren unverändert genutzt. Das in dem AUI verwendete *elementary-connection*-Element zur abstrakten Beschreibung der Übergänge zwischen den Seiten eines Szenarios unterscheidet anhand der vergebenen *interactor ids, forward* und *back,* ob ein Seitenwechsel zur Folgeseite oder zur vorhergehenden Seite stattfindet. In diesem Szenario wird die nächste anzuzeigende Seite jeweils aufgrund der Bestätigung durch einen Klick auf den Button mit der Beschriftung *yes* beziehungsweise durch einen Klick auf *no* ausgewählt. Die vorhandene Realisierung der Seitenwechsel im Transformator eignet sich auch für diese Art der Anwendung. Die restlichen verwendeten Interaktoren sind die *activator-*, *text-* und *navigator*-Elemente. Mit dem *activator*-Element werden benötigte Funktionen mit Hilfe des Webservices geladen und mit dem *text*-Element die Fragen des Dialogs in der abstrakten Beschreibung definiert. Das *navigator*-Element dient der Einbindung der Buttons zur Umsetzung der Seitenwechsel. Für dieses Szenario werden Funktionen benötigt, die zum Beispiel die Berechnung des Preises entsprechend der gewählten Tarifzone und Ermäßigung durchführen oder das Datum des gewählten Abreisetages bestimmen. Derartige Funktionen sind bisher im Katalog nicht vorhanden und müssen für dieses Szenario hinzugefügt werden.

Möchte der Anwender den Dialog dieses Szenarios per Sprachein- und ausgabe führen, muss die AUI-Beschreibung um ein *navigator*-Element auf jeder Seite erweitert werden, mit dem die Modalität gewechselt werden kann. Gestartet wird das Szenario im Browser mit der visuellen Nutzerschnittstelle. Auf jeder Seite ist ein Button vorhanden, der einen Wechsel

der Modalität ermöglicht. Die Abfragen werden akustisch ausgegeben, und der Nutzer antwortet per Sprachbefehl mit *yes* oder *no*. Anschließend wird die Folgeseite im Browser angezeigt, und der Anwender hat wieder die Wahl zwischen beiden Modalitäten. Auf diese Weise ist es möglich, die Zwischenspeicherung der Eingaben wie in den anderen Szenarien mit Hilfe von Cookies umzusetzen. Durch den Wechsel zurück zur visuellen Bedienoberfläche kann der Name der aufzurufenden Funktion, die aus der durch die Spracheingabe erfolgten Antwort abgeleitet wird, als an die URL angefügter Parameter übergeben und vor dem Laden der Folgeseite ausgeführt werden.

5.5.2 Transformatoren für XHTML+Voice

Um die Templates für den Transformator zur Generierung der in einem XHTML+Voice-Dokument vorkommenden Elemente erstellen zu können, wird im ersten Schritt der generelle Aufbau eines mit XHTML+Voice realisierten Dialoges analysiert.

Aufbau eines XHTML+Voice-Dokumentes

```
<html>
      <head>
      <form>
            <field name="">
                  <grammar>…</grammar>
                  <prompt>… </prompt>
            </field>
            <filled>
                  <prompt>…</prompt>
                  <if cond="…">
                        <assign name="" expr="URL&value=functionName1" />
                  <else />
                        <assign name="" expr="URL&value=functionName2" />
                  </if>
            </filled>
      </form>
      </head>
      <body>
            <button>start</button>
      </body>
</html>
```

Listing 17: Aufbau eines XHTML+Voice-Dokumentes

Das Grundgerüst einer XHTML+Voice-Seite besteht aus den *html*-Tags und den darin enthaltenen *head*- und *body*-Elementen. In den *head*-Teil werden die für den Sprachdialog notwendigen Elemente eingebettet.
Eingeleitet wird jeder Sprachdialog mit einem *form*-Tag. Alle innerhalb dieses Tags definierten Elemente werden zur Ein- beziehungsweise Ausgabe von Informationen verwendet. Während dies bei HTML in visueller Form erfolgt, werden bei XHTML+Voice die Ausgaben akustisch über Lautsprecher wiedergegeben und Eingaben mit Hilfe eines Mikrofons entgegengenommen. Jeder aus Frage und Antwort bestehende Teil eines solchen Dialogs wird innerhalb des *field*-Tags definiert. Der unter dem Namensattribut dieses Elementes abgelegte Wert dient zum einen zur Identifizierung des Dialogteils, um diesen von einem anderen Teil aus zu adressieren und zum anderen bezeichnet der Name gleichzeitig die Variable, unter welcher eine erfolgte Eingabe als String abgelegt wird. Die innerhalb der *field*-Elemente definierten *prompt*-Tags dienen der akustischen Ausgabe von in Form von Strings definierten Texten. Wird der Nutzer durch eine solche Ausgabe zu einer Antwort aufgefordert, muss eine Grammatik definiert werden, die mögliche Antworten vorgibt. Diese

Grammatik wird in der AUI-Beschreibung durch Angabe des *feedback*-Elementes, welches bei der abstrakten Beschreibung für die visuelle Umsetung keine Verwendung findet, abgelegt und in die finale Nutzerschnittstelle als Textknoten innerhalb des *grammar*-Elementes übernommen. Nach akustischer Eingabe einer Antwort wird der passende in der Grammatik abgelegte String durch den Interpreter identifiziert und der oben beschriebenen Variablen als Wert zugewiesen. Das nachfolgende *filled*-Tag veranlasst die Aktivierung des Mikrofons und setzt automatisch einen Timer, der die Zeit festlegt, in der die Antwort durch den Nutzer erfolgen soll. Antwortet der Nutzer innerhalb dieser Zeit, wird der oben beschriebene Abgleich der Antwort mit der Grammatik durchgeführt. Darauf basierend wird nachfolgend mit Hilfe des *if*-Tags die Entscheidung getroffen, welcher Funktionsname, der entsprechend der Antwort aus dem AUI ausgelesen wird, an die URL, mit der die Adressierung des Transformators zum Generieren der Nachfolgeseite erfolgt, als Parameter angehangen wird. Wenn der Nutzer in der durch den Timer definierten Zeitspanne dem System nicht antwortet oder das System keinen passenden String zu dem Signal in der Grammatikdefinition findet, werden entsprechende Ausnahmen ausgelöst. Innerhalb der dazugehörigen *noinput*- und *nomatch*-Tags können Rückmeldungen an den Nutzer abgelegt werden.

Ausgehend von der Zielstellung, XHTML+Voice-Dokumente mit dem beschriebenen Aufbau zu erzeugen, werden die Konfigurationsdatei, der Transformator für diese und der generische Transformator entwickelt.

Konfigurationsdatei und Transformator

Die Konfigurationsdatei für XHTML+Voice basiert auf dem gleichen XML Schema, welches zum Erstellen der Konfigurationsdateien für die visuelle Nutzerschnittstelle verwendet wird. Definiert werden die Elemente *param*, *device* und *modality* sowie das *header*-Tag. Das zum letztgenannten Tag gehörige Template erzeugt im finalen Transformator wieder ein Template, welches für jede im AUI definierte Präsentation eine HTML-Seite mit einem Sprachdialog anlegt. Dazu werden die *head*- und *form*-Tags erstellt und das *body*-Tag mit dem *button*-Interaktor zum Starten der Sprachausgabe generiert. Durch die Konfigurationsdatei und den dazugehörigen Transformator werden folgenden Elemente für die finale Nutzerschnittstelle erzeugt:

```
<html>
      <head>
      <form>
            …
      </form>
      </head>
      <body>
            <button>start</button>
      </body>
</html>
```

Listing 18: aus der Konfigurationsdatei erzeugter Teil eines XHTML+Voice-Dokumentes

Im Template befindet sich innerhalb der form-Tags das *apply-templates*-Element, welches weitere Templates, die im finalen Transformator vorhanden sind, aufruft. Die Definition dieser Templates erfolgt im generischen Transformator, welcher beim Erstellen des finalen Transformators eingebunden wird.

Generischer Transformator

Nach dem durch den Transformator für die Konfigurationsdatei das Grundgerüst einer XHTML+Voice-Datei erzeugt wurde, dienen die Templates des generischen Transformators zur Erzeugung der *field*-Tags und der dazugehörigen Kindelemente:

```
<field name=“”>
      <grammar>…</grammar>
      <prompt>… </prompt>
</field>
<filled>
      <prompt>…</prompt>
      <if cond=“…“>
            <assign name=”” expr=”URL&value=functionName1” />
      <else />
            <assign name=”” expr=”URL&value=functionName2” />
      </if>
</filled>
```

Listing 19: aus der AUI-Beschreibung erzeugter Teil eines XHTML+Voice-Dokumentes

Die Grundlage dafür bildet die ebenfalls für die visuelle Umsetzung eines Szenarios verwendbare AUI-Beschreibung.

AUI-Element	Funktion des zugehörigen Templates im Transformator
relation	• Erzeugung eines *field*-Elementes • Zuweisung der *id* des *relation*-Elementes als Attributwert für *name*
feedback	• Erzeugung eines *grammar*-Elementes • Zuweisung der in der AUI-Beschreibung abgelegten Grammatikdefinition
text	• Erzeugung eines *prompt*-Elementes • Auslesen des unter *data reference* abgelegten Strings
navigator	• Erzeugung des *filled*-Elementes • Erzeugung des ersten *prompt*-Tags, um dem Nutzer die getätigte Eingabe zu bestätigen • Erzeugung des *if*-Elementes, Überprüfung mittels *cond*-Attribut, ob der Wert, der unter *data reference* abgelegten Variable mit der Abfrage übereinstimmt • Erzeugung der *assign*-Elemente, dem Attribut *expr* wird die URL des Transformators zugewiesen und der Name der aufzurufenden an die URL angefügt

Tabelle 14: AUI-Elemente für multimodales Szenario

Der vergebene *field*-Name dient bei XHTML+Voice gleichzeitig der Deklaration der Variablen, in welcher die getätigten Eingaben nach Erkennung mit Hilfe der Grammatik durch den Interpreter abgelegt werden. In dem AUI wird ein *field* durch das *relation*-Tag beschrieben, da innerhalb dieses Elementes alle weiteren Elemente wie beispielsweise *text* definiert werden können. Dadurch ist die Verschachtelung der Tags in dem AUI mit der für XHTML+Voice erforderlichen übereinstimmend. Die Bestätigung der Erkennung der durch den Nutzer getätigten Spracheingabe erfolgt innerhalb des *filled*-Elementes, welches durch

die Definition des *navigator*-Elementes in der AUI-Beschreibung erzeugt wird. Für die *relation*- und *navigator*-Elemente existiert im Transformator jeweils ein eigenes Template. Der aus dem *name*-Attribut des *relation*-Tags erzeugten Variablen wird nach erfolgter Eingabe der durch den Erkenner mit Hilfe der Grammatik ermittelte Wert zugewiesen. Durch das nachfolgende *if*-Tag wird überprüft, ob diese Variable den Wert *yes* hat. Der Code zur Durchführung der Überprüfung wird durch das zum *navigator*-Element gehörige Template erzeugt. Damit die Namen der Variablen in beiden Templates übereinstimmen, wird in der AUI-Beschreibung beim *navigator*-Element das Attribut *data reference* ergänzt und diesem der schon beim *name*-Attribut des *field*-Elementes verwendete Bezeichner zugewiesen. Das *data-reference*-Attribut wird bei der Transformation zur Erzeugung der visuellen finalen Nutzerschnittstelle nicht benötigt. Der Transformator liest diesen Attributwert nicht aus, so dass die AUI-Beschreibung für beide Modalitäten verwendbar bleibt. Der folgende Code ist ein Auszug aus dem Template für das *navigator*-Element:

```
<xsl:element name="if">
        <xsl:attribute name="cond">
                <xsl:value-of select="@data_reference" />=='yes'
        </xsl:attribute>
        <xsl:element name="assign" xmlns="http://www.w3.org/2001/vxml">
                <xsl:attribute name="name">
                        window.location
                </xsl:attribute>
                <xsl:attribute name="expr">
                        'http://localhost:8080/XSLT/…&amp;value=
                        <xsl:value-of select="@function_reference" />'
                </xsl:attribute>
        </xsl:element>

<xsl:element name="else" />
        …
</xsl:element>
…
```

Listing 20: Auszug aus Transformator für XHTML+Voice, Template für AUI-Element *navigator*

Nach Aufruf des Transformators über die beim *assign*-Element angegebene URL ruft dieser die aktuelle Seite erneut in der visuellen Modalität auf und führt die angegebene Funktion aus. Durch den Funktionsaufruf wird die passende Präsentation zur Generierung der Folgeseite ausgewählt. Diese wird in visueller Form im Browser angezeigt. Ein Wechsel zur akustischen Modalität ist über den entsprechenden Button möglich.

Alternative zur XHTML+Voice

Neben der Umsetzung mit XHTML+Voice und dem Einsatz von Opera zum Interpretieren des Codes kann alternativ der von Dirk Schnelle-Walke entwickelte Open Source VoiceXML-Interpreter JVoiceXML [58] verwendet werden. Wird dieser eingesetzt, kann ein Großteil der im VoiceXML-Standard [32] definierten Elemente verwendet werden. Da der Interpreter auf Java basiert, kann die Interpretation des VoiceXML-Codes auf dem für die Durchführung der Transformationen mit dem XSLT-Parser Saxon schon vorhandenen Tomcatserver erfolgen. Von Nachteil ist in diesem Fall, dass kein direkter Zugriff auf den Interpreter vom Browser aus möglich ist. Die Ausführung der definierten Sprachdialoge kann entweder über ein Ant-Skript [59] oder über einen mit Java implementierten Client erfolgen. Zudem ist die Qualität des Spracherkenners bei JVoiceXML nicht ausreichend, um eine beliebig große Auswahl an Antwortmöglichkeiten, die in einer Grammatik definiert werden können, korrekt zu unterscheiden. Durch die Beschränkung der Eingaben auf *yes* und *no* beim Beispielszenario kann dieses mit JVoiceXML ausgeführt werden. Da jedoch ein Wechsel der Modalität vom

Browser aus nicht möglich ist, wurde zur Umsetzung dieses Szenarios XHTML+Voice verwendet.

Im Rahmen dieses Kapitels wurden die verwendeten Technologien und die Umsetzung des konzipierten Systems vorgestellt. Als Format zur Definition der abstrakten Beschreibungen, Konfigurationsdateien und der Daten, die für ein Szenario benötigt werden, wird XML [7] verwendet. Die Transformatoren bestehen aus XSL-Templates [40]. Die Umsetzung der finalen visuellen Nutzerschnittstelle erfolgt mit HTML [41] und Javascript [6]. Für die akustische Umsetzung des FUI wird mit XHTML+Voice [56] ein Subset des VoiceXML-Standards [32] verwendet. Zur Durchführung der Transformationen findet der in ein Java Servlet [46] eingebundene XSLT-Parser Saxon [47] Anwendung. Die dazugehörige Ausführungsumgebung ist ein Apache Tomcat Server [45]. Für den Zugriff auf Funktionalitäten, die auf einem externen Server abgelegt sind, werden als Java Servlets realisierte Proxykomponenten verwendet. Zum Laden des Codes von benötigten Funktionen, welcher in das FUI eingebunden wird, wird ein Java Webservice eingesetzt. Für die Umsetzung typischer Web2.0-Interaktionsmöglichkeiten werden die Javascript-Bibliotheken Prototype [3] und Scriptaculous [4] sowie das Framework Dojo [52] verwendet. Zur Realisierung der Funktionalität zum Einbinden von Widgets werden beispielhaft zwei verschiedene Kalenderwidgets genutzt. Die Erzeugung von Mashups wird unter Verwendung der Kartendienste Google Maps [1] und Microsoft Virtual Earth [2] umgesetzt. Abschließend werden in diesem Kapitel die Entwicklung der Transformatoren für die Generierung von FUIs im XHTML+Voice-Format, welche die akustische Interaktion ermöglichen und die Umsetzung eines multimodalen Szenarios beschrieben.

6 Evaluierung

In diesem Kapitel werden die im Rahmen dieser Arbeit entwickelte Konzeption und die prototypische Umsetzung evaluiert. Gezeigt wird dabei im ersten Schritt, inwieweit das erstellte System und die Transformatoren wiederverwendet werden können und für die **Umsetzung weiterer Szenarien** geeignet sind. Dazu wird die AUI und Konfigurationsdatei für ein Beispielszenario erzeugt und anschließend die in zwei Schritten ablaufende ad-hoc Transformation durchgeführt. Zudem wird anhand des Szenarios eine Gegenüberstellung der erzeugten Interaktoren und ihrer abstrakten Beschreibungen vorgenommen.

Eine allgemeine Darstellung der Funktionalitäten, die aus der AUI-Beschreibung abgeleitet werden sollen, erfolgt bereits im Rahmen der Konzeption im Kapitel 4.9 der Arbeit. Im nächsten Schritt der Evaluierung wird anhand von Beispielen analysiert, inwieweit es gelungen ist, benötigte **Funktionalitäten aus der AUI-Beschreibung abzuleiten**. Dazu wird nicht jedes in der abstrakten Beschreibung vorkommende Element erneut beschrieben, sondern die Evaluierung an solchen durchgeführt, bei denen besonders viele beziehungsweise wenige Funktionalitäten abgeleitet werden können. An einem Beispielelement wird der Umfang des Codes zur abstrakten Beschreibung desselben mit dem für die finale Nutzerschnittstelle verglichen und erläutert, welche Relevanz der Umfang der AUI-Beschreibung hat. Ziel des Kapitels ist es zu zeigen, dass ein sinnvoller Kompromiss zwischen der generischen Gestaltung des Transformators und dem Umfang der AUI-Beschreibung gefunden wurde.

Anschließend erfolgt eine Übersicht über die aus der gegebenen Sprache verwendeten Elemente und die Aufstellung der **Elemente und Attribute, um die MARIA XML erweitert wurde**.

Im nächsten Schritt wird evaluiert, inwieweit die zum Anfang der Arbeit in Kapitel 2.5. aufgestellten **Anforderungen durch das prototypisch umgesetzte System erfüllt werden**.

Abschließend erfolgt eine **Effizienzmessung**, durch welche der durch die ad-hoc Transformationen verursachte zeitliche Mehraufwand zum Anzeigen einer angeforderten Seite ermittelt wird.

6.1 Evaluierungsszenario

Zur Überprüfung der Eignung der Transformatoren für weitere Webanwendungen, die nicht vor der Konzeption und Umsetzung definiert wurden, wird die AUI-Beschreibung für ein weiteres Szenario erstellt. Ausgehend von dieser Beschreibung soll es unter Verwendung der bestehenden Transformatoren möglich sein, die konkrete Nutzerschnittstelle zu erstellen.

6.1.1 Beschreibung des Szenarios

Die zu generierende Webanwendung soll es beispielsweise einem Lieferunternehmen erlauben, sich aus vorhandenen Kundenaufträgen mehrere mittels Drag&Drop zur Bearbeitung auswählen zu können. Anschließend werden dem Lieferant die zugehörigen Kundendaten einzelner Aufträge angezeigt. Er soll Zugriff auf die Kartendienste Google Maps [1] und Microsoft Virtual Earth [2] haben, um sich nachfolgend die Lieferadressen auf der Karte darstellen zu lassen. Über ein Kalenderwidget legt er das Datum fest, an dem die Auslieferung der zuvor ausgewählten Aufträge erfolgt und bestätigt damit die Annahme des Auftrages. Abschließend werden die getätigten Eingaben zusammengefasst dargestellt.

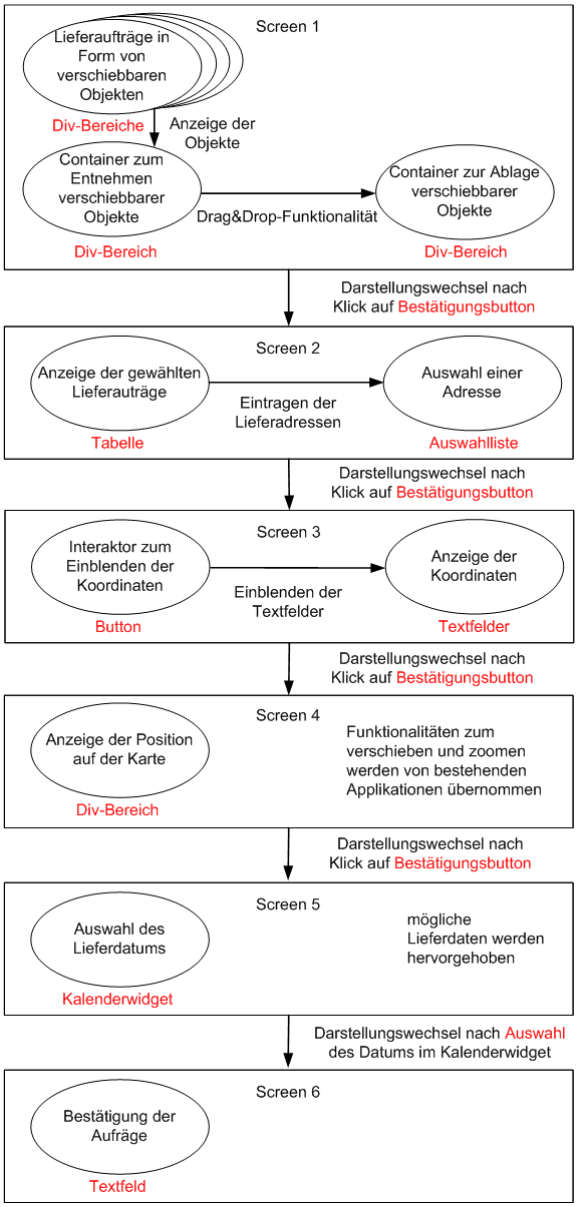


Abbildung 13: Ablauf und Interaktoren des Evaluierungsszenarios

6.1.2 Umsetzung des Szenarios

Die finalen Nutzerschnittstellen der am Anfang der Arbeit vorgestellten Szenarien wurden zuerst unabhängig von der späteren AUI-Beschreibung mit HTML und Javascript umgesetzt. Ziel dieses Vorgehens war es neben dem Definieren der Anforderungen aus dem Code der einzelnen Seiten abzuleiten, welche Interaktoren für die Umsetzung notwendig sind und wie

diese durch Transformation erzeugt werden können. Für dieses Szenario wird das AUI definiert ohne den Code der konkreten Nutzerschnittstelle zu kennen, und die vorhandenen Transformatoren werden verwendet und evaluiert. Zudem sollen die bestehenden Funktionen unverändert wiederverwendbar sein. Für die Drag&Drop-Funktionalität auf der ersten Seite des Szenarios werden, wie in der Konzeption beschrieben, in die AUI-Beschreibung die Elemente *multiple choice* und *object edit* aufgenommen. Die dazugehörigen Funktionen zum Einfügen der wählbaren Elemente in einen Container und zur Speicherung der per Drag&Drop getroffenen Auswahl können an dieser Stelle ohne Änderungen über den Webservice geladen und eingesetzt werden. Auf der Folgeseite erfolgt die Darstellung der Aufträge in Tabellenform. Abstrakt beschrieben wird die Tabelle durch das Textelement, welches die *id table* erhält. Die Funktion zum automatischen Einfügen der Informationen in eine Tabelle wird im Newstickerszenario bereits verwendet, um die gewählten Produkte und ihre Preise anzuzeigen, wobei am Ende der Tabelle die berechnete Summe der Preise angezeigt wird. Sie kann an dieser Stelle wiederverwendet werden, allerdings entfällt das Summieren der Einträge in der letzten Spalte, da die entsprechende Variable für die Summenbildung nicht definiert ist. Auf der gleichen Seite wird eine Auswahlliste angeboten, die sämtliche Adressen, zu denen die Auslieferungen erfolgen sollen, enthält. Die Funktion zum Extrahieren der Adressinformationen aus der XML-Datei, in der sie zusammen mit den Auftragsbezeichnungen abgelegt sind, muss an dieser Stelle neu erstellt werden. Sie realisiert das Aufteilen des Adressstrings, um die Werte für Straße, Nummer und Stadt einzelnen Variablen zuzuweisen. Eine solche Funktion, in der eine direkte Abhängigkeit zwischen den benötigten Bezeichnungen der Variablen und deren Inhalt besteht, lässt sich nicht allgemein definieren. Zum Eintragen der Adressinformationen in die Auswahlliste existiert eine Funktion, die in den vorhandenen Szenarien bereits mehrfach verwendet wird und auch hier unverändert eingebunden werden kann. Die oben beschriebene Speicherung der Adressinformation in einzelne Variable ist für die nach Auswahl einer Adresse stattfindende Ermittlung der Koordinaten und die Anzeige der Position auf der Karte notwendig. Dadurch ist es möglich, die definierten Kartendienste Google Maps [1] und Microsoft Virtual Earth [2] nur durch Definition des Elementes *external application* in der AUI-Beschreibung und Konfigurationsdatei einzubinden. Durch Einfügen des Elementes *external widget* und Zuweisung des Datentyps *date* in die abstrakte Benutzerschnittstellenbeschreibung finden die Kalenderwidgets zur Auswahl des Lieferdatums Anwendung. Die zu den Kartendiensten und Widgets gehörenden Funktionen können für dieses Szenario unverändert wiederverwendet werden. Für die zusammenfassende Darstellung der getätigten Angaben existiert ebenfalls eine Funktion, welche die den entsprechenden Variablen zugewiesenen Werte ausliest und anzeigt.
Die Umsetzung dieses Szenarios zeigt, dass an den Transformatoren keine Änderungen oder Erweiterungen notwendig sind, um mit diesen weitere Szenarien ausgehend von einer AUI-Beschreibung und Konfigurationsdatei zu generieren. Die erstellten Funktionen sind ebenfalls wiederverwendbar. Nur für das Extrahieren der Adressinformation muss eine neue Funktion eingefügt werden, da an dieser Stelle eine direkte Abhängigkeit zum Inhalt besteht.
Im Folgenden werden die für das Evaluierungsszenario erzeugten Interaktionsmöglichkeiten (Abbildung 14 - Abbildung 20) und die dazugehören abstrakten Beschreibungen gegenübergestellt.

multiple choice + object edit **Drag&Drop**

```
<ns:multiple_choice id="dragDiv">
       <ns:events>
       <ns:selection_change>
       <ns:handler>
       <ns:change_property>
       <ns:conditions>
       <ns:condition />
       </ns:conditions>
       </ns:change_property>
       </ns:handler>
       </ns:selection_change>
       </ns:events>
</ns:multiple_choice>

<ns:object_edit id="dropDiv"
       <ns:events>
       <ns:value_change>
       <ns:handler>
              <ns:raise event_name="onDrop" />
       </ns:handler>
       </ns:value_change>
       </ns:events>
</ns:object_edit>
```

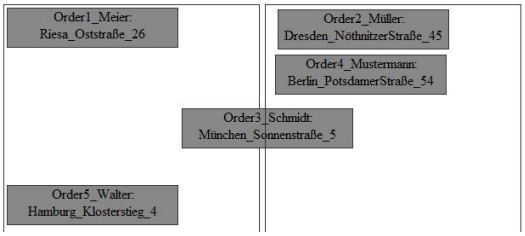


Abbildung 14: Umsetzung von Drag&Drop

Zur abstrakten Definition der Drag&Drop-Funktionalität werden die Elemente *multiple choice* und *object edit* verwendet. Im Transformator wird anhand der *ids* dieser beiden Elemente identifiziert, dass die Interaktion mittels Drag&Drop erfolgen soll. Da die Funktionen zum Eintragen der verschiebbaren Objekte in den linken, durch *multiple choice* angelegten Bereich und zum Aufnehmen der Objekte in den rechten, durch *object edit* definierten Bereich so implementiert sind, dass sie für verschiedene Inhalte wiederverwendet werden können, ist deren Aufruf direkt im Transformator festgelegt. Dadurch ist es nicht notwendig, die Namen der Funktionen in die AUI-Beschreibung aufzunehmen. Zudem kann der Aufruf zum Laden des Funktionscodes vom Server mit Hilfe des Webservices unmittelbar vor Verwendung erfolgen. Es müssen in der AUI-Beschreibung keine *activator*-Elemente angelegt werden, um den Code der Funktionen bereits beim Laden der Seite vom Server anzufordern. In der konkreten Nutzerschnittstelle wird für beide Elemente ein Div-Bereich angelegt, der die in der AUI verwendete *id* zur Identifikation erhält. Die Bezeichnungen der verschiebbaren Objekte sind in einer XML-Datei abgelegt. Diese wird asynchron mittels XmlHttpRequest [43] vom Server geladen. Nachfolgend werden die Namen der Objekte extrahiert und für jedes ein Div-Bereich erzeugt, der verschoben und im benachbarten Container abgelegt werden kann. Platziert der Nutzer eines der Objekte darin, löst dies das Ereignis *onDrop* aus, woraufhin der Name des Objektes in ein Array eingetragen wird.

text **Tabelle zur Anzeige von Inhalten**

```
<ns:text id="table"
       data_reference="elementArray" />
```

Order1_Meier:	Riesa_Oststraße_26
Order2_Müller:	Dresden_NöthnitzerStraße_45
Order3_Schmidt:	München_Sonnenstraße_5
Order4_Mustermann:	Berlin_PotsdamerStraße_54
Order5_Walter:	Hamburg_Klosterstieg_4

Abbildung 15: Umsetzung einer Tabelle zur Anzeige von Inhalten

Wird dem *text*-Element die *id table* zugewiesen, erzeugt der Transformator eine Tabelle zur Darstellung von Informationen. Die Voraussetzung für das dynamische Erzeugen der Tabelle ist die vorherige Speicherung der Inhalte in einem Array. Unter der Bezeichnung *elementArray* liegt ein solches vor. Die Elemente wurden auf der vorherigen Seite durch Auswahl mittels Drag&Drop in das Array eingefügt. Zum Auslesen und Eintragen der Elemente in die Tabelle wird die Funktion *fillTable*, die unabhängig vom Inhalt verwendet werden kann, eingesetzt.

single choice **Auswahlliste**

```
<ns:single_choice id="select"
        continuous_update="true"
        continuous_update_function="
                extractAddress()"
        data_reference="orders_1"
        property_name="tagName"
        property_value="address" >
        <ns:events>
        <ns:selection_change>
        <ns:handler name="selectList">
                <ns:raise event_name="onclick" />
       </ns:handler>
       </ns:selection_change>
       </ns:events>
</ns:single_choice>
```

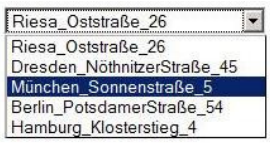


Abbildung 16: Umsetzung einer Auswahlliste

Die zur Auswahl stehenden Adressangaben sind in einer XML-Datei abgespeichert. Durch Setzen des Attributwertes von *continuous update* auf *true* wird ein XMLHttpRequest [43] ausgeführt. Damit wird die XML-Datei, deren Name unter dem Attribut *data reference* abgelegt ist, asynchron im Hintergrund vom Server geladen, die notwendigen Daten daraus extrahiert und in die Liste eingefügt. Die für das Extrahieren der Daten notwendige Funktion *fillLists* wird nicht explizit angegeben, da sie unabhängig vom Inhalt verwendet werden kann und der Aufruf dieser Funktion automatisch erfolgt, wenn eine Auswahlliste erzeugt wird. Allerdings sollen meist nicht alle in den XML-Dateien vorhandenen Daten in die Liste übernommen werden, sondern nur Einträge, die innerhalb bestimmter Tags abgelegt sind. Unter dem Attribut *property name* wird deshalb angegeben, dass im konkreten Fall nur die *address*-Informationen von Interesse sind. Wählt der Nutzer einen Eintrag aus der Liste, wird das Ereignis *onclick* ausgelöst und mit der unter *continuous update function* abgelegten Funktion fortgefahren. Diese speichert die Einzelangaben Straße, Hausnummer und Ort der gewählten Adresse in entsprechende Variablen und erzeugt die passenden Cookies, mit deren Hilfe die Werte zusammen unter entsprechenden Bezeichnern zur Verwendung auf den Folgeseiten abgelegt werden.

external application

```
<ns:navigator id="activate"
        value="show coordinates" >
        <ns:events>
        <ns:activation>
        <ns:handler name="geoService">
                <ns:raise event_name="onclick" />
        </ns:handler>
        </ns:activation>
        </ns:events>
</ns:navigator>

<ns:text id="coordinates" />

<ns:external_application id="geoService"
        enabled="false">
        <ns:events>
        <ns:value_change>
        <ns:handler>
                <ns:raise/>
        </ns:handler>
        </ns:value_change>
        </ns:events>
</ns:external_application>
```

Ermitteln der Koordinaten

Abbildung 17: Umsetzung der Koordinatenermittlung

Bei der Verwendung einer Applikation zur Ermittlung der Koordinaten einer Adresse wird davon ausgegangen, dass die zur Adresse gehörigen Angaben in den Variablen für Straße, Hausnummer und Ort abgelegt sind. Bei diesem Szenario wurden diese Angaben auf der vorgergehenden Seite in einem Cookie abgelegt und stehen zur Verfügung. Der Wert *geoService* des *id*-Attributes weist den Transformator an, in die konkrete Nutzerschnittstelle einen Kartendienst einzubinden. Mit einer der Applikationen Google Maps [1] oder Microsoft Virtual Earth [2] werden bei diesem Szenario die Koordinaten ermittelt. Die abstrakte Definition mittels *external-application*-Element erfolgt unabhängig von der verwendeten Applikation. Durch Setzen des Attributes *enabled* auf *false* wird festgelegt, dass die Koordinaten zwar beim Laden der Seiten ermittelt werden, die Anzeige derselben jedoch erst erfolgt, nachdem der Nutzer dies durch Interaktion mittels eines gesondert definierten *navigator*-Elementes aktiviert.

external application

```
<ns:external_application id="mapService"
        enabled="true">
        <ns:events>
        <ns:value_change>
        <ns:handler>
                <ns:raise/>
        </ns:handler>
        </ns:value_change>
        </ns:events>
</ns:external_application>
```

Anzeigen der Adresse auf der Karte

Abbildung 18: Umsetzung der Anzeige einer Adresse auf der Karte

Das *external-application*-Element wird auch zur abstakten Definition für die Einbindung von Kartendiensten verwendet. Im Gegensatz zur oben beschriebenen Koordinatenermittlung wird dem *id*-Attribut der Wert *mapService* zugewiesen. Die Entscheidung für eine der Applikationen Google Maps [1] oder Microsoft Virtual Earth [2] trifft der Anwender in der Konfigurationsdatei. Die Änderung dieses Parameters ist zur Laufzeit möglich. Zur Anzeige einer Adresse auf der Karte müssen zuvor deren Koordinaten ermittelt werden. Aus diesem Grund muss vor der abstrakten Definition des Kartendienstes die Applikation zum

Bestimmen der Koordinaten im AUI definiert werden.

external widget

```
<ns:external_widget
       data_reference="'date'"
       id="calendar-container">
       <ns:events>
       <ns:value_change>
       <ns:handler name="dateSelect">
              <ns:change_property  />
        </ns:handler>
       </ns:value_change>
      </ns:events>
</ns:external_widget>
```

Einbinden des Kalenderwidgets

Abbildung 19: Umsetzung eines Kalenderwidgets

Der Aufwand zur abstrakten Beschreibung der Einbindung eines Widgets ist ähnlich gering wie bei der Kombination eines Szenarios mit einer vorhandenen Webapplikation zu einem Mashup. Im AUI wird das *external-widget*-Element dazu eingesetzt. Dessen Attribut *data reference* hat im konkreten Fall den Datentyp *date* als Wert. Aus diesem Datentyp wird abgeleitet, dass ein Widget in die konkrete Nutzerschnittstelle einzubinden ist, welches die Eingabe eines Datums ermöglicht. Der unter dem *id*-Attribut abgelegte Wert wird dem Containerelement, welches das Kalenderwidget aufnehmen soll, als *id* zur Identifizierung übergeben. Für dieses Szenario stehen der JSCalendar [51] und das im Dojo-Framework [52] enthaltene Kalenderwidget zur Verfügung. Die Auswahl eines dieser Widgets erfolgt über den gesetzten Parameter des Framework-Tags in der Konfigurationsdatei.

text

```
<ns:text id="heading"
       data_reference="Conclusion" />
<ns:text id="text"
       data_reference="orders "
       function_reference="
              getValue(DDElements, 'orders ')" />
<ns:text id="text"
       data_reference="year of delivery date "
       function_reference="getValue(depy,
              'year of delivery date ' )" />
<ns:text id="text"
       data_reference="month of delivery date "
       function_reference="getValue(depm,
              'month of delivery date ' )" />
<ns:text id="text"
       data_reference="day of delivery date "
       function_reference="
              getValue(depd,
              'day of delivery date ' )" />
```

Zusammenfassung der Eingaben

Abbildung 20: Umsetzung der Anzeige von Inhalten

Nachdem der Lieferant die notwendigen Eingaben getätigt hat, werden diese auf der letzten Seite des Szenarios zusammengefasst. Verwendet wird das schon weiter oben beschriebene *text*-Element, allerdings diesmal mit dem *id*-Wert *text*. In der konkreten Nutzerschnittstelle erzeugt der Transformator für jedes *text*-Element ein HTML-Paragraph-Tag, in welchem der Text abgelegt wird. Der unter *data reference* abgelegte Wert wird unverändert übernommen, wohingegen die *getValue*-Funktion die Werte der übergebenen Variablen ausliest und ebenfalls anzeigt. Im konkreten Fall wird die Adresse des Lieferauftrages und das Lieferdatum ausgelesen. Die Funktion *getValue* kann für jedes beliebige Szenario zum Auslesen von Variablewerten und Anzeige derselben verwendet werden. Durch die

Weitergabe der Eingaben mit Hilfe von Cookies stehen die in Variablen abgelegten Werte auf jeder Seite zur Verfügung.

6.2 Evaluierung abgeleiteter Funktionalitäten

Im Kapitel 4.9 der Arbeit wird beschrieben, welche Funktionalitäten sich aus den im AUI definierten Attributwerten ableiten lassen. Ein Beispiel für eine komplexe Interaktionsmöglichkeit, bei welcher der Aufwand der abstrakten Definition gering ausfällt, ist Drag&Drop. Es werden lediglich die beiden Elemente *multiple choice* und *object edit* angelegt und ihnen die entsprechenden *ids* zugewiesen. Der Verzicht auf weitere Attribute zur abstrakten Beschreibung dieses Interaktionselementes ist durch die Wiederverwendbarkeit der eingesetzten Funktionen gegeben. Soll die Interaktion mit Drag&Drop erfolgen, werden, unabhängig von den vorliegenden Daten, die gleichen Funktionen mit Hilfe des Webservices geladen und zur Durchführung aufgerufen. Aus diesem Grund sind diese Funktionsaufrufe direkt im Transformator angegeben und werden nicht aus dem AUI ausgelesen. Die andere Möglichkeit ist in diesem Fall, die Namen der Funktionen im AUI anzugeben. Dann existiert im Transformator bei den Templates der Elemente *multiple choice* und *object edit* kein Bezug zur Drag&Drop-Funktionalität. Sie können für verschiedene Aufgaben eingesetzt werden, weil immer die übergebene Funktion aufgerufen wird. Allerdings wird dann keinerlei Funktionalität mehr abgeleitet und der Vorteil der abstrakten Definition, mit wenig Aufwand viel Funktionalität zu erhalten, kommt nicht zum Tragen. Wenn für jede benötigte Funktionalität ein Attribut angelegt wird, dann ist zudem die Notwendigkeit des Einsatzes verschiedener Elemente zur abstrakten Beschreibung nicht mehr gegeben. Durch die Angabe sämtlicher aufzurufender Funktionen im AUI zur Umsetzung benötigter Funktionalitäten entfällt die Semantik der Elemente zur abstrakten Beschreibung von Interaktoren. Deshalb werden die abstrakten Elementdefinitionen so einfach wie möglich gestaltet und benötigte Funktionalitäten durch den Transformator hinzuzufügt. Die Templates der beiden zur Umsetzung von Drag&Drop verwendeten Elemente im Transformator enthalten dadurch zwar einen Bezug zu dieser Interaktionsmöglichkeit, können jedoch problemlos für weitere Aufgaben ergänzt werden.
Das Verhältnis zwischen dem geringen Aufwand der abstrakten Definition und den Funktionalitäten der dazugehörigen erzeugten Interaktionselemente ist nicht bei allen Interaktoren vorhanden. Wenn aus der abstrakten Beschreibung eines *navigator*-Elementes beispielsweise ein Button erzeugt wird, dann umfassen die Attribute die Angabe des Typs, in dem Fall *button,* die Bezeichnung des Ereignisses, beispielsweise *onclick* und den Namen der Funktion, die nach Eintreten desselben aufgerufen wird. Betrachtet man die konkrete Umsetzung eines Buttons in HTML, sind die Anzahl der verwendeten Attribute und damit der Aufwand bei der Definition identisch. Die Angaben sind in diesem Fall allerdings notwendig und können nicht weggelassen und abgeleitet werden. Die durch das XML Schema von MARIA XML vorgegebene Struktur erfordert folgenden Aufbau für das *navigator*-Element im AUI:

```
<ns:navigator id="button" function_reference="collectDDElements()" value="confirm">
	<ns:events>
		<ns:activation>
			<ns:handler>
				<ns:raise event_name="onclick" />
			</ns:handler>
		</ns:activation>
	</ns:events>
</ns:navigator>
```

Listing 21: abstrakte Beschreibung eines *navigator*-Elementes

Die durch den Transformator erzeugte Umsetzung des Buttons hat den folgenden Aufbau:

```
<input type="button" onclick="collectDDElements()" id="button" value="confirm">
</input>
```

Listing 22: Umsetzung eines *navigator*-Elementes

Es sind keine Kindelemente vorhanden und die Anzahl der Attribute ist identisch. Die abstrakte Beschreibung eines Buttons ist umfangreicher als der Code der konkreten Umsetzung. Allerdings ist davon auszugehen, dass die AUI-Beschreibungen für Applikation mit MARIA XML toolbasiert erzeugt oder beispielweise aus der WSDL-Datei [50] eines Webservice generiert werden. In diesem Fall ist der Umfang der abstrakten Beschreibung eines Elementes nicht relevant, sondern es muss ein verbindliches Schema zur Generierung der Elemente vorliegen, aus denen dann konkrete Nutzerschnittstellen in verschiedenen Modalitäten und für unterschiedliche Endgeräte erzeugt werden können.
Die beiden Beispiele zeigen die Fälle, in denen sehr viel beziehungsweise sehr wenig Funktionalitäten durch den Transformator hinzugefügt werden. Bei der abstrakten Beschreibung anderer Interaktoren werden vor allem inhaltsbezogene und damit szenariospezifische Angaben in dem AUI mit Hilfe von Attributen definiert und wiederverwendbare Funktionalitäten abgeleitet. Mit dem Element *single choice* kann beispielsweise eine Auswahlliste abstrakt beschrieben werden. Das Laden der Funktion zum Extrahieren der Werte aus der XML-Datei und Eintragen dieser in die Liste wird aus der Belegung des *id*-Attributes mit dem Wert *select* abgeleitet und erfolgt automatisch. Sollen aus der XML-Datei allerdings nur die Werte, die zwischen bestimmten Tags stehen dazu verwendet werden, besteht eine Abhängigkeit zum Inhalt, und der Name der relevanten Tags muss mit Hilfe von Attributen in der abstrakten Beschreibung angegeben werden. Beim Erstellen der Templates des Transformators wird einerseits darauf geachtet, diese möglichst generisch zu gestalten, so dass sie für verschiedene Arten von Interaktionsmöglichkeiten eingesetzt werden können und andererseits aber auch eine kompakte abstrakte Beschreibung ermöglichen. Das Ziel beim Erstellen der Templates des Transformators ist es, einen sinnvollen Kompromiss zwischen den beiden genannten Anforderungen zu finden.

Beim Erstellen der abstrakten Beschreibung müssen die benötigten Funktionen vom Webservice geladen werden, bevor ihr Aufruf erfolgt. Abgesehen von den für verschiedene Szenarien verwendbaren Funktionen, die unmittelbar vor ihrer Verwendung durch den entsprechenden Interaktor geladen werden, ist für jede zu ladende Funktion zu Beginn der AUI-Beschreibung einer Seite ein *activator*-Element zu definieren. Als Ereignis, nach dessen Eintreten der Ladevorgang beginnt, wird dem Attribut *event name* der Wert *window.onload* zugewiesen, wodurch der Code der Funktionen bereits beim Laden der gesamten Seite vom Server zum Client übertragen wird. Verknüpft man den Aufruf zum Laden der Funktionen direkt mit den Interaktoren, die diese dann ausführen, werden auf einigen Seiten die gleichen Funktionen mehrfach geladen. Wird im Transformator beispielsweise im Template für das *navigator*-Element der Aufruf zum Laden der Funktion zur Durchführung eines Seitenwechsels hinterlegt, erfolgt die Übertragung des Codes für jeden Button, der auf einer Seite erzeugt wird, erneut. Um dies zu vermeiden, werden die Funktionen wie beschrieben mit dem *activator*-Element zu Beginn einmal geladen und können dann mehrfach auf einer Seite verwendet werden. Geht man davon aus, dass später die Erstellung der AUI-Beschreibungen toolbasiert erfolgt, dann ist es Aufgabe des verwendeten Tools zu erkennen, ob eine Funktion für die aktuell zu beschreibende Seite bereits geladen wird oder das

Anlegen eines *activator*-Elementes notwendig ist. Im XML Schema von MARIA XML ist ein Element mit dem Namen *external function* vorhanden, welches sich der Terminologie nach zur abstrakten Beschreibung des Aufrufes zum Laden von Funktionen anbietet. Die durch das gegebene Schema vorhandene Struktur erlaubt die Verwendung des Elementes allerdings nur außerhalb der *presentation*-Tags, welche die Szenarien in einzelne Seiten gliedern. Dadurch wird, bei Verwendung des *external function*-Elementes der Code sämtlicher Funktionen, die für ein Szenario benötigt werden, auf jeder Seite geladen. Es werden Funktionen auf einer Seite bereitgestellt, die nicht aufgerufen werden, damit mehr Transfervolumen erzeugt als unbedingt notwendig ist und die Zeit, bis eine Seite vollständig geladen ist, vergrößert.

Allgemein kann eine benötigte Funktionalität immer dann abgeleitet werden, wenn die zur Umsetzung benötigten Funktionen sich so gestalten lassen, dass sie für verschiedene Applikationen wiederverwendbar sind. Kommt eine solche Funktion zum Einsatz, dann muss ihr Name nicht explizit in dem AUI angegeben werden, um sie auszuführen. Die Zuweisung eines bestimmten *id*-Wertes oder die Belegung entsprechender Attribute mit dem Wert *true* reicht aus, um komplexe Funktionalitäten abzuleiten. Vorausgesetzt wird dazu meist, dass die zu verarbeitenden Daten in einem bestimmten Format, beispielsweise als Array bei Drag&Drop, abgelegt sind. Existiert ein direkter Bezug zum Inhalt, dann ist keine Generalisierung des Codes möglich, und der Funktionsaufruf kann nicht aus der Elementbeschreibung im AUI abgeleitet werden. In dem Fall ist es erforderlich, den Namen der aufzurufenden Funktion in der abstrakten Beschreibung explizit anzugeben. Bei der Erstellung der Templates wurde der Fokus darauf gelegt, möglichst viele Funktionalitäten durch den Transformator einem abstrakt beschriebenen Element hinzuzufügen. Dieses Vorgehen mindert zwar die Generik des Transformators, die Templates können jedoch problemlos für weitere Einsatzmöglichkeiten ergänzt werden.

6.3 Änderungen an MARIA XML

Die folgenden Elemente von MARIA XML wurden zur Umsetzung der Beispielszenarien verwendet:

verwendetes Element	**Beispiele für die Umsetzung in der finalen Nutzerschnittstelle**
interface	Wurzelelement einer AUI-Beschreibung
data	Definition von Variablen einfachen Typs
presentation, relation, connection	Seiten und Seitenwechsel
activator	Ladevorgänge, Ausführen von Funktionen
text	Überschriften, Textabschnitte, Tabellen
text edit	Texteingabefelder
single choice	Auswahllisten, Radiobuttons, Newsticker
multiple choice + object edit	Drag&Drop
navigator	Buttons
description	Angabe der Dateien für die Transformation
feedback	Grammatik für XHTML+Voice [56]

Tabelle 15: Einsatz von in MARIA XML vorhandenen Elementen

Ergänzt wurde MARIA um die folgenden Elemente:

neu erstelltes Element	**Beispiele für die Umsetzung in der finalen Nutzerschnittstelle**
external widget	Kalenderwidgets
external application	Applikationen zur Positionsbestimmung

Tabelle 16: Einsatz neu erstellter Elemente

Die Terminologie beider Elemente weist einen direkten Bezug zu dem Interaktor, der durch den Transformator für die finale Nutzerschnittstelle erzeugt wird, auf. Das Element *external widget* beschreibt zum Beispiel abstrakt die Einbindung eines beliebigen Widgets. Betrachtet man vergleichend das Element *activator*, welches in MARIA XML vorhanden ist, dann kann dort der Einsatz nicht unmittelbar aus der Terminologie abgeleitet werden. Zudem wurden in MARIA XML die in Kapitel 4.5.2 vorgestellten Attribute *validation function*, *continue function*, *needs validation* und *event triggered update* ergänzt. Durch die Erweiterung von MARIA XML um die beiden genannten Elemente und die beschriebenen Attribute weist die letztendlich in dieser Arbeit zur Beschreibung von Web2.0-Applikationen verwendete Sprache eine Tendenz zu einem CUI auf. Die zu erzeugenden Applikationen können jedoch trotzdem unabhängig von der später verwendeten Plattform beschrieben werden.

6.4 Anforderungen und ihre Umsetzung

In der folgenden Übersicht ist die Erfüllung der in Kapitel 2.5 genannten Anforderungen dargestellt. Anschließend erfolgt zu jeder der genannten Anforderungen die Beschreibung der Umsetzung anhand von Beispielen, um im Detail zu zeigen, inwieweit diese erfüllt sind.

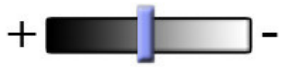
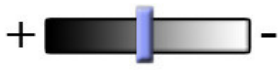
Anforderung	Grad der Erfüllung
1. Multimodalität	
2. asynchroner Datentransfer	
3. Einsatz von Widgets	
4. Web2.0-Nutzerinteraktion	
5. Einsatz von Sessions	
6. Erzeugen von Mashups	
7. Verwendung von Standards	
8. Abstraktion von der Zielplattform	
9. Verwendung einer gegebenen Sprache zur abstrakten Beschreibung	

Tabelle 17: Anforderungen und Grad der Erfüllung

Die Anforderungspunkte zwei bis sechs beschreiben Funktionalitäten, deren Einsatz typisch für Web2.0-Applikationen ist. Diese fünf Anforderungen sind erfüllt. Der Grad der Erfüllung bei den Punkten acht und neun ist hingegen nicht bei einhundert Prozent. Aus der Übersicht lässt sich demnach schlussfolgern, dass eine Zunahme der Funktionalität zu einer Abnahme der Generik bei den Transformatoren führt.

1. Multimodalität 

Neben der visuellen Umsetzung zur Interaktion mit der Anwendung über einen Webbrowser ist es möglich, vordefinierte Texte per Sprache auszugeben und durch Sprachkommandos den Verlauf der Applikation zu steuern. Während die mit HTML und Javascript erstellten Webseiten der visuellen Nutzerschnittstelle von einem beliebigen aktuellen Webbrowser, bei dem Javascript aktiviert ist, interpretierbar sind, muss für die akustische Interaktion auf den Browser Opera zurückgegriffen werden. Dieser unterstützt mit XHTML+Voice [56] ein Subset von VoiceXML und ermöglicht die Interaktion per Sprachein- und ausgabe. Um die Ausführbarkeit eines Szenarios nicht durch die Qualität des eingesetzten Spracherkenners einzuschränken, werden für die Interaktion in diesem Fall nur die Befehle *yes* und *no* verwendet. Der im Operabrowser enthaltene Spracherkenner für XHTML+Voice kann zwar auch weitere in der Grammatik definierte Worte voneinander unterscheiden, mit JVoiceXML

[58] ist dies aber beispielsweise nicht zuverlässig möglich. Die Dialoge eines Szenarios müssen entsprechend dieser Einschränkung so aufgebaut werden, dass der Anwender lediglich Entscheidungsfragen per Spracheingabe beantworten muss. Ein Wechsel der Modalität innerhalb eines Szenarios ist möglich, allerdings ist der Ausgangspunkt die visuelle Nutzerschnittstelle. Mit dem entwickelten System ist es nicht möglich rein akustische Szenarien umzusetzen, denn die Speicherung getätigter Eingaben und die von der Interaktion abhängige Wahl der nachfolgenden Seite werden durch die mit HTML und Javascript erstellten Seiten durchgeführt. Die Anforderung, eine Synchronisation zwischen beiden Modalitäten zu ermöglichen, ist erfüllt. Aus dem erkannten Sprachbefehl wird die aufzurufende Funktion abgeleitet, deren Name an die Webseite übergeben und daraus der weitere Verlauf eines Szenarios bestimmt. Andere Modalitäten wie beispielsweise die Erkennung von Gesten werden nicht betrachtet. Aufgrund der Einschränkung der Eingaben auf *yes*- und *no*-Befehle und der Abhängigkeit der akustischen Nutzerschnittstelle von der visuellen ist der Grad der Erfüllung dieser Anforderung als neutral einzustufen.

2. asynchroner Datentransfer 

Die Möglichkeit des asynchronen Datenverkehrs durch Einsatz des XMLHttpRequest-Objektes [43] wird sowohl zum Laden benötigter Daten als auch zum Transferieren von benötigtem Code vom Server zum Client eingesetzt. Voraussetzung ist, dass die Daten auf dem Server im XML-Format vorliegen. Denkbare Alternativen zur Speicherung der Daten auf dem Server sind die Formate JSON [60] und Fast Infoset [61] oder reiner Text. Das erstellte System unterstützt jedoch nur das Auslesen von im XML-Format abgelegten Daten. Das asynchrone Laden ermöglicht es, die Informationen erst anzufordern und in eine Seite einzubinden, wenn sie benötigt werden. Es müssen nicht sämtliche Daten, die auf einer Seite Anwendung finden können, unmittelbar nach dem Laden der eigentlichen Seite zur Verfügung stehen, sondern es wird aus der Nutzerinteraktion abgeleitet, welche Datensätze benötigt werden und nur diese geladen. Das gleiche Prinzip wird beim Laden des Codes der für eine Seite notwendigen Javascriptfunktionen angewendet. Durch den eingesetzten Webservice ist es möglich, auf einer Seite nur den Code der Funktionen bereitzustellen, die aufgerufen werden. Dadurch werden das Transfervolumen und die Zeit beim ersten Laden einer Seite minimiert. Da sich der Einsatz des asynchronen Datenverkehrs nicht auf die in XML-Dateien abgelegten Informationen beschränkt, sondern zudem zum Laden von Funktionscode eingesetzt wird, ist diese Anforderung erfüllt.

3. Einsatz von Widgets 

Das Einbinden externer UI-Komponenten ist nach Erweiterung von MARIA XML um ein neues Element zur abstrakten Beschreibung derselben möglich. Die Auswahl bereitgestellter Widgets erfolgt abhängig vom in der abstrakten Beschreibung angegebenen Datentyp. In der Beispielimplementierung wird ein Kalenderwidget geladen, wenn das Attribut für den Datentyp den Wert *date* hat. Es können Widgets aus verschiedenen Frameworks eingesetzt werden. Der Austausch eines Widgets ist durch Anpassung eines Parameters in der Konfigurationsdatei möglich. Zudem wird vor dem Laden eines Widgets der die Anfrage stellende User Agent ermittelt und gegebenenfalls automatisch auf die Anzeige eines Widgets verzichtet und eine alternative Interaktionsmöglichkeit eingebunden. Dies ist beispielsweise dann der Fall, wenn aus der Bezeichnung des User Agents abgeleitet werden

kann, dass die Größe des vorhandenen Displays nicht ausreicht, um ein Widget darzustellen. Da externe UI-Komponenten durch die Definition eines Elementes im AUI eingebunden werden können und zudem durch Änderung eines Parameters austauschbar sind, ist auch diese Anforderung erfüllt.

4. Web2.0-Interaktion 

In der Anforderung zur Web2.0-Interaktion steht, dass wiederkehrende Codefragmente identifiziert werden sollen, um durch Generalisierung die Wiederverwendbarkeit dieser zu gewährleisten. In die Szenarien wurden neben der Möglichkeit Objekte per Drag&Drop auszuwählen, die Unterbreitung von Vorschlägen und sofortige Validierung bei Texteingabefeldern sowie ein Newsticker beispielhaft für typische Web2.0-Interaktionsmöglichkeiten eingebunden. Die für die Umsetzung von Drag&Drop benötigten Funktionen wurden so angepasst, dass sie für beliebige Inhalte, vorausgesetzt diese sind in einem Array abgelegt, wiederverwendet werden können. Dadurch ist es möglich die Interaktion per Drag&Drop durch die Definition von zwei einfach aufgebauten Elementen in der AUI-Beschreibung bereitzustellen. Die Unterbreitung von Vorschlägen und Validierung von Texteingaben wird durch Angabe von je zwei Attributwerten bei den entsprechenden *text-edit*-Elementen im AUI realisiert. Einzige Voraussetzung zum Laden der Textbausteine, die als Vorschläge bei Texteingabefeldern oder im Newsticker angezeigt werden, ist ein Webserver, der PHP interpretieren kann. Durch die Verwendung von Proxykomponenten, welche die Anfragen vom Tomcatserver an einen Apache Webserver zur Ausführung weiterleiten, ist dies gegeben. Die Anforderung ist erfüllt, da die genannten Web2.0-Interaktionsmöglichkeiten mit wenig Aufwand abstrakt beschrieben werden können und unabhängig von den verwendeten Daten einsetzbar sind.

5. Einsatz von Sessions 

Das Erzeugen der einzelnen Seiten eines Szenarios beim Aufruf durch ad-hoc Transformation ist durch den Einsatz von Sessions möglich. Die finale Nutzerschnittstelle (FUI) wird dabei direkt aus der abstrakten Beschreibung ohne weitere Zwischenschritte erzeugt. Die Verwaltung der Sessions übernimmt ein Java Servlet, weshalb vorausgesetzt wird, dass auf Serverseite Java interpretiert werden kann. Durch das Sessionkonzept ist zudem eine flexible Ablaufgestaltung eines Szenarios möglich. Die Auswahl der nachfolgenden Seite kann in Abhängigkeit von der Nutzerinterkation erfolgen. Muss vor einem Seitenwechsel eine Bedingung überprüft werden, ist es erforderlich die benötigte Funktion zu erstellen und im Funktionskatalog abzulegen, da beim Überprüfen meist ein direkter Bezug zum Inhalt besteht und eine solche Funktion deshalb nicht wiederverwendbar ist. In die AUI-Beschreibung eingebunden wird die Funktion über ein, ergänzend zu den in MARIA XML vorhandenen, neu erstelltes Attribut. Die getätigten Eingaben werden in Cookies zwischengespeichert und stehen auf allen Folgeseiten eines Szenarios zur Verfügung. Mit der Erzeugung der Seiten durch ad-hoc Transformation und der flexiblen Gestaltung des Ablaufs eines Szenarios ist die Anforderung erfüllt.

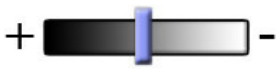
6. Erzeugen von Mashups

Als Beispiele für Web2.0-Anwendungen, die mit einem erstellten Szenario zu kombinieren sind, werden die Kartendienste Google Maps [1] und Microsoft Virtual Earth [2] verwendet. Beide Applikationen können durch Anpassung eines Parameters in der Konfigurationsdatei ausgetauscht werden. Auf die Variablen, in denen die zur Positionsermittlung notwendigen Adressdaten abgelegt sind, und die Container, in denen die Anzeige der Koordinaten und Karte erfolgt, können beide Applikationen zugreifen. Zur abstrakten Beschreibung der einzubindenden externen Applikationen wurde MARIA XML um das Element *external application* erweitert. Um anzugeben, welche Art von Applikation verwendet werden soll, wird lediglich dessen Bezeichnung dem *id*-Attribut als Wert im AUI zugewiesen. Im Rahmen der prototypischen Umsetzung der Szenarien werden in dieser Arbeit nur die Kartendienste eingesetzt. Zum Erstellen von Mashups unter Verwendung weiterer Webapplikationen ist der Transformator entsprechend zu erweitern, wobei sich an der Umsetzung der Einbindung der Kartendienste orientiert werden kann. Dass das Erstellen von Mashups mit dem entwickelten System möglich ist, wurde mit den beispielhaft verwendeten Applikationen gezeigt und somit ist auch diese Anforderung erfüllt.

7. Verwendung von Standards

Für die Umsetzung werden die durch das W3C definierten Standardsprachen XML [7], XSL [40], und HTML [41] verwendet. Die Verwendung der durch den ECMA-Standard [62] definierten Sprache Javascript [6] auf Clientseite ergibt sich aus dem Ziel, Ajax-basierte Web2.0-Applikationen zu entwickeln. Zudem sind die auf Clientseite eingebundenen Bibliotheken, wie beispielweise Prototype [3] und Scriptaculous [4], mit Javascript erstellt. Die Umsetzung der Sprachein- und ausgaben erfolgt mit XHTML + Voice [56], welches ein Subset des VoiceXML-Standards [32] ist. XML dient zur Erstellung der AUI-Beschreibungen nach dem durch MARIA XML gegebenen XML Schema [39] sowie zum Definieren der notwendigen Parameter in den Konfigurationsdateien. Zum Erstellen der Templates für die Transformatoren wird XSL [40] verwendet. Die Transformation wird mit dem XSLT-Parser Saxon [47] durchgeführt. Die Anforderung ist erfüllt, denn es werden zur abstrakten Beschreibung der Nutzerschnittstelle und bei der konkreten Umsetzung dieser keine proprietären Sprachen verwendet.

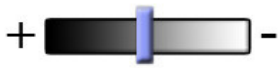
8. Abstraktion von Zielplattform



Zur Anzeige und Interaktion mit der visuellen konkreten Nutzerschnittstelle können die aktuellen Versionen der am Markt vorhandenen Browser verwendet werden. Die einzige Bedingung ist, dass der Nutzer in den Einstellungen Javascript nicht deaktiviert hat. Anpassungen des UI an den User Agent werden durch das Servlet gegebenenfalls automatisch vorgenommen. Die abstrakte Definition von Interaktoren erfolgt unabhängig von den bei der Transformation für Widgets verwendeten Frameworks. Die Entscheidung für ein solches Framework trifft der Anwender in der Konfigurationsdatei. Für die Umsetzung der Sprachein- und ausgabe mit XHTML+Voice [56] ist der Einsatz des Browsers Opera [57] erforderlich, wenn die akustische Interaktion ohne die Installation zusätzlicher Plug-ins unterstützt werden soll. Dadurch kann auf proprietäre Lösungen zur Realisierung der Sprachein- und ausgabe verzichtet werden. Der Grad der Erfüllung ist neutral. Wird die

Erfüllung der Anforderung allein an der visuellen Umsetzung gemessen, dann gilt diese als vollständig erfüllt.

9. Verwendung einer gegebenen Sprache zur abstrakten Beschreibung



Wie bereits erwähnt wurde MARIA XML zur Umsetzung der Szenarien an einigen Stellen erweitert. Die gegebenen Elemente reichen nicht aus, um beispielsweise den Einsatz von Widgets so abstrakt zu beschreiben, dass dies aus der Terminologie abgeleitet werden kann. Jedoch besteht der generische Transformator nur aus Templates für die Elemente, die in MARIA XML gegeben sind. Die Templates zur Umsetzung der neu hinzugefügten Elemente sind im Transformator für die Konfigurationsdatei definiert. Damit ist eine klare Trennung vorhanden. Durch die in zwei Schritten ablaufende Transformation werden beide Transformatoren zu einem Temporären kombiniert, mit dem alle abstrakt definierten Interaktoren in die finale Nutzerschnittstelle überführt werden. Da zum Erstellen der AUI-Beschreibungen für die Szenarien nicht ausschließlich auf durch MARIA XML vorgegebene Elemente zurückgegriffen wird, aber bei der Umsetzung durch die beschriebene Trennung erkennbar ist, welche Elemente neu hinzugekommen sind, ist der Grad der Erfüllung dieser Anforderung neutal.

6.5 Effizienzmessung

Als eine weitere Anforderung wurde die effiziente Durchführung der Transformationen definiert. In diesem Kapitel wird der Zeitaufwand für die Durchführung der ad-hoc Transformationen mit der Gesamtzeit ins Verhältnis gesetzt, die zum Übertragen und Anzeigen der einzelnen Seiten benötigt wird. Für die Messungen wird der Browser Mozilla Firefox [63] und dessen Erweiterung Firebug [64] verwendet. Die Durchführung erfolgt lokal mit den drei Szenarien, die in Kapitel 2.1 vorgestellt wurden. Die Zwischenspeicherung von Transformationsergebnissen ist auf Serverseite deaktiviert. Der Cache des Browser wird auf Clientseite vor jeder Messreihe gelöscht. Es werden jeweils fünf Messungen durchgeführt und darüber der Mittelwert gebildet und die Standardabweichung berechnet. In Tabelle 18 sind die Zeitwerte für die Durchführung der Transformationen angegeben und in Tabelle 19 die zum Laden und Anzeigen einer Seite benötigte Gesamtzeit.

Szenario1	screen1	screen2	screen3	screen4	screen5	screen6
Zeit in ms	52	56	69	52	52	51
	61	67	60	56	56	49
	54	52	49	52	51	50
	61	48	51	51	49	48
	51	49	51	55	50	45
Mittelwert in ms	55,8	54,4	56	53,2	51,6	48,6
Standardabweichung	4,86826458	7,70064932	8,42614977	2,16794834	2,70185122	2,30217289
Szenario2	49	45	45			
Zeit in ms	47	50	49			
	49	47	46			
	46	46	44			
	46	49	43			
Mittelwert in ms	47,4	47,4	45,4			
Standardabweichung	1,51657509	2,07364414	2,30217289			
Szenario3	44	45	49	43		
Zeit in ms	44	45	44	43		
	44	45	45	42		
	45	45	43	44		
	44	44	47	44		
Mittelwert ins ms	44,2	44,8	45,6	43,2		
Standardabweichung	0,4472136	0,4472136	2,40831892	0,83666003		

Tabelle 18: Zeit für die Durchführung der Transformationen

Szenario1	screen1	screen2	screen3	screen4	screen5	screen6
Zeit in ms	1730	2380	2280	2220	1900	78
	2060	2210	2260	2110	1940	76
	1890	2300	2810	2740	2460	77
	2030	2150	2310	2080	2400	75
	1670	2310	2380	2550	1900	73
Mittelwert in ms	1876	2270	2408	2340	2120	75,8
Standardabweichung	174,298594	90,2773504	229,281486	291,118533	284,253408	1,92353841
Szenario2	1950	4310	3140			
Zeit in ms	1940	3870	3260			
	1960	4250	3210			
	1980	4060	3320			
	2030	4160	3230			
Mittelwert in ms	1972	4130	3232			
Standardabweichung	35,6370594	173,349358	66,1059755			
Szenario3	620	683	484	488		
Zeit in ms	617	625	490	479		
	643	645	442	445		
	648	629	458	490		
	643	656	470	503		
Mittelwert ins ms	634,2	647,6	468,8	481		
Standardabweichung	14,5155089	23,383755	19,4730583	21,8746429		

Tabelle 19: Gesamtzeit zum Laden und Anzeigen der Seiten

Die Zeitwerte in Tabelle 18 wurden mit dem Netzwerkmodul von Firebug ermittelt. Es wurde jeweils die unter dem Punkt *„Auf Antwort warten“* gemessene Zeit der ersten GET-Anfrage in die Tabelle übernommen. Durch diese Anfrage wird das Servlet, welches die ad-hoc Transformation auf dem Server durchführt, aufgerufen. Anschließend erfolgt die DOM-Verarbeitung der gelieferten Daten, welche den größten Teil der Gesamtzeit beansprucht. Parallel dazu wird der Code für die benötigten Funktionen an den entsprechenden Stellen über den Webservice geladen. Die Zeiten zum Laden der einzelnen Funktion variieren, sind aber im Vergleich zur jeweiligen Gesamtzeit gering. Beim zweiten Szenario wurden durch die Anfragen an den Kartendienst Microsoft Virtual Earth [2] Gesamtzeiten von 4,1 Sekunden zur Ermittlung der Koordinaten und 3,2 Sekunden zur Anzeige des Ortes auf der Karte ermittelt. Diese Anfragen erfolgen jedoch unabhängig davon, ob die entsprechenden Seiten durch Transformation erzeugt oder statisch aufgerufen werden. Deshalb wird dieses Szenario in die Messungen einbezogen.

Betrachtet man zum Beispiel die Gesamtzeiten des ersten Szenarios zur Anzeige der fünften und sechsten Seite, dann benötigt die Anzeige der Zusammenfassung der eingegebenen Daten auf der fünften Seite 2120 ms, während die einfache Bestätigungsmeldung auf der letzten Seite nach 75,8 ms angezeigt wird. Auf der fünften Seite werden zwei Funktionen benötigt. Der dazugehörige Ladevorgang ist nach 150 ms abgeschlossen. Die restlichen 2 Sekunden benötigt der Browser zur DOM-Verarbeitung, durch welche die anzuzeigenden Daten aus Variablen ausgelesen und anschließend dargestellt werden. Die Verarbeitung auf der letzten Seite beschränkt auf die Anzeige eines einfachen Textes, der direkt aus der AUI-Beschreibung durch den Transformator in die erzeugte HTML-Seite eingebunden wird und im Browser unverändert angezeigt werden kann. Zieht man von dem Mittelwert der

Gesamtzeit die für die Transformation benötigten 48,6 ms ab, dann ergibt sich bei dieser Seite eine Verarbeitungszeit von 27,2 ms. Aus der Tabelle ist jedoch zu entnehmen, dass die Gesamtzeit bei allen anderen Seiten, auf denen komplexe Interaktoren und mehr Informationen angezeigt werden, erheblich größer ist als die Transformationszeit. Das Rendern der Interaktoren und die Ausführung von Skripten im Browser auf Clientseite beansprucht insgesamt die meiste Zeit. Der Mittelwert über alle gemessenen Gesamtzeiten beträgt 1743 ms. Für die Zeit zur Durchführung der Transformationen wurden im Mittel 49 ms gemessen. Insgesamt beanspruchen die Transformationen rund 2,8% der Gesamtzeit und beeinflussen damit die Zeit, die ein Anwender warten muss, bis eine Seite einer Anwendung vollständig angezeigt wird, nur sehr gering.
Wird die Anwendung nicht lokal ausgeführt, vergrößert sich das Verhältnis zwischen Transformationszeit und der Zeit für die anderen Vorgänge weiter, da in diesem Fall zusätzliche Zeit zum Übertragen der Daten über die Internetleitung notwendig ist.

7 Zusammenfassung und Ausblick

Im Rahmen der Arbeit wurde gezeigt, dass aus einer interaktorbasierten Beschreibung mulitmodale Web2.0-Applikationen dynamisch generiert werden können. Um die Anforderungen an eine interaktorbasierte UI-Beschreibung zu spezifizieren, wurden in Kapitel 2.1 vier Szenarien vorgestellt, die typische Web2.0-Interaktionsmöglichkeiten enthalten. Nach der Identifikation der Anforderungen in Kapitel 2.2 erfolgt in Kapitel 3.2 die Vorstellung der Sprache MARIA XML, die zur abstrakten Beschreibung von UIs eingesetzt wird. Im Kapitel 3.3 werden vorhandene Ansätze zur Generierung von finalen Nutzerschnittstellen aus abstrakten modellbasierten Beschreibungen analysiert und gezeigt, dass diese Ansätze den definierten Anforderungen nicht genügen. Die Durchführung von ad-hoc Transformationen, durch welche die Konfigurierbarkeit der zu generierenden Applikationen und ein Wechsel der Modalität zur Laufzeit möglich wird, ist bei diesen Ansätzen nicht vorgesehen. Zudem werden zur Erzeugung von FUIs mehrere Abstraktionsebenen durchlaufen. Eine direkte Erzeugung einer finalen Nutzerschnittstelle aus einer AUI-Beschreibung ist bei diesen Ansätzen nicht möglich. Im Rahmen der Konzeption wird in Kapitel 4.1 ein System zur Durchführung der ad-hoc Transformationen entwickelt. Anschließend werden die entwickelten Szenarien in Kapitel 4.3 im Detail analysiert, um daraus in Kapitel 4.4 und Kapitel 4.5 abzuleiten, welche Elemente und Attribute von MARIA XML zur abstrakten Beschreibung von Interaktoren verwendet werden können. Notwendige Erweiterungen der abstrakten Beschreibungssprache zur Generierung von multimodalen Web2.0-Applikationen werden in Kapitel 4.6 beschrieben. Anschließend werden in Kapitel 4.7 die Templates des generischen Transformators zur Generierung der Interaktoren für die finale Nutzerschnittstelle aus den abstrakt definierten Elementen von MARIA XML konzipiert. Um die Transformationsregeln für die neu eingeführten Elemente von denen für die in MARIA XML enthaltenen Elementen zu trennen und die Möglichkeit zur einfachen Konfiguration der Anwendung bereitzustellen, erfolgt in Kapitel 4.8 die Konzeption eines Schemas zur Erstellung von Konfigurationsdateien und die Entwicklung des dazugehörigen Transformators. In Kapitel 4.9 werden die Funktionalitäten, welche aus den Elementen der abstrakten Beschreibung durch die konzipierten Transformatoren abgeleitet werden, vorgestellt. Im sich anschließenden Kapitel 5 wird die konkrete Umsetzung des konzipierten Systems beschrieben. Er erfolgt die Vorstellung der verwendeten Technologien und die Beschreibung der Umsetzung des mulitmodalen Szenarios. Im Rahmen der Evaluierung wird in Kapitel 6 gezeigt, dass die Transformatoren zur Generierung von FUIs weiterer Szenarien geeignet sind. In Kapitel 6.4 wird die Erfüllung der spezifizierten Anforderungen beschrieben. Durch die in Kapitel 6.5 durchgeführten Effizienzmessungen wird nachgewiesen, dass der Zeitaufwand für die Durchführung der Transformationen im Vergleich zur benötigten Gesamtzeit gering ist.

In der Motivation zur Arbeit wurden vier Kernfragen gestellt, welche nachfolgend beantwortet werden.

Die erste Frage bezieht sich auf die Mächtigkeit der AUI-Sprache MARIA XML. Während der Konzeption hat sich gezeigt, dass die Ergänzung von MARIA XML um zwei Elemente und vier Attribute erforderlich ist, um Web2.0-Applikationen zu erzeugen. Zudem wurden kleine Änderungen an der Struktur im vorhandenen XML Schema vorgenommen, um bereits in MARIA XML vorhandene Attribute bei weiteren als den vorgesehenen Elementen zur Verfügung zu stellen. Es wurde die beschriebene Konfigurationsdatei eingeführt, um Angaben zum Einbinden von Frameworks und Styleinformationen ablegen zu können. Durch

die aufgeführten Erweiterungen weist die zur Beschreibung der Szenarien verwendete Sprache eine Tendenz zu einem CUI auf.

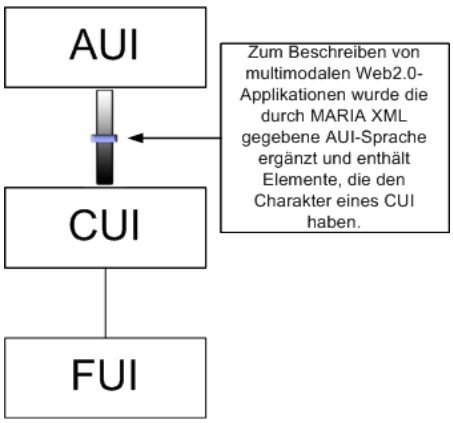


Abbildung 21: Schema: AUI, CUI, FUI

Die zu erzeugenden Applikationen können jedoch unabhängig von der später verwendeten Plattform beschrieben werden. Durch die Einführung der Konfigurationsdatei können die Templates für die in MARIA XML enthaltenen Elemente getrennt von denen für die neu eingeführten Elemente in einem eigenen Transformator abgelegt werden. Für die zur Beschreibung eines Szenarios notwendige Konfigurationsdatei existiert ein XML Schema, um diese korrekt erstellen zu können.

Bei der Analyse der Erfüllung der Anforderungen im Rahmen der Evaluierung wurde festgestellt, dass eine Zunahme der abgeleiteten Funktionalitäten zu einer Abnahme der Generik bei den entwickelten Transformatoren führt. Trotzdem können die Transformatoren für verschiedene Web2.0-Applikationen verwendet werden. Es wird die Erzeugung einer Vielzahl von Interaktoren bereits unterstützt, und die Templates müssen lediglich für spezifische Aufgaben erweitert werden, was problemlos möglich ist, da sich an den vorhandenen Transformationsregeln orientiert werden kann. Der Umfang zur abstrakten Beschreibung einiger Interaktoren ist größer als der letztendlich für die finale Nutzerschnittstelle erzeugte Code. Da die auf MARIA XML basierenden AUI-Beschreibungen allerdings in Zukunft toolbasiert erstellt oder aus Dienstbeschreibungen von Webservices abgeleitet werden sollen, ist der Umfang der abstrakten Beschreibung weniger relevant. In dem Fall ist es wichtig, dass ein verbindliches Schema existiert, in welchem der Aufbau zur Beschreibung der Interaktoren festgelegt ist.

Eine weitere Fragestellung, die in der Motivation formuliert wurde, ist die Konfigurierbarkeit der zu erzeugenden Web2.0-Applikationen. Im Rahmen der vorliegenden Arbeit wurde die Ermittlung des User Agents eingebunden um daraus abzuleiten, ob beispielsweise die Displaygröße des Clients ausreicht, um ein Widget anzuzeigen oder ob es von Vorteil ist, einen platzsparerenden Interaktor in die finale Nutzerschnittstelle einzubinden. Zudem ist das System, welches die ad-hoc Transformationen durchführt, so konzipiert, dass die verwendeten Frameworks für die Widgets und die eingebunden externen Applikationen durch Änderung eines einzelnen Parameters zur Laufzeit ausgetauscht werden können.

Mit Hilfe des zur Durchführung der ad-hoc Transformationen konzipierten Systems können multimodale Web2.0-Applikationen aus einer abstrakten Beschreibung generiert werden.

Ein Wechsel von der visuellen zur akustischen Modalität ist zur Laufzeit möglich. Zudem werden die FUIs direkt aus den abstrakten Beschreibungen generiert. Durch die Verwendung eines Webservices kann benötigter Code bei Bedarf geladen werden und muss nicht im generischen Transformator abgelegt werden. Der Einsatz von Proxies ermöglicht den Zugriff auf externe Funktionalitäten. Ein flexibler auf der Interaktion des Anwenders basierender Ablauf eines Szenarios ist durch Einführung des Sessionkonzeptes möglich.

Damit sind die in der Motivation gestellten Kernfragen beantwortet und es folgt die Vorstellung von möglichen Erweiterungen.

Die Konfiguration einer Anwendung kann durch Entwicklung eines UI zur Administration der Konfigurationsdatei eines Szenarios komfortabler gestaltet werden. Zudem können neben der Ermittlung des User Agents und der Auswahl von Frameworks und einzubindenden Applikationen auch Nutzerpräferenzen und Kontextinformation wie beispielsweise die Umgebungshelligkeit die Transformation beeinflussen, um die erzeugte Nutzerschnittstelle entsprechend anzupassen. Neben der Interaktion per Maus und Tastatur oder Sprachein- und ausgabe ist der Einsatz weiterer Modalitäten denkbar. Das Generieren einer finalen Nutzerschnittstelle aus einer abstrakten Beschreibung, bei welcher zum Beispiel über einen Touchscreen eingegebene Gesten erkannt werden, ist vorstellbar. Wenn bei zukünftigen Szenarien mehrere Widgets eines Frameworks eingebunden sind, dann kann das gesamte Widget-Set durch Ändern des entsprechenden Paramaters gegen eines aus einem anderen Framework ausgetauscht werden. Der Einsatz alternativer Formate wie JSON [60] oder Fast Infoset [61] zur Speicherung der Daten, die mittels XMLHttpRequest geladen werden, kann die Effizienz der erzeugten Web2.0-Applikationen steigern. Die abstrakte Beschreibung muss in dem Fall um einen Parameter zur Angabe des gewünschten Datenformates ergänzt werden. Das konzipierte System und die Transformatoren sind so gestaltet, dass derartige Erweiterungen problemlos vorgenommen werden können.

Anhang

XML Schema für Konfigurationsdatei

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema xmlns="http://config.maria" elementFormDefault="qualified" targetNamespace="http://config.maria"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="config_type">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="params" type="params_type" />
      <xs:element minOccurs="0" maxOccurs="unbounded" name="devices" type="devices_type" />
      <xs:element minOccurs="0" maxOccurs="unbounded" name="modalities" type="modalities_type" />
      <xs:element minOccurs="0" maxOccurs="unbounded" name="data" type="data_type" />
      <xs:element minOccurs="0" maxOccurs="unbounded" name="header" type="header_type" />
      <xs:element minOccurs="0" maxOccurs="unbounded" name="widgets" type="widgets_type" />
      <xs:element minOccurs="0" maxOccurs="unbounded" name="external_apps" type="external_apps_type" />
    </xs:sequence>
  </xs:complexType>
<!--
data
-->
  <xs:complexType name="data_type">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="datatext" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
<!--
params
-->
  <xs:complexType name="params_type">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="param" type="param_type" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="param_type">
    <xs:attributeGroup ref="name_attributes" />
  </xs:complexType>
<!--
header
-->
  <xs:complexType name="header_type">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="scriptimports" type="scriptimports_type" />
      <xs:element minOccurs="0" maxOccurs="unbounded" name="styleimports" type="styleimports_type" />
      <xs:element minOccurs="0" maxOccurs="unbounded" name="cookies" type="cookies_type" />
      <xs:element minOccurs="0" maxOccurs="unbounded" name="functions" type="functions_type" />
      <xs:element minOccurs="0" maxOccurs="unbounded" name="variables" type="variables_type" />
      <xs:element minOccurs="0" maxOccurs="unbounded" name="frameworks" type="frameworks_type" />
      <xs:element minOccurs="0" maxOccurs="unbounded" name="ext_apps" type="ext_apps_type" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="scriptimports_type">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="JSscript" type="JSscript_type" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="styleimports_type">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="CSSstyle" type="CSSstyle_type" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="JSscript_type">
    <xs:attributeGroup ref="script_attributes" />
  </xs:complexType>
  <xs:complexType name="CSSstyle_type">
    <xs:attributeGroup ref="style_attributes" />
  </xs:complexType>
<!--
devices
-->
  <xs:complexType name="devices_type">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="device" type="device_type" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="device_type">
    <xs:attributeGroup ref="name_attributes" />
  </xs:complexType>
<!--
modality
-->
  <xs:complexType name="modalities_type">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="modality" type="modality_type" />
    </xs:sequence>
```

```
  </xs:complexType>
  <xs:complexType name="modality_type">
    <xs:attributeGroup ref="name_attributes" />
  </xs:complexType>
<!--
cookies
-->
  <xs:complexType name="cookies_type">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="cookie" type="cookie_type" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="cookie_type">
    <xs:attributeGroup ref="name_attributes" />
  </xs:complexType>
<!--
variables
-->
  <xs:complexType name="variables_type">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="variable" type="variable_type" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="variable_type">
    <xs:attributeGroup ref="name_attributes" />
    <xs:attribute name="type" type="xs:string" use="optional" />
  </xs:complexType>
<!--
framework
-->
  <xs:complexType name="frameworks_type">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="framework" type="framework_type" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="framework_type">
    <xs:attributeGroup ref="name_attributes" />
  </xs:complexType>
<!--
external application
-->
  <xs:complexType name="ext_apps_type">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="ext_app" type="ext_app_type" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ext_app_type">
    <xs:attributeGroup ref="name_attributes" />
  </xs:complexType>
<!--
functions
-->
  <xs:complexType name="functions_type">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="function" type="function_type" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="function_type">
    <xs:attributeGroup ref="name_attributes" />
  </xs:complexType>
<!--
widgets
-->
  <xs:complexType name="widgets_type">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="widget" type="widget_type" />
    </xs:sequence>
    <xs:attributeGroup ref="name_attributes" />
  </xs:complexType>
  <xs:complexType name="widget_type">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="code" type="xs:string" />
    </xs:sequence>
    <xs:attributeGroup ref="name_attributes" />
  </xs:complexType>
<!--
external_app
-->
  <xs:complexType name="external_apps_type">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="external_app" type="external_app_type" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="external_app_type">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="code" type="xs:string" />
    </xs:sequence>
    <xs:attributeGroup ref="name_attributes" />
    <xs:attribute name="service" type="xs:string" use="optional" />
```

```
  </xs:complexType>
<!--
attribute groups
-->
  <xs:attributeGroup name="name_attributes">
    <xs:attribute name="name" type="xs:string" use="required" />
  </xs:attributeGroup>
  <xs:attributeGroup name="script_attributes">
    <xs:attribute name="src" type="xs:string" use="required" />
    <xs:attribute name="djConfig" type="xs:string" use="optional" />
  </xs:attributeGroup>
  <xs:attributeGroup name="style_attributes">
    <xs:attribute name="href" type="xs:string" use="required" />
  </xs:attributeGroup>
<!--
root element
-->
  <xs:element name="config" type="config_type" />
</xs:schema>
```

Tabellenverzeichnis:

Abbildungsverzeichnis:

Quellcodeverzeichnis:

Quellen- und Literaturverzeichnis:

[1] Google Maps: http://maps.google.de/ (15.07.2009)

[2] Microsoft Virtual Earth: http://www.microsoft.com/maps/ (15.07.2009)

[3] Prototype: http://www.prototypejs.org/ (15.07.2009)

[4] Scriptaculous: http://script.aculo.us/ (15.07.2009)

[5] AJAX: http://de.wikipedia.org/wiki/Ajax_(Programmierung) (15.07.2009)

[6] Javascript: http://de.selfhtml.org/javascript/index.htm (15.07.2009)

[7] XML: http://www.w3.org/XML/ (15.07.2009)

[8] Veröffentlichung: MARIA: A Universal, Declarative, Multiple Abstraction Level Language for Service-Oriented Applications in Ubiquitous Environments, Fabio Paterno, Carmen Santoro, Lucio Davide Spano, ISTI-CNR, 2009

[9] Modelling and Generating Ajax Applications: A Model-Driven Approach, 2008, Vahid Gharavi, Ali Mesbah, Arie van Deursen, TU Delft: http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2008-024.pdf (15.07.2009)

[10] ANDROMDA: http://www.andromda.org/ (15.07.2009)

[11] Apache Velocity: http://velocity.apache.org/ (15.07.2009)

[12] Hibernate: https://www.hibernate.org/ (15.07.2009)

[13] A first draft of a Model-driven Method for Designing Graphical User Interfaces of Rich Internet Applications, 2006, IEEE, Francisco J. Martinez-Ruiz, Jaime Munoz Arteaga, Jean Vanderdonckt, Juan M. Gonzalez-Calleros, Ricardo Mendoza: http://ieeexplore.ieee.org/iel5/4022074/4022075/04022089.pdf?arnumber=4022089 (15.07.2009)

[14] Cameleon Framework: http://giove.isti.cnr.it/cameleon/deliverable1_1.html (15.07.2009)

[15] UsiXML: http://www.usixml.org/ (15.07.2009)

[16] XSLT: http://www.w3.org/TR/xslt (15.07.2009)

[17] Microsoft .Net: http://www.microsoft.com/NET/ (15.07.2009)

[18] WebDSL – A Domain-Specific Language for Dynamic Web Applications, TUDelft, 2008, D.M. Groenewegen, Z. Hemel, L.C.L. Kats, E. Visser:

http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2008-040.pdf (15.07.2009)

[19] WebWorkFlow: http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2008-029.pdf (15.07.2009)

[20] Stratego XT: http://strategoxt.org/ (15.07.2009)

[21] Designing Rich Internet Applications with Web Engineering Methodologies, 2007, IEEE, J.C. Preciado, M. Linaje, S. Comai, F. Sanchez-Figueroa: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4380240 (15.07.2009)

[22] WebML: http://www.webml.org/webml/page1.do (15.07.2009)

[23] RUX-Model: http://wcat.unex.es/wcat07/documents/Preciado-Linaje-Sanchez-Figueroa.pdf (15.07.2009)

[24] UWE: http://uwe.pst.ifi.lmu.de/ (15.07.2009)

[25] Designing Rich Internet Applications Combining UWE und RUX-Method, IEEE, 2008, Juan Carlos Preciado, Marino Linaje, Rober Morales-Chaparro, Fernando Sanchez-Figueroa, Gefei Zhang, Christian Kroiß, Nora Koch: http://ieeexplore.ieee.org/iel5/4577854/4577855/04577878.pdf?arnumber=4577878 (15.07.2009)

[26] Migrating Multi-page Web Applications to Single-page AJAX Interfaces, 2007, IEEE, Ali Mesbah, Arie van Deursen: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4145036 (15.07.2009)

[27] Refactoring to Rich Internet Applications: A Model-Driven Approach, 2008, Gustavo Rossi, Matias Urbieta, Jeronimo Gnizburg, Damiano Distante, Alejandra Garrido: http://www.lifia.info.unlp.edu.ar/papers/2008/Rossi2008.pdf (15.07.2009)

[28] Amazon.de: http://www.amazon.de/ (15.07.2009)

[29] Conceptual Modeling and Code Generation for Rich Internet Applications, 2006, ACM, Alessandro Bozzon, Piero Fraternali, Sara Comai, Giovanni Toffetti Carughi: http://portal.acm.org/citation.cfm?id=1145649 (15.07.2009)

[30] WebRatio: http://www.webratio.com/ (15.07.2009)

[31] The TERESA XML Language for the Description of Interactive System at Multiple Abstraction Levels, Silvia Berti, Francesco Correani, Fabio Paterno, Carmen Santoro: http://giove.isti.cnr.it/cameleon/cp25.html (15.07.2009)

[32] VoiceXML: http://www.voicexml.org/, http://www.w3.org/TR/voicexml20/ (15.07.2009)

[33] Dialog Modelling with interactors and UML Statecharts – A hybrid approach, 2003, Hallvard Traetteberg, NTNU Norwegen: http://www.springerlink.com/index/7qvl1t097bw06h0v.pdf (15.07.2009)

[34] Query View Transformation (QVT): http://www.omg.org/spec/QVT/1.0/ (15.07.2009)

[35] ATLAS Transformation Language (ATL): http://www.eclipse.org/m2m/atl/ (15.07.2009)

[36] Laszlo LZX: http://openlaszlo.org/ (15.07.2009)

[37] Adobe Flex: http://www.adobe.com/de/products/flex/ (15.07.2009)

[38] Microsoft XAML: http://msdn.microsoft.com/en-us/library/ms752059.aspx (15.07.2009)

[39] XML Schema: http://www.w3.org/XML/Schema (15.07.2009)

[40] XSL: http://www.w3.org/Style/XSL/ (15.07.2009)

[41] HTML 4.01: http://www.w3.org/TR/html401/ (15.07.2009)

[42] CSS: http://www.w3.org/Style/CSS/ (15.07.2009)

[43] XMLHttpRequest: http://www.w3.org/TR/XMLHttpRequest/ (15.07.2009)

[44] Java: http://java.sun.com/ (15.07.2009)

[45] Apache Tomcat: http://tomcat.apache.org/ (15.07.2009)

[46] Java Servlet: http://java.sun.com/products/servlet/ (15.07.2009)

[47] Saxon: http://saxon.sourceforge.net/ (15.07.2009)

[48] Apache HTTP Server: http://httpd.apache.org/ (15.07.2009)

[49] SOAP: http://www.w3.org/TR/soap/ (15.07.2009)

[50] WSDL: http://www.w3.org/TR/wsdl (15.07.2009)

[51] JSCalendar: http://www.dynarch.com/projects/calendar/old/ (15.07.2009)

[52] Dojo: http://www.dojotoolkit.org/ (15.07.2009)

[53] Livevalidation: http://www.livevalidation.com/ (15.07.2009)

[54] PHP: http://www.php.net/ (15.07.2009)

[55] PeriodicalUpdater: http://www.prototypejs.org/api/ajax/periodicalUpdater/ (15.07.2009)

[56] XHTML+Voice: http://www.w3.org/TR/xhtml+voice/, http://dev.opera.com/articles/voice/ (15.07.2009)

[57] Opera (Version 9.63): http://de.opera.com/ (15.07.2009)

[58] JVoiceXML: http://jvoicexml.sourceforge.net/ (15.07.2009)

[59] Ant Skript: http://ant.apache.org/ (15.07.2009)

[60] JSON http://www.json.org/ (15.07.2009)

[61] Fast Infoset http://java.sun.com/developer/technicalArticles/xml/fastinfoset/ (15.07.2009)

[62] ECMA-Standard http://www.ecma-international.org/ (15.07.2009)

[63] Firefox http://www.mozilla-europe.org/de/firefox/ (15.07.2009)

[64] Firebug https://addons.mozilla.org/de/firefox/addon/1843 (15.07.2009)

[63] Buch: Javascript: Einführung, Programmierung und Referenz, 4. Auflage, Stefan Koch, dpunkt Verlag 2007

[64] Buch: Ajax: Grundlagen, Frameworks, Praxislösungen, 1. Auflage, Stefan Mintert, Christoph Leisegang, dpunkt Verlag 2007

[65] Buch: Web2.0: Webseiten intelligent verknüpfen, 1. Auflage, Shu-Wai Chow, Franzis Verlag 2008

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Riesa, den 29.07.2009 ……

Unterschrift des Verfassers