



MASTER-THESIS

im Studiengang Kommunikation und Medienmanagement

Konzeption und Umsetzung einer
komponentenbasierten Publikationsplattform

JAN OEVERMANN (46594)

BETREUER:

Prof. Dr. Wolfgang Ziegler
Prof. Martin Schober
Daniel Heideck

ZEITRAUM:

02.02. – 01.08.2015

MASTER-THESIS:

Konzeption und Umsetzung einer
komponentenbasierten Publikationsplattform
Design and implementation of a component-based publishing platform

AN DER:

Hochschule Karlsruhe - Technik und Wirtschaft
Fakultät für Informationsmanagement und Medien
Studiengang Kommunikation und Medienmanagement

BETREUER:

Prof. Dr. Wolfgang Ziegler, Hochschule Karlsruhe
Prof. Martin Schober, Hochschule Karlsruhe
Daniel Heideck, arvato Bertelsmann

ORT:

Düsseldorf

ZEITRAUM:

02.02. – 01.08.2015

ABSTRACT

Die fortschreitende Digitalisierung führt in Unternehmen zu einem rasanten Wachstum an Informationen, die für interne und externe Nutzergruppen effizient bereitgestellt werden müssen. Inhalte stammen einerseits aus einer steigenden Zahl von Quellen und Datenstrukturen, während andererseits eine Ausgabe in immer mehr Kanäle erforderlich ist. Mechanismen zur automatisierten Publikation dieser Informationen werden deshalb zunehmend komplexer.

Mit den Publikationsmethoden von Content-Management-Systemen und den Aggregierungsfunktionen von Content-Delivery-Portalen bestehen bereits Lösungen am Markt, die jeweils einen Teil des Problems abdecken. Umfassende Systeme, die Informationen sowohl bündeln als auch verteilen, werden in der Regel jedoch als Individuallösungen umgesetzt. Einen Ausweg können komponentenbasierte Systeme bieten, die sich flexibel an neue Anforderungen und Umgebungen anpassen lassen. Diese Arbeit stellt ein Konzept vor, das einen solchen Lösungsansatz verfolgt: Eine Publikationsplattform auf Basis von austauschbaren Komponenten für Quellen, Verarbeitungen und Ausgaben.

Nach eingehender Untersuchung grundlegender Konzepte aus den Bereichen Publishing und komponentenbasierte Entwicklung werden die konkreten Anforderungen an eine Publikationsplattform formuliert. Aufbauend darauf wird ein neutrales Konzept entwickelt und in einer prototypischen Umsetzung konkret implementiert. Die entstandene Lösung wird abschließend anhand ausgewählter Anwendungsfälle verifiziert.

INHALTSVERZEICHNIS

1 EINLEITUNG	1
1.1 Motivation	1
1.2 Methodik	2
1.3 Ausgangssituation	2
1.4 Problemstellung und Ziele	4
1.5 Abgrenzung	5
2 GRUNDLAGEN UND ÜBERSICHT	7
2.1 Content	7
2.1.1 Überblick	7
2.1.2 Definition	7
2.1.3 Klassifizierung	9
2.1.4 Methoden	10
2.2 Datenformat	12
2.2.1 Überblick	12
2.2.2 XML	13
2.2.3 JSON	13
2.2.4 RDF	14
2.2.5 HTML5	14
2.3 Publishing	20
2.3.1 Überblick	20
2.3.2 Definition	21
2.3.3 Methoden	21
2.3.4 Technologien	22
2.4 Komponentenbasierte Systeme	24
2.4.1 Überblick	24
2.4.2 Definition	25
2.4.3 Klassifizierung	26
2.4.4 Datenfluss	27
2.4.5 Komponenten-Frameworks	29
2.4.6 Konzepte und Prinzipien	30
2.4.7 Flow-based Programming	33
2.5 Komponentenbasiertes Publizieren	34
2.5.1 Überblick	34
2.5.2 Stand der Technik	34
2.5.3 Weitere Ansätze	38
3 ANFORDERUNGEN	39
3.1 Allgemein	39
3.1.1 Ziel des Konzepts	39
3.1.2 Herkunft der Anforderungen	39
3.1.3 Kernfunktionalität	40

3.2	Nichtfunktionale Anforderungen	40
3.2.1	Konzeption	40
3.2.2	Umsetzung	42
3.2.3	Freie Software	42
3.2.4	Anpassbarkeit (Customizing)	43
3.3	Funktionale Anforderungen	43
3.3.1	Formate	43
3.3.2	Aufgaben	44
3.3.3	Cloud-Fähigkeit	46
3.3.4	Bedienung	46
4	KONZEPTION	47
4.1	Architektur	47
4.1.1	Modell	47
4.1.2	Schichten	48
4.1.3	Szenarien	49
4.1.4	Anmerkung	49
4.2	Grundlagen	50
4.2.1	Hauptbegriffe	50
4.2.2	Weitere Begriffe	50
4.3	Publikationsprozess	51
4.3.1	Definition	51
4.3.2	Phasen	51
4.3.3	Informationsfluss	53
4.3.4	Teilprozesse	53
4.4	Komponenten	54
4.4.1	Definition	54
4.4.2	Aufbau	55
4.4.3	Klassifizierung	55
4.4.4	Identifizierung	57
4.4.5	Schnittstellen	57
4.4.6	Lebenszyklus	58
4.4.7	Kontextinformationen	59
4.4.8	Prozessbeschreibung	59
4.5	Publikationsdefinitionen	62
4.5.1	Definition	62
4.5.2	Aufbau	62
4.5.3	Identifizierung	62
4.5.4	Lebenszyklus	62
4.6	Repository	63
4.6.1	Definition	63
4.6.2	Aufbau	63
4.6.3	Erstellung	63
4.6.4	Funktion	64
4.7	Interne Informationsstruktur	64
4.7.1	Bedarf	64
4.7.2	Anforderungen	65

5 TECHNISCHE UMSETZUNG	67
5.1 Arbeitsname	67
5.2 Informationsstruktur	67
5.2.1 Anforderungsabbildung	67
5.2.2 Aufbau	67
5.3 Umgang mit Content	70
5.3.1 Vorüberlegungen	70
5.3.2 Überführung	71
5.3.3 Strukturierung	72
5.3.4 Kombination	73
5.3.5 Beispiele	73
5.4 Schnittstellenmodell	74
5.4.1 Vorüberlegungen	74
5.4.2 Datenfluss	75
5.4.3 Klassifizierung	75
5.4.4 Media types	75
5.4.5 Spezifikation	76
5.4.6 Validierung	78
5.5 Publikationssystem	78
5.5.1 Architektur	78
5.5.2 Technologie	78
5.5.3 Host-System	79
5.5.4 Aufbau	81
5.5.5 Bootstrapping	83
5.5.6 Benutzerschnittstelle	84
5.6 Komponenten	85
5.6.1 Identifier	85
5.6.2 Aufbau	85
5.6.3 Schritte	87
5.6.4 Parameter	88
5.6.5 Kontextinformationen	89
5.6.6 Klassifizierung	90
5.6.7 Beispiel	90
5.6.8 Abhängigkeiten	91
5.6.9 Liste der Komponenten	94
5.7 Publikationsdefinitionen	96
5.7.1 Identifizierung	96
5.7.2 Aufbau	96
5.7.3 Beispiel	97
5.8 Repository	98
5.8.1 Aufbau	98
5.8.2 Erzeugung	99
5.9 Architektur	99
5.9.1 Technologien	100
5.10 Web-API	101
5.10.1 Aufgaben	101

5.10.2 Technologie	101
5.10.3 Klassifizierung der Schnittstelle	101
5.10.4 Datenaustausch	103
5.10.5 API-Kurzreferenz	104
5.10.6 Publishing mit der Web-API	104
5.11 Web-Client	107
5.11.1 Aufgaben	107
5.11.2 Technologie	108
5.11.3 Publishing mit dem Web-Client	108
6 USE CASES	111
6.1 Überblick	111
6.2 DITA nach PDF mit Strukturänderung	111
6.2.1 Beschreibung	111
6.2.2 Konfiguration	111
6.2.3 Ablauf	112
6.3 PDF nach HTML mit Index-Erstellung	114
6.3.1 Beschreibung	114
6.3.2 Konfiguration	114
6.3.3 Ablauf	115
6.4 Word-Dokumente nach Android-App	117
6.4.1 Beschreibung	117
6.4.2 Konfiguration	117
6.4.3 Ablauf	118
6.5 Bewertung	119
7 FAZIT UND AUSBLICK	121
7.1 Zusammenfassung	121
7.2 Fazit	122
7.3 Ausblick	123
A ANHANG	125
A.1 Online-Demo	125
A.2 Lokale Installation	125
A.2.1 Publikationsserver	126
A.2.2 Web-API	126
A.2.3 Web-Client	126
A.3 Verwendete Programmteile	126
A.4 Zusatzdokumente	129
A.5 Statistik zur Umsetzung	129
LITERATURVERZEICHNIS	130

ABBILDUNGSVERZEICHNIS

Abb. 1	Konzept: CMS und CDP	3
Abb. 2	Konzept: Kombinierte Publikationsplattform . . .	4
Abb. 3	Dateibasierte Schnittstelle: Beispiel XSL-FO . . .	29
Abb. 4	Hub and Spoke: Schematische Darstellung	31
Abb. 5	Pipes and Filters: Schematische Darstellung	32
Abb. 6	DITA OT: Publikationsprozess nach dita-ot.org . .	36
Abb. 7	Pandoc: Hub-and-Spoke-Architektur	38
Abb. 8	Anforderung: Content-Zusammenführung	45
Abb. 9	Anforderung: Content-Restrukturierung	45
Abb. 10	Anforderung: Content-Anreicherung	45
Abb. 11	Plattform: Konzeptionelle Architektur	47
Abb. 12	Publikationssystem: Konzept	51
Abb. 13	Publikationsprozess: Phasen und Hauptfluss . . .	52
Abb. 14	Publikationsprozess: Nebenflüsse	53
Abb. 15	Publikationsprozess: Teilprozesse	54
Abb. 16	Komponenten: Lebenszyklus	58
Abb. 17	Publikationsprozess: Kontextinformationen	59
Abb. 18	Publikationsprozess: Informationsfluss	60
Abb. 19	Publikationsprozess: Schritte/Abhängigkeiten . .	61
Abb. 20	Host-System: Aufbau und Kommunikation	80
Abb. 21	Publikationssystem: Aufbau Dateisystem	82
Abb. 22	Aufbau Komponentenpaket	85
Abb. 23	Schema eines Komponenten-Manifests	86
Abb. 24	Komponenten: Alle Abhängigkeiten	92
Abb. 25	Abhängigkeiten: Strukturieren	93
Abb. 26	Abhängigkeiten: Kopieren	93
Abb. 27	Schema einer Publikationsdefinition	97
Abb. 28	Schema des Repository	99
Abb. 29	Umsetzung: Architektur und Technologien	100
Abb. 30	JSON-Repräsentation Publikationsdefinition . . .	103
Abb. 31	Web-API: Publikationsprojekt	106
Abb. 32	Web-Client: Quellen hinzufügen	109
Abb. 33	Web-Client: Verarbeitungen hinzufügen	109
Abb. 34	Use Case 1: Konfiguration	111
Abb. 35	Use Case 1: Quellen	112
Abb. 36	Use Case 1: Struktur bearbeiten	113
Abb. 37	Use Case 1: Ergebnis-PDF	114
Abb. 38	Use Case 2: Konfiguration	115
Abb. 39	Use Case 2: Verarbeitungen	116
Abb. 40	Use Case 2: Ergebnis-HTML mit Index	117
Abb. 41	Use Case 3: Konfiguration	118

Abb. 42	Use Case 3: App von Web-Client installieren . . .	119
Abb. 43	Use Case 3: Ergebnis-App auf Android Tablet . .	119
Abb. 44	Demo: Eigenschaften eintragen	125

TABELLENVERZEICHNIS

Tabelle 1	HTML5: Arten von Syntax	15
Tabelle 2	Komponenten: Arten von Kopplungen	28
Tabelle 3	Komponenten: Arten von Innerer Bindung . .	28
Tabelle 4	Informationsstruktur: Anforderungsabbildung .	68
Tabelle 5	Informationsstruktur: Attribute an Root-Element	71
Tabelle 6	infoflow Kommandozeilen-Parameter	84
Tabelle 7	Komponenten: Schlüssel-Typ-Zuordnung	85
Tabelle 8	Komponenten: Schritt-Typen	87
Tabelle 9	Komponenten: Parameter-Typen	88
Tabelle 10	Komponenten: Kontextinformations-Typen . . .	89
Tabelle 11	Liste der Basis-Komponenten	94
Tabelle 12	Liste der System-Komponenten	95
Tabelle 13	Liste der Eingabe-Komponenten	95
Tabelle 14	Liste der Verarbeitungs-Komponenten	95
Tabelle 15	Liste der Ausgabe-Komponenten	96
Tabelle 16	Plattform: Eingesetzte Technologien	100
Tabelle 17	API: Umwandlung XML - JSON	104
Tabelle 18	API: Kurzreferenz	105
Tabelle 19	Verwendete Software: Publikationssystem . . .	127
Tabelle 20	Verwendete Software: Optionale Installationen .	127
Tabelle 21	Verwendete Software: Web-API	128
Tabelle 22	Verwendete Software: Web-Client	128
Tabelle 23	Zusatzdokumente	129
Tabelle 24	Statistik Umsetzung: Entstandener Code	129

CODEAUSSCHNITTE

Code 1	Notierte Struktur in HTML5	16
Code 2	Interpretierte Struktur in HTML5	17
Code 3	DITA in XML	19
Code 4	DITA mit RDFa in (X)HTML5	19
Code 5	Sectioning in HTML4	20
Code 6	Sectioning in HTML5	20
Code 7	Makrostruktur H5	69
Code 8	Schnittstellenspezifikation Eingabe	76
Code 9	infoflow-Kommandozeile: Publikation	84
Code 10	infoflow-Kommandozeile: Validierung	84
Code 11	Parameter-Übergabe	88
Code 12	Parameter-Definition	89
Code 13	XML-Notation einer Komponente	90
Code 14	XML-Notation einer Publikationsdefinition	98
Code 15	REST via HTTP - Anfrage	102
Code 16	REST via HTTP - Antwort	102

ABKÜRZUNGSVERZEICHNIS

AIIM	Association for Information and Image Management
API	Application Programming Interface
APK	Android Application Package
ASF	Apache Software Foundation
CAD	Computer-aided Design
CBD	Component Based Development
CBSE	Component Based Software Engineering
CCM	Component Content Management
CM	Content Management
CMS	Content-Management-System
CDP	Content-Delivery-Portal
CMP	Cross-Media-Publishing
CORBA	Common Object Request Broker Architecture
CORS	Cross-Origin Resource Sharing
CSS	Cascading Style Sheets
DITA	Darwin Information Typing Architecture
DITA OT	DITA Open Toolkit
DOM	Document Object Model
ECM	Enterprise Content Management
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IANA	Internet Assigned Numbers Authority
IETF	Internet Engineering Task Force
IRI	Internationalized Resource Identifier
ISO	International Organization for Standardization
JS	JavaScript
JSON	JavaScript Object Notation
J ₂ EE	Java Platform, Enterprise Edition
MSP	Multi-Source-Portal
OASIS	Organization for the Advancement of Structured Information Standards

OMG	Object Management Group
OSS	Open Source Software
PDF	Portable Document Format
RDF	Resource Description Framework
REST	Representational State Transfer
RUP	Rational Unified Process
SDK	Software Development Kit
SGML	Standard Generalized Markup Language
SSP	Single-Source-Publishing
TD	Technische Dokumentation
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
W3C	World Wide Web Consortium
XHTML	Extensible Hypertext Markup Language
XML	Extensible Markup Language
XSD	XML Schema Definition
XSL	Extensible Stylesheet Language
XSL-FO	XSL Formatting Objects
XSLT	XSL Transformation
XPath	XML Path Language

EINLEITUNG

1.1 MOTIVATION

In den letzten Jahren haben anhaltende Trends auf dem Gebiet der Technischen Kommunikation neue Herausforderungen für Personal und Technik mit sich gebracht. Durch eine generell zunehmende Menge an Informationen und einen branchenübergreifenden Trend zur Digitalisierung (vgl. IDC 2014) hat sich die Zahl von Informationsquellen sowie möglicher Ausgabeformen tendenziell erhöht (vgl. ZIEGLER/BEIER 2014:50).

Der Technische Redakteur wandelt sich immer mehr zum modernen Medienmanager, der neben seiner traditionell schreibenden Arbeit auch Dritt-Informationen sammeln und bündeln muss, um diese anschließend verschiedenen Ausgabemedien zuzuführen (vgl. OEHMIG 2011:163).

Diese Entwicklung erscheint konsequent, denn in den TD-Abteilungen einer Firma ist in der Regel das *Know-How* zur Verarbeitung und Strukturierung von komplexen Informationen angesiedelt. Die Ausgabe modularen *Contents* in verschiedene Ausgabemedien, das sogenannte Cross-Media-Publishing (CMP), ist mit dem Einsatz von Content-Management-Systemen zur Routine geworden und methodisch gut untersucht (vgl. ZIEGLER 2013:22). Bisherige Ansätze basieren allerdings auf dem Prinzip des Single-Source-Publishing (SSP); der Umgang mit einer steigenden Anzahl heterogener und oft unzureichend strukturierter Informationsquellen ist neu.

Für neuartige Informationsprodukte wie Mobile Apps oder Portale werden nunmehr aus unterschiedlichen Abteilungen eines Unternehmens Inhalte zugesteuert, die von Marketing-Videos über Konstruktionsmodellen bis hin zu Programmteilen reichen können (vgl. CLOSS 2014:109). Die Branche reagiert auf diese Bewegung mit der Entwicklung von sogenannten Content-Delivery-Portalen, die Inhalte verschiedener Art aggregieren und webbasiert bereitstellen können (vgl. ZIEGLER/BEIER 2014:50); ein Ansatz, der auch als Multi-Source-Publishing bezeichnet werden kann.

Nicht betrachtet wird bisher jedoch die Kombination der beiden Ansätze zu einem integrierten und offenen Konzept, das mehrere Quellen aufnehmen, sie verarbeiten und anschließend in verschiedene Zielmedien publizieren kann.

Die Lösung zur Umsetzung einer solchen Multi-Source-/Multi-Target-Plattform können Komponentensysteme sein, deren konkrete Zusammenstellung sich dynamisch anpassen kann. Die Wiederverwendung von in sich abgeschlossenen Einheiten auf Inhaltsebene ist bereits üblich. Deshalb liegt es nahe, ein ähnliches Konzept auch auf Ebene der Inhaltsverarbeitung anzuwenden.

1.2 METHODIK

Die Struktur der Arbeit gliedert sich neben Einleitung und Fazit in zwei Blöcke: Einen theoretischen Teil, der die Kapitel 2 bis 3 umfasst und einen praktischen Teil, der von Kapitel 4 bis Kapitel 6 reicht.

Im ersten Teil wird in thematisch wichtige Grundlagen eingeführt, ein Überblick über den aktuellen Stand von Wissenschaft und Technik gegeben sowie Anforderungen an eine Lösung formuliert. Im zweiten Teil wird ein Konzept spezifiziert, eine konkrete technische Umsetzung entwickelt sowie deren Funktionalität anhand von ausgewählten Anwendungsfällen verifiziert. Die Einordnung eines Kapitels in den Gesamtzusammenhang wird für jedes Kapitel in Form eines *Advance Organizer* vorangestellt.

Durch die Neuartigkeit des Themas existiert bisher nur wenig Literatur, die sich konkret mit einer Lösung beschäftigt, die die geforderten Eigenschaften mit sich bringt. Deshalb verfolgt die Arbeit den Ansatz, aus angrenzenden und zum Teil gut erforschten Bereichen Erkenntnisse in ein Konzept zu übertragen, das im Bereich der Technischen Dokumentation (TD) eingesetzt werden kann. Aus diesem Grund stammen viele der verwendeten Quellen aus anderen Fachbereichen (im Wesentlichen aus der angewandten Informatik).

Der besondere Fokus der Arbeit soll darauf liegen, die Problemstellung praxisorientiert zu lösen und dabei, aufbauend auf bewährten Methoden und Konzepten, moderne Technologien und Möglichkeiten einzusetzen.

1.3 AUSGANGSSITUATION

Derzeit sind zwei Strömungen bei der Bereitstellung von Technischen Informationen zu beobachten: Die „klassische“ Kombination aus SSP und CMP, bei der aus einer medieneutralen Quelle medienspezifische Publikationen erzeugt werden (vgl. DREWER/ZIEGLER 2011:296) und der neue Trend hin zu Content-Delivery-Portalen, bei denen aus mehreren medienspezifischen Quellen eine einheitliche Darstellung in Form eines Portals erzeugt wird (vgl. ZIEGLER/BEIER 2014:50).

Single-Source-Publishing gilt als erprobt und gut erforscht (vgl. CLOSS 2007:20f). Mit dem Siegeszug von SSP in Technischen Redaktionen

hat sich auch die Technologie auf der Ausgabeseite einer Publikation weit entwickelt: Mit modernen Content-Management-Systemen (CMS) werden Inhalte in zahlreiche Medien publiziert (vgl. DREWER/ZIEGLER 2011:432). Doch durch die Erstellung des Contents direkt im Quellformat wurde alternativen Eingabeformaten bisher nur wenig Beachtung geschenkt.¹ Content-Delivery-Portale (CDP) decken diesen Teil mit ihren Multi-Source-Fähigkeiten zwar ab, sind jedoch (nur) für Retrieval- und Recherche-Prozess optimiert (vgl. ZIEGLER/BEIER 2014:51f) und haben nur begrenzte Möglichkeiten zum Publizieren in andere Medien oder Formate. Aufgrund der Neuartigkeit der Systeme sind Prozesse zur Integration von unzureichend strukturierten Daten nur wenig untersucht worden.

Ein Ansatz, der mit Hilfe einer zentralen Plattform diese beiden Strömungen vereint und Informationen verschiedener Strukturierungsgrade mit Hilfe festgelegter Prozesse vereint, fehlt bisher am Markt.

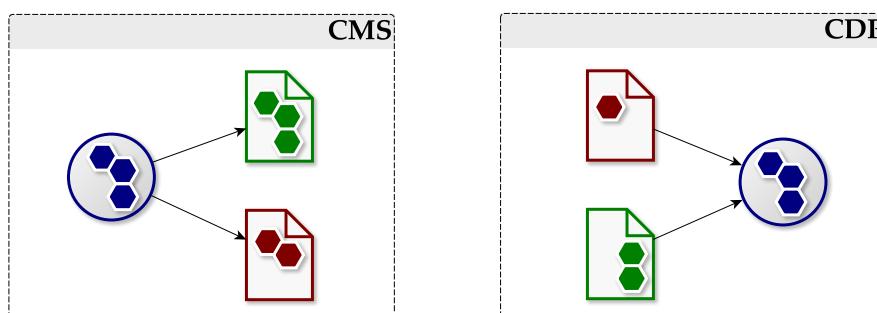


Abb. 1: Konzept: CMS und CDP

Die Lösung für die Umsetzung einer solchen Plattform können komponentenbasierte Systeme sein: Funktionen und Prozesse werden in Komponenten gekapselt, die je nach Anforderungen zusammenge stellt werden können. Vorteile ergeben sich durch die Möglichkeiten zur Wiederverwendung, dem Hinzufügen neuer Komponenten sowie den flexiblen Anpassungsmöglichkeiten an neuen Anforderungen. Die Konfiguration des jeweiligen Systems ergibt sich aus der Zusammenstellung der Komponenten.

Komponentenbasierte Softwareentwicklung ist weit verbreitet und wird auch speziell im Umfeld von Open Source Software (OSS) auf grund seiner offenen Natur eingesetzt (vgl. SLYNGSTAD 2011:23). In der Technischen Dokumentation (TD) konnten sich aufgrund der hohen Anforderungen fast ausschließlich proprietäre Systeme durch setzen, deren interne Applikationsstruktur sich nur vermuten lässt. Einer der wenigen Vorreiter im Bereich der offenen und konfigu rierbaren Publikationslösungen ist das DITA Open Toolkit (siehe Ab schnitt 2.5.2.3).

¹ Diese wurden entweder manuell in das CMS übertragen oder individuell migriert.

1.4 PROBLEMSTELLUNG UND ZIELE

Diese Arbeit hat die Konzeption und prototypische Umsetzung einer komponentenbasierten Publikationsplattform zum Ziel, die aus mehreren Quellen in mehrere Ziele publizieren kann. Während des Publikationsprozesses soll es möglich sein, den *Content* zu manipulieren (filtern, anreichern, umstrukturieren, etc.).

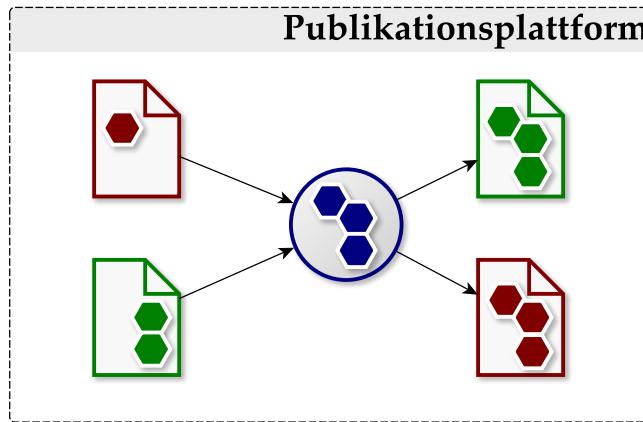


Abb. 2: Konzept: Kombinierte Publikationsplattform

Dieses Ziel soll durch ein komponentenorientiertes Konzept erreicht werden. Das heißt, Funktionen und Prozesse werden in wiederverwendbare und austauschbare Einheiten, sog. Komponenten, gekapselt. Der Teil der Plattform, der als Publikationssystem agiert, soll dabei vollständig aus Komponenten aufgebaut sein.

Zielerreichung

Die Zielerreichung kann in folgende Arbeitspakete unterteilt werden:

- Definition von Anforderungen an eine mögliche Lösung
- Spezifikation eines komponentenbasierten Konzepts
 - Architektur einer Publikationsplattform
 - Entwurf eines Prozess- und Datenflussmodells
 - Vorgabe eines Modells zur Abbildung von Komponenten und Publikationsprozessen
- Umsetzung des Konzepts in einem lauffähigen Prototypen
 - Auswahl einer internen Informationsstruktur
 - Spezifikation eines Schnittstellenmodells
 - Implementierung der Plattform
 - Umsetzung von Beispielkomponenten
- Verifikation der Ergebnisse mit ausgewählten Use Cases

Problemstellung

Die Problemstellung der Arbeit ist die Beantwortung der folgenden Fragen:

- Ist eine komponentenbasierte Architektur für eine solche Art von Publikationsplattform geeignet?
- Wie können unzureichend strukturierte Informationen mit strukturierten Informationen sinnvoll kombiniert werden?
- Können gängige Anwendungsfälle aus der Praxis mit der Plattform abgebildet werden?
- Wo liegen Grenzen und Limitierungen des Konzepts?
- Welche alternativen Nutzungsszenarien und -möglichkeiten ergeben sich auf Basis der Umsetzung?

1.5 ABGRENZUNG

Diese Arbeit konzentriert sich auf die Entwicklung einer konzeptuellen Publikationsplattform, die prototypisch umgesetzt werden soll. Die Lösung ist ein *Proof of Concept* und soll Möglichkeiten und Grenzen eines solchen Konzepts im Bereich der TD aufzeigen sowie als Basis für weitere Entwicklungen dienen.

Um den Umfang einer Master-Thesis zu wahren, werden einige untergeordnete Themen des klassischen Content-Management nicht behandelt. Dazu gehören Mehrsprachigkeit, Variantenmanagement und Versionierung. Für die Bereitstellung des Contents werden ausschließlich Publikationsmechanismen, jedoch keine *On-Demand-Auslieferung* (*Content-Delivery*) betrachtet.

2

GRUNDLAGEN UND ÜBERSICHT

In diesem Kapitel werden grundlegende Begriffe und Methoden aus den Bereichen Content und Publishing behandelt. Komponentenbasierte Systeme im Allgemeinen und Publishing-Systeme im Speziellen werden vorgestellt.

2.1 CONTENT

2.1.1 Überblick

Inhalte sind der zentrale Bestandteil jeder Publikation, unabhängig von deren Ausprägung und (technischer) Umsetzung. Diese Inhalte werden im Bereich der professionellen Inhaltsverwertung meist als *Content* (engl.: *Inhalt*) bezeichnet.

Content kann produziert² (*Content Creation*) (vgl. ROCKLEY 2003:82), erworben oder getauscht werden (*Content Syndication*) (vgl. LOHR/DEPPE 2013:17f). Der vorliegende *Content* kann anschließend von sog. Inhaltsverwertern verwaltet (*Content Management*), kombiniert, verteilt (*Content Distribution* oder auch *Content Delivery*) und veröffentlicht (*Publishing*) werden (vgl. DREWER/ZIEGLER 2011:301).

Typische Inhaltsverwerter sind Medienhäuser aller Art (TV, Radio, Print, Internet, etc.), Marketing-Abteilungen oder Agenturen aber auch spezialisierte Ausprägungen dieser, wie etwa die Technische Dokumentation (TD) einer Firma oder TD-Dienstleister.

In der vorliegenden Arbeit wird ausschließlich der Bereich der Inhaltsverwertung im Bereich der TD betrachtet.

2.1.2 Definition

Laut DUDEN 2015a handelt es sich bei *Content* um „qualifizierte[n] Inhalt“ und den „Informationsgehalt“ von Medien. Diese allgemeine Definition umfasst auch das Verständnis für *Content*, welches im Umfeld der TD vorherrscht. Dort bezeichnet *Content* alle Inhalte textueller und visueller Art, die modular in einem Ausgabemedium verwendet werden können (vgl. DREWER/ZIEGLER 2011:297).

² Unter die industriell geprägte Bezeichnung *produzieren* fällt z.B. das klassische Schreiben eines Textes.

In einiger Literatur aus dem Bereich des Dokumentenmanagements wird *Content* auch schlicht beschrieben als „Information [...], die in elektronischen Systemen zur Nutzung bereitgestellt wird.“ (KAMPFF-MEYER 2003:47).

Unter diese Definitionen fallen die häufig in XML erfassten textuellen Inhalte, inklusive Tabellen und Listen, sowie alle informationstragenden³ *Grafik-, Audio- und Video-Elemente*. Weiterhin können auch Mischformen der genannten Arten oder interaktive Anwendungen zu Content zählen (vgl. ZIEGLER 2013:23).

Gemeinsamkeit aller Ausprägungen von Content ist die Eigenschaft, Wissen oder Informationen zu tragen, die unabhängig vom Ausgabemedium für den Anwender von Nutzen sein können. Damit werden bei Anwendung des SSP-Prinzips alle rein gestalterischen oder medienspezifischen Elemente ausgeschlossen.

2.1.2.1 *Verwandte Begriffe*

Speziell im Bereich der TD hat sich mit der Zeit eine bestimmte Terminologie durchgesetzt, um die verschiedenen Ausprägungen von Content exakt zu bezeichnen:

MODUL

Als *Module* werden wiederverwendbare Informationseinheiten von Content bezeichnet (vgl. DREWER/ZIEGLER 2011:295). Sie sind Grundlage für Single-Source-Publishing (SSP) und Content Management (CM) bzw. CCM.

Eine spezielle Ausprägung von Modulen sind *Topics*, die hauptsächlich im DITA- und Software-Dokumentations-Umfeld anzutreffen sind.

INFORMATIONSOBJEKT

Informationsobjekte sind Module einschließlich der ihnen zugehörigen Verwaltungs- und Metadaten. Module liegen in einem CMS in der Regel als Informationsobjekte vor.

DOKUMENT

Ein *Dokument* ist eine festgelegte Zusammensetzung mehrerer Module (vgl. DREWER/ZIEGLER 2011:298). Die Aggregation der Module kann hierbei automatisiert oder manuell erfolgen. Dokumente sind unabhängig von Zielmedium und Layout (vgl. DREWER/ZIEGLER 2011:298).

PUBLIKATION

Unter einer *Publikation* versteht man in der Regel die Instanz eines Dokuments in einem Medium, also dessen medienspezifische Ausprägung (vgl. DREWER/ZIEGLER 2011:298). Eine Publikation entsteht durch *Publishing* (Publizieren) eines Dokuments.

³ *Informationen tragen* wird hier analog zu *Informationsgehalt haben* (vgl. DUDEK 2015a) verwendet

In dieser Arbeit wird mit *Publikation* immer die Gesamtheit aller Ergebnisse eines Publikationsprozesses beschrieben. Diese (etwas abweichende) Definition ergibt sich daraus, dass beim Cross-Media-Publishing mehrere medienspezifische Ausgaben gleichzeitig erstellt werden können, die in ihrer Gesamtheit als Publikation betrachtet werden sollen.

2.1.3 Klassifizierung

Content kann auf viele verschiedene Arten klassifiziert werden. In der Redaktionsarbeit spielen hierbei vor allem praktische Gesichtspunkte eine Rolle, wie etwa die Verwertbarkeit, Strukturierung oder Herkunft der Quellen.

2.1.3.1 Strukturierungsgrad

Da in der Technischen Dokumentation die Modularisierbarkeit von Content eine wichtige Rolle spielt, ist eine Klassifizierung nach *Strukturierungsgrad* sinnvoll. Nach KAMPFFMEYER 2003 kann hier zwischen drei Stufen unterschieden werden: *Strukturiert*, *schwach strukturiert* und *unstrukturiert*. Angepasst auf die Gegebenheiten im Bereich der TD kann Content aufbauend auf DREWER/ZIEGLER 2011:297 wie folgt eingeteilt werden:

STRUKTURIERTER CONTENT

Relationale Daten oder hierarchisch strukturierte Informationen, die in standardisierter Form aus oder in einem elektronischen System zur Verfügung gestellt werden (z.B. XML-Daten).

SCHWACH STRUKTURIERTER CONTENT

Informationen und Dokumente, die in nicht standardisierter Form zur Verfügung gestellt werden, jedoch zum Teil Struktur- und Metainformationen tragen können (z.B. Daten aus Textverarbeitungsprogrammen).

UNSTRUKTURIERTER CONTENT

Daten- oder Medienobjekte, die keine Strukturinformationen tragen oder deren Struktur nicht direkt ersichtlich ist (z.B. binäre Daten)

2.1.3.2 Herkunft

Wie schon in Abschnitt 2.1.1 beschrieben, kann Content intern erstellt oder aus einer externen Quelle bezogen werden. Im Bereich der TD wird Content im Regelfall in einem Informationsmodell erfasst. Durch den anhaltenden Trend zur Digitalisierung und Datenintegration in Unternehmen kommt es aber immer öfter vor, dass Content von externen Quellen zugeliefert wird, der nicht dem verwen-

deten Haupt-Informationsmodell entspricht bzw. abweichend oder überhaupt nicht strukturiert ist (vgl. ZIEGLER/BEIER 2014:50).

Diese externen Quellen können sowohl außerhalb (z.B. Zulieferer) als auch innerhalb des Unternehmens (z.B. Marketing-Abteilung) liegen. Für die spätere Verwertung (etwa die Integration in ein Dokument) spielen Informationsmodell und Strukturierungsgrad (siehe Abschnitt 2.1.3.1) des Contents aber eine große Rolle. Aus diesem Grund wird in vielen Redaktionen (speziell bei Dienstleistern) eine Einteilung in die beiden Herkunftsformen vorgenommen.

PRIMÄRCONTENT

Als *Primärcontent* wird der Teil des Gesamtcontents eines *Dokuments* beschrieben, der im Haupt-Informationsmodell der Publikation erfasst wird und dessen Strukturierung damit sichergestellt ist.

SEKUNDÄRCONTENT

Bei *Sekundärcontent* handelt es sich um die Teile einer Publikation, deren Struktur und Informationsmodell nicht denen des *Primärcontents* entsprechen. Bei Technischen Publikationen sind dies in der Regel Zusatzdokumente, wie Stücklisten oder Konstruktionsdaten, die in das Dokument integriert werden sollen, jedoch nicht aus der TD stammen.

2.1.4 Methoden

2.1.4.1 Content Management

Als Content Management (CM) bezeichnet man das „Erfassen, Verwalten und Publizieren von unternehmensinternen oder -externen technischen Informationen [...]“ (DREWER/ZIEGLER 2011:291). CM zeichnet sich in der TD vor allem durch das Prinzip der modularen Wiederverwendung von Informationseinheiten (SSP) und dem automatisierten medienspezifischen Publizieren (CMP) aus (siehe Abschnitt 2.3.3).

Im englischsprachigen Raum wurde zur Abgrenzung von *Document Management* und domänenspezifischen Ausprägungen wie etwa *Web Content Management* die Bezeichnung Component Content Management (CCM) eingeführt, um die modulare, komponentenbasierte Inhaltsverwaltung zu betonen (vgl. ANDERSEN 2011:386f).

In der Regel wird CM mit Hilfe von sogenannten Content-Management-Systemen (CMS) umgesetzt, die Redakteure aus technischer Sicht bei der Umsetzung der entsprechenden Methoden (Bedatung, Retrieval und Verwaltung von Modulen bzw. Informationsobjekten) unterstützen. Im deutschsprachigen Raum ist neben CMS noch die synonym verwendete Bezeichnung *Redaktionssystem* im Umlauf (vgl. DREWER/ZIEGLER 2011:292).

CMS dienen als zentrale Datenquelle für modulbasierten Content im Redaktionsprozess. Dieser wird fast immer in Form von XML-codierten Objekten gespeichert und verwaltet (vgl. STRAUB/ZIEGLER 2008:204).

2.1.4.2 *Content Delivery*

Durch die stetig steigende Informationsmenge in Unternehmen, der Verbreitung von mobilen Endgeräten wie Smartphones und Tablets sowie dem allgemeinen Trend zur Digitalisierung haben sich in den letzten Jahren sogenannte Content-Delivery-Portale (CDP) am Markt platziert (vgl. ZIEGLER/BEIER 2014:50).

Bei diesen Systemen werden Informationen aus einer zentralen Quelle (etwa dem CMS) *on-Demand* über eine datenbasierte Schnittstelle zur Verfügung gestellt oder der Content wird aus mehreren, auch heterogenen Quellen aggregiert. Dargestellt werden die Inhalte in der Regel in einem Anzeigerahmen (etwa in Form eines *Portals* oder eines *Viewers*) auf Nutzerseite (vgl. OEVERMANN 2014b:86). Nach ZIEGLER/BEIER 2014 bieten Content-Delivery-Portale „die webbasierte Bereitstellung von modularen, aggregierten oder dokumentbasierten Informationen [...]\". Sie vereinen das Prinzip der zentralen Datenhaltung (SSP) mit den von Multi-Source-Portalen (MSP) bekannten Zugriffsmöglichkeiten.

In den von CMS-Anbietern vertriebenen Content-Delivery-Lösungen kommt in der Regel ein proprietäres XML-basiertes Format (meist angelehnt an das Standard-Informationsmodell des CMS), zum Einsatz. Lösungen, die außerhalb der TD entstanden sind, setzen aufgrund der geringeren Dateigröße tendenziell eher auf JSON als Datenaustauschformat (vgl. CONTENTFUL GMBH 2015).

2.1.4.3 *Content Enhancement*

Als *Content Enhancement* kann allgemein die automatisierte und konzeptbasierte Anreicherung bestehenden Contents oder seiner Metadaten verstanden werden (vgl. ZIEGLER/BEIER 2014:53). Dies können Erweiterungen des Modulinhalts sein, etwa in Form von (neu entstandener) Interaktivität (vgl. OEVERMANN 2014b:84), durch das Einblenden von Zusatzinformationen wie Begriffserklärungen (vgl. OEVERMANN 2013:25) oder durch automatisierte Klassifizierungen anhand von statistischen, semantischen oder linguistischen Analysen (vgl. OEVERMANN 2014a).

Die Anreicherung kann dabei nur auf den vorliegenden Informationen beruhen oder externe Quellen miteinbeziehen (z.B. Referenzkataloge). *Content Enhancement* gewinnt aufgrund der steigenden Informationsmengen besonders im Bereich der automatischen Verschlagwortung und Indexierung von Modulen und Dokumenten (vgl. ZIEGLER/BEIER 2014:53f) immer mehr an Bedeutung.

Zur systematischen Umsetzung von *Content Enhancement* im Bereich der TD gibt es (noch) keine spezialisierten Systeme. In den meisten Fällen werden die Methoden innerhalb eines Content-Delivery-Portal (CDP)s umgesetzt oder durch vorgelagerte Verarbeitungen außerhalb des Systems generiert (vgl. HEIDECK/STEURER 2013). Durch das große Potential von Content Enhancement z.B. im Bereich der automatisierten Indexierung sind hier jedoch Innovationen zu erwarten (vgl. ZIEGLER/BEIER 2014:55).

2.1.4.4 *Enterprise Content Management*

Enterprise Content Management (ECM) wird von der Association for Information and Image Management (AIIM) in einem Essay definiert:

„Enterprise Content Management (ECM) is the strategies, methods and tools used to capture, manage, store, preserve, and deliver content and documents related to organizational processes.“ (AIIM 2012)

Im Unterschied zu CM schließt diese Definition jeden Content ein, der in einem Unternehmen produziert wird (vgl. ROCKLEY 2003:4). Explizit wird auch der Bezug zu Unternehmensprozessen betont, der auch in enger Verbindung zum wirtschaftlichen Erfolg steht: „Content must be managed so that it is used to achieve business goals.“ (AIIM 2012).

Sowohl AIIM 2012 als auch ROCKLEY 2003 betonen bei der Beschreibung von ECM den Fokus auf einer unternehmensweiten ganzheitlichen Strategie, die verfolgt werden muss, um Informationen in einem Unternehmen effizient und abteilungsgrenzenübergreifend (vgl. ROCKLEY 2003:5) zu verteilen.

Dazu werden in vielen Firmen sog. *Enterprise-Content-Portale* eingesetzt, die „internen und externen Nutzergruppen den differenzierten Zugriff auf Content und Dokumente aus unterschiedlichen Quellen“ (ZIEGLER/BEIER 2014:52) bieten.

2.2 DATENFORMAT

2.2.1 Überblick

Nach der Definition aus Abschnitt 2.1.2 kann Content in nahezu jedem Format abgebildet oder enthalten sein. Im Bereich der Inhaltsverwertung und des Publishing (siehe Kapitel 2.3) sind jedoch vor allem *Markup-Formate* von großer Bedeutung (vgl. CLOSS 2007:15). In diesem Kapitel werden einige für die Arbeit wichtige Formate vorgestellt und bewertet. Eine besondere Rolle nimmt dabei HTML5 ein, dessen Neuerungen und Besonderheiten herausgearbeitet werden.

2.2.2 XML

XML steht für Extensible Markup Language und ist eine seit 1998 standardisierte Auszeichnungssprache für hierarchisch strukturierte Daten (vgl. W3C 2008). Extensible Markup Language (XML) selbst ist eine *Metasprache*, durch deren Einschränkung andere Formate definiert werden können. Die Syntax baut auf einer Untergruppe der Standard Generalized Markup Language (SGML) (vgl. ISO 8879 1986) auf.

Im praktischen Einsatz wird XML fast immer mit einem Informationsmodell eingesetzt, das die zugelassene Struktur und deren Knoten in einem Regelwerk bestimmt. Durch eine solche Strukturdefinition wird die XML-Syntax zur konkreten XML-Ausprägung eines Formats (z.B. XHTML). Um die Austauschbarkeit von Content zu gewährleisten, sollte auf standardisierte Informationsmodelle zurückgegriffen werden (vgl. KRÜGER/ZIEGLER 2008:15).

XML ist ein textbasiertes Format und wird durch die ausgereifte Standardisierung und die hohe Flexibilität auch in Zukunft das präferierte Format für die Speicherung von Content im Umfeld der TD bleiben (vgl. KRÜGER 2012:108). Primärcontent liegt in der Regel als XML vor.

In einem CMS werden Content-Module in den meisten Fällen als XML-kodierte Informationsobjekte gespeichert und verwaltet (vgl. STRAUB/ZIEGLER 2008:204). Auch wenn ein Ansatz ohne Redaktionssystem gewählt wird, ist die Ablage von Modulen als einzelne XML-Dateien nicht unüblich (etwa im DITA-Umfeld).

2.2.3 JSON

Die JavaScript Object Notation (JSON) ist ein textbasiertes Austauschformat für strukturierte Daten. Spezifiziert wird JSON in konkurrierenden Standards von ECMA⁴ (vgl. ECMA-404 2013) und der IETF (vgl. IETF RFC 7158 2013). Die Syntax entspricht der Objektnotation in JavaScript (JS). Damit ist es möglich, JSON-Dateien in JS-Programmen direkt und ohne Parser auszuwerten. Dennoch ist das Format unabhängig von einer Programmiersprache.

Besonders im Bereich der webbasierten Technologien konnte sich JSON gegenüber XML aufgrund der geringeren Dateigrößen, einfacheren Syntax und leichteren Maschinenlesbarkeit durchsetzen. Im Bereich der TD hingegen wird JSON kaum eingesetzt, da wichtige Funktionalitäten wie Strukturdefinition, Typsystem⁵ und Transformationssprache fehlen.

⁴ ECMA stand ehemals für *European Computer Manufacturers Association*.

⁵ Bei XML ist ein echtes Typsystem nur in Kombination mit *XML Schema* möglich.

2.2.4 RDF

Das *Resource Description Framework (RDF)* wurde vom World Wide Web Consortium (W3C) ursprünglich als Standard zur Beschreibung von Metadaten konzipiert. Das Konzept kann Sachverhalte (*Aussagen*) über beliebige Dinge (*Ressourcen*) beschreiben. Diese Informationen werden in *Tripeln* (3er-Informationssätzen) gespeichert. Tripel bestehen aus *Subjekt* (das Beschriebene), *Prädikat* (die Beziehung oder die Art der Aussage) und *Objekt* (das Objekt oder das Ausgesagte).

Für universelle Eindeutigkeit wird eine Ressource in der Regel mit einem Uniform Resource Identifier (URI) oder einem Internationalized Resource Identifier (IRI) repräsentiert. Durch das Einbinden standardisierter Namensräume, kann auf vordefiniertes Vokabular aus Metadaten-Katalogen zum Bilden von Aussagen zurückgegriffen werden.

Die Herangehensweise von RDF ist grundsätzlich unabhängig von einem Format oder einer textuellen Repräsentation findet jedoch häufig als eine XML-Sprache ihre Anwendung (vgl. W3C 2014b). Über RDFa kann das Prinzip auch in Hypertext Markup Language (HTML)-Dokumenten verwendet werden (vgl. W3C 2015b). RDF eignet sich jedoch weniger zur Erfassung von Content, sondern eher zur Beschreibung von Beziehungen und Prozessen.

2.2.5 HTML5

Bei HTML5 handelt es sich um die aktuelle Fassung der *Dokumentenbeschreibungssprache* Hypertext Markup Language (HTML), die gegenüber ihren Vorgängern um zahlreiche wichtige Funktionen ergänzt wurde (vgl. W3C 2015a). Der HTML-Standard wird vom World Wide Web Consortium (W3C) in sogenannten *Recommendations* gepflegt. Mit der neuen Spezifikation wurden auch viele Grenzfälle und Ungenauigkeiten früherer Versionen ausgeräumt.

HTML selbst versteht sich als „World Wide Web's core markup language“ (W3C 2015a:1.1) und gehört zu den am meisten eingesetzten Technologien im Internet.

Hinweis: Die Zitationen in diesem Teil beziehen sich auf die Abschnitte der jeweiligen Spezifikationen.

2.2.5.1 Syntax

HTML entstand in seiner ursprünglichen Form als Untermenge der standardisierten Auszeichnungssprache SGML (ISO 8879 1986). Um Datenaustausch und maschinelle Verarbeitung zu verbessern und mit dem Ziel, die beiden Standards zu vereinheitlichen entstand nach einer Zeit die Extensible Hypertext Markup Language (XHTML), eine Variante, die HTML in valider XML-Syntax abbildet (vgl. W3C 2010).

Die beiden Syntax-Varianten von HTML bestehen bis heute nebeneinander und führen zu Verwirrung und Inkonsistenz. So ist selbst in der offiziellen W3C-Spezifikation vermerkt:

„It must be admitted that many aspects of HTML appear at first glance to be nonsensical and inconsistent.“

(W3C 2015a:1.5)

Laut Spezifikation versteht sich HTML5 als übergeordnetes Konzept, das Inhalte mit einem bestimmten Vokabular auszeichnet (vgl. W3C 2015a:1.6). Die Repräsentation dieses Vokabulars kann auf verschiedene⁶ Arten erfolgen, von denen drei näher definiert wurden: Als HTML, als XHTML oder im Document Object Model (DOM) (vgl. W3C 2015a:1.6). Da es sich beim DOM nur um eine abstrakte Repräsentation⁷ handelt, kann HTML5 demnach in zwei verschiedenen Syntaxen ausgezeichnet werden: HTML und XHTML.

Beide Syntax-Varianten haben den selben Namensraum:

„The HTML namespace is the namespace both for documents in the HTML syntax and for documents in the XML syntax.“ (W3C 2013b:3.2)

<http://www.w3.org/1999/xhtml>

	HTML5	XHTML 1.1
Basierend auf	SGML	XML
XML-Deklaration	<i>keine</i>	<i>empfohlen</i>
Namensraum	HTML (<i>implizit</i>)	HTML (<i>explizit</i>)
Doctype	<!DOCTYPE html>	<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN">
MIME Type	text/html	application/xhtml+xml

Tabelle 1: HTML5: Arten von Syntax

Die XHTML-Syntax von HTML5 wird manchmal umgangssprachlich auch als XHTML5 bezeichnet. Die vom W3C bevorzugte Syntax ist HTML. HTML5 wird (unabhängig von der Syntax) von fast allen modernen Browsern schon unterstützt (vgl. SCHOBER 2012:36).

Einer Verbesserung der Interoperabilität zwischen HTML- und XML-basierten Systemen ist noch immer Gegenstand vieler Kontroversen und Untersuchungen (vgl. W3C 2012).

⁶ „There are various concrete syntaxes that can be used to transmit resources that use this abstract language [...]“ (W3C 2015a:1.6)

⁷ Das DOM ist das Modell des Dokuments, das z.B. ein Browser in den Speicher lädt und auf dessen Element per definierter Schnittstellen zugegriffen werden kann (die sog. Web Application API).

2.2.5.2 Semantik

HTML entstand ursprünglich als semantische Auszeichnungssprache für wissenschaftliche Dokumente (vgl. W3C 2015a), jedoch waren Element-Bezeichnungen zu Beginn fast ausschließlich darstellungsorientiert (vgl. W3C 2013b). Mit HTML5 wurden neue semantische Elemente und Attribute eingeführt, mit denen sich Content nach Funktion und Inhalt auszeichnen lässt (vgl. MOZILLA DEVELOPER NETWORK 2015a). Dazu gehören z.B. Makrostrukturelemente, wie `<section>`, `<aside>` oder `<footer>` aber auch Inline-Elemente, wie `<kbd>` (*Keyboard Input*), `<var>` (*Variable*) oder `<samp>` (*Sample Output*) (vgl. MOZILLA DEVELOPER NETWORK 2015a).

2.2.5.3 Implizitität

Der Erfolg und die rasche Verbreitung von HTML beruhen mit hoher Wahrscheinlichkeit auch auf der gegenüber XML weniger strikt ausgelegten Parser-Interpretierung, die so auch in den jeweiligen Spezifikationen festgehalten wurde (W3C 2015a, W3C 2013b). So werden Strukturen, Elemente oder Werte, die der Autor nicht angegeben hat, implizit an der entsprechenden Stelle eingefügt oder die Groß-/Kleinschreibung von Element- und Attributnamen freigestellt.

Dadurch soll es Autoren leichter gemacht werden, Dokumente möglichst einfach und mit wenig *Overhead* zu erstellen und Parsern (z.B. im Browser) die Verarbeitung von eigentlich korrupten HTML-Dokumenten noch zu ermöglichen.⁸

Beispiele für die Implizitität auf Elementebene sind die *Empty attribute syntax* (`<element attribut>`), die *Unquoted attribute value syntax* (`<element attribut=wert>`) und die *Optional tags* (das Auslassen diverser Start- und Endtags) (vgl. W3C 2015a:8.1.2.3f).

Die Implizitität auf Strukturerbene eines Dokuments geht so weit, dass es möglich ist, nahezu die gesamte Makrostruktur eines HTML-Dokuments (`<head>`, `<body>`, etc.) auszulassen (siehe Code 1 und Code 2). Die fehlenden Strukturen werden vom Parser zur Laufzeit eingefügt (vgl. W3C 2013b:3.3.1.1).

```

1 <!DOCTYPE html>
2 <title>Titel des Dokuments</title>
3 <p>Text des Dokuments</p>

```

Code 1: Notierte Struktur in HTML5

⁸ Im Gegensatz zu XML, wo z.B. ein Syntax-Fehler einen Abbruch des Vorgangs zur Folge hätte.

```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Titel des Dokuments</title>
5   </head>
6   <body>
7     <p>Text des Dokuments</p>
8   </body>
9 </html>
```

Code 2: Interpretierte Struktur in HTML5

Alle aufgeführten Techniken gelten ausdrücklich nur für die HTML-Syntax von HTML5, da XHTML-Dokumente wohlgeformtes XML sein müssen. Dokumente in dieser Syntax müssen ein `<html>`-Element im korrekten Namensraum als Wurzelement besitzen (vgl. W3C 2013b:3.3.2).

2.2.5.4 Erweiterbarkeit

Schon seit seiner ersten Fassung kannte das HTML-Vokabular Attribute und Elemente, mit denen sich die eigentliche Syntax nach Bedarf erweitern ließ: `<meta>`-Tags im Kopf eines Dokuments, `rel`-Attribute für die Beschreibung von Links oder das `class`-Attribut mit dessen Einsatz an neutralen Elementen wie `<div>` oder `` *de facto* neue Elementtypen geschaffen werden konnten (vgl. W3C 2015a).

In Begleitdokumenten zur HTML5-Spezifikation wurden neue semantische Erweiterungen des Basisvokabulars definiert, die von Autoren zur Auszeichnung benutzt werden können, um HTML-Dokumente valide zu erweitern.

DATA-ATTRIBUTE

Mit HTML5 wurde die Möglichkeit eingeführt, für beliebige Elemente Attribute zu definieren, deren Name mit `data-` beginnt und von mindestens einem Zeichen gefolgt wird (wie z.B. `data-length`). Diese `data`-Attribute sind explizit nur für den *privaten*⁹ Gebrauch innerhalb des Dokuments vorgesehen (z.B. für administrative oder programmatische Zwecke) und nicht für die Auswertung durch *externe* Software wie sog. *Crawler* (vgl. W3C 2015a:3.2.5.9). Soll eine externe Auswertung, z.B. von semantischen Auszeichnungen ermöglicht werden, so verweist die Spezifikation auf alternative Methoden wie z.B. HTML Microdata (vgl. W3C 2015a:3.2.5.9).

⁹ *Privat* bedeutet in diesem Zusammenhang, dass die Daten nur für den Besitzer des Dokuments oder Entitäten, die dem Besitzer des Dokuments bekannt sind, gedacht sind und nicht öffentlich spezifiziert werden.

Pro Element können beliebig viele `data-Attribute` angefügt werden. Diese haben keine Auswirkung auf die Darstellung des Elements im Browser.

MICRODATA

Mit *HTML Microdata* wurden neue Mechanismen eingeführt, die es erlauben, maschinenlesbare Daten mit einer einfach zu schreibenden Syntax in HTML-Dokumente zu integrieren (vgl. W3C 2013a:4.1). *HTML Microdata* soll dabei kompatibel zu anderen Datenformaten wie JSON oder RDF bleiben.

Microdata erweitert die Idee der Auszeichnung mit `data-Attributen` um *Scopes*, *Types* und *Properties*. Diese können als `itemscope`-`itemtype`- und `itemprop`-Attribute an Elementen deren Inhalte näher spezifizieren. Ziel ist es dabei, die *menschlesenbare* Darstellung (als gerendertes HTML) von der *maschinenlesbaren* Repräsentierung zu trennen. *Types* dienen bei HTML als Namensräume für *Properties*. Um einen Austausch mit Microdata erweiterter Dokumente zu ermöglichen, können diese Namensräume den standardisierter Metadaten-Kataloge entsprechen (z.B. der *Dublin Core Metadata Initiative*).

Über bestimmte festgelegte Algorithmen kann so aus einem mit *Microdata* versehenen HTML-Dokument z.B. eine maschinenlesbare JSON-Datei erzeugt werden, die zum Austausch zwischen Programmen benötigt wird (vgl. W3C 2013a:7.1).

RDFa CORE

Mit *RDFa Core* (kurz: *RDFa*) soll dem Problem begegnet werden, dass zwar eine enorme Menge an strukturierten Informationen als HTML-Seiten im Internet existiert, diese jedoch nicht durch Programme verarbeitet werden können, da die benötigten semantischen Auszeichnungen fehlen (vgl. W3C 2015c:1).

Dazu wird das Konzept von RDF auf Web-Dokumente übertragen und mit Hilfe von Attributen umgesetzt. Verwendet werden dazu entweder schon bestehende HTML-Attribute wie `href` oder `src` oder neu eingeführte Attribute wie `vocab` oder `property` (vgl. W3C 2015b:1.2).

Die Beschreibung des Contents erfolgt mit Hilfe der aus RDF bekannten IRIs (Internationalized Resource Identifier). Diese werden entweder direkt oder in Form von CURIEs (Compact URI Expressions)¹⁰ verwendet (vgl. W3C 2015c:2).

Durch die neu hinzugekommenen Erweiterungen, die eine echte semantische Auszeichnung von Content und den Einsatz von standardisierten Metadaten in HTML-Dokumenten zulassen, wird das Format auch für die TD interessanter. Bei einer Publikation von XML nach

¹⁰ CURIEs entsprechen in etwa den Namespace-Prefixes aus XML

HTML können so semantische Merkmale z.B. als *RDFa* erhalten bleiben (siehe Code 3 und Code 4).

```

1 <?xml version="1.0" encoding="utf-8" ?>
2 <concept>
3   <title>Konzept</title>
4   <conbody>
5     <p>Beschreibung des Konzepts</p>
6   </conbody>
7 </concept>
```

Code 3: DITA in XML

```

1 <html xmlns="http://www.w3.org/1999/xhtml"
2   prefix="dita: http://dita.xml.org">
3   <title>HTML-Dokument</title> <!-- nötig für Validität -->
4   <section property="dita:concept">
5     <h1 property="dita:title">Konzept</h1>
6     <p property="dita:conbody">Beschreibung des Konzepts</p>
7   </section>
8 </html>
```

Code 4: DITA mit RDFa in (X)HTML5

2.2.5.5 Strukturierung

Die Content-Struktur von HTML-Dokumenten ist in der Regel flach, was im Gegensatz zur hierarchisch strukturierten Natur von XML-Dokumenten steht (vgl. WALMSLEY 2011:4). Hierarchie wird in HTML durch *Heading*-Elemente erzeugt, deren Hierarchiestufe fester Namensbestandteil ist (z.B.: `<h1>`, `<h2>`, etc.). Dies führt zu diversen Problemen, insbesondere beim Zusammenführen verschiedener Dokumente (vgl. MOZILLA DEVELOPER NETWORK 2015c).

Eine automatische Gliederung von HTML-Dokumenten über blockbildende Elemente (z.B. `<div>`) war bisher sehr fehleranfällig, da Elemente, die nicht strukturbildend, sondern nur für gestalterische Zwecke eingesetzt waren, nicht eindeutig erkannt werden konnten (vgl. MOZILLA DEVELOPER NETWORK 2015c).

Im Zuge der Einführung neuer semantischer Elemente in HTML5 wurde auch die Gliederungslogik (engl.: *outline*) von Dokumenten überdacht. Strukturbildend (für die Gliederung) sind nun ausschließlich `<section>`-Elemente (vgl. W3C 2015a:4.3.3). Für Inhalte, die nicht strukturell relevant sind (z.B. die Seitenavigation), stehen andere Elemente zur Verfügung (z.B. `<nav>`).

Um die Zusammenführung von Dokumenten zu vereinfachen oder z.B. den Austausch von *sections* zu ermöglichen, wird ein `<section>`-Element durch eine Überschrift erster Ebene identifiziert (`<h1>`) (vgl.

W3C 2015a:4.3.3). Jedes <section>-Element kann indes seine eigene Überschriften-Hierarchie haben (vgl. MOZILLA DEVELOPER NETWORK 2015c). Damit lassen sich diese beliebig schachteln und ein hierarchisch strukturiertes Dokument in validem HTML5 erstellen.

```

1 <h1>Erste Ebene</h1>
2 <p>Text der ersten Ebene</p>
3 <h2>Zweite Ebene</h2>
4 <p>Text der zweiten Ebene</p>
5 <h3>Dritte Ebene</h3>
6 <p>Text der dritten Ebene</p>
```

Code 5: Sectioning in HTML4

```

1 <section>
2   <h1>Erste Ebene</h1>
3   <p>Text der ersten Ebene</p>
4   <section>
5     <h1>Zweite Ebene</h1>
6     <p>Text der zweiten Ebene</p>
7     <section>
8       <h1>Dritte Ebene</h1>
9       <p>Text der dritten Ebene</p>
10      </section>
11    </section>
12  </section>
```

Code 6: Sectioning in HTML5

2.3 PUBLISHING

2.3.1 Überblick

Publishing (engl.: *veröffentlichen*) oder *Publizieren* bezeichnet den Vorgang, „etwas öffentlich zu machen“ und ist in dieser ursprünglichen Bedeutung älter als der Buchdruck selbst (vgl. BHASKAR 2013:16). Obwohl der Begriff des *Publishing* schon immer eng mit Printmedien verbunden war, bezog er sich schon von Beginn an auf alle Medienarten (Musik, Sprache, Bilder, etc.).

Auch wenn im ursprünglichen Sinne nicht notwendig, war Technologie schon immer Treiber und Grundlage von Publishing (vgl. BHASKAR 2013:43). Entsprechend beschäftigen sich *Publisher*, egal ob Medienhäuser oder Technische Redaktionen, in der heutigen Zeit mit neuen Technologien, um Informationen öffentlich zugänglich zu machen.

2.3.2 Definition

Im Bereich der TD bezeichnet *Publishing* den Vorgang, aus einer Sammlung von modularem Content eine ausgerichtete *Publikation* zu erstellen (siehe dazu auch Abschnitt 2.1.2.1). Diese Ausrichtung kann z.B. nach den Faktoren Zielmedium, Zielgruppe, Dokumenttyp oder Sprache erfolgen (vgl. DREWER/ZIEGLER 2011:302f). Die methodische Grundlage dafür bietet das Cross-Media-Publishing (CMP) (vgl. DREWER/ZIEGLER 2011:296).

In dieser Arbeit wird *Publishing* analog zu Cross-Media-Publishing verwendet, jedoch mit dem Unterschied, dass der Vorgang der Veröffentlichung des Contents nicht zwingend mit dem Übergang in ein Zielmedium verbunden sein muss.

2.3.3 Methoden

2.3.3.1 Single-Source-Publishing

SSP gehört zu den Kernmethoden des *Content Management* und beschreibt die kontrollierte Wiederverwendung von Informationseinheiten (vgl. DREWER/ZIEGLER 2011:295). Gleiche Inhalte werden nur einmal erstellt, um danach aus einer zentralen Quelle (*Single Source*) in verschiedenen Zusammenstellungen wiederverwendet zu werden (*Publishing*) (vgl. CLOSS 2007:28). Voraussetzung für eine effiziente Nutzung der Methode ist die Festlegung der geeigneten Größe der Informationseinheiten (*Module* oder *Topics* genannt) (vgl. DREWER/ZIEGLER 2011:295) und ggf. die technische Unterstützung zur Verwaltung und Verwendung dieser z.B. mit einem CMS (vgl. Closs 2007:28).

2.3.3.2 Multi-Source-Publishing

Multi-Source-Publishing ist eine Methode, die erst in den letzten Jahren ihren Weg in die TD gefunden hat. Durch die gestiegene Informationsflut im Pre-und After-Sales-Bereich, die einer gestiegenen Produktkomplexität geschuldet ist, sowie der zunehmenden Prozess- und Datenintegration in Unternehmen (vgl. ZIEGLER/BEIER 2014:50) wuchs der Bedarf nach Portal-ähnlichen Lösungen, die neben der primären Quelle für technische Informationen, dem CMS, auch weitere Quellen (siehe Abschnitt 2.1.3.2) mit einbinden können (*Multi Source*).

Dies steht zunächst im Gegensatz zur Methode des SSP, ist aber bei Publikationen, die nicht rein technischer Natur sind, oft nicht zu vermeiden, da Inhalte aus anderen Abteilungen oder Bereichen integriert werden müssen, die nicht der Kontrolle der TD unterliegen.

2.3.3.3 *Cross-Media-Publishing*

Cross-Media-Publishing (CMP) baut in der Regel auf Single-Source-Publishing auf und beschreibt das Publizieren von medienneutralen Content-Modulen in verschiedene spezifische Ausgabemedien (vgl. DREWER/ZIEGLER 2011:296). Durch den Einsatz von XML als Auszeichnungsformat für strukturierte und klassifizierte Inhalte konnte sich das Prinzip in der TD durchsetzen, so dass mittlerweile der Großteil von Unternehmen ihre Inhalte in zwei bis vier verschiedene Medien publizieren (vgl. STRAUB/ZIEGLER 2008:44).

2.3.4 *Technologien*

Um *Publishing* in die verschiedenen Zielmedien und deren Formate und Spezifikationen möglich zu machen, bedarf es verschiedener Technologien. Die in der TD und angrenzenden Feldern üblichen und für diese Arbeit relevanten Werkzeuge sind im Folgenden kurz aufgelistet.

2.3.4.1 *Transformation*

XSLT

XSL Transformation (XSLT) ist eine XML-basierte Transformationssprache. Sie wird eingesetzt, um XML-basierte Dokumente in ein anderes XML-Format, HTML oder Text zu transformieren. XSLT ist eng verknüpft mit der XML Path Language (XPath), die zur Knotenadressierung eingesetzt wird. Der aktuell einsetzbare Standard ist XSLT 2.0 bzw. XPath 2.0.

Im Bereich der TD hat XSLT eine große Bedeutung bei der Transformation von XML-Dateien (vgl. HAUSER 2010:71). Alternativen (z.B. *MetaMorphosis*) werden nur vereinzelt eingesetzt.

JAVASCRIPT

Die dynamische Skriptsprache JavaScript (JS) (ECMA-262 2015) kann zum Transformieren und Manipulieren von JSON-Dateien bzw. JS-Objekten verwendet werden und kommt dort häufig zum Einsatz.

Zeitweise wurden spezielle Methoden für die Manipulation von XML in den JS-Standard übernommen (vgl. ECMA-357 2005), jedoch mittlerweile durch generische DOM-Methoden ersetzt. Mit diesen ist ebenso möglich XML-Dokumente zu lesen, zu transformieren oder zu erstellen, jedoch wird aufgrund der besseren Integration eine Alternative mit JSON als Austauschformat oft bevorzugt verwendet (siehe dazu auch Abschnitt 2.2.3).

2.3.4.2 Datenzugriff

Der Zugriff auf den eigentlichen Content kann bei einer Publikation auf verschiedene Arten und Weisen erfolgen:

DATEIBASIERT

Der Content steht in Form von Dateien zur Verfügung und kann über ein lokales Dateisystem oder über ein Netzwerk bezogen werden. Diese Variante stellt die einfachste und am häufigsten vorkommende Zugriffsform für Sekundärcontent dar.

SCHNITTSTELLE

Der Content wird von einem System über eine Schnittstelle bezogen, die Parameter entgegennimmt (z.B. eine ID) und den Content als Datenobjekt zurückgibt. Diese Variante ist vor allem bei Web-CMS (vgl. OEVERMANN 2012:6f) oder bestimmten API-basierten CDP (vgl. CONTENTFUL GMBH 2015) verbreitet.

DATENBANK

Der Content liegt in einer Datenbank als Datenobjekt und kann mit einer Abfrage (*query*) bezogen werden. Diese Variante ist die Basis vieler Content-Anwendungen (z.B. CMS) (vgl. DREWER/ZIEGLER 2011:437) und damit häufigste Zugriffsform auf Primärcontent.

Oft stehen auch mehrere Zugriffsarten zur Verfügung. Bei einem klassischen CMS sind etwa alle drei Varianten denkbar: Dateibasiert durch Export aus CMS, Schnittstellen-basiert durch Bereitstellung einer API durch das CMS und Datenbank-basiert durch Umgehung des CMS und direkten Zugriff auf die Datenbank.

2.3.4.3 Stapelverarbeitung

Der Publishing-Prozess besteht neben dem Transformationsteil oft auch aus Zwischenschritten, die der Automatisierbarkeit dienen und Dateioperationen oder Aufrufe selbstständig tätigen. Darunter fallen z.B. Kopieren, Verschieben, Löschen und Umbenennen von Dateien, das Komprimieren oder Dekomprimieren von Archiven, das Aufrufen oder Ausführen von Transformationen sowie das Herstellen von Verbindungen zu Datenbanken oder Servern.

Im Bereich der TD werden diese Zwischenschritte oft durch *Batch*- oder *Shell*-Skripte automatisiert, um ein manuelles Eingreifen unnötig zu machen (vgl. DREWER/ZIEGLER 2011:436). Bei komplexeren Publikationsstrecken werden auch Build-Tools aus der Softwareentwicklung eingesetzt (z.B. *Apache Ant* oder *Apache Maven*).

2.4 KOMPONENTENBASIERTE SYSTEME

2.4.1 Überblick

Im Folgenden soll ein kurzer Überblick über komponentenbasierte Systeme gegeben werden, der besonders den Bereich der komponentenbasierten Entwicklung von Software beleuchtet. Dabei werden die für diese Arbeit relevanten Definitionen formuliert.

2.4.1.1 Komponenten und Systeme

Das Wort *Komponente* stammt vom lateinischen *componens* („das Zusammensetzende“) und bezeichnet im Allgemeinen einen Bestandteil bzw. ein Element eines Ganzen (vgl. DUDEN 2015b). Im technischen Umfeld bilden Komponenten funktionale Einzelteile eines komplexeren *Systems* ab und können sowohl physischer als auch virtueller Natur (wie z.B. bei Software) sein. In der Regel wird versucht, einer Komponente in einem System genau eine spezifische Aufgabe zuzuordnen (vgl. KRUCHTEN 2004:284).

Je nach Domäne wird das Konzept einer Komponente auch als *Objekt* oder *Modul* bezeichnet (obwohl es hier wiederum je nach Fachgebiet genauere Abgrenzungen gibt). In dieser Arbeit wird eine *Komponente* als eine gekapselte funktionale Untereinheit eines größeren Systems betrachtet und in Abgrenzung zu *Modul* (in der TD eine wiederverwendbare inhaltliche Informationseinheit) und *Objekt* (in der TD eine technische Verwaltungseinheit) definiert.

Als *System* bezeichnet man eine Gruppe von interagierenden oder interdependenten Komponenten, die einen gemeinsamen (höheren) Zweck erfüllen (vgl. MERRIAM-WEBSTER 2015).

2.4.1.2 Komponentenbasierte Entwicklung

Das Prinzip, komplexe Software aus Komponenten aufzubauen, ist schon seit Jahrzehnten populär und hat sich seitdem unter verschiedenen Bezeichnungen schnell verbreitet. Dies hat zur Entstehung der *Komponentenbasierten (Software-)Entwicklung* geführt, einem Teilgebiet der angewandten Informatik. Weit verbreitet ist die englische Bezeichnung Component Based Development (CBD) oder auch das synonym verwendete Component Based Software Engineering (CBSE).

In der komponentenbasierten Entwicklung entstehen Softwarelösungen durch das Zusammenstellen von unabhängigen Komponenten, wobei das Verhalten des dadurch entstehenden Systems durch das Zusammenwirken der Komponenten erreicht wird (vgl. DE CESARE/LYCETT/MACREDIE 2006:4).

Klassische Beispiele für weitverbreitete komponentenbasierte Technologie sind z.B. Web-Services auf Basis der J2EE- oder .NET-Plattform. Beispiele für eine bekannte Software-Komponenten-Spezifikationen sind *JavaBeans* bzw. *Enterprise JavaBeans*.

Im Allgemeinen bietet die komponentenbasierte Entwicklung von Software eine Methode zur Applikationserstellung, die weniger anfällig gegenüber schnell wechselnden Anforderungen ist, eine hohe Wiederverwendungsrate zulässt und damit Entwicklungszeiten wesentlich verkürzt sowie Kosten für die Instandhaltung des Systems verringert (vgl. KROLL/KRUCHTEN 2003:42f).

2.4.2 Definition

Auch wenn keine universell anerkannte Definition für Software-Komponenten verfügbar ist, so können aus der Literatur die entscheidenden Charakteristika zusammengefasst werden (vgl. DE CESARE/LYCETT/MACREDIE 2006:5). Grundlage dafür ist eine oft referenzierte Definition von SZYPERSKI:

„A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties“

In SZYPERSKI 2002:36 wird eine Software-Komponente weitergehend durch drei Eigenschaften charakterisiert:

A component:

1. is a unit of independent deployment
2. is a unit of third-party composition
3. has no (externally) observable state

Daraus wird Bezug nehmend auf die drei Anforderungen vom Autoren geschlussfolgert (vgl. SZYPERSKI 2002:36):

1. Eine Komponente ist abgegrenzt von ihrer Umgebung und anderen Komponenten und kann nur als Ganzes verteilt werden
2. Eine Komponente muss in sich abgeschlossen sein, klar spezifizierte *Abhängigkeiten* und Funktionen enthalten und über definierten *Schnittstellen* kommunizieren
3. Eine Komponente muss nach außen zustandsunabhängig sein, darf sich also nicht von Kopien seiner selbst unterscheiden.

Als *context dependencies* werden die von den Komponenten benötigten Schnittstellen (*contractually specified interfaces*) sowie das standardisierte Framework¹¹ verstanden, in welches die Komponenten eingesetzt

¹¹ Als (Component) Framework wird in der komponentenbasierten Entwicklung das System bezeichnet.

werden (vgl. DE CESARE/LYCETT/MACREDIE 2006:6). Es dient als Infrastruktur und definiert die Rahmenbedingungen für die Komponentenstruktur (siehe auch Abschnitt 2.4.5).

Andere Definitionen heben weitere Eigenschaften hervor:

„A component is a non-trivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture.“

(KRUCHTEN 2004:284)

Hierbei wird die funktionelle Kapselung hervorgehoben, die einen Austausch einer Komponente innerhalb eines Systems ermöglicht. In einem komponentenbasierten System muss zur Kommunikation mit einer Komponente nur deren Schnittstelle definiert und bekannt sein; es muss kein Wissen über die inneren Vorgänge der Komponente vorliegen (vgl. KROLL/KRUCHTEN 2003:42). Durch diese Kapselung ist es z.B. möglich, dass eine Komponente komplett neu programmiert oder ausgetauscht werden kann, ohne dass sich das Verhalten des Gesamtsystems ändert, so lange sich die Schnittstelle der Komponente nicht verändert hat.

Aus technischer Sicht setzen sich Software-Komponenten aus sogenannten *atomaren Komponenten* zusammen, die jeweils aus einem Modul und zugehörigen Ressourcen bestehen (vgl. DE CESARE/LYCETT/MACREDIE 2006:26). Software-Komponenten sind weiterhin auf ein bestimmtes *Framework* abgestimmt, welches die Zusammensetzung der Komponenten vorgibt und überwacht (vgl. SZYPERSKI 2002:548).

„Eine Komponente (component) ist eine Softwarebasierte Zusammenfassung von Funktionalität, die nach technologischen Aspekten, wie der einfachen Integrationsmöglichkeit und jeweils nur einer Schnittstellenverbindung konzipiert wurde.“

(DUMKE 2013:58)

DUMKE definiert aus konzeptioneller Sicht und spezifiziert weiterhin die Arten von Kopplungen und inneren Bindungen, die Komponenten haben können, um Daten zu verarbeiten (vgl. DUMKE 2013:58f.). Siehe dazu Abschnitt 2.4.3.

2.4.3 Klassifizierung

Software-Komponenten können nach verschiedenen Merkmalen klassifiziert werden. Oft wird dies analog zum klassischen Begriff des *Software-Moduls* in den Bereichen *Kopplung* und *Innere Bindung* (Kohäsion) getan (vgl. DUMKE 2013:58).

2.4.3.1 Arten von Kopplung

Tabelle 2 auf Seite 28 zeigt die Arten von *Kopplung* nach DUMKE in aufsteigender Intensität.

2.4.3.2 Arten von Innerer Bindung

Tabelle 3 auf Seite 28 zeigt die Arten von *Innerer Bindung* (Kohäsion) nach DUMKE in aufsteigender Stärke der Bindung.

2.4.3.3 Empfohlene Konzeption

Nach DUMKE sollte bei der Konzeption von Komponenten in einem System „eine schwache Kopplung und eine starke Bindung angestrebt werden“ (DUMKE 2013:59). Dadurch sollen eine hohe Wiederverwendungsrate der Komponente bei gleichzeitig robuster Kapselfung erreicht werden.

2.4.4 Datenfluss

Einer der wichtigsten Aspekte in einem komponentenbasierten System ist der Datenfluss von Start- zu Endpunkt. Er wird durch die Kommunikation zwischen den Komponenten definiert, die Daten aneinander weitergeben und ist folglich Teil der Komponentenschnittstelle.

2.4.4.1 Dateibasiert

Bei einer dateibasierten Schnittstelle (*file based interface*) werden Daten in Form von Dateien weitergegeben (vgl. STOYE 2011:16). Dabei müssen die Dateien einem Datenmodell entsprechen, das von allen beteiligten Komponenten verarbeitet werden kann. In der Regel werden textbasierte Formate eingesetzt (im Gegensatz zu binären Dateien).

Ein dateibasierter Austausch von Informationen ist insbesondere beim Verknüpfen von heterogenen Systemen üblich. Oft wird dabei auf temporäre Dateien zurückgegriffen, die als Produkt der Zwischenschritte entstehen und nach Verarbeitung der gesamten Prozesskette wieder gelöscht werden können.

Ein Anwendungsfall ist z.B. die Verarbeitung von XML-Dateien zu PDF-Dateien via XSL-FO (siehe Abb. 3). Hierbei wird zunächst aus einer XML-Datei eine FO-Datei erzeugt, aus der anschließend eine PDF-Datei generiert wird. FO-Datei dient hierbei als dateibasierte Schnittstelle, die nach der Transformation wieder gelöscht werden kann.

BEZEICHNUNG	BESCHREIBUNG
Keine Kopplung	Zwischen den Komponenten besteht keine Kopplung
Datenkopplung	Komponenten übergeben nur Daten
Datenstrukturkopplung	Komponenten übergeben ganze Datenstrukturen
Steuerflusskopplung	Komponenten übergeben Steuerparameter für die Ablaufsteuerung
Datenexterne Kopplung	Komponenten sind über einen gemeinsamen Datenbereich gekoppelt
Datenflusskopplung	Komponenten modifizieren interne Daten von anderen Komponenten bzw. umgehen die eigentliche Schnittstelle

Tabelle 2: Arten von Kopplungen nach DUMKE

BEZEICHNUNG	BESCHREIBUNG
Keine Bindung	Aneinanderreihung von Funktionalitäten ohne jeglichen Bezug zueinander
Logische Bindung	Es besteht ein logischer Zusammenhang, z.B. als Menge aller Eingabefunktionen
Zeitliche Bindung	Es besteht ein zeitlicher Zusammenhang, z.B. für die Systeminitialisierung
Prozedurale Bindung	Es besteht die Forderung nach einer funktionellen Reihenfolge
Kommunikative Bindung	Es besteht eine gemeinsame Datennutzung
Sequentielle Bindung	Es besteht eine strenge Folge der Reihenfolge, da z.B. Funktionen Eingaben von Vorgängerfunktionen benötigen
Funktionale Bindung	Es besteht eine monolithische Zusammenfassung der Funktionalität

Tabelle 3: Arten von Innerer Bindung nach DUMKE

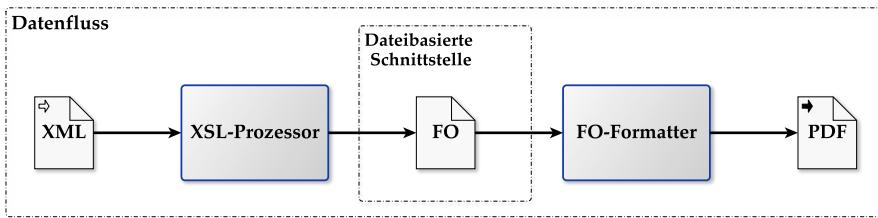


Abb. 3: Dateibasierte Schnittstelle: Beispiel XSL-FO

Vorteile dieser Methode sind die *hohe Flexibilität*¹² sowie eine mögliche *Persistenz* der Ergebnisdaten von Zwischenschritten. Nachteile können eine *langsamere Geschwindigkeit* (Lese-/Schreibzugriffe auf Speichermedium) sowie bei persistenten Daten ein *hoher Speicherbedarf* sein.

2.4.4.2 Datenstrombasiert

Bei einer datenstrombasierten Schnittstelle (*stream based interface*) werden Daten über die sogenannten *Standard-Datenströme* ausgegeben: `stdin` (Standardeingabe), `stdout` (Standardausgabe) und `stderr` (Standardfehlerausgabe). Ein- und Ausgabe sind normalerweise mit Tastatur bzw. Monitor verbunden, können aber auch umgeleitet oder gekoppelt werden. Der Vorgang des Koppelns zweier Komponenten über ihre Standard-Datenströme ist insbesondere bei Unix-basierten Betriebssystemen üblich und nützlich. Die Datenströme enthalten in der Regel Text.

Vorteile dieser Methode sind die *hohe Geschwindigkeit* (Datenströme werden nur im Arbeitsspeicher transportiert) sowie die *simple Kopp lung* über *Pipes* (siehe Abschnitt 2.4.6.3) direkt im Aufruf. Nachteile sind eine *fehlende Persistenz* (für Zwischenschritte) und die *geringere Verbreitung* von Datenstromschnittstellen im Windows-Umfeld.

2.4.5 Komponenten-Frameworks

Ein *Komponenten-Framework*, also das System, in dem Software-Komponenten agieren, ist eine Ansammlung von Regeln und Schnittstellen, die die Interaktion der einzelnen Komponenten miteinander steuert (vgl. SZYPERSKI 2002:548). In der Regel setzen Komponenten-Frameworks entscheidende Konventionen durch die Kapselung von Informationen um.

Frameworks können auch selbst Komponenten in einem höherrangigen Framework sein. Neben zweigliedrigen (z.B. Workflow- oder Integrationsserver) sind auch dreigliedrige Systeme im praktischen

¹² Der Großteil aller existierenden Programme kann standardmäßig Dateien einlesen und Dateien schreiben.

Einsatz anzutreffen (vgl. SZYPERSKI 2002:548). Die verschiedenen Hierarchien werden mit dem englischen Ausdruck *tier* in aufsteigender Reihenfolge bezeichnet. So ist z.B. ein *tier-1*-Framework eine Komponente in *tier 2*.

Bekanntester Vertreter von Frameworks für komponentenbasierte Entwicklung im Software-Bereich ist die Common Object Request Broker Architecture (CORBA) (siehe unter OMG 2012b). Dabei handelt es sich um einen von der Object Management Group (OMG) entwickelten Standard, der unabhängig von Programmiersprache und Implementierung eine komponentenbasierte Kommunikation in verteilten Systemen ermöglicht.

Der Rational Unified Process (RUP), ein kommerzielles Produkt, das von IBM vertrieben wird, beinhaltet unter Anderem ein Vorgehensmodell zur objektorientierten Softwareentwicklung und zählt zur ersten Generation der CBD-Frameworks. Zwar verfolgt der RUP einen allgemeinen Ansatz, es werden jedoch auch Methoden zur Komponentenkonzepion definiert. Im RUP sind Komponenten Implementierungseinheiten, die andere (fremde) Software-Artefakte beinhalten können und über ein Netzwerk verteilt werden (vgl. DE CESARE/LYCETT/MACREDIE 2006:26).

2.4.6 Konzepte und Prinzipien

Im Folgenden sollen Prinzipien vorgestellt werden, die nicht zwingend aus der komponentenbasierten Softwareentwicklung kommen, sondern ähnliche Ideen aus angrenzenden Bereichen einbringen. Gemeinsamkeiten aller Konzepte ist das Prinzip des *Separation of Concerns*, also der Aufteilung von Belangen in eigene Zuständigkeiten, was wiederum ein übergeordnetes Paradigma der komponentenbasierten Systeme ist.

2.4.6.1 Divide and Conquer

In der Informatik beschreibt das *Divide-and-Conquer-Prinzip* (dt.: Teile- und-herrsche-Prinzip) die Aufteilung eines Problems in kleinere und einfacher zu lösende (Teil-) Probleme. Dies geschieht rekursiv so lange, bis die Problemfragmente gelöst werden können. Anschließend wird aus den Teillösungen wieder eine Gesamtlösung rekonstruiert (vgl. HOFFMANN 2014:224).

Bezogen auf eine komponentenbasierte Architektur bietet dieses Prinzip einen Anhaltspunkt für Größe und Komplexität einer Komponente. So sollten bei einer Problemstellung die Komponenten den Lösungen der ermittelten lösbarren Teilprobleme entsprechen.

2.4.6.2 Hub and Spoke

Das *Hub-and-Spoke-Prinzip* (dt.: *Nabe und Speiche* oder *Speichenarchitektur*) bezeichnet ein in der Logistik (insbesondere der Luftfahrt) und dem Informationswesen verwendetes Konzept, bei dem in einem Netzwerk Verbindungen (*Spokes*) zwischen Start- und Endknoten nicht direkt, sondern über einen zentralen Knoten (*Hub*) hergestellt werden (vgl. BUSSLER 2003:10).

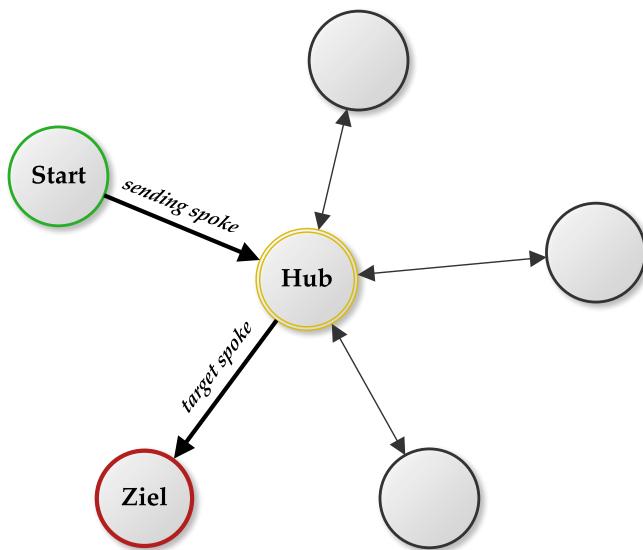


Abb. 4: Hub and Spoke: Schematische Darstellung

Bei einer komponentenbasierten Architektur kann dieses Prinzip auf den Informationsfluss innerhalb eines Systems bezogen werden. Der Prozess bzw. die Komponente von Dateneingang zu Datenausgang wird nicht direkt definiert, sondern in zwei Teile aufgegliedert: Dateneingang → *Hub* und *Hub* → Datenausgang. Diese Datenflüsse werden dann als *Spokes* bezeichnet.

Es gibt zwei Varianten des Informationsflusses innerhalb einer Hub-and-Spoke-Architektur (vgl. BUSSLER 2003:10):

1. Eingangskomponenten (*sending spoke*) bestimmen entsprechende Ausgangskomponente (*target spoke*), die vom Hub gewählt werden sollen.
2. Es findet keine direkte Adressierung statt und der Hub entscheidet basierend auf Regeln oder Inhalt über welche Ausgangskomponente die Daten der Eingangskomponente fließen.

Bei Variante 2 können Komponenten wesentlich unabhängiger voneinander definiert werden, da keine Informationen über weitere Komponenten vorliegen müssen.

Muss eine Transformation der Daten durchgeführt werden, so kann diese vor Übergabe an den Hub stattfinden oder danach. Der ideale Zeitpunkt hängt vom jeweiligen Anwendungsfall ab.

Der initiale Aufwand zum Erstellen der Teilprozesse (Komponenten) ist bei einer Hub-and-Spoke-Architektur höher als bei vergleichbaren Punkt-zu-Punkt-Lösungen, die Wiederverwendungswahrscheinlichkeit ist jedoch wesentlich höher. Werden neue *Spokes* an den Hub angeschlossen bleiben bestehende *Spokes* unbeeinflusst. Zur Integration ist nur eine Registrierung am Hub notwendig.

2.4.6.3 Pipes and Filters

Bei *Pipes-and-Filters* handelt es sich um ein Muster für Systeme, die Datenströme verarbeiten (vgl. BUSCHMANN 1998:54). Dabei wird jeder Verarbeitungsschritt in sogenannten *Filtern* gekapselt, welche wiederum durch *Pipes* (dt.: Kanäle) miteinander verbunden sind. *Filter*, die nicht aufeinander folgen tauschen demnach keine Informationen (direkt) aus (vgl. BUSCHMANN 1998:55).

Eine definierte Folge von *Pipes* und *Filtern* wird als *Pipeline* bezeichnet. Am Anfang einer *Pipeline* steht eine Datenquelle (*data source*), am Ende eine Datensenke (*data sink*) (vgl. BUSCHMANN 1998:56).

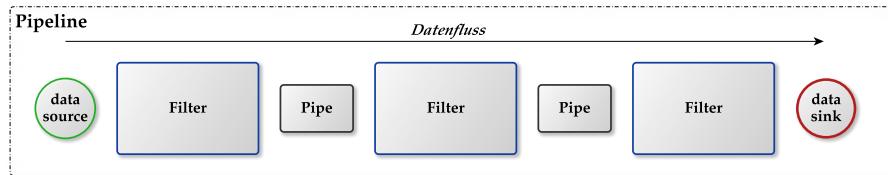


Abb. 5: Pipes and Filters: Schematische Darstellung

Durch eine Neuordnung der Filter wird der Datenstrom auf andere Weise verarbeitet, wodurch ein verwandtes System gebildet oder ein bestehendes System erweitert werden kann (vgl. BUSCHMANN 1998:55). Dies trägt sehr zur Konfigurierbarkeit einer Anwendung sowie zur Wiederverwendung von Filtern bei (vgl. BUSCHMANN 1998:69). Jedoch muss beim praktischen Einsatz sichergestellt werden, dass die *Filter* eines Systems entweder beliebige Datentypen verarbeiten können oder eine Validierung der Reihenfolge stattfindet.

2.4.6.4 Dependency Injection

Mit *Dependency Injection* (dt.: Abhangigkeitsinjektion) wird in der Informatik ein Konzept bezeichnet, das eine dynamische Abhangigkeitsauflosung von Komponenten oder Softwareobjekten beschreibt (vgl. FOWLER 2004). Im Gegensatz zu anderen Ansatzen wird hierbei die Verantwortung fur die Verwaltung von benotigten Komponenten

an einen Ausführungsrahmen (*Framework*) übertragen (vgl. OEHLSLE 2013:69). In der Regel findet die Auflösung der Abhängigkeit zur Laufzeit des Programms statt und das Framework bezieht die benötigte Komponente von einem zentralen Ort.

Bei der Architektur eines komponentenbasierten Systems kann *Dependency Injection* etwa dann eingesetzt werden, wenn eine Komponente aus anderen Komponenten zusammengesetzt wird oder zur Datenverarbeitung eine weitere Komponenten benötigt.

2.4.7 *Flow-based Programming*

Unter *flow-based programming* (dt.: flussbasierte Programmierung) versteht man ein Programmierparadigma, bei dem eine Applikation nicht als einzelner Prozess verstanden wird, sondern als ein Netzwerk asynchroner *Black-Box-Prozesse*¹³, die über Datenströme miteinander kommunizieren (vgl. MORRISON 2013). Dieses Netzwerk wird von einem externen Prozess definiert, den MORRISON als *Scheduler*¹⁴ bezeichnet.

Die *Black-Box-Prozesse* können neu verbunden oder kombiniert werden, um neue Applikationen zu formen. Sie sind eine Ausprägung von Komponenten. *Flow-based programming* zählt damit zur *komponentenbasierten Software-Entwicklung*.

Arbeiten im Bereich des *flow-based programming* befassen sich unter Anderem mit der Abstraktion komplexer Kontrollfluss-Operation (*Pipes, Filters, Buffer, etc.*) für Applikationen, die Datenströme verarbeiten (vgl. KOSTER et al. 2001:1f). Hierbei wird die Komplexität der Anwendungsentwicklung reduziert, indem ein Framework (oder eine Plattform) domänenspezifische Funktionen zur Verfügung stellt oder automatisch ausführt (vgl. KOSTER et al. 2001:2).

Eine solche domänenspezifische Anwendung, die nach den Prinzipien des *flow-based programming* Datenströme verarbeitet, ist auch für das Publizieren von Technischen Informationen denkbar und Grundlage dieser Arbeit.

¹³ Mit *Black Box* werden hier die typischen Eigenschaften von Komponenten bezeichnet: Die internen Prozesse sind für das Komponentensystem irrelevant, solange die Schnittstelle korrekt definiert ist.

¹⁴ In manchen Software-Projekten auch *Dispatcher* genannt.

2.5 KOMPONENTENBASIERTES PUBLIZIEREN

2.5.1 Überblick

Mit seinen ständig wechselnden Zusammenstellungen von ähnlichen Verarbeitungsschritten ist das automatisierte Publizieren von Informationen prädestiniert für die Umsetzung als komponentenbasiertes System. Was dabei als Komponente bezeichnet wird, hängt von der jeweiligen Umsetzung ab und kann vom einfachen Stylesheet bis hin zum voll integrierten Publikationsserver reichen.

Da sich das in dieser Arbeit vorgestellte Konzept mit komponentenbasiertem Publizieren beschäftigt, sollen neben einem kurzen Überblick über den aktuellen Stand der Technik auch die Möglichkeiten und Grenzen in diesem Bereich aufgezeigt werden.

2.5.2 Stand der Technik

2.5.2.1 Content-Management-Systeme

CMS haben in der Regel einen *Publikationsserver* integriert, der von den *Clients* des Systems Publikationsaufträge annimmt und abarbeitet. Die Quelldaten sind bei einer CMS-Publikation meist Inhalte aus der Datenbank des Systems und ggf. externe Dateien (vgl. DREWER/ZIEGLER 2011:437). Der textuelle Content liegt fast immer strukturiert als XML vor, binäre Dateien als verknüpfte Objekte.

Beim Starten der Publikation wird vom Benutzer das entsprechende Ausgabemedium gewählt. Je nach Medium wählt das CMS den passenden Publikationsprozess aus. Art und Format der Prozessdefinition im System sind durch die geschlossene Natur der Systeme fast immer proprietär, jedoch teilweise auf XML-basiert und konfigurierbar (vgl. OEVERMANN 2013:51).

Der eigentlich Publikationsprozess besteht meist aus einem linearen Ablauf von Verarbeitungsschritten, wie Dateioperationen, (mehrstufigen) Transformationen oder dem Aufruf von externen Programmen (z.B: *Compilern*).

Es ist anzunehmen, dass Systemhersteller bei ihren Publikationsservern zumindest teilweise einen komponentenbasierten Ansatz verfolgen, um auch neue Zielmedien schnell implementieren zu können. Jedoch sind diese durch die geschlossene Systemarchitektur oder die fehlende Dokumentation (vgl. OEVERMANN 2013:51) nicht durch den Benutzer frei konfigurierbar.

2.5.2.2 Content-Delivery-Portale

Zusammengefasst unter der (relativ neuen) Bezeichnung Content-Delivery-Portal (CDP) werden seit einiger Zeit Publikationslösungen auf dem Markt angeboten, die im Portal-Charakter und optimiert für den mobilen Gebrauch Informationen für Nutzer zur Verfügung stellen (vgl. ZIEGLER/BEIER 2014:50).

Gerade im Bereich der TD haben sich diese Systeme darauf spezialisiert modulare Daten aus dem CMS mit Sekundärcontent beliebiger Art zu kombinieren (sog. Multi-Source-Portale). Beispiele für Produkte aus diesem Bereich finden sich z.B. bei DOCUFY GMBH 2015, EASY-BROWSE GMBH 2015, SCHEMA GMBH 2015 und ARVATO CIM 2015.

CDP sind bei der Verarbeitung heterogener Quelldaten zwar sehr flexibel, in ihrer Content-Ausgabe jedoch meist auf eine proprietäre Portallösung oder einen Content-Reader spezialisiert. Einblicke in die internen Vorgänge der kommerziellen Lösungen sind nicht möglich; ein komponentenbasierter Import-Vorgang lässt sich nur vermuten.

2.5.2.3 DITA Open Toolkit

Um aus dem standardisierten Informationsmodell DITA (Darwin Information Typing Architecture) verschiedene Medien publizieren zu können, wurde eine Sammlung von modularen Werkzeugen entwickelt: das *Open Toolkit*. Das ursprünglich von IBM entwickelte DITA OT wurde 2005 als Open-Source-Software zur Verfügung gestellt (vgl. DAY/PRIESTLEY/SCHELL 2005:11) und wird seitdem von einer Gemeinschaft Freiwilliger weiterentwickelt.

Das DITA Open Toolkit (DITA OT) ist die einzige freie, plattformübergreifende DITA-Implementierung und gilt damit als die Referenzimplementierung des Standards (vgl. KIMBER 2012). Es dient als Publikationssystem für DITA, mit dessen Hilfe sich diverse Ausgabeformate erzeugen lassen (vgl. DITA OT 2015b).

Aus technischer Sicht setzt sich das DITA OT aus mehreren Bestandteilen zusammen: *Apache Ant*, *XSLT* und *Java* (vgl. DITA OT 2015a).

Das DITA OT setzt bei der Verarbeitung auf ein zweistufiges Konzept. Zunächst wird der DITA-Content unabhängig vom Ausgabekanal vorverarbeitet (*Preprocess*) und danach je nach Zielmedium weiterverarbeitet (siehe Abb. 6).

Im Bereich der komponentenbasierten Publishing ist DITA OT das wohl am besten getestete und erprobte System, das als freie Software zur Verfügung steht.

Das *Open Toolkit* selbst kann erweitert werden und damit auch andere Anwendungsfälle abdecken, ist jedoch immer an eine spezielle strukturierte Eingabe - DITA-Dateien - gebunden. Damit können nicht alle

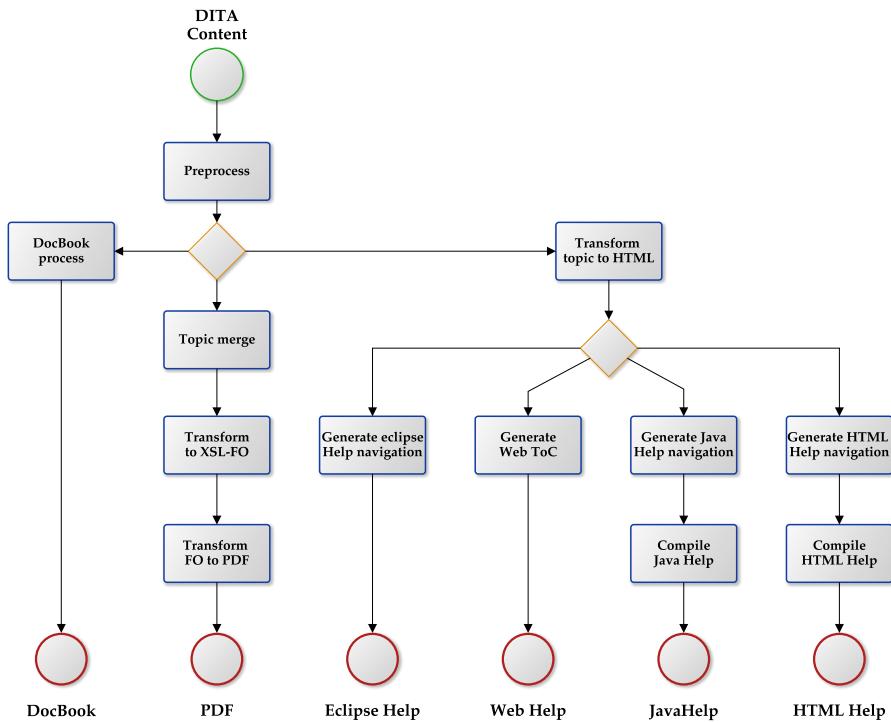


Abb. 6: DITA OT: Publikationsprozess nach dita-ot.org

Anforderungen an eine Publikationsplattform abgedeckt werden, die auch unstrukturierte Quellen verarbeiten kann.

2.5.2.4 Apache XML Project

Unter dem Name *Apache XML Project* wurden bis 2012 mehrere Projekte der *Apache Software Foundation (ASF)* zusammengefasst, die sich mit XML-Technologien beschäftigten. Ein Großteil dieser Projekte wird mittlerweile als *top level project* in der XML-Kategorie der ASF geführt (vgl. APACHE SOFTWARE FOUNDATION 2015b). Unter ihnen befinden sich auch Projekte, die sich mit dem komponentenbasierten Publizieren von heterogenem Content beschäftigt haben:

A P A C H E C O C O O N

Bereits seit 1998 wird *Apache Cocoon* als XML-basiertes Publikations-Framework entwickelt (vgl. BECHER 2004). Cocoon beruht auf sogenannten *component pipelines*, die sich zu einer Publikationsstrecke zusammensetzen lassen. In Pipelines werden die Informationsflüsse von Quelle zu Ziel beschrieben, innerhalb einer Pipeline werden Inhalte als XML repräsentiert. Mehrere Pipelines werden in *Sitemap*-Dateien zusammengefasst (vgl. APACHE SOFTWARE FOUNDATION 2013). Um die mitgelieferten Funktionalitäten zu erweitern, können sogenannte *Blocks* verwendet werden, die als Komponenten des Frameworks dienen. Diese

werden als Java-Archive (`.jar`) verteilt. Das letzte *Stable Release* von *Apache Cocoon* wurde Anfang 2013 veröffentlicht.

APACHE FORREST

Bei *Apache Forrest* handelt es sich um ein XML-basiertes Publikations-Framework, das die Eingabe verschiedener Informationsquellen in ein einheitliches Format umwandelt, um es anschließend in ein oder mehrere Ausgabeformate zu überführen (vgl. APACHE SOFTWARE FOUNDATION 2011). Das Framework ist modular und erweiterbar aufgebaut und basiert auf *Apache Cocoon*. Das Projekt entstand im Jahr 2002 mit dem Ziel, einfach und konsistent Webseiten für die Projekte des *Apache XML Project* zu generieren und hat sich bis zum derzeit letzten *Stable Release* Anfang 2011 zu einem generischen Publishing-Tool entwickelt. Als Hauptanwendungen werden die Publikation der Dokumentation von Software-Projekten sowie Intranets und Websites genannt.

APACHE XANG

Apache Xang war ein bis Ende 2009 aktives Projekt der ASF (vgl. APACHE SOFTWARE FOUNDATION 2009). Es handelte sich um ein XML-Web-Framework, das mehrere Datenquellen aggregieren und zusammenführen konnte, um die kombinierten Inhalte danach über HTTP-Methoden adressierbar zu machen. Eingesetzt wurde *Apache Xang* für das schnelle Erstellen datenbasierter Web-Anwendungen.

Alle vorgestellten Projekte aus der Apache-XML-Gruppe haben komponentenbasierte Ansätze im Bereich Publishing verfolgt. Getrieben waren die Entwicklungen in der Regel immer aus dem Bedürfnis heraus, Dokumentationen oder Websites für Software aus dem Apache-Umfeld zu erstellen. Die Konzepte verfolgen deshalb pragmatische Konzepte bezüglich der Handhabung von Content und dessen Struktur, die nicht alle Anforderungen, die aus der TD kommen, abdecken können.

Die komplizierte Handhabung der Programme und der Umstand, dass keines der Projekte derzeit aktiv weiterentwickelt wird, lässt sie als Basis für eine Publikationsplattform ausscheiden.

2.5.2.5 Pandoc

Die von John MacFarlane entwickelte Software *Pandoc* ist ein universelles Konvertierungstool für Dokumente in verschiedenen Markup-Sprachen (vgl. MACFARLANE 2014). Unterstützt wird unter anderem die Umwandlung zwischen HTML, Office-Dokumenten (z.B. `docx`), L^AT_EX (und darüber PDF), EPUB und DocBook.

Die Software wird als Kommandozeilenwerkzeug oder als Haskell-Bibliothek unter der GNU General Public License zur Verfügung gestellt.

Pandoc ist vollständig modular aufgebaut und verfolgt dabei einen *Hub-and-Spoke-Ansatz* (siehe Abschnitt 2.4.6.2):

„[P]andoc has a modular design: it consists of a set of readers, which parse text in a given format and produce a native representation of the document, and a set of writers, which convert this native representation into a target format. Thus, adding an input or output format requires only adding a reader or writer.“ (MACFARLANE 2015)

Die Module (oder Komponenten) aus denen Pandoc aufgebaut ist, unterteilen sich in sogenannte *Reader* und *Writer*, also Module, die Daten in einem bestimmten Format lesen oder schreiben können. Schnittstelle zwischen den beiden Modultypen ist eine „native representation“ des Contents, also ein intern verwendetes Informationsmodell (siehe Abb. 7).

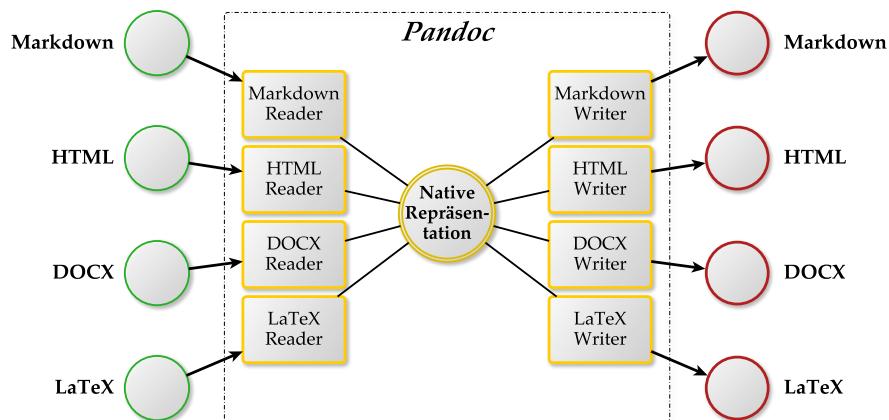


Abb. 7: Pandoc: Hub-and-Spoke-Architektur

Pandoc ist sehr beliebt in der Software-Dokumentation und wird dort vor allem für das Publishing von Markdown-Dokumenten eingesetzt. Durch den komponentenbasierten Ansatz ist eine hohe Flexibilität und Erweiterbarkeit gewährleistet. Da Pandoc ein reiner Konvertierer ist, können jedoch keine weiteren Verarbeitungen oder Anpassungen am Content bei der Publizierung vorgenommen werden. Dadurch kann es als Endpunkt einer Publikationsstrecke eingesetzt werden, jedoch nicht alle benötigten Anforderungen abdecken.

2.5.3 Weitere Ansätze

Weitere Publikationsmechanismen, die teilweise komponentenbasiert arbeiten, sind unter Anderem bei (DREWER/ZIEGLER 2011:434ff) aufgelistet. Des Weiteren entstehen derzeit neue Lösungen auf dem Markt, die sich in den meisten Fällen auf einen Content-Delivery-Ansatz spezialisieren und auf mobiles Publishing fokussiert sind

3

ANFORDERUNGEN

In diesem Kapitel werden auf Basis der Erkenntnisse aus Kapitel 2 Anforderungen definiert, erläutert und eingeteilt.

3.1 ALLGEMEIN

3.1.1 Ziel des Konzepts

Übergeordnetes Ziel des Konzepts ist es, Technischen Redakteuren und Informationsmanagern ein Werkzeug an die Hand zu geben, das die Erstellung von Publikationsstrecken erheblich vereinfacht und beschleunigt. Dieses Ziel soll erreicht werden, indem eine Abstraktionsschicht Konzept und technische Umsetzung einer Publikationsstrecke voneinander löst und bekannte Prinzipien aus der TD wie Modularität und Wiederverwendung begünstigt. Besonders berücksichtigt werden soll dabei die Integration von Sekundärcontent und die Integration in bestehende Redaktionsprozesse.

Die Publikationsplattform agiert dabei als eine Art Setzkasten, in dem Komponenten in verschiedenen Zusammenstellungen kombiniert werden können. Die Komponenten übernehmen jeweils eine Aufgabe innerhalb der Publikation (Import, Export, Verarbeitung, etc). Die Zusammenstellung aus Komponenten, Plattform und konkreten Datenquellen ergibt damit die Definition einer Publikationsstrecke.

Die Konzeption soll dabei aufbauend auf Erkenntnissen aus angrenzenden Bereichen und dem Wissen um spezielle Bedürfnisse der Technischen Dokumentation erfolgen.

3.1.2 Herkunft der Anforderungen

Die Anforderungen an die Konzeption einer komponentenbasierten Publikationsplattform stammen zum größten Teil aus Erfahrungen der täglichen Arbeit in Unternehmen, die sich mit TD beschäftigen müssen. Dabei sind besonders TD-Dienstleistungsunternehmen gefordert, da die große Vielfalt an Kunden und Projekten auch erhebliche Unterschiede in Art und Tiefe der Zusammenarbeit mit sich bringt (vgl. LINDNER/NITSCHE 2013:71). Für die vorliegende Arbeit sind besonders die Lieferung von Quelldaten in verschiedenen Formaten, die Vorstellungen hinsichtlich der Ausgabemedien sowie die Automatisierbarkeit der Prozesse hervorzuheben.

Diese Anforderungen sind speziell im Bereich der Publikationsstrecken-Entwicklung ausgeprägt und können von der Anpassung einer bestehenden Ausgabe in einem CMS (vgl. DREWER/ZIEGLER 2011:434f) bis hin zu kompletten Neuentwicklungen (vgl. OEVERMANN 2012) reichen. Die Problemstellung hängt dabei immer von der spezifischen Ausgangslage, den Quelldaten und den verfügbaren Systemen ab.

Bezüglich des Entwurfs von Softwaresystemen wird zum Teil auf die in den üblichen Normen festgehaltenen Anforderungen zurückgegriffen (siehe DIN EN ISO 9241-210 2011 und ISO/IEC 25010 2011).

3.1.3 Kernfunktionalität

Die Kernfunktionalität der Publikationsplattform lässt sich in wenigen Punkten zusammenfassen:

- Es kann Content aus mehreren, auch heterogenen Quelldaten in das System importiert werden
- Der Content kann im System manipuliert, verarbeitet und ggf. mit Informationen angereichert werden
- Der Content kann aus dem System heraus in mehrere Ziele exportiert werden.

In dieser Arbeit werden explizit nicht die Bezeichnungen Format, Informationsmodell oder Medium verwendet, sondern stattdessen umfassender von *Quellen* (bzw. *Eingabe*) und Zielen (bzw. *Ausgabe*) gesprochen. Für das interne Modell des Contents wird die Bezeichnung *interne Informationsstruktur* verwendet.

Die Anforderungen an Konzept und Umsetzung werden in *funktionale* und *nichtfunktionale* Anforderungen unterteilt.

3.2 NICHTFUNKTIONALE ANFORDERUNGEN

3.2.1 Konzeption

3.2.1.1 Komponentenbasiertes System

Als Lösung soll ein komponentenbasiertes System (also ein *Komponenten-Framework* mit *Komponenten*) nach den in Abschnitt 2.4.5 aufgestellten Kriterien entstehen. Dabei soll auf bestehenden Kenntnissen der Literatur aufgebaut werden. Insbesondere Empfehlungen zur Konzeption eines solchen Systems (siehe z.B. Abschnitt 2.4.3.3) sollen befolgt werden; etablierte Modelle und Prinzipien sollen angewendet werden (siehe Abschnitt 2.4.6).

Bestehende Systeme aus dem Bereich des Publishing und der TD sollen als Beispiele und Untersuchungsgegenstände dienen.

3.2.1.2 Flexibilität, Wartbarkeit, Änderbarkeit

Durch eine komponentenbasierte Form soll eine größtmögliche Flexibilität erreicht werden. Komponenten sollen neu kombiniert, ausgetauscht oder komplett neu erstellt werden können. Es muss möglich sein, auf ständig wechselnde Anforderungen in der TD reagieren zu können und ein zukunftssicheres Werkzeug zur Hand zu haben, das offen zur Weiterentwicklung ist. Komponenten sollen auch als gekapseltes Wissen verstanden werden, das wiederverwendet oder weitergegeben werden kann. Dadurch kann das Gesamtsystem (in diesem Fall die Publikationsplattform) leichter gewartet und mögliche Fehlerquellen besser isoliert werden. Eine Änderbarkeit des Systems soll durch eine vollständige Dokumentation gewährleistet sein.

3.2.1.3 Basierend auf Standards

Das Konzept soll auf bewährten und offenen Standards aufbauen, die im Umfeld der Technischen Dokumentation etabliert sind und nur wenn nötig, neue Modelle oder Formate einführen.¹⁵ Dadurch soll erreicht werden, dass verfügbares Wissen von oder über ähnliche Lösungsansätze in die Konzeption mit einfließen, um damit ein Auftreten des *Not-Invented-Here-Syndroms* zu verhindern, welches unter Umständen zur Ablehnung von externem Wissen und der Entwicklung proprietärer Lösungen führt (vgl. ODERMATT 2009:102). Standards bringen Neutralität, Sicherheit und langfristige Verfügbarkeit mit sich (vgl. CLOSS 2007:23), weshalb das Konzept nicht abhängig von proprietären Technologien sein soll. Eine Anpassbarkeit des Systems soll basierend auf Kenntnissen über die eingesetzten Standards möglich sein.

3.2.1.4 Fokus auf Content

Das Konzept soll eine ganzheitliche Betrachtung der Problemstellung verfolgen. Neben den technischen Aspekten (etwa der Umwandlung von Dateiformaten) soll der Fokus vor allem auf dem eigentlichen Content bzw. dessen innerer Struktur liegen. Das TD-spezifische konzeptionelle Wissen über strukturierte Inhalte soll schon in der Konzeption einfließen und damit eine rein technische getriebene Lösung verhindern. Sowohl semantische als auch darstellungsorientierte Ansätze zur Strukturierung von Content sollen bei der Entwicklung Beachtung finden. Verfahren zur Modellierung und Überführung des Contents sollen definiert werden.

Ein besonderer Fokus soll auf dem Umgang mit unstrukturiertem Content und dessen Kombination mit strukturiertem Content liegen.

¹⁵ Siehe dazu auch die humoristische Aufarbeitung von MUNROE 2011

3.2.2 Umsetzung

3.2.2.1 Ganzheitlicher Architektur

Schon in der Konzeptionsphase soll eine ganzheitliche Architektur bedacht werden, die funktionale Elemente der Lösung logisch auftrennt und getrennt nutzbar macht (z.B. durch eine Aufteilung in Schichten).

Die Lösung soll so aufgebaut sein, dass sie möglichst viele Anwendungsfälle abdecken kann. Deshalb soll bei der Konzeption verschiedenen Betriebs- und Anwendungsszenarien bedacht werden, die vom gezielten Einsatz einzelner Schichten bis hin zum umfassenden Betrieb einer kompletten Lösung reichen.

Die Architektur soll neben der eigentlichen Datenverarbeitung auch die Bereiche Benutzeroberfläche sowie lokale und netzwerkbasierte Schnittstellen umfassen, um sich optimal in bestehende Redaktionsprozesse einpassen zu können.

3.2.2.2 Ergonomische Bedienung

Die Umsetzung soll intuitiv bedienbar sein. Dazu soll dem Benutzer eine Oberfläche zur Verfügung stehen, die ihn bei der Bedienung unterstützt und leitet. Anweisungen und Hilfetexte sollen an entsprechenden Stellen in der Oberfläche integriert sein und sinnvolle Default-Werte oder implizite Konfigurationen automatisch vorgenommen werden, um dem Nutzer so viel Komplexität wie möglich zu verbergen.

3.2.2.3 Portierbarkeit und Unabhängigkeit

Wenn möglich sollen die Grundsätze des Konzepts unabhängig von einer technischen Umsetzung einsetzbar sein. Aber auch die eigentliche Implementierung soll soweit wie möglich plattformunabhängig und portabel sein. Dies soll durch Konformität und Standardisierung der zugrunde liegenden Technologien erreicht werden.

3.2.3 Freie Software

Die Umsetzung soll als *Freie Software* entwickelt werden und in der Standard-Ausführung nur auf Freier Software basieren. Dadurch sollen eine einfache Verbreitung und eine uneingeschränkte Anpassbarkeit gewährleistet werden.

Abhängigkeiten zu proprietärer Software sollen für Basisaufgaben nicht bestehen. Es soll jedoch prinzipiell möglich sein, proprietäre Programmteile in Komponenten zu integrieren.

3.2.4 Anpassbarkeit (Customizing)

Jedem Nutzer, der Zugriff auf die entsprechenden Quelldaten hat, soll es möglich sein, Anpassungen am System vorzunehmen. Dazu müssen sowohl ein theoretisches Konzept als auch eine technische Spezifikationen vorhanden sein.

Der Quellcode der Umsetzung muss ausreichend kommentiert und Beschreibungen für einzelne Bestandteile verfügbar sein.

3.3 FUNKTIONALE ANFORDERUNGEN

3.3.1 Formate

In einer Publikation muss neben den typischen CMS-Inhalten oft auch nicht zureichend strukturierter Sekundärcontent verwertet werden, der z.B. aus anderen Abteilungen oder Systemen stammt. Im Folgenden soll ein kurzer Überblick¹⁶ gegeben werden, welche Dateiformate dort zum Einsatz kommen:

- Dokumente: Portable Document Format (PDF), `indd`/`idml` (*In-Design-Dokument*)
- Datenaustausch: XML, JSON (im Web-Bereich), `YAML` (selten)
- Markup: HTML, Markdown (im Bereich der Software-Dokumentation), `LATEX` (im akademischen Bereich), XML-Informationsmodelle (z.B. DITA)
- Textverarbeitung: `docx`, `odt`, `rtf`, `doc` (nur für Altdaten relevant)
- Tabellen: `xlsx`, `csv`, `xls` (nur für Altdaten relevant)
- Grafiken (Pixel): `jpg`, `png`, `gif`, `bmp` (selten)
- Grafiken (Vektor): `eps`, `svg`, `ai` (*Adobe Illustrator*)
- Videos: `mp4`, `ogg`, `webm`, `mov`, `wmv`
- CAD-Daten: `dxf`, `obj`, `3ds`, `vrml`, `x3d`

Das Konzept soll es ermöglichen, theoretisch jedes der aufgeführten Datenformate verarbeiten zu können. In der Umsetzung soll dies gezeigt werden, indem aus den Kategorien Dokumente, Markup, Textverarbeitung, Grafiken und Videos jeweils mindestens ein Format technisch umgesetzt wird.

Durch die komponentenbasierte Architektur soll die Fähigkeit, ein neues Format verarbeiten zu können, per Komponente nachrüstbar sein.

¹⁶ Diese Liste soll lediglich einen Eindruck über die Formatvielfalt geben. Sie erhebt keinen Anspruch auf Vollständigkeit. Abkürzungen werden aus Platzgründen nicht ausgeschrieben.

3.3.2 Aufgaben

Alle folgenden Aufgaben sollen über oder in Komponenten innerhalb des Systems abgebildet werden.

3.3.2.1 Dateioperationen

Die Anforderungen aus Basis-Dateioperationen ergeben sich zu größten Teilen aus den Eigenschaften der zu verarbeitenden Dateiformate.

KOPIEREN, LÖSCHEN

Die beiden Grundfunktionen *Kopieren* und *Löschen* von Dateien und Ordnern müssen als Funktionen im System abgebildet sein.

Aus ihnen lassen sich auch die Operationen Umbenennen (Kopieren in Datei mit neuem Namen + Löschen der alten Datei) als auch Verschieben (Kopieren an neuen Ort + Löschen der alten Datei) abbilden.

TRANSFORMATION

Das System muss XSL-Transformationen für die Verarbeitung von XML-Dateien unterstützen. Dabei soll mindestens der Standard XSLT 2.0 / XPath 2.0 unterstützt werden.

ARCHIVIERUNG (ZIP)

Sowohl das Entpacken (Microsoft Word docx) (vgl. MEINIKE 2014:4) als auch das Komprimieren von Formaten (z.B. ePub) (vgl. MEINIKE 2010) soll als Funktion im System abgebildet sein.

LISTENERSTELLUNG

Es soll möglich sein Listen von Dateien in einem Ordner und von Ordnern in einem Ordner zu erstellen.

Diese Listen können z.B. als Grundlage für eine Stapelverarbeitung der enthaltenen Dateien/Ordner dienen.

DATEIABRUF

Das System soll eine Möglichkeit bereitstellen, nicht-lokale Dateien (z.B. aus dem Internet) abrufen und speichern zu können (vgl. OEVERMANN 2012:17).

DATEIERSTELLUNG

Dem System soll es möglich sein, neue Dateien zu erstellen oder in bestehende Dateien zu schreiben (Inhalt anhängen).

ORDNERVERWALTUNG

Das System kann Ordner und die enthaltenen Dateien erkennen, erstellen, löschen oder verschieben.

SUCHEN / ERSETZEN

Der Inhalt einer Datei kann durchsucht werden. Es ist möglich die Ergebnisse einer Suche durch ein vorgegebenes Muster zu ersetzen.

3.3.2.2 Datenverarbeitung

CONTENT-ZUSAMMENFÜHRUNG

Es soll möglich sein, den Content aus verschiedenen Quellen in einer Struktur zusammenzuführen. Sowohl Einzel- als auch Gesamtcontent sollen in gleichem Format und gleicher Informationsstruktur vorliegen. Strukturbildende Elemente des Gesamtcontents sollen mit IDs eindeutig identifiziert werden können.

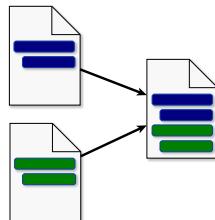


Abb. 8: Anforderung: Content-Zusammenführung

NEU- UND RESTRUKTURIERUNG DES CONTENTS

Es soll möglich sein, den Gesamtcontent (siehe oben: Content-Zusammenführung) basierend auf einer Strukturdefinition neu zu strukturieren. Ebenso soll es möglich sein, eine Strukturdefinition des aktuellen Contents zu exportieren. Diese kann dann als Grundlage für ein Umsortieren dienen.

Die Umstrukturierung soll neben der Reihenfolge auch die Hierarchie beeinflussen können, sowie das Löschen hierarchischer Blöcke ermöglichen.

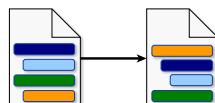


Abb. 9: Anforderung: Content-Restrukturierung

CONTENT-ANREICHERUNG UND -MANIPULATION

Es soll möglich sein, den Gesamtcontent mit Informationen anzureichern (z.B. Metadaten, Definitionen) oder ihn regelbasiert zu manipulieren (z.B. Referenzierungen). Auch sollen Verarbeitungen, die abgeleitete Informationen als Ergebnis haben (z.B. Indizes, Inhaltsverzeichnisse, Verschlagwortung) möglich sein (vgl. OEVERMANN 2014a).

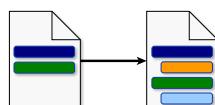


Abb. 10: Anforderung: Content-Anreicherung

3.3.3 *Cloud-Fähigkeit*

Die konkrete technische Implementierung der System-Architektur soll so gewählt werden, dass der Betrieb der Lösung als Cloud-Anwendung möglich ist. Dies kann z.B durch eine klare Trennung von Client und Server-Code, sowie einer Aufteilung von statischen und dynamischen Bestandteilen erfolgen.

Als Orientierung sollen die übliche Infrastruktur- und Serviceangebote von Cloud-Diensten (vgl. z.B. AWS Inc. 2015) dienen.

3.3.4 *Bedienung*

Es soll die Möglichkeit bestehen, die Umsetzung der Lösung über ein *Web-Interface* (Internetseite) bedienen zu können. Das heißt, eine komplette Publikationsstrecke kann im Browser definiert und ausgeführt¹⁷ werden.

¹⁷ Ausführen bezieht sich hierbei auf das Anstoßen des Rechenprozesses, nicht das Ausführen (berechnen) des Prozesses. Dies kann von einem entfernten Server übernommen werden.

4

KONZEPTION

In diesem Kapitel wird, basierend auf den Ergebnissen aus Kapitel 2, ein neutrales Konzept spezifiziert, das alle in Kapitel 3 genannten Anforderungen erfüllt und als Grundlage für eine technische Implementierung dienen wird.

4.1 ARCHITEKTUR

4.1.1 Modell

Die Architektur der Publikationsplattform besteht aus drei Schichten: *Publikationsserver*, *Web-API* und *Web-Client*. Die Schichten kommunizieren über definierte Schnittstellen miteinander. Ausgangspunkt für die Publikationsplattform ist ein abstraktes *Dateisystem*.

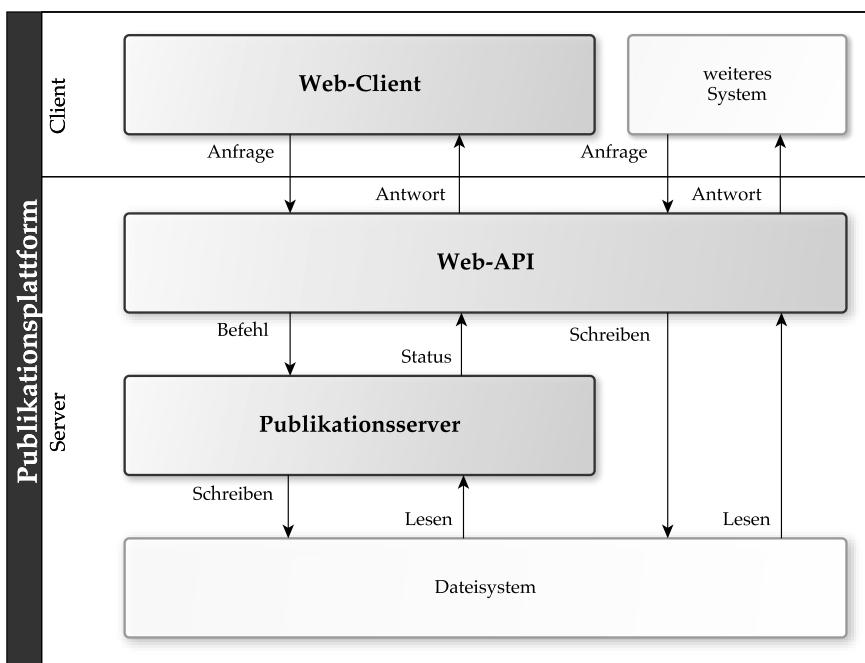


Abb. 11: Publikationsplattform: Konzeptionelle Architektur

4.1.2 Schichten

Die Plattform wird durch ihre Schichtenarchitektur funktional strukturiert. Die Schichten lassen sich entsprechend ihrer Funktion wie folgt beschreiben und voneinander abgrenzen:

PUBLIKATIONSSERVER

Der Publikationsserver¹⁸ bildet die Basisschicht der Architektur und enthält die *Business-Logik* des Systems. Er entspricht einem *Komponenten-Framework* entsprechend der Definition aus Abschnitt 2.4.5 mit zugehörigen *Komponenten*. In dieser Schicht finden alle Publikationsprozesse statt. Der Publikationsserver schreibt (z.B. *Publikationen*) und liest (z.B.: *Publikationsdefinitionen*) Dateien auf dem Dateisystem.

Eingaben sind *Befehle*, Ausgaben sind *Status-Meldungen*.¹⁹ Alle Funktionalitäten werden über Komponenten abgebildet.

WEB-API

Die Web-API dient als webbasierte Schnittstelle und Abstraktionschicht zwischen dem Publikationsserver und anderen Anwendungen. Sie entspricht einem Vermittler zwischen den verknüpften Systemen. In dieser Schicht finden die Verarbeitung und Beantwortung von Anfragen, Fehlerbehandlung und Datenumformungen statt. Die Web-API kann je nach Befehl auch direkt mit dem Dateisystem kommunizieren und dort Dateien schreiben und lesen.

Eingaben sind *Anfragen*, Ausgaben sind *Antworten*, jeweils bestehend aus Daten.

WEB-CLIENT

Der Web-Client stellt die webbasierte Benutzeroberfläche der Publikationsplattform dar. Er kommuniziert mit dem Publikationsserver über die Web-API. Innerhalb des Web-Clients finden Benutzerinteraktion, Anfrageformulierung und Antwortauswertung statt. Des Weiteren unterstützt der Web-Client den Benutzer durch sinnvolle Standardkonfigurationen und Eingabe-Validierungen.

Eingaben sind *Benutzereingaben* (Interaktionen), Ausgaben sind *Anfragen* an die Web-API .

¹⁸ Server wird hier im klassischen Sinne von *Dienstleister* für einen *Client* verwendet.

¹⁹ Der erfolgreiche Status einer Publikation kann dann z.B. das Lesen der Publikationsergebnisse zum Anlass haben.

4.1.3 Szenarien

Die drei Schichten der Plattform bauen in der Reihenfolge

Publikationsserver → Web-API → Web-Client

aufeinander auf. Daraus ergeben sich die folgenden drei Betriebsszenarien.

PUBLIKATIONSSERVER

Der Publikationsserver wird alleine betrieben, z.B. als integriertes Modul in einem CMS oder als Kommandozeilenwerkzeug für die Entwicklung oder die direkte Ausführung von Publikationsdefinitionen.

PUBLIKATIONSSERVER + WEB-API

Der Publikationsserver wird zusammen mit der Web-API betrieben, z.B. als netzwerkbasierter Publikationsservice für andere Content-Systeme, die ohne separaten Client direkt auf die Funktionalität zugreifen.

PUBLIKATIONSSERVER + WEB-API + WEB-CLIENT

Der Publikationsserver, die Web-API und der Web-Client werden gemeinsam betrieben, z.B. als umfassende Publikationslösung oder direkt für Endanwender in Form von *Publication as a Service (PaaS)*²⁰.

In dieser Arbeit wird hauptsächlich der gleichzeitige Betrieb aller drei Schichten betrachtet, da dieses Szenario die Anforderung an eine umfassende Lösung (sieh Abschnitt 3.2.2.1) am besten abdeckt. Die Erkenntnisse und getroffenen Aussagen lassen sich aber für alle drei Szenarien anwenden.

4.1.4 Anmerkung

Die funktionale Trennung in Schichten folgt den *Best Practices* aus dem Bereich der (webbasierten) Anwendungsentwicklung und kann analog zum Architekturmuster *Presentation-Abstraction-Control (PAC)* (vgl. BUSCHMANN 1998:145) betrachtet werden.²¹

Durch die Kommunikation über definierte Schnittstellen sind die einzelnen Schichten jeweils austauschbar oder in verschiedenen Zusammensetzungen kombinierbar (siehe Abschnitt 4.1.3). Eine Möglichkeit ist z.B. die Verwendung eines anderen Clients, um auf die Web-API zugreifen, oder eine Neu-Implementierung des Publikationsservers.

²⁰ Das es solche Services bisher nicht auf dem Markt gibt, werden sie in dieser Arbeit analog zu den ähnlichen Konzepten *Software as a Service (SaaS)* oder *Infrastructure as a Service (IaaS)* bezeichnet

²¹ Allerdings folgen die Schichten intern teilweise anderen Mustern, z.B: *Model-View-Controller (MVC)* beim Web-Client (siehe auch Abschnitt 5.11.2).

4.2 GRUNDLAGEN

4.2.1 Hauptbegriffe

Konzeptionelle Grundlage ist das Zusammenspiel von *Publikationssystem*, *Publikationsdefinitionen* und *Komponenten* mit den jeweiligen Quelldaten:

PUBLIKATIONSSYSTEM

Das *Publikationssystem* ist eine Sammlung von Konventionen, die festlegt, wie *Publikationsdefinitionen* und *Komponenten* aufgebaut sind, wie sie interpretiert werden und über welche Schnittstellen sie kommunizieren. Dies entspricht der Definition eines Komponenten-Frameworks (siehe Abschnitt 2.4.5).

Die Prüfung und Umsetzung dieser Konventionen ist wiederum in *Komponenten* abgebildet (konkret in der Gruppe der *Systemkomponenten*, siehe Abschnitt 4.4.3).

PUBLIKATIONSDEFINITION

Eine *Publikationsdefinition* ist eine Datei, in der eine spezifische Konstellation von Quelldaten und *Komponenten* in einem *Publikationsprozess* beschrieben wird. Der beschriebene *Publikationsprozess* wird vom *Publikationssystem* ausgeführt. Das Ergebnis kann eine *Publikation* sein.

KOMPONENTE

Eine *Komponente* ist eine Datei oder ein Dateipaket, das eine spezifische Funktion im *Publikationssystem* erfüllt. Dies entspricht der Definition einer klassischen System-Komponente (siehe Abschnitt 2.4.2).

Komponenten können aus anderen Komponenten zusammengesetzt sein oder als *Basiskomponenten* auf native Funktionen des *Host-Systems* zurückgreifen.

Sie können einzeln ausgeführt (z.B. *Systemkomponenten*) oder in *Publikationsdefinitionen* miteinander kombiniert werden.

4.2.2 Weitere Begriffe

Weitere in diesem Kapitel verwendete Bezeichnungen, werden im Folgenden für den Kontext dieser Arbeit definiert:

PUBLIKATION

Eine *Publikation* ist das Ergebnis der Ausführung einer vollständigen *Publikationsdefinition* in einem *Publikationsprozess*, also die Gesamtheit aller publizierten Medien in diesem Prozess (analog zur Definition in Abschnitt 2.1.2.1).

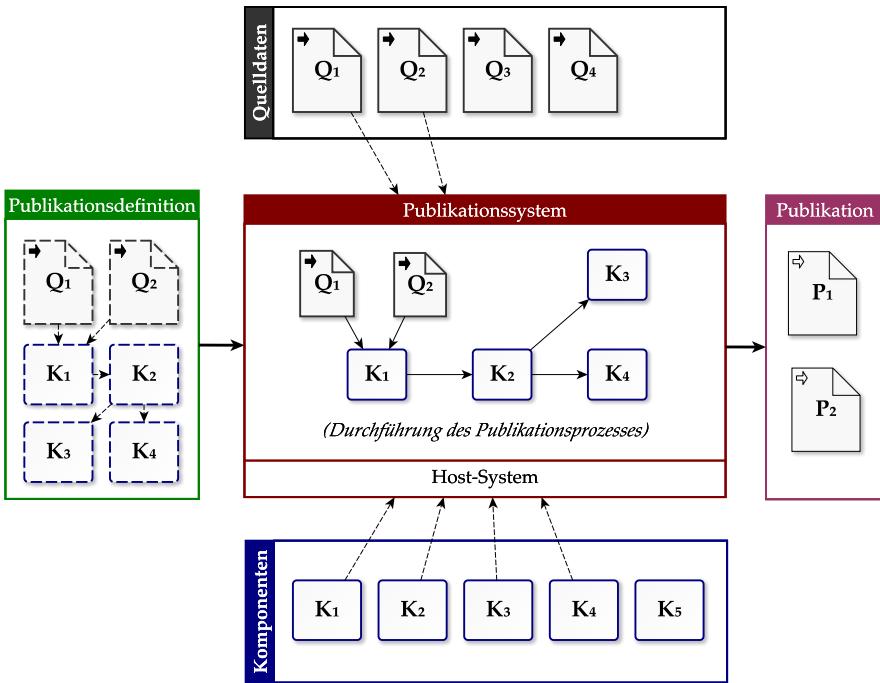


Abb. 12: Publikationssystem: Konzept

HOST-SYSTEM

Das *Host-System* ist das System, auf dem das Publikationssystem ausgeführt wird. Es gibt die konkrete Implementierung des Konzepts und dessen Funktionsumfang vor.

PUBLIKATIONSSERVER

Als *Publikationsserver* wird die Umsetzung des *Publikationssystems* als eine der drei Schichten der *Publikationsplattform* bezeichnet.

PUBLIKATIONSPLATTFORM

Die Kombination aus Publikationsserver, Web-API und -Client zu einer umfassenden Anwendung (siehe Kapitel 4.1).

4.3 PUBLIKATIONSPROZESS**4.3.1 Definition**

Ein *Publikationsprozess* ist der in einer *Publikationsdefinition* beschriebene Ablauf, um referenzierte Daten mit Komponenten zu verarbeiten.

4.3.2 Phasen

Zur Definition des Ablaufs wird der Publikationsprozess in drei Phasen eingeteilt: *Eingabe*, *Verarbeitung* und *Ausgabe*. In jeder dieser Pha-

sen können Komponenten eingesetzt werden. Der Hauptfluss der Informationen durchfließt bei einem vollständigen Publikationsprozess alle drei Phasen.

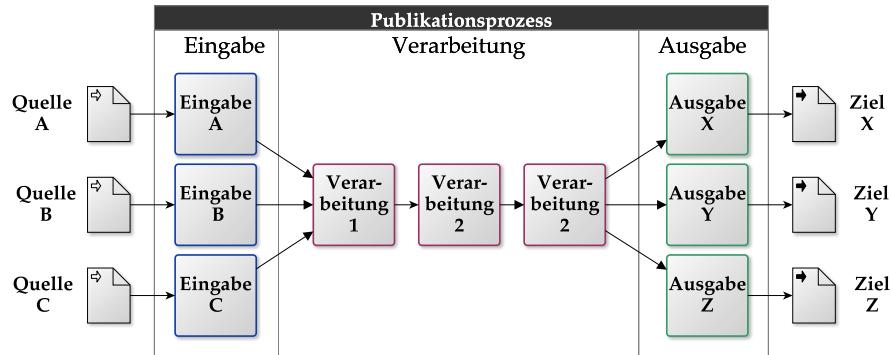


Abb. 13: Publikationsprozess: Phasen und Hauptfluss

EINGABE

Alle Komponenten der Eingabephase werden parallel verarbeitet. Sie nehmen Quelldaten entgegen und geben die intern genutzte Informationsstruktur aus. Jeder Komponente in dieser Phase wird eine Datenquelle zugeordnet.

VERARBEITUNG

Alle Komponenten der Verarbeitungsphase werden sequentiell verarbeitet. Sie arbeiten ausschließlich²² mit der internen Informationsstruktur. Eingabe einer Komponenten ist die Ausgabe der vorangegangenen Komponente.

AUSGABE

Alle Komponenten der Ausgabephase werden parallel verarbeitet. Sie nehmen die intern genutzte Informationsstruktur entgegen und geben Zieldaten aus. Jeder Komponente in dieser Phase wird ein Datenziel zugeordnet.

Das Prinzip, alle Quelldaten zunächst in eine einheitliche Informationsstruktur zu überführen und von dort aus weiterzuverarbeiten entspricht dem *Hub-and-Spoke-Prinzip* (siehe Abschnitt 2.4.6.2) und wird auch bei vergleichbaren Systemen eingesetzt (vgl. *Apache Cocoon* (Abschnitt 2.5.2.4) und *Pandoc* (Abschnitt 2.5.2.5)).

Durch diese Architektur ist es möglich, die Verarbeitung des Contents von Quell- und Zieldaten zu entkoppeln und damit eine echte Wiederverwendbarkeit einzelner Komponenten zu ermöglichen. Die sequentielle Verarbeitung in dieser Phase und die einheitliche Ein- und Ausgabe erlauben ein *Piping* von Komponenten (siehe Abschnitt 2.4.6.3).

²² Dies gilt für den Hauptfluss. In Nebenflüssen können alternative Informationsstrukturen verwendet werden.

4.3.2.1 Anmerkung

Durch die parallele Verarbeitung der Eingabedaten muss nach erfolgreicher Ausführung aller Komponenten eine Zusammenführung des Contents erfolgen, der dann als Basis für Komponenten der Verarbeitungsphase dient. Diese Funktionalität muss als eigene Komponente zur Verfügung stehen, die nach der oben eingeführten Trennung nach Phasen an erster Stelle der Verarbeitungsphase steht. Dies stellt eine Ausnahme für Verarbeitungskomponenten dar, da diese laut Konzept in ihrer Position austauschbar sein müssen.

4.3.3 Informationsfluss

Neben dem in Abb. 13 gezeigten Hauptfluss sind auch Nebenflüsse von Daten möglich. Nebenflüsse können durch Komponenten in alle Phasen gefordert oder erzeugt werden und enthalten Zusatzinformationen zu oder über den Hauptfluss (z.B. Metadaten). Die Datenart der Nebenflüsse kann von der Datenart des Hauptflusses abweichen.

Nebenflüsse sind immer optional, das heißt, ein erfolgreicher Abschluss des Publikationsprozesses darf nicht von ihnen abhängig sein.

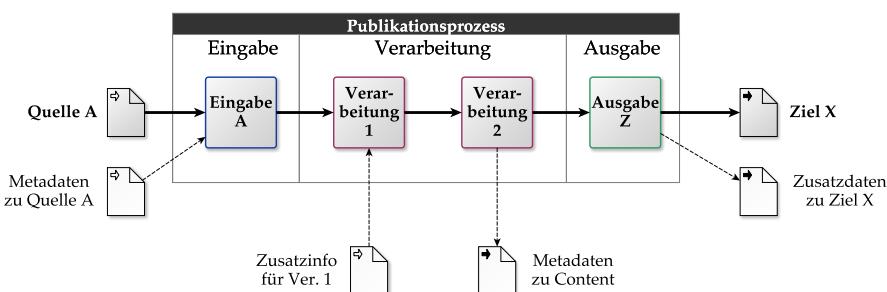


Abb. 14: Publikationsprozess: Nebenflüsse

4.3.4 Teilprozesse

Durch die sequentielle Kapselung der Publikationphasen ist es möglich, nur bestimmte Publikationsphasen auszuführen. Wird z.B. nur die Eingabe-Phase ausgeführt, ist das Ergebnis die Überführung der Quelldateien in die interne Informationsstruktur. Liegt der Content aber z.B. schon in der internen Informationsstruktur vor, können die Verarbeitungsphase, die Ausgabephase, oder beide Phasen kombiniert angewendet werden.

Dieses Modell erlaubt neben einer großen Flexibilität auch weiterführende Funktionen, wie das Umstrukturieren des Contents während des Publikationsprozesses (siehe Abb. 15). Dort kann die erzeugte

Strukturdefinition verändert werden, bevor der zweite Teilprozess angesstoßen wird.

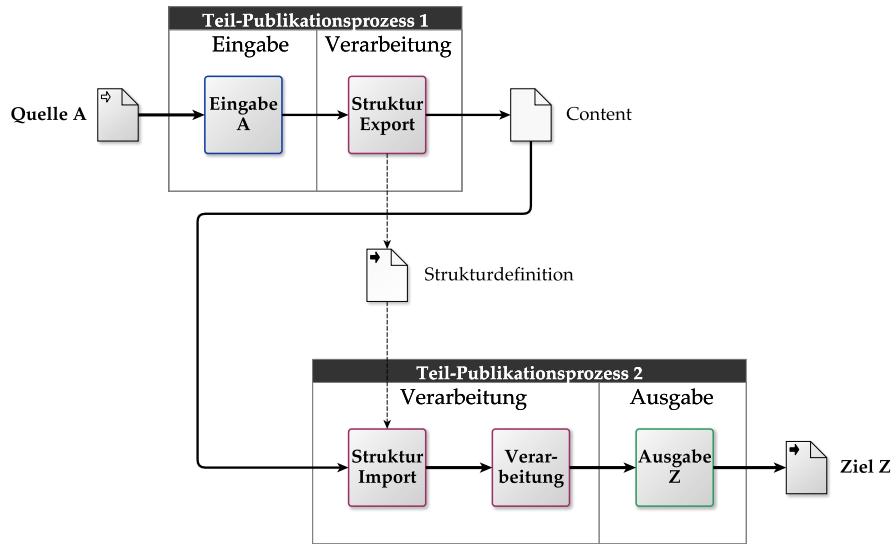


Abb. 15: Publikationsprozess: Teilprozesse bei Strukturumformung

Zunächst wird auf die Quelldaten die Eingabephase und eine verkürzte Verarbeitungsphase mit entsprechender Komponente angewendet. Es liegt nun eine Strukturdefinition vor, die verändert werden kann. Mit der veränderten Strukturdefinition kann nun eine Teilpublikation auf den schon vorliegenden Content angewendet werden, die aus dem Rest der Verarbeitungsphase (inklusive spezieller Komponente zur Verarbeitung der Strukturdefinition) und der Ausgabephase besteht.

4.4 KOMPONENTEN

4.4.1 *Definition*

Komponenten sind die Bausteine, aus denen sowohl das *Publikationssystem* als auch die *Publikationsprozesse* aufgebaut sind. Sie kapseln einzelne Funktionalitäten für eine größtmögliche Wiederverwendung und Flexibilität.

Komponenten definieren sich über ihre Schnittstelle, das heißt die Art der Ein- und Ausgangsdaten im Hauptfluss. Diese resultieren wiederum aus der internen Verarbeitung der Daten, die in einem Prozess²³ schrittweise beschrieben wird.

²³ In Abgrenzung zu dem in einer *Publikationsdefinition* beschriebenen *Publikationsprozess* wird der interne Prozess einer Komponenten nur als *Prozess* bezeichnet.

4.4.2 Aufbau

Eine Komponente besteht aus genau einem *Manifest*²⁴ und optionalen *Zusatzinhalten*.

MANIFEST

Enthält *Metadaten* (eindeutiger Bezeichner, Name, Version, Autor und Beschreibung der Komponente), eine *Schnittstellen-Definition* (die Art der Ein- und Ausgangsdaten in Haupt- und Nebenfluss) sowie die interne *Prozessbeschreibung* (sequentieller Ablauf von Operationen innerhalb der Komponente) in einer vom Publikationssystem vorgegebenen Syntax.

ZUSATZINHALTE

Als Zusatzinhalte werden alle Dateien, Programme und Objekte bezeichnet, die für die Ausführung des in der Komponente beschrieben Prozesses notwendig sind. Sie werden innerhalb des Manifests referenziert.

Das Komponentenpaket (Manifest und Zusatzinhalte) wird in einem Ordner zusammengefasst, dessen Name dem eindeutigen Bezeichner der Komponente entspricht.

4.4.3 Klassifizierung

Für die Umsetzung des Publikationssystems und zur Ausführung von Publikationsprozessen werden fünf verschiedene Komponententypen benötigt: *Basis-, System-, Eingabe-, Verarbeitungs- und Ausgabekomponenten*. Diese fünf Typen können in zwei *Komponentenklassen* eingeteilt werden: *Publikationskomponenten* und *Funktionskomponenten*.

PUBLIKATIONSKOMPONENTEN

Der Klasse *Publikationskomponenten* gehören alle Komponententypen an, die als Komponenten höherer Ordnung in einer der Phasen des *Publikationsprozesses* eingesetzt werden können. Zu ihnen gehören: *Eingabe-, Verarbeitungs- und Ausgabekomponenten*. Diese Komponententypen qualifizieren sich über ihre Schnittstellendefinition (siehe unten).

FUNKTIONSKOMPONENTEN

Der Klasse *Funktionskomponenten* gehören alle Komponententypen an, die Funktionen der darunterliegenden Systeme abbilden. Zu ihnen gehören *Basis- und Systemkomponenten*. Diese Komponententypen qualifizieren sich über das System, das sie abbilden (siehe unten).

²⁴ In der Informationstechnik enthalten *Manifest-Dateien* wichtige Metadaten und zentrale Informationen zur Registrierung an einem übergeordneten System.

Komponententypen erben alle Eigenschaften der Klasse, der sie angehören. Die jeweilige Ausprägung wird nach folgenden Gesichtspunkten bestimmt:

PUBLIKATIONSKOMPONENTEN

EINGABEKOMPONENTEN

Eingabekomponenten nehmen eine beliebige Datenquelle entgegen, wandeln diese in die intern verwendete Informationsstruktur um und geben diese weiter.

Beispiel ist etwa eine Komponente, die Dateien, die aus einem Textverarbeitungsprogramm stammen, in die interne Informationsstruktur überführen kann.

VERARBEITUNGSKOMPONENTEN

Verarbeitungskomponenten nehmen die intern verwendete Informationsstruktur entgegen, verarbeiten diese und geben die verarbeiteten Daten wieder in der intern verwendeten Informationsstruktur weiter.

Beispiel ist etwa eine Komponente, die die Struktur des Contents entsprechend bestimmter Vorgaben ändern oder umsortieren kann.

AUSGABEKOMPONENTEN

Ausgabekomponenten nehmen die intern verwendete Informationsstruktur entgegen, wandeln diese in ein beliebiges Datenformat oder Publikationsziel um und geben dieses aus.

Beispiel ist etwa eine Komponente, die den Content als printorientiertes Dokument ausgibt.

FUNKTIONSKOMPONENTEN

BASISKOMPONENTEN

Basiskomponenten bilden eine Funktion des *Host-Systems* ab. Die abgebildeten Funktionen stehen allen anderen Komponententypen zur Verfügung. Basiskomponenten können immer nur als untergeordnet, also als Bestandteil von Komponenten höherer Ordnung eingesetzt werden.

Durch die Unterordnung von Basiskomponenten ist es möglich, von Komponenten höherer Ordnung Parameter zu empfangen, die die auszuführende Funktion näher definieren oder Dateien referenzieren.

Beispiel ist etwa eine Komponente, die eine Transformationsfunktion zur Verfügung stellt. Parameter ist in diesem Fall die Transformationsbeschreibung.

SYSTEMKOMPONENTEN

Systemkomponenten bilden eine Funktion des *Publikationssystems* ab. Die abgebildeten Funktionen stehen keinen an-

deren Komponententypen zur Verfügung. Systemkomponenten bestimmen die Charakteristik des Publikationssystems und geben dessen Verhalten vor.

Sie werden entweder direkt (durch Nutzer oder Web-API) oder durch dynamisch erzeugte Ablaufpläne (sog. assemblierte Publikationsdefinitionen) ausgeführt.

Beispiel ist etwa eine Komponente, die eine *Publikationsdefinition* in einen ausführbaren Ablaufplan übersetzt (assembliert).

4.4.4 Identifizierung

Jede Komponente wird durch einen *eindeutigen Bezeichner* identifiziert. Mit Hilfe dieses Bezeichners kann die Komponente in anderen Komponenten oder in Publikationsdefinitionen referenziert werden.

Der eindeutige Bezeichner entspricht einem *Klassenpfad*²⁵, der beginnend bei der Komponentenklasse die Art der Komponente definiert. Dabei können beliebige Untergruppen gebildet werden. Der Name der Komponente muss dabei innerhalb seiner Klasse oder Untergruppe eindeutig sein.

4.4.5 Schnittstellen

Die Schnittstellen einer Komponente definieren ihre Klasse und Funktion innerhalb des Publikationssystems. Komponenten besitzen immer eine Eingabe- und eine Ausgabeschnittstelle für den Hauptfluss. Optional sind Ein- und Ausgänge eines Nebenflusses (siehe auch Abschnitt 4.3.3). Alle Schnittstellen müssen im Manifest der Komponente beschrieben sein.

Da die Aufgabe von Komponenten innerhalb des Publikationssystems in den meisten Fällen eine Umwandlung von Dateien in eine andere Form ist, werden die Schnittstellen über Daten- bzw. Medientypen beschrieben. Dadurch ist es möglich, den Informationsfluss innerhalb einer Komponente zu validieren (außer es werden native Schritte verwendet).

Zur Definition der Schnittstelle muss für eine Umsetzung ein geeigneter Standard zur Beschreibung der Medientypen gefunden werden.

Dieses Vorgehen wurde gewählt, um eine größtmögliche Flexibilität in Aktualisierung und Austausch von Komponenten zu erreichen. So lange die Schnittstellendefinition gleich bleibt, können die internen Vorgänge komplett verändert werden.

²⁵ Ein Klassenpfad gibt in der Informatik an, wo eine Laufzeitumgebung nach benötigten Komponenten suchen soll.

4.4.6 Lebenszyklus

Eine Komponente, bzw. ihr *Manifest*, kann in drei Formen vorliegen: Als Vorlage (*Grundform*), als Variante, die in ausführbaren Code übersetzt wurde (*Kompilierte Form*) oder als ausgeführte *Instanz*.

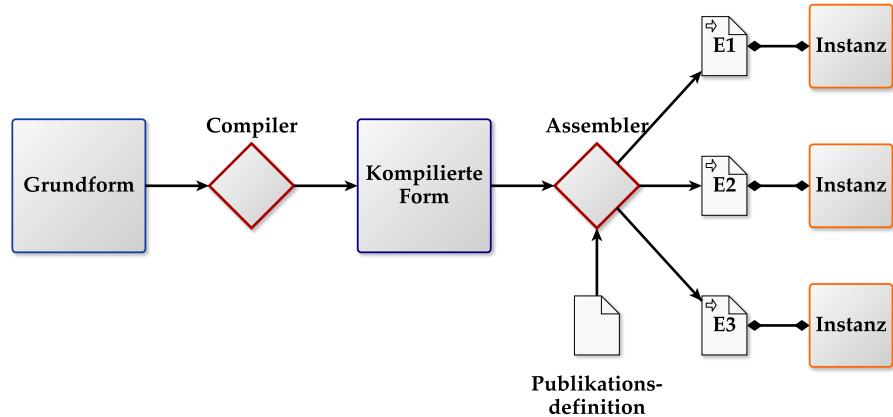


Abb. 16: Komponenten: Lebenszyklus

Die Übersetzung der Grundform in die ausführbare Variante sowie die Ausführung der Instanz mit konkreten Eingaben wird vom *Komplierer* bzw. dem *Assembler* übernommen.

GRUNDFORM

Grundform ist das unveränderte Manifest einer Komponente in einer vom Publikationssystem vorgegebenen Syntax. Sie enthält unaufgelöste Referenzen zu anderen Komponenten und variable Kontextinformationen.

KOMPILIERTER FORM

Die Grundform wurde in eine vom Host-System ausführbare Sprache übersetzt (*kompiliert*). Referenzen zu anderen Komponenten wurden vollständig²⁶ aufgelöst und Kontextinformationen festgelegt. Die einzige verbleibende nicht aufgelöste Parameter sind die konkrete Eingabe-Datei und der übergeordnete Publikationsprozess. Die kompilierte Form tritt pro Publikationsprozess nur einmal auf und kann in beliebig viele Instanzen überführt werden.

INSTANZ

Die kompilierte Form wird innerhalb eines Publikationsprozesses mit einer bestimmten Eingabe-Datei ausgeführt (*instanziert*). Für jede Eingabe-Datei einer Komponente wird eine eindeutige Instanz generiert. Diese existiert nur während der Ausführung des Publikationsprozesses.

²⁶ Vollständig bedeutet in diesem Fall eine rekursive Auflösung in beliebige Tiefe.

Analog zur Funktionsweise eines Compilers, wird der Vorgang des Übersetzens in dieser Arbeit *kompilieren* genannt, die zuständige Systemkomponente wird als *Compiler* bezeichnet. In der Regel wird die Kompilierung unmittelbar vor ihrer Ausführung durchgeführt. Die kompilierte Variante wird nach erfolgreicher Ausführung wieder gelöscht.

4.4.7 Kontextinformationen

Komplizierte Komponenten werden immer in einem bestimmten *Kontext* ausgeführt. In der Regel ist das die Publikation, in der eine Komponente eingesetzt wird. Der Kontext bestimmt eine Menge an *Parametern* in verschiedenen *Geltungsbereichen*.

Die konkreten Kontextinformationen können je nach Implementierung voneinander abweichen, in jedem Fall notwendig sind aber (hierarchisch absteigend):

PUBLIKATIONSPROZESS-KONTEXT

UUID der Publikationsdefinition

KOMPONENTEN-KONTEXT

Eindeutiger Bezeichner (*Identifier*) der Komponente und Eingabedaten der Komponente (Quelldaten)

PROZESSSCHRITT-KONTEXT

Eingabedaten des Schritts (Quelldaten oder Ausgabedaten des Vorgängerschritts) und Ausgabedaten des Schritts (Ergebnisdaten oder Eingabedaten des Nachfolgerschritts)

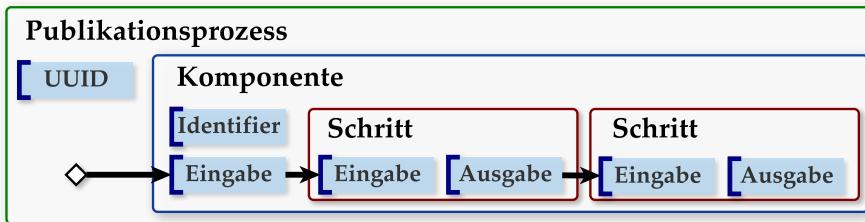


Abb. 17: Publikationsprozess: Kontextinformationen

Diese Informationen sind notwendig, um die einzelnen Bestandteile in einem Publikationsprozess zueinander in Beziehung setzen zu können. Die Kontextinformationen kaskadieren in der Hierarchie nach unten.

4.4.8 Prozessbeschreibung

Der Prozess, den eine Komponente beschreibt, ist aus aufeinanderfolgenden Schritten aufgebaut, deren Verarbeitung sequentiell erfolgt.

Eine Komponente beinhaltet genau eine Prozessbeschreibung, die aus mindestens einem Schritt besteht.

4.4.8.1 Informationsfluss

Innerhalb einer Komponente fließen Informationen streng linear. Alle Schritte nehmen Daten entgegen und geben Daten weiter. Eingabe des ersten Schrittes ist die Quelldatei, Ausgabe des letzten Schrittes die Ergebnisdatei. Im Regelfall ist die Eingabe eines Schrittes die Ausgabe des Vorgängerschrittes.

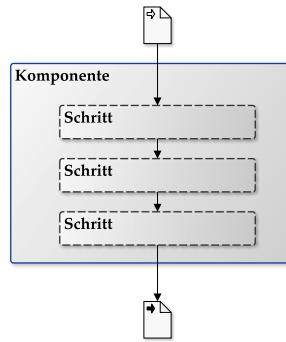


Abb. 18: Publikationsprozess: Informationsfluss bei Schritten

Um einen validen Informationsfluss zu gewährleisten,¹ müssen deshalb Komponenten im Hauptfluss auch immer den Content, den sie verarbeiten, ausgeben. Schritte, die Dateien erzeugen, die für weitere Schritte innerhalb der Komponente relevant sind, müssen diese im Nebenfluss ausgeben. Schritte, die Prozesse auslösen, die keine direkte Bearbeitung des Contents zur Folge haben (z.B. kopieren), müssen den Content unverändert *durchschleusen*, um einen Bruch im Informationsfluss zu vermeiden.

Dieser Umstand erscheint zunächst an manchen Stellen ungewohnt in der Umsetzung, ist aber wesentlicher Teil des freien, komponentenbasierten Konzepts.

4.4.8.2 Arten von Prozessschritten

Schritte können in zwei verschiedenen Ausprägungen auftreten:

KOMPONENTEN

Der Schritt verweist auf den Prozess einer anderen Komponente, der an Stelle des Schrittes eingebunden werden soll (*Dependency Injection*, siehe Abschnitt 2.4.6.4). Diese Verschachtelung kann beliebig tief sein.

NATIVES SKRIPT

Der Schritt enthält ein Skript in einer vom Host-System ausführbaren Sprache sowie die Information, um welche Sprache

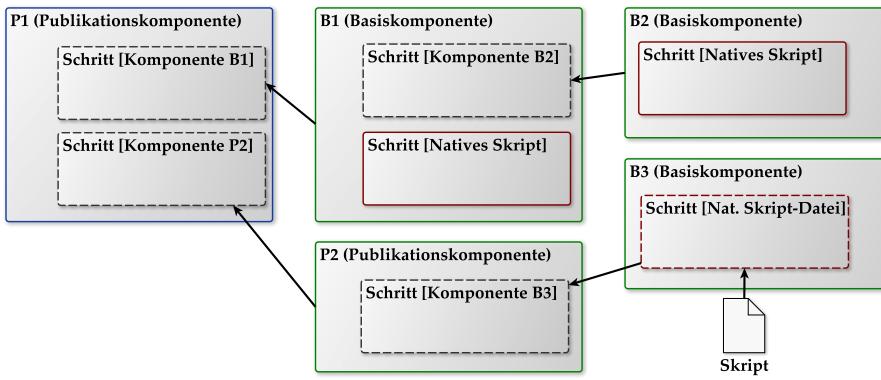


Abb. 19: Publikationsprozess: Schritte und Abhängigkeiten

es sich handelt. Schritte, die native Skripte enthalten, sind nur in Basiskomponenten zulässig.²⁷

Bei der Kompilierung einer Komponente werden alle Abhängigkeiten dynamisch aufgelöst. Das bedeutet, dass jede kompilierte Komponente nur noch aus den Inhalten von *nativen* Schritten besteht. Dadurch entsteht ein vollständig vom Host-System ausführbares Programm - die Komponentensyntax wurde in ausführbaren Code kompiliert.

4.4.8.3 Anmerkung

Dieses Vorgehen bedeutet, dass alle Prozesse, die in *Publikationskomponenten* beschrieben werden ausschließlich aus Referenzen zu Komponenten bestehen. Da Komponenten immer ein Konzept oder eine Funktion abbilden, sind Publikationskomponenten somit *implementierungsunabhängig* (unabhängig vom Host-System). Ändert sich das Host-System müssen zwar Funktionskomponenten angepasst werden, jedoch können alle Publikationskomponenten unverändert übernommen werden.

Native Skripte sind ausschließlich in Basiskomponenten zugelassen. Dadurch wird nicht nur eine Modularisierung erzwungen, sondern auch eine Validierung von Komponenten ermöglicht bzw. vereinfacht. Das Skript wird durch die Basiskomponente mit eindeutigen Schnittstellen versehen. Wird das Skript als Schritt wiederverwendet, kann der Gesamtprozess der übergeordneten Komponente validiert werden. Dies wäre nicht möglich, wenn es sich um einen nativen Schritt handeln würde, der direkt eingesetzt wird (da das System aus dem Skript nicht dessen Schnittstellen erkennen kann).

²⁷ Diese Regel erzwingt eine Ausgliederung von Funktionen, die durch Basiskomponenten auch anderen Komponententypen zur Verfügung gestellt werden können.

4.5 PUBLIKATIONSDEFINITIONEN

4.5.1 *Definition*

Eine *Publikationsdefinition* beschreibt die Verknüpfung von Eingangsdaten zu Komponenten sowie die Einteilung der Komponenten in die drei Phasen des Publikationsprozesses.

4.5.2 *Aufbau*

Eine Publikationsdefinition ist eine *Manifest-Datei*.

MANIFEST

Enthält *Metadaten* (universal eindeutige ID, Name und Beschreibung) und eine Einteilung in die drei *Phasen* des Publikationsprozesses (Eingabe, Verarbeitung, Ausgabe).

Jeder der Phasen können Referenzen zu Komponenten zugeordnet werden. Jeder Komponente können Referenzen zu Eingabedateien (Haupt- und Nebenfluss) zugeordnet werden.

4.5.3 *Identifizierung*

Da eine Publikationsdefinition potentiell auf verschiedenen Publikationssystemen ausführbar sein soll und damit eindeutig bezeichnet werden muss, wird sie über einen sogenannten Universally Unique Identifier (UUID) identifiziert (vgl. IETF RFC 4122 2005).

4.5.4 *Lebenszyklus*

Eine Publikationsdefinition kann in zwei Formen vorliegen: Als Vorlage (*Grundform*) oder als Variante, die als Ablaufplan dient und in ausführbaren Code übersetzt wurde (*Assemblierte Form*).

Im Gegensatz zur Kompilierung bei Komponenten werden allerdings keine konkreten Operationen abgebildet, sondern lediglich Aufrufe in einem bestimmten Ablauf festgelegt. Dieser Vorgang kann auch als dynamische Dienste- oder Service-Komposition betrachtet werden (vgl. ERL 2008:411).

GRUNDFORM

Grundform ist das unveränderte Manifest einer Publikationsdefinition in einer vom Publikationssystem vorgegebenen Syntax. Die Grundform enthält unaufgelöste Referenzen zu Komponenten und Verweise auf Quelldateien.

ASSEMBLIERTE FORM

Die Grundform wurde vollständig in einen vom Host-System ausführbaren Ablaufplan zusammengesetzt (engl.: *assemble*).

Die Phasen wurden entsprechend ihrer inneren Logik (siehe Abschnitt 4.3.2) als Ablauf umgesetzt (sequentielle oder parallele Verarbeitung). Der Komponenten-Lebenszyklus wird für jede Komponente innerhalb einer Phase abgebildet. Das bedeutet, dass für jede Komponente zunächst die Kompilierung angestoßen wird und die kompilierte Variante anschließend für jede mit der Komponente assoziierten Eingabe-Datei instantiiert wird. Die *assemblierte Form* tritt pro Publikationsprozess nur einmal auf und wird in der Regel nach der Publikation verworfen.

4.6 REPOSITORY**4.6.1 Definition**

Als *Repository* (engl.: Lager, Depot) wird das Verzeichnis aller in einem Publikationssystem verfügbaren Komponenten bezeichnet. Es dient als Repräsentation des Funktionsumfangs von Host- und Publikationssystem und damit als Basis für das Erstellen von Publikationsdefinitionen.

4.6.2 Aufbau

Ein Repository ist eine *Verzeichnis-Datei*.

VERZEICHNIS

Enthält die *Metadaten* der Komponente (Komponententyp, Name, Version und Beschreibung), alle Abhängigkeiten zu anderen Komponenten und die zulässigen Informationsflüsse (Eingabe, Ausgabe, jeweils Haupt und ggf. Nebenfluss).

Der Dateiname der aktuellen Version des Verzeichnisses entspricht **repository**.

4.6.3 Erstellung

Das Repository wird aus vorhanden Informationen von einer Systemkomponente generiert. Metadaten und Ein-/Ausgänge können direkt aus der Komponente übernommen werden und die Abhängigkeiten können über eine rekursive Prüfung der referenzierten Komponenten gewonnen werden.

Bei der Erstellung muss darauf geachtet werden, dass eine Vorvalidierung der verzeichneten Komponenten vorgenommen wird. Dabei müssen alle der folgenden Bedingungen erfüllt sein:

- Ein wohlgeformtes Manifest der Komponente ist verfügbar.
- Alle referenzierten Abhängigkeiten müssen vollständig aufgelöst werden können.
- Der eindeutige Bezeichner in den Metadaten und der Dateiname des Manifests müssen übereinstimmen.

Wird eine dieser Bedingungen verletzt, darf die Komponente nicht in das Repository aufgenommen werden.

4.6.4 Funktion

Im Plattformbetrieb (siehe Kapitel 4.1) ist das Repository ein wichtiges Werkzeug, um die aktuell gültige Komponentenkonfiguration zwischen Server und Client auszutauschen. Es dient aber auch als Grundlage für das manuelle Erstellen oder Anpassen von Komponenten.

4.7 INTERNE INFORMATIONSSTRUKTUR

4.7.1 Bedarf

Durch den Aufbau der Publikationsplattform im *Hub-and-Spoke*-Muster wird eine interne Informationsstruktur benötigt, in die zunächst alle Quellen transformiert werden. Auf Basis des internen Content-Modells sollen in der Verarbeitungsphase der Publikation weitere Bearbeitungen durchgeführt werden können. Der anschließende Export in die Zielmedien hat entsprechend immer die intern genutzte Informationsstruktur als Basis.

Durch eine gemeinsame, von allen Verarbeitungskomponenten genutzte Informationsstruktur entstehen folgende Vorteile:

EINHEITLICHKEIT

Die Publikationsplattform hat sich zum Ziel gesetzt, auch heterogene Quelldaten verarbeiten zu können. Dies ist erst möglich, wenn alle Quelldaten vereinheitlicht wurden und damit in eine interne Informationsstruktur übertragen wurden.

WIEDERVERWENDBARKEIT

In Publikationsstrecken gibt es einige Operationen, die fast immer ausgeführt werden müssen, unabhängig von Quell- und Ergebnis-Daten. Dazu gehört z.B. das Auflösen von Referenzen

zu Mediendateien. Durch die Verwendung einer internen Informationsstruktur kann dieser Prozessschritt in eine Verarbeitungskomponente ausgelagert werden und für beliebige Kombinationen von Eingabe- und Ausgabekomponenten wieder verwendet werden.

KOPPLUNG / PIPING

Da die Datenstruktur bei Verarbeitungskomponenten für Eingabe und Ausgabe identisch ist, können diese in beliebiger Reihenfolge gekoppelt oder getauscht werden. Sie entsprechen dadurch den *Filtern* eines Pipes-and-Filters-Modells (siehe dazu Abschnitt 2.4.6.3).

4.7.2 Anforderungen

STANDARD

Entsprechend des Hauptziels, die Publikationsplattform auf standardisierten Technologien aufzubauen (siehe Abschnitt 3.2.1.3), soll auch die Informationsstruktur ein bekannter Standard sein oder auf einem Standard basieren. Dieser Standard soll von einer unabhängigen Organisation gepflegt und spezifiziert sein. Dadurch soll die Schaffung eines neuen proprietären Formates vermieden, die Erweiterung der Plattform erleichtert und das Nutzen von schon bestehenden Werkzeugen ermöglicht werden. Um Offenheit und Kompatibilität zu gewährleisten soll es sich um ein textbasiertes Format handeln.

NEUTRAL

Die Informationsstruktur des Formats muss prinzipiell jede Art von Informationsart, die aus den Quellen stammt, abbilden können. Dazu darf sie selbst nicht zu dogmatisch sein, sondern muss einen neutralen Rahmen für die verschiedensten Ausprägungen von Content bieten (Text, Bild, Video, Interaktivität, etc.).

Moderne XML-Informationsarchitekturen sind zwar stark semantisch orientiert, die Integration von nicht-semantischem Sekundärcontent in diese Modelle ist jedoch nur ungenügend gelöst.²⁸ Für eine Publikationsplattform ist ein darstellungsorientiertes Modell oder eine Mischform besser geeignet.

MODERN

Das Modell muss soweit erweiterbar sein, dass eine Anpassung an neue Anforderungen möglich ist. So muss es möglich sein, neue Medienformen einzubinden zu können oder neue Auszeichnungsformen zu definieren. Der zugrundeliegende Standard soll mit Hinblick auf Zukunftssicherheit gewählt werden.

²⁸ Was unter Anderem auf der spezifischen Domänenausrichtung der Informationsmodelle beruht.

QUERSCHNITT

Die interne Informationsstruktur dient als *Lingua franca* zwischen verschiedenen Sprachen und Formaten. Um den Transformationsaufwand möglichst gering zu halten, soll sie in etwa einem Querschnitt der Formate entsprechen, die verarbeitet werden. Hier ist mit Hinblick auf die Digitalisierung besonders für die Übertragung in der Exportphase ein modernes Modell zu wählen.

SPEICHERBEDARF

Die Struktur soll möglichst wenig *Overhead* in der Speicherplatzbelegung erzeugen. Das bedeutet, dass die Basisstrukturen zur Erzeugung eines validen Dokuments so klein wie möglich sein sollen und alle nicht-inhaltstragenden Elemente so kompakt wie möglich abgebildet werden können.

5

TECHNISCHE UMSETZUNG

In diesem Kapitel wird das in Kapitel 4 definierte Konzept in einer möglichen Implementierung umgesetzt und diese spezifiziert. Beispielkomponenten werden auf Basis der technischen Umsetzung entworfen.

5.1 ARBEITSNAME

Der Arbeitsname der Umsetzung lautet: *infoflow*.

infoflow setzt sich aus den Bestandteilen *info* (für Informationen) und *flow* (für Fluss der Informationen) zusammen. Der Name soll verdeutlichen, dass der Hauptgesichtspunkt der Umsetzung der funktionale *Fluss von Informationen* durch die Phasen eines Publikationsprozesses (in der Umsetzung *flows* genannt) ist. Dabei werden gezielt Ansätze des *flow-based programming* angewendet, einer Ausprägung der komponentenbasierten Softwareentwicklung (siehe Abschnitt 2.4.7).

5.2 INFORMATIONSSTRUKTUR

5.2.1 Anforderungsabbildung

Auf Basis der in Abschnitt 4.7.2 genannten Anforderungen wurde als Informationsstruktur für die interne Repräsentation des Contents eine Variante des Web-Standards HTML5 gewählt. Die Abdeckung der Anforderungen ist in Tabelle 4 auf Seite 68 zusammengefasst.

Mit den in HTML5 eingeführten Verbesserungen der Markup-Sprache HTML (vgl. SCHOBER 2012:36ff) steht ein darstellungsorientiertes Modell zur Auszeichnung von Content zur Verfügung, das über standardisierte Erweiterungen um semantische Funktionen ergänzt werden kann und somit sowohl zukunftssicher als auch flexibel ist.

5.2.2 Aufbau

Um allen Anforderungen gerecht zu werden, wird eine restriktive Variante von HTML5 verwendet, die bestimmte Vorgaben zur Strukturbildung des Dokuments macht. Die Informationsstruktur bleibt jedoch weiterhin valides HTML5, welches in der XML-Syntax abgebildet wird

ANFORDERUNG	ABBILDUNG IN HTML5
Standard	HTML5 wird vom W3C unabhängig spezifiziert und standardisiert. Die erste Version der HTML-Spezifikation wurde bereits 1992 veröffentlicht und seitdem kontinuierlich verbessert bis hin zur heutigen Version 5
Neutral	HTML5 ist eine darstellungsorientierte Markup-Sprache, die durch semantische Elemente erweitert werden kann. Durch die Fülle an verschiedenen Content-Arten im World Wide Web bietet HTML5 eine Vielzahl an Funktionen und Schnittstellen zur Einbindung von multimedialen und interaktiven Inhalten.
Modern	HTML5 wurde im Oktober 2014 als Standard verabschiedet und wird seitdem kontinuierlich als <i>lebender Standard</i> um neue Funktionen ergänzt.
Querschnitt	Ein Großteil der neu entstehenden Ausgabeziele sind webbasiert oder können webbasiert umgesetzt werden. Durch den Einsatz von HTML5 als Basis entstehen hier nur geringe <i>Reibungsverluste</i> bei einer Umwandlung.
Speicherbedarf	HTML5 kennt Möglichkeiten zur impliziten Darstellung von Strukturen (siehe Abschnitt 2.2.5.3) und ist im Gegensatz zu semantischen XML-Modellen als eher <i>schlank</i> einzuordnen.

Tabelle 4: Informationsstruktur: Anforderungsabbildung

(siehe Abschnitt 2.2.5.1). Um Content und Teile davon mit Metainformationen anzureichern, wird auf semantische Erweiterungen (insbesondere `data-Attribute`) zurückgegriffen (siehe Abschnitt 2.2.5.4).

Für den weiteren Verlauf der Arbeit wird die verwendete restriktive Variante zur besseren Abgrenzung als `h5` bezeichnet.

5.2.2.1 Makrostruktur

Die Makrostruktur eines `h5`-Dokuments entspricht weitgehend den Mindestanforderungen an ein valides HTML5-Dokument (siehe Abschnitt 2.2.5.3).

Neben dem HTML-Doctype und einem für die XML-Validität notwendigen Wurzelement `<html>` mit Namespace-Deklaration muss mindestens ein `<title>`-Element im impliziten Kopf des Dokument vorhanden sein (vgl. W3C 2015a:4). Bei `h5`-Content enthält dieses Element den Titel der gesamten Publikation.

Im impliziten Körper des Dokuments muss mindestens ein strukturbildendes Element (`<section>`) vorkommen. Jeder dieser Strukturblöcke muss mit einem Titel im Element `<h1>` ausgezeichnet werden. Dies entspricht dem neu eingeführten Outline-Algorithmus von HTML5 (vgl. MOZILLA DEVELOPER NETWORK 2015c).

```

1 <!-- H5 Makrostruktur -->
2 <!DOCTYPE html>
3 <html xmlns="http://www.w3.org/1999/xhtml">
4   <title>Titel der Publikation</title>
5   <section>
6     <h1>Titel des Strukturblocks</h1>
7     ...
8   </section>
9 </html>

```

Code 7: Makrostruktur H5

Strukturblöcke können beliebig geschachtelt werden und entsprechen den Hierarchien aus anderen Informationsmodellen. Innerhalb der Strukturblöcke können die nicht-strukturbildenden Block-Elemente `<p>` für Absätze und `<div>` für Gruppierungen (z.B. Warnhinweise) verwendet werden. Andere nicht-strukturbildenden Block-Elemente wie `<nav>` oder `<article>` sind nicht zulässig. Alle Inline-Elemente aus HTML5 sowie Tabellen, Listen und eingebettete Medien (``, `<video>`, etc.) sind erlaubt.

5.2.2.2 Metadaten

Beim Einsatz von *Metadaten* am Content wird je nach Art unterschieden, in welcher Form sie abgebildet werden:

VERARBEITUNGS-METADATEN

Metadaten, die zur (internen) Verarbeitung des Contents durch die Publikationsplattform benötigt werden, sind durch data-Attribute abgebildet. Entsprechend der empfohlenen Verwendung enthalten diese lediglich Daten, deren Auswertung durch externe Parteien nicht vorgesehen ist (siehe Abschnitt 2.2.5.4).

Beispiele sind etwa die in der Umsetzung verwendeten Attribute `data-hash` (Prüfsumme des Dokuments) an `<html>` sowie `data-origin` (Komponente, die den Inhalt erzeugt hat) und `data-source` (Quelldatei des Inhalts) an `<section>`.

DOKUMENT-METADATEN

Metadaten, die eine dokumentweite Gültigkeit haben, sollen wenn möglich über `<meta>`-Elemente abgebildet werden. Wenn das nicht möglich ist, sind RDFa-Attribute am `<html>`-Element möglich. Die Auswertung dieser Daten durch externe Parteien ist vorgesehen. Beispiel sind etwa Schlagworte zur Kategorisierung eines Dokuments in einem `meta`-Tag vom Typ *keywords*.

STRUKTURBLOCK-METADATEN

Metadaten, die jeweils nur für einen bestimmten Strukturblock gültig sind, sollen wenn möglich über Microdata-Attribute am <section>-Element abgebildet werden. Wenn das nicht möglich ist, sind RDFa-Attribute am jeweiligen Element möglich. Die Auswertung dieser Daten durch externe Parteien ist vorgesehen. Beispiel ist die DITA-Semantik in Code 4.

ANDERE METADATEN

Metadaten, die auf tieferer Ebene, z.B. an Inline-Elementen oder nicht-strukturbildenden Block-Elementen, gültig sind, werden über RDFa-Attribute am jeweiligen Element abgebildet. Die Auswertung dieser Daten durch externe Parteien ist vorgesehen.

Ein Beispiel für die Erhaltung von Semantik durch den Einsatz von Metadaten auf der Ebene von Strukturböcken und anderen Elementen findet sich unter Abschnitt 2.2.5.4 (bzw. Code 3 und Code 4).

5.2.2.3 Validierung

`h5`-Dokumente können mit Standard-Validierungstools für HTML5 validiert werden. Diese prüfen Syntax und Wohlgeformtheit des Dokuments nach der offiziellen Spezifikation (vgl. SIVONEN/SMITH 2015). Eine Validierung der speziellen Restriktionen des `h5`-Modells kann durch spezielle (nachgeschaltete) Stylesheets erfolgen.

Der intern verwendete Medien-Typ (siehe Kapitel 5.4) für Content lautet: `application/vnd.infoflow.content+xml`

Die Validierung der `h5`-Dokumente ist in der Verarbeitungskomponente `mid.validate` umgesetzt. Diese kann an beliebiger Stelle in der Verarbeitungsphase der Publikation eingesetzt werden.

5.3 UMGANG MIT CONTENT

5.3.1 Vorüberlegungen

Zentraler Bestandteil einer Publikationsplattform ist der Umgang mit Content, also dessen Verarbeitung, Umwandlung und Interpretation. Mit der in Kapitel 5.2 definierten Informationsstruktur auf Basis von HTML5 steht ein flexibles Modell zur internen Repräsentation des Contents zur Verfügung.

Zur effektiven Nutzung dieser Struktur gibt die Plattform einige Methoden vor, die innerhalb der jeweiligen Komponenten umgesetzt werden müssen, aber auch ausgenutzt werden können. Prinzipiell ist dabei zu beachten, dass Komponenten den Umgang mit einer bestimmten Content-Art, im Rahmen der vorgegebenen Richtlinien, immer selbst bestimmen.

5.3.2 Überführung

Unter *Überführung des Contents* wird in dieser Arbeit die Umwandlung und Anpassung des Contents bezeichnet. Eine Überführung ist sowohl in Eingabe- als auch Ausgabephase des Publikationsprozesses vorgesehen. Dabei ist neben der simplen Umwandlung des Datenformats vor allem auch die Struktur von großer Bedeutung (diese wird in Abschnitt 5.3.3 gesondert behandelt). Im folgenden wird speziell die Überführung von Content bei der Eingabephase betrachtet.

Das Ergebnis einer Eingabekomponente muss ein Dokument(-Teil) sein, das der in Kapitel 5.2 beschriebenen Struktur entspricht und mit der entsprechenden Klasse versehen ist (siehe Tabelle 5). Um nachvollziehen zu können, woher ein Content-Fragment nach der Zusammenführung stammt und eine Filterung nach Quellen zu ermöglichen, muss die Herkunft mit Attributen gekennzeichnet werden.

Die folgenden Attribute am Root-Element (`html`) müssen gesetzt sein:

ATTRIBUT	WERT
<code>class</code>	<code>content</code>
<code>data-origin</code>	Identifier d. erzeugenden Publikationskomponente
<code>data-source</code>	Absoluter Pfad zur Quelldatei (z.B. in <code>sources</code>)

Tabelle 5: Informationsstruktur: Attribute an Root-Element

Das obligatorische `title`-Element im Dokument wird, wenn möglich, aus den Metadaten der Quelldatei bestimmt. In Fällen, in denen das nicht möglich ist, wird der Dateiname ohne Erweiterung verwendet. Bei der Zusammenführung werden die `title`-Angaben der Content-Teile zugunsten des Titels für die Gesamt-Publikation verworfen.

Es wird empfohlen, die Komponente `base.xhtml.structure` zu verwenden, die eine korrekte Struktur sicherstellt, das `title`-Element erzeugt und die geforderten Attribute an `html` automatisch setzt.

Das von Eingabekomponenten generierte Ergebnisdokument wird von der Plattform im Ordner `content/parts` (siehe Dateisystem in Abb. 21) unter dem original Dateinamen (+ Dateiendung) mit angehängter Dateiendung `.part` abgelegt. Beispiel:

`example1.pdf → example1.pdf.part`

Entstehen neben dem eigentlichen Hauptdokument noch weitere Dateien (z.B. Bilder), können diese unter `content/parts` in einem Unterordner abgelegt werden, dessen Name dem der Quelldatei (+ Dateiendung) entspricht. Werden diese Dateien im Hauptdokument referenziert, muss der Verweis bei der Eingabe entsprechend angepasst werden.

5.3.3 Strukturierung

Die Plattform muss Content unterschiedlichster *Strukturierungsgrade* (vgl. Abschnitt 2.1.3.1) entgegennehmen und kombinieren können. Um auch schwach oder unstrukturierte Inhalte mit in die Gesamtstruktur aufnehmen zu können, müssen bestimmte Strategien zur Rekonstruktion oder Neuschaffung von Strukturen verfolgt werden.

Als Struktur wird in dieser Arbeit die hierarchische Gliederung (*outline*) eines Dokuments oder Moduls verstanden. Die Gliederungselemente werden als Struktureinheiten bezeichnet. In vielen Formaten und Informationsmodellen werden Struktureinheiten je nach Hierarchiestufe unterschiedlich benannt (vgl. z.B.: book, chapter, etc. bei *DocBook*; section, subsection, etc. bei *LATEX*, Überschrift 1, Überschrift 2, etc. bei *Microsoft Word*). In der von der Plattform verwendeten Informationsstruktur wird eine einheitliche Struktureinheit verwendet: section (siehe dazu Kapitel 5.2)

Prinzipiell müssen Eingabekomponenten versuchen, so viel Struktur wie möglich aus Quelldaten zu extrahieren oder zu rekonstruieren. Sind die Quelldaten strukturiert, sollten alle Struktureinheiten übernommen werden, um eine größtmögliche Flexibilität in der Kombination von Inhalten zu gewährleisten (siehe Abschnitt 5.3.4). Content, der sich mit den vorhandenen Mitteln nicht weiter automatisiert strukturieren lässt (z.B. Videos), wird als eine Struktureinheit betrachtet. Diese kann dann in der Hierarchie der Gesamtstruktur entsprechend eingesortiert werden (Komponente mid.sort).²⁹ Bei schwach strukturiertem Content muss grundsätzlich entschieden werden, bis zu welcher Hierarchietiefe eine nachträgliche automatisierte Strukturierung sinnvolle Ergebnisse erzielt. Beispiele für Strategien zur Strukturierung von schwach strukturiertem Content sind in Abschnitt 5.3.5 gelistet.

Die Festlegung der Größe von Struktureinheiten orientiert sich an grundlegenden Modularisierungsansätzen der Technischen Dokumentation (vgl. DREWER/ZIEGLER 2011:308). Durch die heterogenen Quelldaten und die daraus folgende weniger stark semantische Ausrichtung fallen bei der Plattform die Struktureinheiten im Gegensatz zu Modulen beim Einsatz von CMS jedoch tendenziell größer aus und entsprechen oft den Strukturen dokumentenorientierter Publikationen („Gliederungsbasierte Modularisierung“) (vgl. DREWER/ZIEGLER 2011:310). Des Weiteren ist zu beachten, dass gerade bei der Eingabe von Dokumenten (Word, PDF, etc.) die Struktureinheiten den Kapitelstrukturen entsprechen und der enthaltene Inhalt deshalb keine klassischen Moduleigenschaften (Unabhängigkeit vom Kontext, Innere Abgeschlossenheit) besitzt.

²⁹ Dieser Vorgang muss allerdings manuell vorgenommen werden, z.B. über die Strukturansicht des Web-Clients.

Eine weitere Entscheidung, die bei der Strukturierung getroffen werden muss, ist die der höchsten Hierarchieebene. So gibt es z.B. bei einem Word-Dokument, das mehrere Überschriften erster Ebene (*Überschrift 1*) besitzt (jedoch keinen *Titel*) prinzipiell zwei Möglichkeiten: *Überschrift 1* wird als höchste Hierarchieebene verwendet und auch entsprechend in der Gesamtstruktur eingeordnet oder eine Hierarchieebene wird eingeführt, die den *Dokumentrahmen* abbildet (dann jedoch keinen echten Titel hat). Bei den im Rahmen der Arbeit umgesetzten Komponenten wird so verfahren, dass wenn eine echte Dokument-Ebene mit Titel vorhanden ist, diese als oberste Hierarchieebene verwendet wird, wenn nicht jedoch auf die innerhalb des Dokuments höchsten Hierarchieebenen zurückgegriffen wird.

Bei Ausgabekomponenten muss desweiteren beachtet werden, dass viele Strukturmodelle dokumentenorientierter Formate eine begrenzte Zahl an Hierarchieebenen haben (HTML: 6, L^AT_EX: 7, etc.). Da die intern verwendete Informationsstruktur eine theoretisch unbegrenzte Anzahl von Hierarchieebenen zulässt, müssen hier ab einer gewissen Strukturtiefe die jeweiligen Sektionen verflacht werden.

5.3.4 Kombination

Durch die Einteilung in Struktureinheiten ist es möglich, auch über Hierarchiestufen hinweg Inhalte verschiedener Quellen miteinander zu kombinieren. Neben der Standard-Zusammenführung der Komponente `mid.merge`, die alle Quellen mit der jeweils obersten Hierarchie-Ebene aneinander reiht, ist es über die durch `mid.outline` erzeugte Strukturdefinition auch möglich, die Struktureinheiten der Quellen miteinander zu kombinieren. Denkbare Anwendungsfälle sind z.B. das Einfügen oder Austauschen von Sektionen sowie das Hinzufügen erweiterten Inhalts (z.B. Videos). Eine Strukturdefinition kann bei Darwin Information Typing Architecture (DITA)-Dateien auch die Aufgaben von DITA-Maps (vgl. DREWER/ZIEGLER 2011:419) übernehmen und die Konstellation von Topics bestimmen .

5.3.5 Beispiele

5.3.5.1 Video-Import

Mit der Komponente `in.video` lassen sich Videodateien in die Gesamtstruktur integrieren. Da Videos keine weiteren Hierarchieebenen besitzen, werden sie als einzelne Struktureinheiten betrachtet. Die Datei wird in die interne Informationsstruktur eingebunden und kann, falls vorhanden, mit Metadaten angereichert werden. Diese müssen in einem Nebenstrom (`meta`) vorliegen und enthalten einen Titel (der auch für die Struktureinheit verwendet wird) und eine Beschreibung.

5.3.5.2 PDF-Strukturrekonstruktion

Die Komponente `in.pdf` ist spezialisiert auf Dokumente mit nummerierten Überschriften. Diese werden z.B. häufig in Betriebsanleitungen, wissenschaftlichen Arbeiten oder Spezifikationen eingesetzt.

In einem ersten Prozessschritt wird der gesamte Inhalt der PDF als schwach strukturierter Text extrahiert (vermutete Absätze und Seitennummernbrüche werden von *Apache PDFBox* ausgezeichnet). Das Ergebnis wird anschließend gefiltert und umgeformt:

- Titelblatt wird entfernt (konfigurierbare Zahl von Seiten ab Start)
- Inhaltsverzeichnis wird entfernt (Seiten, die einen bestimmten *Token* enthalten). Dies funktioniert nur, wenn im Inhaltsverzeichnis, Einträge und Seiten durch z.B. Punkte voneinander getrennt werden (wie in dieser Arbeit).
- Je nach Konfiguration werden Kopf- und Fußzeile entfernt (der jeweils erste oder letzte Absatz einer Seite)
- Bildunterschriften werden anhand von Tokens (z.B. *Abb.*) entfernt, da keine Bilder extrahiert werden.
- Listsymbole wie *Bullets* (siehe diese Liste) oder *Dashes* am Anfang eines Absatzes werden zu Listenpunkten umgeformt

Die Struktur wird erzeugt, indem zunächst nach Absätzen gesucht wird, die mit einem bestimmten Zeichenmuster beginnen (z.B. *Zahl*, *Punkt*, *Leerzeichen*). Da nummerierte Überschriften in der Regel einen Punkt als Trennung zwischen Hierarchieebenen verwenden (5.3.5), lässt sich anhand der Anzahl von Punkten die Strukturtiefe erkennen. Die Nummerierung wird entfernt und die entsprechende Ebene ausgezeichnet. Anschließend wird das Dokument anhand der Auszeichnungen strukturiert.

5.4 SCHNITTSTELLENMODELL

5.4.1 Vorüberlegungen

Eine der Grundlagen für Funktion und Flexibilität eines Komponentensystems ist eine gemeinsame Schnittstellendefinition. Da Komponenten sowohl in Publikationsphasen als auch in anderen Komponenten eingesetzt werden können, muss es sich um eine *universelle* Schnittstellendefinition handeln, die in beiden Fällen gültig ist.

Wie bereits im Konzept beschrieben, ist es bei einer Publikationsplattform, deren Hauptaufgabe ist, Daten und deren Strukturen umzuformen, sinnvoll auf Medientypen zurückzugreifen, um die Verarbeitung einer Komponente zu beschreiben und damit ihre Ein- und Ausgabeschnittstelle festzulegen. Diese Medien-Typen sollen gemäß den Anforderungen auf Standards basieren (Abschnitt 3.2.1.3).

5.4.2 Datenfluss

Der Datenfluss innerhalb des Publikationssystems ist komplett *dateibasiert* (siehe Abschnitt 2.4.4). Dies gewährleistet einen stabilen Publikationsprozess, höchste Kompatibilität zwischen Komponenten und gute Debugging-Eigenschaften.³⁰

Das bedeutet, dass bei der Verarbeitung des in einer Komponente festgelegten Prozesses für jeden Zwischenschritt eine (temporäre) Datei angelegt wird. Die Festlegung von Speicherort und Benennung dieser Dateien sowie das Weitergeben dieser Informationen an die entsprechenden Schritte bzw. Komponenten ist Aufgabe des Publikationssystems (bzw. des *Compilers system.compile*).

5.4.3 Klassifizierung

Nach der Klassifizierung von DUMKE handelt es sich bei der umgesetzten Schnittstelle um eine *Datenkopplung*. Dies entspricht einer *schwachen Kopplung* und ist das empfohlene Muster bei Komponentensystemen (vgl. DUMKE 2013:59).

5.4.4 Media types

Eine der am meisten verbreiteten Standards auf dem Gebiet der Content- und Medien-Klassifizierung sind die von der Internet Assigned Numbers Authority (IANA) verwalteten *media types*³¹ (Medien-Typen), deren Aufbau und Zuordnung unter anderem von der Internet Engineering Task Force (IETF) spezifiziert werden.

Der Aufbau von Medien-Typen wird wie folgt beschrieben:

„The mechanism used to label such content is a media type, consisting of a top-level type and a subtype, which is further structured into trees. Optionally, media types can define companion data, known as parameters.“

(IETF RFC 6838 2013)

Beispiele für oft vorkommende Medien-Typen sind z.B. `text/html`, `application/xml` oder `image/jpeg`.

Ein Verzeichnis bereits standardisierter Medien-Typen kann bei IANA 2015 gefunden werden. Für proprietäre Medien-Typen steht der sogenannte *Vendor-Tree* zur Verfügung dessen *sub type tree* mit dem Präfix `vnd.` beginnt (vgl. IETF RFC 6838 2013:3.2).

³⁰ Alle Zwischenergebnisse von Verarbeitungsschritten werden gespeichert und lassen sich bei Bedarf leicht überprüfen.

³¹ Entsprechend ihres Ursprungs wurden *media types* früher als *MIME types* bezeichnet und wurden für die Deklarierung von Mail-Anhängen verwendet. Andere verwendete Bezeichnungen sind: *Content-Type* und *Internet media type*

Für die Beschreibung der zugrundeliegenden Struktur kann das Basisformat mit einem + angehängt werden (vgl. IETF RFC 3236 2002), wie z.B. bei application/xhtml+xml.

5.4.5 Spezifikation

Alle Eingabe- und Ausgabeschnittstellen (Schnittstellen-Typ) müssen durch ihren Fluss-Typ (Haupt- oder Nebenfluss) und ihren Medien-Typ bestimmt werden. Durch diese Bestimmung wird die Schnittstelle einer Komponente festgelegt, ihre Zuordnung zu einem Komponententyp bestimmt und ihre Funktion in einem Publikationsprozess eingeschränkt.

Beispiel für die Schnittstelle einer Eingabekomponente, die PDF-Daten verarbeitet:

```

1 <!-- Auszug aus Komponenten-Manifest -->
2 <interface>
3   <input type="main" template="application/pdf" />
4   <output type="main" template="vnd.infoflow.content+xml" />
5 </interface>
```

Code 8: Schnittstellenspezifikation Eingabe

Validierungen von Publikationsdefinitionen und Komponenten basieren auf den Angaben über die Schnittstellen.

5.4.5.1 Schnittstellen-Typen

Der Typ der Schnittstelle wird im Element <interface> (Schnittstellen-Definition einer Komponente) definiert:

- <input>: Eingabe-Schnittstelle
- <output>: Ausgabe-Schnittstelle

Die Kombination aus allen Eingabe- und Ausgabeschnittstellen wird als *Schnittstelle* der Komponente bezeichnet.

5.4.5.2 Fluss-Typen

Der Fluss-Typ eines Schnittstellen-Typs wird im Attribut type des jeweiligen Elements mit einem Schlüssel angegeben. In der derzeitigen Umsetzung sind zwei Schlüssel einsetzbar:

MAIN

Der Hauptfluss der Daten. Pro Komponente ist mindestens³² eine Eingabeschnittstelle vom Typ main und genau eine Ausgabeschnittstelle vom Typ main erforderlich.

³² Mehrere Eingabeschnittstellen vom Typ main werden bei der Kompilierung zusammengefasst. Dadurch können mehrere Medien-Typen kombiniert werden.

META

Komponenten können für Nebenflüsse von Daten den Fluss-Typ `meta` verwenden. Durch diesen können der Komponente z.B. für die Verarbeitung verwendete *Metadaten* zugeführt werden. Nebenflüsse sind per Definition optional, das bedeutet, dass eine Komponente auch ohne Daten des Nebenflusses fehlerfrei³³ arbeiten muss.

Prinzipiell sind für die Nebenflüsse weitere Arten vorstellbar und sinnvoll (z.B. für Konfigurationen oder Referenzdateien). In der technischen Implementierung wurde bisher jedoch nur der `meta`-Fluss umgesetzt. Dieser kann aber für jede Art von Nebenfluss-Dateien verwendet werden.

5.4.5.3 *Medien-Typen*

Der Medien-Typ eines Schnittstellen-Typs wird im Attribut `template` des jeweiligen Elements mit einem *Template* angegeben. In der derzeitigen Umsetzung sind drei Template-Varianten möglich:

MEDIA-TYPE

Ist der Medien-Typ der Schnittstellen-Daten eindeutig, kann der bei der IANA registrierte³⁴ Template-String verwendet werden (z.B. `application/xml`). Dies ist die bevorzugte Vorgehensweise. Handelt es sich um ein proprietäres Format oder einen intern verwendeten Medientyp, wird der Vendor-Tree im `subtype` benutzt (z.B. `application/vnd.infoflow.content+xml`).

WILDCARD

Ist der Medien-Typ der Schnittstellen-Daten nicht eindeutig festgelegt, kann mit *Wildcards* (Platzhaltern) gearbeitet werden. Das Wildcard-Zeichen ist: `*`. Wildcards können sowohl nur im `subtype` eingesetzt werden (z.B. `image/*`) als auch global (`*` oder `*/*`, beide Varianten werden gleich behandelt).

EXTENSION

Ist der Medien-Typ der Schnittstellen-Daten nicht als Template-String registriert, jedoch über die Datei-Erweiterung (*Extension*) erkennbar, kann die Dateierweiterung mit einem vorangestellten `.` (*Punkt*) angegeben werden (z.B. `.dita`).

Generell sollten Schnittstellen immer so genau wie möglich spezifiziert werden, um eine korrekte Validierung zu ermöglichen und Kompatibilitätsprobleme zu vermeiden. Die angegebenen Daten werden z.B. auch vom Web-Client ausgewertet, um eine Filterung erlaubter Eingabedateien im Frontend zu ermöglichen.

³³ Das kann auch bedeuten, dass eine Komponente im Fall des Fehlens einer Metadatei, den Content im Hauptfluss nur unverändert durchschleust.

³⁴ Ein Verzeichnis der registrierten Typen findet sich bei IANA 2015.

5.4.6 Validierung

Die Validierung von Informationsflüssen ist in der Systemkomponente `system.validate` umgesetzt. Sollen alle Komponenten des Publikationssystems per Stapelverarbeitung überprüft werden, kann die Komponente `system.validate-all` eingesetzt werden. Beide Komponenten werden unabhängig von einer Publikationsdefinition direkt ausgeführt.

5.5 PUBLIKATIONSSYSTEM

5.5.1 Architektur

In einer *Publikationsdefinition* sind Komponenten mit zugehörigen Eingabedateien in jeweils eine der drei Publikationsphasen eingeordnet. Bei Ausführung der Publikation (*Publishing*) wird der Publikationsprozess zunächst in einen Ablaufplan übersetzt (*assembliert*), der nach Regeln der Publikationsphasen Komponenten *kompiliert* und anschließend mit den entsprechenden Eingabedateien *instanziert*. Der generierte Ablaufplan wird anschließend ausgeführt und die Publikation dadurch gestartet.

Kompilierungen und Instanzierungen von Komponenten sind Unterprozesse des Publikationsprozesses. Instanzierte Komponenten können selbst auch Unterprozesse starten (z.B. externe Programme) oder Skripte erzeugen und diese ausführen. Dadurch sind beliebig tiefe Prozessschachtelungen möglich.

5.5.2 Technologie

5.5.2.1 Übersicht

Die Umsetzung des Publikationssystems mitsamt seiner Umsetzung durch Komponenten basiert auf XML-Technologien.

Das bedeutet konkret, dass alle Manifest-Dateien (Komponenten, Publikationsdefinitionen) und Verzeichnisse (Repository) als XML-Dateien repräsentiert werden. Die Kompilierung und Assemblierung von Manifesten, sowie die Generierung von Verzeichnissen erfolgt durch XSL-Transformationen. Die vom Host-System ausführbare Zielsprache ist die von *Apache Ant* verwendete XML-Syntax.

5.5.2.2 XML

XML wurde als Basis für die Umsetzung des Publikationssystems gewählt, da es als ausgereifter und flexibler Standard zukunftssicher

und im TD-Umfeld etabliert ist (vgl. PELSTER 2011:54). Die Erstellung von Manifesten oder Stylesheets kann sowohl automatisiert als auch *von Hand* erfolgen. Trotz der strukturellen Stärken des Formats bleibt es *menschenlesbar*.

Andere Formate kamen wegen zu geringer Funktionalität (JSON) oder zu hoher Komplexität (RDF) nicht in Frage (siehe Abschnitt 2.2.3f).

5.5.2.3 XSL

Zur Verarbeitung der XML-basierten Teile des Publikationssystems kommt XSLT 2.0 als Transformationssprache zum Einsatz. Vorteile ergeben sich vor allem durch die tiefen Integrierungsmöglichkeiten in *Apache Ant* als Bestandteil des Host-System und der Nähe zu allen XML-verwandten Sprachen.

5.5.2.4 Ant-XML

Als Syntax für die ausführbare (kompilierte) Variante einer Komponente kommt die von *Apache Ant* verwendete XML-Syntax zum Einsatz (im folgenden Ant-XML genannt). Durch die syntaktische Nähe von Grundform und kompilierter Variante eines Manifests ist eine sehr effiziente Kompilierung zur Laufzeit möglich.

Eine Implementierung mit einer anderen Sprache oder einem anderen System ist ebenfalls denkbar (siehe Kapitel 7.3).

5.5.3 Host-System

In der technischen Umsetzung wird das Host-System aus einer Kombination von *Apache Ant*, dem installierten Betriebssystem und den darauf vorhandenen *Programmen und Funktionen* gebildet. Das Host-System ist damit, wie in Abb. 20 dargestellt, selbst aus verschiedenen Schichten aufgebaut.

5.5.3.1 Apache Ant

Bei *Apache Ant* (im folgenden: *Ant*) handelt es sich um ein *Build-Tool*, dessen Entwicklung von der Apache Software Foundation (ASF) koordiniert wird.

Unter *Build* versteht man den Erstellungsprozess eines Programmes, also das Kompilieren und Referenzieren von Code-Blöcken. Von den Entwicklern wird das Tool wie folgt beschrieben:

„[Ant's] mission is to drive processes described in build files as targets and extension points dependent upon each other. The main known usage of Ant is the build of Java applications.“ (APACHE SOFTWARE FOUNDATION 2014b)

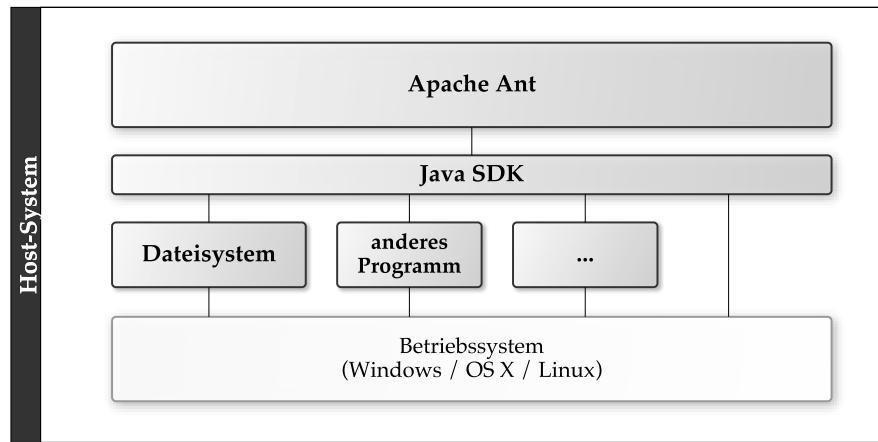


Abb. 20: Host-System: Aufbau und Kommunikation

Doch nicht nur der Build-Prozess von Java-Anwendungen kann mit Ant automatisiert werden, sondern auch diverse Operationen, die mit der Verwaltung, Umwandlung und Ausführung von Dateien und Programmen zusammenhängen (siehe Abschnitt 3.3.2). Aus diesen Gründen wird Ant mittlerweile auch in angrenzenden Bereichen wie der Web-Entwicklung eingesetzt (vgl. FULLER/FULLER 2010). In der offiziellen Beschreibung der Software heißt es inzwischen deshalb allgemeiner:

„More generally, Ant can be used to pilot any type of process which can be described in terms of targets and tasks.“
 (APACHE SOFTWARE FOUNDATION 2014b)

Ant basiert auf Java und ist damit plattformunabhängig einsetzbar (Systemvoraussetzung ist ein installiertes *Java Developer Kit (JDK)*). Prozesse werden in XML-Dateien in Form von Projekten (*projects*), Zielen (*targets*) und Aufgaben (*tasks*) beschrieben.

Bei der Umsetzung des Konzepts ist die XML-Notation einer der wesentlichen Ansatzpunkte. Ant-Prozessbeschreibungen (sog. *build files*) in XML können automatisiert durch XSL-Transformationen erzeugt werden. Da Ant-Prozesse selbst auch XSL-Transformationen anstoßen können, ist es also auch möglich, dass diese wiederum andere Ant-Prozesse generieren und/oder starten. Diese Idee wird in Begleitdokumenten zu Ant kurz beschrieben:

„XSLT can be used to dynamically generate build.xml files from a source XML file, with the XSLT task controlling the transform. This is the current recommended strategy for creating complex build files dynamically. However, its use is still apparently quite rare - which means you will be on the bleeding edge of technology.“

(vgl. LOUGHAN 2005)

Die hier beschriebene dynamische Erzeugung von Ant-Prozessen ist die Basis für das Assemblieren von Publikationsdefinitionen und das Kompilieren von Komponenten. In beiden Fällen wird aus einer formalen Syntax in eine Sprache übersetzt, die vom System ausführbar ist (Programmcode).

5.5.3.2 *Betriebssystem*

Das Betriebssystem stellt als Hauptfunktionalität ein Dateisystem zur Verfügung, auf das Apache Ant und ggf. die Web-API Zugriff haben. Des Weiteren dient es als Plattform für weitere Programme, auf die das Publikationssystem Zugriff hat (siehe Abschnitt 5.5.3.3).

Durch den Einsatz von Ant als Abstraktionsschicht spielt die genaue Art des Betriebssystems nur eine untergeordnete Rolle. Beim Erstellen von Komponenten muss deshalb darauf geachtet werden, keine plattformspezifischen Konventionen oder Funktionalitäten zu benutzen.

Erfolgreich getestet wurde das Publikationssystem auf den Betriebssystemen: Microsoft Windows 7, Microsoft Windows Server 2012, os x 10.9 (Mavericks) und os x 10.10 (Yosemite).

5.5.3.3 *Weitere Funktionalitäten*

Wie im Konzept beschrieben dürfen Komponenten, mit Ausnahme des Host-Systems, keine externen Abhängigkeiten haben. Werden in einer Komponenten Funktionen abgebildet, die externe Programme benötigen, sollte deshalb immer das entsprechende Programm mit der Komponente gekapselt werden (z.B. als .jar-Datei im contents-Ordner der Komponente). In manchen Fällen ist dies jedoch nicht möglich, da das benötigte Programm nicht gekapselt oder ohne vorherige Installation nicht ausgeführt werden kann (z.B. iOS- und Android-SDKs oder eine L^AT_EX-Distribution). In diesem Fall müssen die Programme auf dem Host-System installiert werden und über eine Basis-Komponente dem Publikationssystem zur Verfügung gestellt werden. Dadurch unterliegen diese Funktionen der Abhängigkeitsverwaltung des Publikationssystems.

Beispiele sind z.B. die Komponenten base.latex oder base.cordova.

5.5.4 *Aufbau*

Das Publikationssystem setzt sich aus zwei Haupt-Bestandteilen zusammen, die den Aufbau bestimmen: *Komponenten* und *Publikationen* (bzw. deren zugehörige Dateien).

Der schematische Aufbau des zentralen Ordners eines *Publikationssystems* entspricht der folgenden Ordnung:

Darstellung Dateisystem

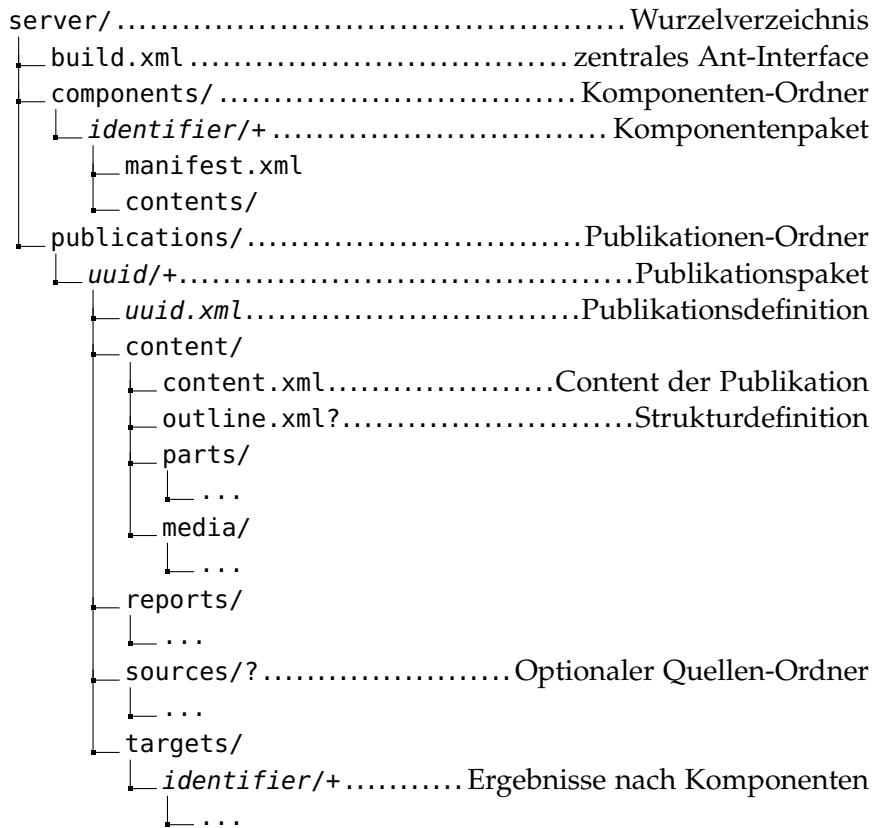


Abb. 21: Publikationssystem: Aufbau Dateisystem

5.5.4.1 Komponenten

Alle Komponentenpakete eines Publikationssystems werden ohne zusätzliche Hierarchieebenen in einem Ordner `components` hinterlegt. Die Bezeichnung der Pakete entspricht dem Identifier der jeweiligen Komponente (siehe Abschnitt 5.6.1). Dadurch ist eine automatisierte Lokalisierung von Komponenten möglich, ohne dass verteilte Speicherorte gepflegt werden müssen.

5.5.4.2 Publikationen

Alle Dateien, die mit einer Publikation zusammenhängen, werden in jeweils einem Ordner, der mit der UUID der Publikation benannt ist, unterhalb des Ordner `publications` zusammengefasst. Die Struktur eines Publikationsordners ist dabei immer gleich:

PUBLIKATIONSDDEFINITION

Auf oberster Ebene wird das Manifest der Publikationsdefinition als XML-Datei abgelegt. Der Dateinamen entspricht der UUID der Publikation.

INHALT

Der Gesamtinhalt der Publikation in der internen Informationsstruktur wird im Ordner `content` abgelegt. Aktuelle Hauptdatei ist dabei immer `content.xml`. Wurde durch eine entsprechende Komponente³⁵ eine Strukturdefinition des Inhalts erzeugt, so ist diese in `outline.xml` gespeichert. In den Unterordnern befinden sich jeweils die von den Eingabekomponenten erzeugten Teil-Inhalte (`parts`) bzw. die zur Publikation zugehörigen Mediendateien (`media`).

BERICHTE

In den Ordner `reports` werden standardmäßig alle Log-Dateien und Fehlerberichte der an der Publikation beteiligten Komponenten geschrieben.

Dieser Ordner ist nur für Debugging-Zwecke relevant.

QUELLEN

Die originalen Quelldateien einer Publikation werden beim Betrieb als Publikationsplattform im Ordner `sources` gespeichert. Da in einer Publikationsdefinition die Eingabedateien aus beliebigen Speicherorten referenziert werden können, kann bei einem Alleinbetrieb des Publikationsserver auf diesen Ordner verzichtet werden.

ZIELE

Die Ergebnisse eines Publikationsprozesses werden in Unterordnern von `targets` gespeichert, die nach der jeweiligen Komponente benannt sind (ohne den `out`-Teil des Klassenpfades).

5.5.5 Bootstrapping

Das Publikationssystem ist vollständig aus Komponenten aufgebaut. Das Kompilieren von Komponenten wird wiederum durch eine spezielle Komponente (den *Compiler*) vorgenommen, der aber zur Ausführung auch kompiliert werden muss. Dies ist ein in der Informatik und speziell bei Compilern verbreitetes *Henne-Ei-Problem*. Ein möglicher Lösungsansatz ist das sogenannte *Bootstrapping*. Dabei wird aus bestehenden, einfachen Mitteln das komplexe Gesamtsystem aktiviert. Bei der Umsetzung des Publikationssystems wird eine solche Bootstrapping-Methode für das Initiiieren des Publikationsprozesses angewandt. In einer statischen Datei (`build.xml`) sind Aufrufe von *Compiler* und *Assembler* vorkompiliert. Diese setzen wiederum den restlichen Publikationsprozess in Gang. Die Bootstrapping-Datei dient gleichzeitig als Benutzerschnittstelle für das Publikationssystem (siehe Abschnitt 5.5.6).

³⁵ Die Komponente `mid.outline` erzeugt eine Strukturdefinition des Inhalts, die verändert und von der Komponente `mid.sort` verarbeitet werden kann.

5.5.6 Benutzerschnittstelle

Das Publikationssystem stellt als Benutzerschnittstelle eine auf Ant basierende Kommandozeilenschnittstelle (*Command Line Interface*) bereit (`build.xml`), deren Aufruf mit entsprechenden Parametern den Publikationsprozess steuern kann. Die verschiedenen Funktionen sind dabei als *Ant-Targets* implementiert. Der Aufruf wird über das Alias `infoflow` gestartet (Linux/os x: `./infoflow`, Windows: `infoflow.bat`).

```
1 $ ./infoflow publish -Dpublication=p-d6c28e22
```

Code 9: infoflow-Kommandozeile: Publikation

```
1 $ ./infoflow run -Dcomponent=system.validate-all
```

Code 10: infoflow-Kommandozeile: Validierung

Die möglichen Parameter sind in Tabelle 6 aufgeführt. K entspricht dabei dem Identifier der jeweiligen Komponente, P der UUID der Publikationsdefinition und Q dem Pfad zur Quelldatei.

AKTION	AUFRUF
Hilfe anzeigen	<code>infoflow help</code>
Web-API starten	<code>infoflow start</code>
Komponente K kompilieren	<code>infoflow compile -Dcomponent=K</code>
Komponente K mit Quelle Q ausführen	<code>infoflow run -Dcomponent=K -Dsource=Q</code>
Publikation P assemblieren	<code>infoflow assemble -Dpublication=P</code>
Publikation P publizieren	<code>infoflow publish -Dpublication=P</code>

Tabelle 6: infoflow Kommandozeilen-Parameter

Neben den beiden Hauptfunktionen *Assemblieren* (`assemble`) und *Kompilieren* (`compile`) ist es möglich, Komponenten oder Publikationsdefinitionen auch direkt auszuführen (`run` und `publish`). In diesen Fällen werden die entsprechenden Umwandlungen zunächst automatisch ausgeführt und anschließend die übersetzte Variante aufgerufen.

Zusätzlich kann über die Benutzerschnittstelle des Publikationssystems auch die Web-API über *Node.js* gestartet werden.

5.6 KOMPONENTEN

5.6.1 Identifier

Wie im Konzept beschrieben, wird Komponenten über einen Klassennpfad ein eindeutiger Bezeichner zugewiesen. Dieser wird innerhalb der Umsetzung als *Identifier* bezeichnet. Erster Bestandteil dieses Pfades ist immer der Schlüssel einer der fünf Komponententypen.

SCHLÜSSEL	KOMPONENTENTYP
base	Basiskomponente
system	Systemkomponente
in	Eingabekomponente
mid	Verarbeitungskomponente
out	Ausgabekomponente

Tabelle 7: Komponenten: Schlüssel-Typ-Zuordnung

Der vollständige Pfad wird durch Punkte getrennt, zusammen- und kleingeschrieben. Beispieldpfad für eine Komponente, die z.B. eine App für das Betriebssystem iOS ausgibt, wäre:

out . app . ios
 ⌄ ⌄ ⌄
 Typ Gruppe Name

Identifier müssen global eindeutig sein. Der Namensteil eines Identifiers muss innerhalb der jeweiligen Gruppe eindeutig sein, so sind z.B. sowohl `in.pdf` als auch `out.pdf` erlaubt.

5.6.2 Aufbau

Komponenten werden dem Konzept entsprechend als Pakete in entsprechend benannten Dateiordnern verteilt. Name des Ordner ist der Identifier der Komponente (siehe Abschnitt 5.6.1). Inhalte des Ordners sind eine Manifest-Datei `manifest.xml` und ein Unterordner mit der Bezeichnung `contents`.

Darstellung Dateisystem

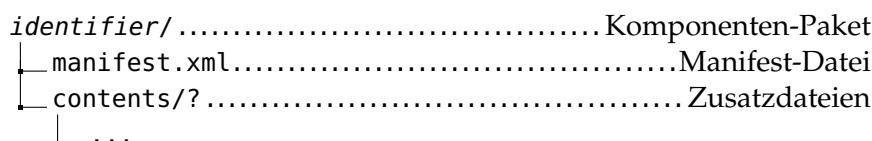


Abb. 22: Aufbau Komponentenpaket

5.6.2.1 *Manifest*

In Abb. 23 wird der schematische Aufbau einer Manifest-Datei spezifiziert. Für die XML-Präsentation wurde ein eigenes Schema gewählt, um die Syntax simpel und die Dateigröße gering zu halten.³⁶

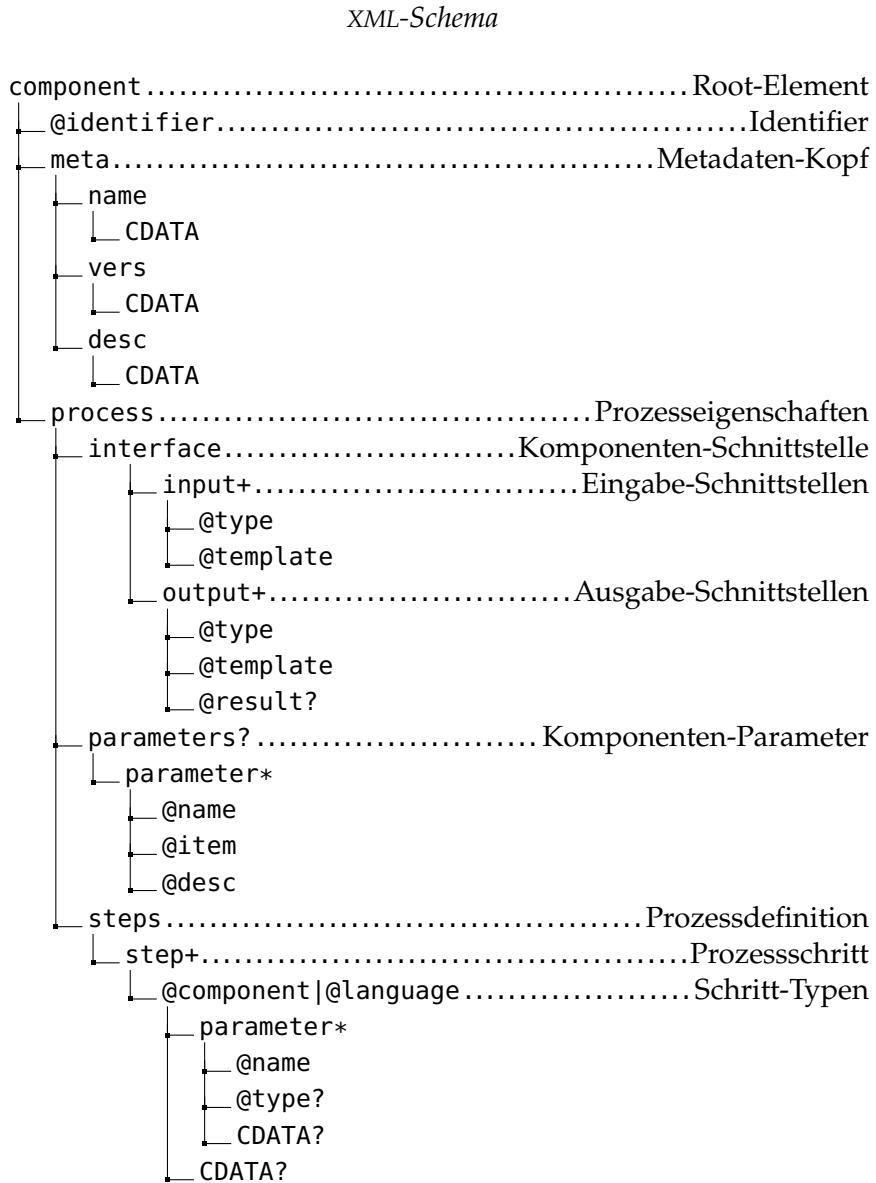


Abb. 23: Schema eines Komponenten-Manifests

Im Schema wurden die Anforderungen entsprechend der konzeptuellen Vorgaben umgesetzt (Abschnitt 4.4.2). Unterhalb des Wurzellements `component`, das im Attribut `identifier` den eindeutigen Bezeichner der Komponente enthält, wird in zwei Teile untergliedert: `meta` (Metadaten-Kopf) und `process` (Prozesseigenschaften).

³⁶ Prozessdefinitionssprachen wie BPMN (Business Process Model and Notation) oder RDF wären auch möglich, verursachen jedoch zu viel Overhead.

Innerhalb der Prozesseigenschaften werden die akzeptierten Parameter (nur bei Basis-Komponenten relevant) und die Schnittstelle der Komponente definiert (*interface*) sowie der Ablauf des Prozesses in Schritten festgelegt (*steps*).

Der Entwurf des Schemas verfolgt das Ziel, eine möglichst flexible Struktur zu bieten, in der alle Komponententypen hinreichend abgebildet werden können. Dennoch gibt es wenige Besonderheiten, die beachtet werden müssen:

- Das Element `parameters` wird nur bei Basiskomponenten eingesetzt und beschreibt die in `steps` verwendeten Parameter.
- Das Attribut `result` an `output`-Elementen wird nur bei Ausgabe-Komponenten eingesetzt und gibt dort die Haupt-Ergebnisdatei (bzw. *Vorschaudatei*) der Publikation an (z.B. *index.html*).
- Das Element `step` kann immer nur eines der beiden Attribute `component` oder `language` besitzen.
- Nur `step`-Elemente, die das Attribut `component` besitzen und eine Basis-Komponente referenzieren, können `parameter`-Kindelemente haben.

5.6.2.2 Contents

Im Unterordner `contents` können beliebige Dateien und Strukturen abgelegt werden. Es können zusätzliche Unterordner mit Inhalten angelegt werden. Der Ordnerinhalt enthält alle Dateien, die zur Ausführung des im Manifest beschriebenen Prozesses benötigt werden (in Kombination mit den vom Host-System zur Verfügung gestellten Funktionen).

Sind keine weiteren Dateien nötig, kann der `contents`-Ordner auch weggelassen werden.

5.6.3 Schritte

Die im Konzept beschrieben Arten von Schritten werden in Attributen am Element `step` festgelegt (siehe Tabelle 8).

TYP	ATTRIBUT	MÖGLICHE WERTE
Komponenten	<code>component</code>	Gültiger <i>Komponenten-Identifier</i>
Natives Skript	<code>language</code>	ant oder javascript

Tabelle 8: Komponenten: Schritt-Typen

Schritte, die Komponenten referenzieren, können als einzigen Inhalt `parameter`-Elemente besitzen (oder keinen Inhalt). Native Schritte enthalten das Skript direkt als textuellen Element-Inhalt (CDATA). Bei

JS-Schritten sollte das Skript in CDATA-Anweisungen eingeschlossen werden, um die Wohlgeformtheit des Manifest-XMLs zu gewährleisten.

Die Möglichkeit Skript-Dateien zu referenzieren ist in den folgenden Basiskomponenten umgesetzt: `base.ant` (Ant-Dateien) sowie `base.js.rhino` und `base.js.node` (JS-Dateien). Die Referenzierung der Datei erfolgt durch den Parameter `script`.

5.6.4 Parameter

Wie im Konzept beschrieben, können Basis-Komponenten Parameter von ihnen übergeordneten Komponenten Parameter entgegennehmen. Diese werden bei der Referenzierung der Basiskomponente in der übergeordneten Komponente mitgegeben:

```

1 <!-- Auszug aus Komponenten-Manifest -->
2 <step component="base.transform">
3   <parameter type="contents" name="xsl">parse.xsl</parameter>
4 </step>
```

Code 11: Parameter-Übergabe

Parameter haben einen Namen (`name`) und einen optionalen Typ (`type`). Der Wert des Parameters wird im Elementinhalt angegeben. Wird kein Typ spezifiziert, wird der Wert des Parameters unverändert weitergegeben.

Parameter-Typen dienen zur einfachen Referenzierung von Dateien innerhalb des Publikationssystems. Durch ihre Zuordnung wird der im Parameter-Element angegebene relative Pfad bei der Kompilierung in einen absoluten Pfad umgewandelt. Zur Verfügung stehende Typen sind:

TYP	INHALT
<i>ohne</i>	Unverändert
<code>contents</code>	Pfad ab <code>contents</code> -Ordner der aktuellen Komponente.
<code>base</code>	Pfad ab Root-Verzeichnis des Publikationssystems
<code>pub</code>	Pfad ab Publikationsverzeichnis der aktuellen Publikation (<code>/publications/[UUID]/...</code>)
<code>tmp</code>	Pfad ab temporärer Ordner des Publikationssystems

Tabelle 9: Komponenten: Parameter-Typen

Innerhalb der Basiskomponenten werden die akzeptierten Parameter im Element `parameters` definiert. Dort wird neben dem Namen des Parameters (`@name`) auch eine kurze Beschreibung (`@desc`) und der

Objekttyp (@item) festgelegt. Diese Angaben werden bei der Kompliierung nicht ausgewertet, sondern dienen der Dokumentation und Verwertung durch potentielle Autorentools für Komponenten.

```

1 <!-- Auszug aus Komponenten-Manifest -->
2 <parameters>
3   <parameter name="xsl" item="file" desc="The XSL stylesheet to
      transform with" />
4 </parameters>
```

Code 12: Parameter-Definition

5.6.5 Kontextinformationen

Nach dem in Abb. 17 beschriebenen Schema stehen in verschiedenen Ebenen des Publikationssystems Kontextinformationen zur Verfügung.

Auf die enthaltenen Daten kann im Manifest und in nativen Ant-Schritten (Umgebung: *Ant*) mit der Ant-Variablen-Notation (\${typ}) zugegriffen werden. In nativen JavaScript-Schritten können die Daten über die Ant-Schnittstelle (`project.getProperty('typ');`) und innerhalb von Extensible Stylesheet Language (XSL) mit übergebenen Stylesheet-Parametern abgerufen werden (`<xsl:param name="typ"/>`).

TYP	KONTEXT	UMGEBUNG	BESCHREIBUNG
base.dir	System	Ant, JS, XSL	Root-Verz. Pub.-System
tmp.dir	System	Ant, JS, XSL	Temp-Verz. Host-System
pub.dir	System	Ant, JS, XSL	Publikationsverzeichnis
publication	Pub.	Ant, JS, XSL	UUID Publikationsdef.
component	Komp.	Ant, JS, XSL	Identifier Komponente
source	Komp.	Ant, JS	Pfad Quelldatei
input.path	Komp.	XSL	Pfad Quelldatei
source.file	Komp.	Ant, JS	Dateiname Quelldatei
input.file	Komp.	XSL	Dateiname Quelldatei
input.meta	Komp.	Ant, JS, XSL	Pfad Nebenfluss-Datei
step.in	Schritt	Ant, JS	Pfad Eingabedatei
step.out	Schritt	Ant, JS	Pfad Ausgabedatei
stepContent	Schritt	JS (native var)	Inhalt des Schrittes

Tabelle 10: Komponenten: Kontextinformations-Typen

5.6.6 Klassifizierung

Nach der Klassifizierung von DUMKE handelt es sich bei der inneren Bindung (Kohäsion) der Komponenten um eine *Sequentielle Bindung*. Dies entspricht einer *starken Bindung* und ist das empfohlene Muster bei Komponentensystemen (vgl. DUMKE 2013:59).

5.6.7 Beispiel

In Code 13 wird das Manifest der Komponente base.xml.clean in der Grundform dargestellt. Die Komponente hat die Aufgabe, XML-Dateien *aufzuräumen* bevor sie weiterverarbeitet werden, um Probleme mit Kodierungen sowie nicht deklarierten Entitäten oder Dokumenttyp-Definitionen (DTDs) zu vermeiden.

```

1 <!-- base.xml.clean/manifest.xml -->
2 <component identifier="base.xml.clean">
3   <meta>
4     <name>Clean XML</name>
5     <vers>0.3</vers>
6     <desc>...</desc>
7   </meta>
8   <process>
9     <interface>
10    <input type="main" template="application/xml" />
11    <output type="main" template="application/xml" />
12  </interface>
13  <parameters />
14  <steps>
15    <step language="ant">
16      <replaceregexp file="${step.in}" match="&lt;!DOCTYPE[\s\S]*?&gt;" replace="" byline="true" />
17      <copy file="${step.in}" tofile="${step.out}" />
18    </step>
19    <step language="javascript"><![CDATA[
20      stepContent = stepContent.replaceAll(' ', ' ');
21    ]]></step>
22    <step component="base.js.node">
23      <parameter type="contents" name="script">fix.js</
24        parameter>
25    </step>
26    <step component="base.transform">
27      <parameter type="contents" name="xsl">clean.xsl</
28        parameter>
29    </step>
30  </steps>
31  </process>
32</component>
```

Code 13: XML-Notation einer Komponente

5.6.7.1 Anmerkungen

In der Schnittstellendefinition der Komponente (*interface*) werden XML-Dateien (*application/xml*) als Ein- und Ausgabe bestimmt. Parameter werden von der Komponente keine verwendet (*parameters*).

In der Komponente wird der Content in mehreren Schritten auf verschiedene Weisen verarbeitet (siehe nachfolgende Liste).

- Ein *nativer* Schritt in Ant-Syntax. Mit dem Befehlt `replaceregexp` wird mit einem regulären Ausdruck nach `DOCTYPE`-Definitionen gesucht und diese entfernt. Da bei dieser Bearbeitung die Eingabedatei verändert wird, muss diese noch in die Ausgabedatei kopiert werden, damit der nächste Schritt Zugriff auf die Ergebnisse hat.
- Ein *nativer* Schritt in JavaScript (JS). Im Content werden *named entities* eines geschützten Leerzeichens () durch eine *numeric entity* () ersetzt. Innerhalb von JS dient die Variable `stepContent` für den Austausch der Daten mit vorherigem und nachfolgendem Schritt (siehe Tabelle 9).
- Das Ausführen einer JS-Datei in `Node.js` durch die Referenzierung der Komponente `base.js.node`. Die Datei im `contents`-Ordner der Komponente wird über den Parameter `script` referenziert.
- Das Ausführen einer XSL-Transformation durch die Referenzierung der Komponente `base.transform`. Die per Parameter referenzierte Datei dient dabei als Stylesheet. Ein- und Ausgabedatei des Schrittes entsprechen denen der Transformation.

Wie zu erkennen, wird die Datenübergabe zwischen den Schritten fast vollständig vom Publikationssystem übernommen. Dadurch ist ein einfaches Aneinanderreihen (*Piping*) der Schritte möglich.

5.6.8 Abhängigkeiten

Die unter den Komponenten bestehenden Abhängigkeiten sind in Abb. 24 dargestellt. Eine interaktive³⁷ Variante der Darstellung kann unter INFOFLOW 2015a erreicht werden.

Wie in der Grafik zu erkennen, bestehen die meisten Abhängigkeiten zur Komponente `base.transform` (XSL-Transformationen). Aufgrund der XML-basierten Umsetzung war diese Häufung zu erwarten. Daraus wird deutlich, dass z.B. die Wahl des verwendeten XSL-Prozessors erhebliche Auswirkungen auf die gesamte Funktionalität des Publikationsservers hat.

³⁷ Abhängigkeiten und Abhängige können durch *MouseOver* pro Komponente separat dargestellt werden.

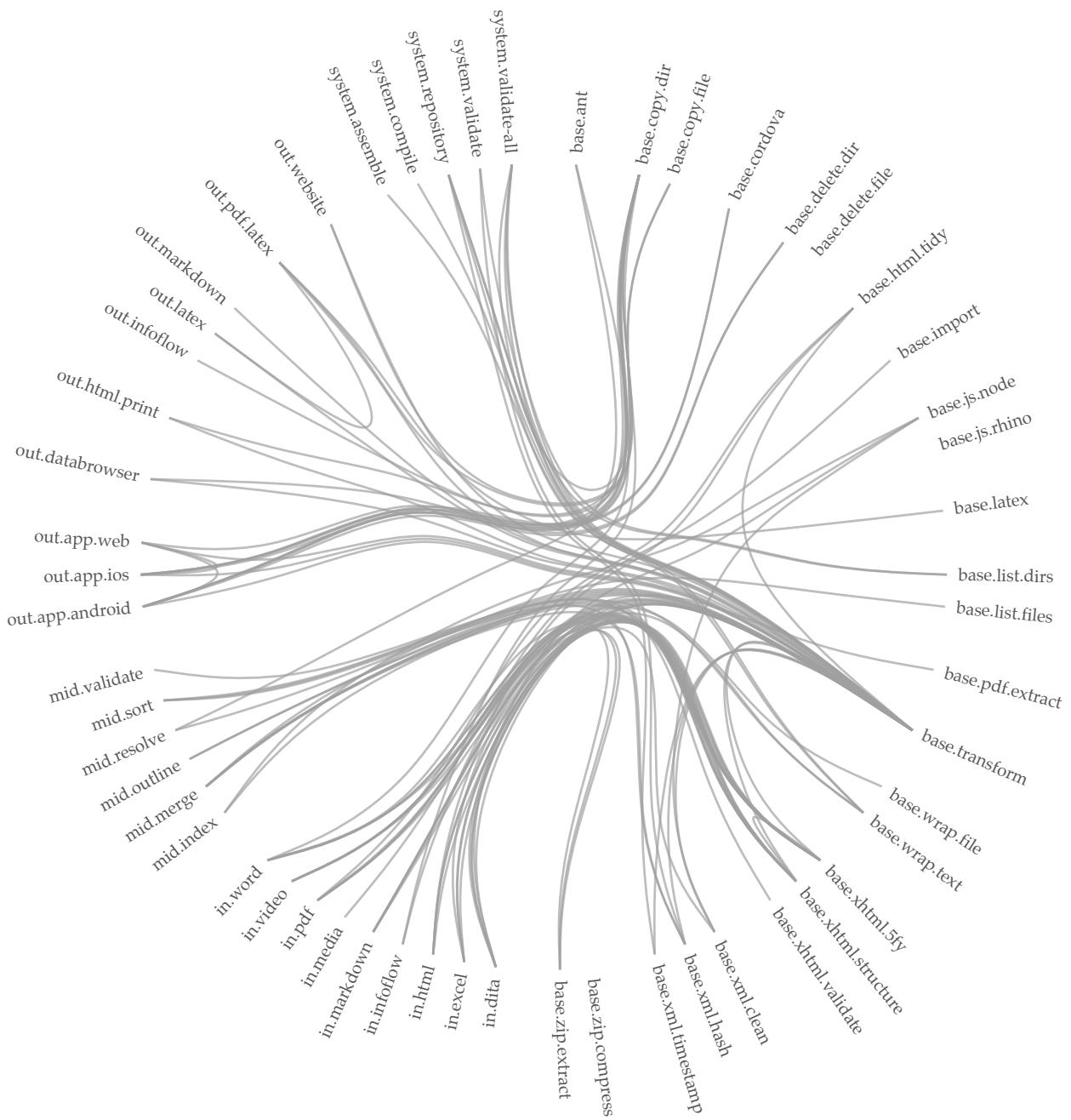


Abb. 24: Komponenten: Alle Abhangigkeiten

Weitere *Cluster* von Abhängigkeiten, die in der Grafik zu erkennen sind, werden im Folgenden aufgelistet:

- Zwischen Eingabe-Komponenten und `base.xhtml.structure`. (siehe Abb. 25, Rot: Abhängigkeiten, Grün: Abhängige, Schwarz: Wurzel)

Da bei der Eingabe in eine strukturierte HTML5-Variante übersetzt werden muss, wandeln viele Komponenten zunächst nach HTML um und setzen danach `base.xhtml.structure` zur Strukturierung ein.

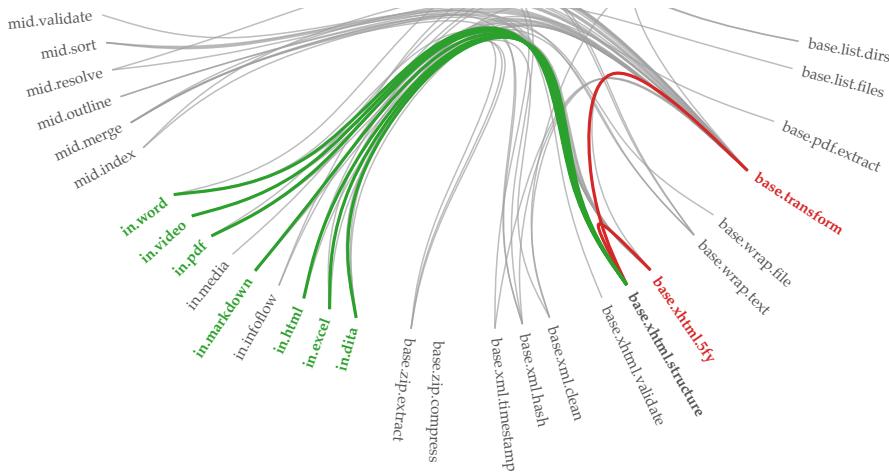


Abb. 25: Abhängigkeiten: Strukturieren

- Zwischen Ausgabe-Komponenten und `base.copy.dir`. (siehe Abb. 26, Grün: Abhängige, Schwarz: Wurzel)

Während der Publikation werden referenzierte Medien im Unterordner `media` gespeichert. Dieser wird bei der Ausgabe von vielen Ausgabekomponenten mit Hilfe von `base.copy.dir` in den Ergebnis-Ordner kopiert.

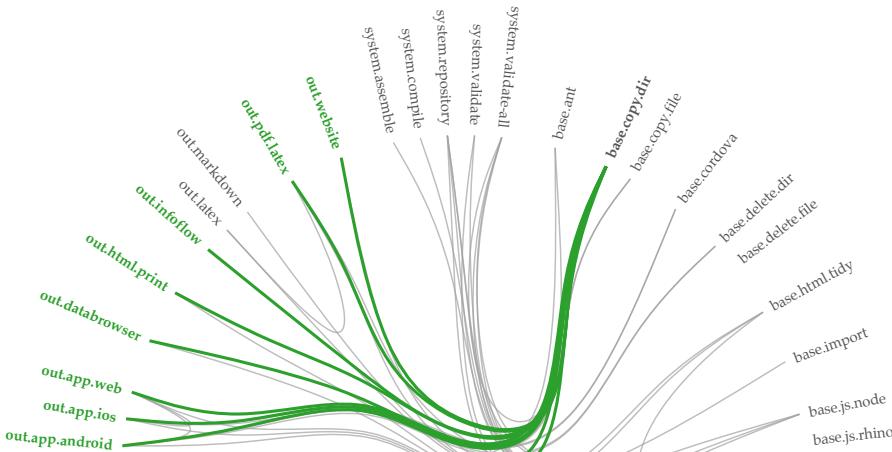


Abb. 26: Abhängigkeiten: Kopieren

5.6.9 Liste der Komponenten

In Tabelle 11 bis Tabelle 15 sind alle 55 in der Umsetzung entstandenen Komponenten sortiert nach Komponententyp aufgeführt.

IDENTIFIER	NAME / BESCHREIBUNG
base.ant	Ant-Datei ausführen
base.copy.dir	Ordner kopieren
base.copy.file	Datei kopieren
base.cordova	Apache Cordova ausführen
base.delete.dir	Ordner löschen
base.delete.file	Datei löschen
base.html.tidy	HTML aufräumen (mit <i>JTidy</i>)
base.import	Datei importieren
base.js.node	JS-Datei in <i>Node.js</i> ausführen
base.js.rhino	JS-Datei in <i>Rhino</i> ausführen
base.latex	L <small>A</small> T <small>E</small> X-Datei ausführen
base.list.dirs	Ordner auflisten (in Ordner)
base.list.files	Dateien auflisten (in Ordner)
base.pdf.extract	Text aus PDF extr. (mit <i>PDFBox</i>)
base.transform	XML transformieren (mit <i>Saxon</i>)
base.wrap.file	Pfad zu Datei in XML abbilden
base.wrap.text	Text in XML abbilden
base.xhtml.5fy	HTML zu HTML5 konvertieren
base.xhtml.structure	HTML(5) strukturieren
base.xhtml.validate	HTML(5) validieren (mit <i>v.Nu</i>)
base.xml.clean	XML bereinigen
base.xml.hash	Prüfsumme für XML erzeugen
base.xml.timestamp	Zeitstempel für XML erzeugen
base.zip.compress	ZIP-Archiv erzeugen
base.zip.extract	ZIP-Archiv entpacken

Tabelle 11: Liste der Basis-Komponenten

IDENTIFIER	NAME / BESCHREIBUNG
system.assemble	Publikationsdef. assemblieren
system.compile	Komponente kompilieren
system.repository	Repository erzeugen
system.validate	Komponente validieren
system.validate-all	Alle Komponenten validieren

Tabelle 12: Liste der System-Komponenten

IDENTIFIER	NAME / BESCHREIBUNG
in.dita	DITA-Dokument lesen
in.excel	<i>Microsoft-Excel</i> -Dokument lesen
in.html	HTML-Dokument lesen
in.infoflow	<i>infoflow</i> -Paket importieren
in.markdown	<i>Markdown</i> -Dokument lesen
in.media	Medien importieren
in.pdf	PDF-Dokument lesen
in.video	Video-Datei importieren
in.word	<i>Microsoft-Word</i> -Datei lesen

Tabelle 13: Liste der Eingabe-Komponenten

IDENTIFIER	NAME / BESCHREIBUNG
mid.index	Dokument-Index erzeugen
mid.merge	Quellen zusammenführen
mid.outline	Struktur exportieren
mid.resolve	Medienreferenzen auflösen
mid.sort	Struktur anpassen
mid.validate	Inhaltsformat validieren

Tabelle 14: Liste der Verarbeitungs-Komponenten

IDENTIFIER	NAME / BESCHREIBUNG
out.app.android	Native <i>Android</i> -App erzeugen
out.app.ios	Native <i>iOS</i> -App
out.app.web	Web-App erzeugen
out.databrowser	Ausgabe für <i>arvato CIM-Portal</i>
out.html.print	HTML-Dokument erzeugen
out.infoflow	<i>infoflow</i> -Paket exportieren
out.latex	<i>LATEX</i> -Dokument erzeugen
out.markdown	<i>Markdown</i> -Dokument erzeugen
out.pdf.latex	PDF-Dokument erzeugen (via <i>LATEX</i>)
out.website	HTML5-Website erzeugen

Tabelle 15: Liste der Ausgabe-Komponenten

5.7 PUBLIKATIONSDEFINITIONEN

5.7.1 Identifizierung

Die Manifest-Datei einer Publikationsdefinition ist nach ihrer UUID (mit Präfix p-) benannt und trägt die Endung .pub (z.B. p-550e8400-e29b-11d4-a716-446655440000.pub). Im Betrieb mit einem Web-Client bleibt die UUID für den Benutzer verborgen.

Es ist möglich eigene IDs zu verwenden (die z.B. aus einem anderen System stammen) oder auf gekürzte UUIDs zurückzugreifen. So werden in der Umsetzung der Publikationsplattform z.B. nur die ersten 8 Stellen einer generierten UUID verwendet (p-dd6ea511). Dies ergibt eine ausreichend große Zahl an möglichen Kombinationen ($2,82 \times 10^{12}$) bei übersichtlicheren IDs.

Um Kollisionen zu vermeiden, prüft die API bestehende Publikationen auf dem Publikationsserver und gibt ggf. einen Fehler zurück.

5.7.2 Aufbau

5.7.2.1 Manifest

In Abb. 27 wird der schematische Aufbau einer Manifest-Datei spezifiziert. Die Syntax orientiert sich am Aufbau von Komponenten.

Die genaue Bedeutung aller Elemente, ihr Vorkommen und gültige Werte, werden in der Spezifikation (siehe Kapitel A.4) festgehalten. Im folgenden werden die Bestandteile der Publikationsdefinition kurz beschrieben:

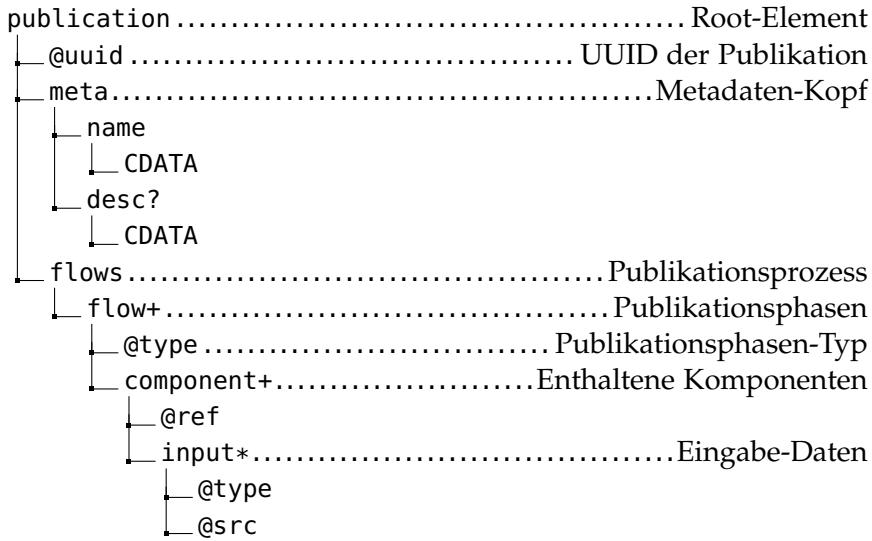
XML-Schema

Abb. 27: Schema einer Publikationsdefinition

METADATEN

Im Element `meta` werden die Metadaten zusammengefasst. Dies sind der Name (bzw. Titel) der Publikation im Element `name` sowie eine optionale Beschreibung im Element `desc`.

PUBLIKATIONSPHASEN

Die in der Konzeption beschriebenen Phasen einer Publikation werden im Element `flows` zusammengefasst. Dieses enthält bis zu drei `flow`-Elemente, die jeweils einer der Phasen entsprechen und mit dem Attribut `type` zugeordnet werden (mögliche Werte: `in`, `mid`, `out`).

KOMPONENTEN

Komponenten können als Elemente `component` in jeder der drei `flow`-Typen über das Attribut `ref` referenziert werden. Handelt es sich um Eingabekomponenten oder um Komponenten, die Daten aus Nebenflüssen verarbeiten, können `input`-Elemente auf entsprechende Dateien verweisen.

5.7.3 Beispiel

In Code 14 wird das Manifest einer Beispiel-Publikationsdefinition p-73e22ca9.pub in der Grundform dargestellt. Die Publikationsstrecke verarbeitet eine Word-Datei (docx) und ein Video (mp4) mit zugehörigen Metadaten. Die Quellen werden zusammengeführt, umstrukturiert und mit externen Medien verknüpft. Der Content wird als PDF-Dokument und als Website ausgegeben.

```

1 <!-- p-73e22ca9.pub -->
2 <publication uuid="p-73e22ca9">
3   <meta>
4     <name>Beispiel</name>
5     <desc>Dies ist eine Beispiel-Publikation, die mit dem Web-
       Client erstellt wurde.</desc>
6   </meta>
7   <flows>
8     <flow type="in">
9       <component ref="in.word">
10      <input type="main" src="/server/publications/p-1/sources/
           document_1434632962260_.docx"/>
11    </component>
12    <component ref="in.video">
13      <input type="main" src="/server/publications/p-1/sources/
           video_1434632968805_.mp4"/>
14      <input type="meta" src="/server/publications/p-1/sources/
           video-mp4_1434632972899_.meta"/>
15    </component>
16  </flow>
17  <flow type="mid">
18    <component ref="mid.merge"/>
19    <component ref="mid.sort">
20      <input type="meta" src="/server/publications/p-1/content/
           outline.xml"/>
21    </component>
22    <component ref="mid.resolve"/>
23  </flow>
24  <flow type="out">
25    <component ref="out.pdf.latex"/>
26    <component ref="out.website"/>
27  </flow>
28 </flows>
29 </publication>
```

Code 14: XML-Notation einer Publikationsdefinition

5.8 REPOSITORY

5.8.1 Aufbau

Das *Repository* wird in einer Datei mit dem Namen `repository.xml` gespeichert. Der Aufbau dieser Datei entspricht nach Vorgaben des Konzepts dem folgenden Schema:

Die im Repository enthaltenen Informationen werden in fast allen Fällen direkt aus der jeweiligen Komponente entnommen. Die Eigenschaften `type` und `requires` werden bei der Erzeugung von `system.repository` dynamisch erzeugt.

XML-Schema

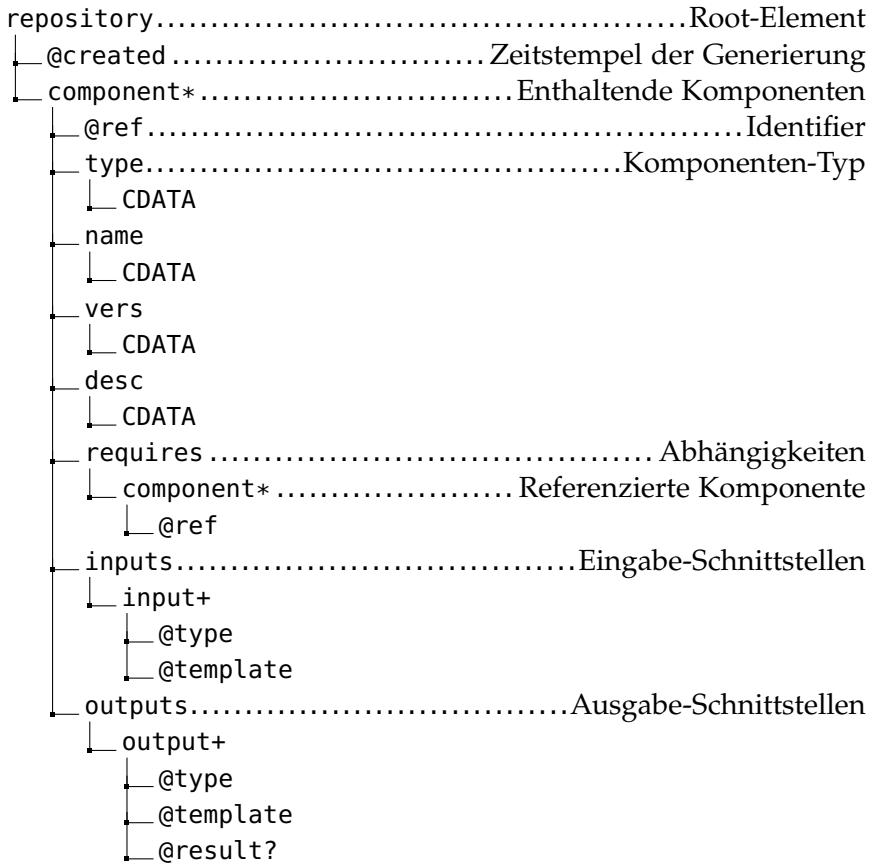


Abb. 28: Schema des Repository

5.8.2 Erzeugung

Das Repository wird von der Systemkomponente `system.repository` erzeugt. Diese prüft alle im `components`-Ordner liegenden Komponenten entsprechend der im Konzept beschriebenen Vorvalidierung (siehe Abschnitt 4.6.3).

5.9 ARCHITEKTUR

Die Architektur der Publikationsplattform entspricht dem im Konzept vorgegebenen Schichtenmodell aus Publikationsserver, Web-API und Web-Client. Diese werden technologisch umgesetzt durch *Apache Ant* (Publikationsserver) und *Node.js* (Web-API) auf der Serverseite und durch einen Browser (Web-Client) auf der Client-Seite. Durch diesen Aufbau ist eine klare Modularisierung der Plattform möglich, die eine Ausrichtung auf *Cloud Computing*³⁸ erlaubt.

³⁸ Unter *Cloud Computing* versteht man das Speichern und Ausführen von Dateien und Programmen auf entfernten und je nach Bedarf skalierten IT-Infrastrukturen.

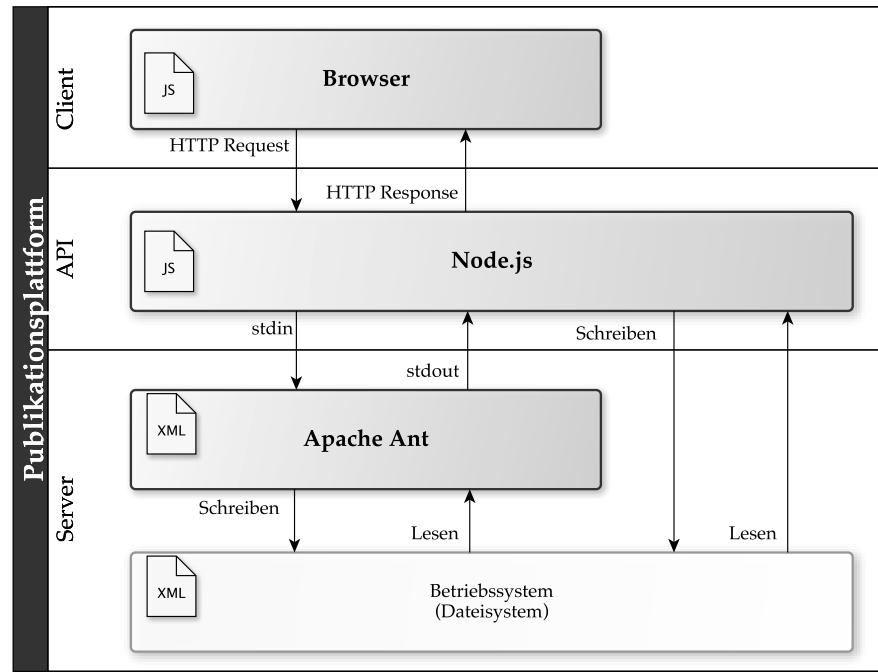


Abb. 29: Umsetzung: Architektur und Technologien

5.9.1 Technologien

Die Umsetzung der Schichten und ihrer Funktionalität unterscheidet sich sowohl in den eingesetzten Technologien als auch im verwendeten *Framework* (siehe Tabelle 16).

	PUB.-SERVER	WEB-API	WEB-CLIENT
Sprache	XML	JS	JS / HTML5
Datenformat	XML	XML / JSON	JSON
Framework	Apache Ant	Node.js	AngularJS

Tabelle 16: Plattform: Eingesetzte Technologien

Wie in Tabelle 16 zu erkennen, wird bei den Web-Schichten der Plattform im Gegensatz zum Publikationsserver auf Technologien aus dem JS-Umfeld gesetzt. Dieses Vorgehen erlaubt schlanke Datenübertragungen, hohe Effizienz und moderne Funktionen.

5.10 WEB-API

5.10.1 Aufgaben

Hauptaufgabe der Web-API ist, die Kommunikation zwischen einem Client und dem Publikationsserver über eine webbasierte Schnittstelle (die API) zu koordinieren (siehe auch Abb. 11).

Umgesetzt wird das mit Hilfe einer Serveranwendung, die Client-Anfragen verarbeiten und Antworten senden kann (`main.js`), einer Schnittstelle zum Dateisystem des Publikationsservers, die Daten zwischen verschiedenen Formaten übersetzen kann (`bridge.js`), sowie einer Schnittstelle zum Publikationsserver selbst, die Prozesse starten und stoppen kann (`connector.js`).

5.10.2 Technologie

Die Web-API wurde als *Node.js*-Anwendung umgesetzt. Bei *Node.js* handelt es sich um eine Umgebung, die es ermöglicht, JavaScript auch außerhalb des Browsers auszuführen (vgl. HUGHES-CROUCHER/WILSON 2012:1). Dies ist besonders im Hinblick darauf interessant, dass die bisher auf clientseitige Entwicklung fokussierte Skriptsprache JavaScript nun auch auf der Serverseite eingesetzt werden kann. *Node.js* selbst basiert wiederum auf der hochperformanten *V8-JS-Runtime*, die im Browser *Google Chrome* zum Einsatz kommt und den in JS geschriebenen Code direkt in Maschinencode übersetzen kann.

Innerhalb des *Node.js*-Ökosystems ist die Wiederverwendung und Verteilung von Code als *Packages* (oder auch *Modules*) sehr verbreitet. Diese Pakete werden als Komponenten³⁹ einer *Node.js*-Anwendung eingesetzt und durch zentrale Repositories verteilt oder lokal verknüpft. Das Einbinden von Paketen erfolgt über `require()`-Aufrufe.

Bei der Umsetzung wurden sowohl externe Pakete (z.B. für Grundfunktionalitäten des Server) als auch interne Pakete (zur besseren Gliederung der Applikation) verwendet. Eine Liste der externen Pakete findet sich in Tabelle 21.

5.10.3 Klassifizierung der Schnittstelle

Die clientseitige Schnittstelle der Web-API ist eine auf HTTP basierte REST-API. Das Datenaustauschformat im Nachrichtenkörper (*payload*) ist JSON. Im *REST Maturity Model* ist die Schnittstelle als *Level 2* einzutragen (vgl. FOWLER 2010).

³⁹ Der *Node.js*-Paketmechanismus war eines der Vorbilder bei der Entwicklung dieser Umsetzung

Mit Representational State Transfer (REST) wird ein Programmierparadigma bezeichnet, das für die zustandslose Kommunikation zwischen Maschinen verwendet wird (vgl. FIELDING 2000:76ff). Dabei werden Ressourcen als URLs adressiert und Operationen als Methoden des darunterliegenden Protokolls abgebildet. REST wird in der Regel mit dem Hypertext Transfer Protocol (HTTP) umgesetzt, dem Standard-Protokoll zur Übertragung von Webseiten (vgl. RICHARDSON/RUBY 2008:13).

Die von HTTP vorgegebenen Anfrage-Methoden bilden dabei die möglichen⁴⁰ Operationen der Schnittstelle ab: GET (eine Ressource anfordern, *nullipotent*), PUT (eine Ressource anlegen, *idempotent*) und POST (neue Subressource oder andere Operationen, *nicht idempotent*). Bei den Methoden PUT und POST kann die Anfrage im Nachrichtenkörper Daten enthalten (reiner Text, JSON, XML, etc.), bei GET können Daten nur in der Uniform Resource Locator (URL) kodiert werden (als Parameter). Die Antworten des Servers enthalten neben Statuscodes (z.B. 200 für OK) fast immer Daten (z.B. HTML-Dateien bei einem Webserver).

```

1 GET /infoflow/status HTTP/1.1
2 Host: www.example.com
3 [...]
```

Code 15: REST via HTTP - Anfrage

```

1 HTTP/1.1 200 OK
2 Content-Length:142
3 Content-Type:application/json; charset=utf-8
4 [...]
5 {
6   "type": "success",
7   "msg": "Infoflow API available",
8   "val": {
9     "version": "0.9.2",
10    "description": "Public Demo (AWS)",
11    "uptime": 173845.929,
12    "client": false
13  }
14 }
```

Code 16: REST via HTTP - Antwort

REST wurde aufgrund des einfachen und schlanken Aufbaus, der Konformität mit bereits etablierten Standards (siehe Abschnitt 3.2.1.3) und der einfachen Zugänglichkeit für Clients gewählt. Durch die Nutzung von HTTP wurde sichergestellt, dass die vorhandene Infrastruktur einfach weitergenutzt werden kann und eine größtmögliche Kompatibilität zu Systemen besteht.

⁴⁰ Aufgeführt werden nur Methoden, die in der Umsetzung eingesetzt werden.

5.10.4 Datenaustausch

Der Datenaustausch der Web-API erfolgt in zwei Richtungen: zum Client und zum Server.

Publikationen und Komponenten werden auf Serverseite in XML erfasst und verarbeitet. Die Steuerung⁴¹ des Publikationssystems erfolgt über Programmaufrufe, die über ein Ant-Interface spezifiziert (siehe Abschnitt 5.5.6) werden.

Zum Datenaustausch mit Clients wird auf JSON zurückgegriffen. Die native Unterstützung durch JS, der geringe *Overhead* des Formats sowie die breite Anwendung in Web-Anwendungen machen JSON zu einem idealen Austauschformat für webbasierte Kommunikation.

Publikationsdefinitionen, Strukturdefinitionen des Inhalts und das Repository haben dadurch neben ihrer XML- auch eine JSON-Definition. Diese entspricht in ihrer Struktur weitgehend den XML-Varianten, wird jedoch an einigen Stellen angepasst, um die Erstellung und Verarbeitung durch Clients zu vereinfachen (siehe Beispiel Abb. 30). Es handelt sich deshalb nicht um 1:1-Umwandlungen wie z.B. mit JXON (vgl. Mozilla Developer Network 2015b).

Darstellung: JSON-Schema

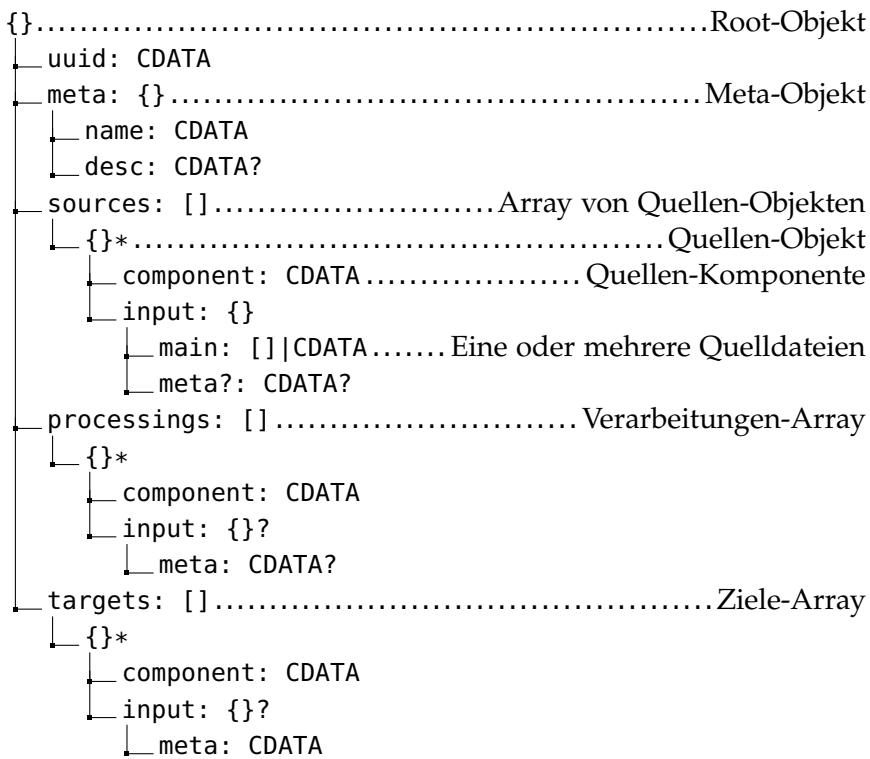


Abb. 30: JSON-Repräsentation einer Publikationsdefinition

⁴¹ Mit Steuerung ist hier das Starten und Stoppen von Prozessen mit entsprechenden Parametern sowie das Überwachen des Prozess-Status gemeint

Die Übersetzung zwischen diesen beiden Repräsentationsarten ist Aufgabe der API und in einem eigenen Modul gekapselt (**bridge**). In der Umsetzung werden diese Umwandlungen nahezu in Echtzeit vorgenommen (siehe Tabelle 17).

TYP	XML → JSON	JSON → XML
Repository	×	-
Strukturdefinition	×	×
Publikationsdefinition	-	×

Tabelle 17: API: Umwandlung XML - JSON

5.10.5 API-Kurzreferenz

In Tabelle 18 wird eine kurze Übersicht über die Funktionen gegeben, die über die REST-API gesteuert werden können (Platzhalter haben einen vorangestellten Doppelpunkt, z.B. :id).

Eine vollständige Auflistung mit Erklärungen findet sich in der API-Referenz (siehe Kapitel A.4).

Die angegebenen Endpunkte beziehen sich immer auf einen *Host* (IP oder URL) mit der API-Root-Adresse (im Standard: /infoflow). Der Aufruf erfolgt mit einer HTTP-Methode.

Beispiel für einen entfernten API-Server:

```
[GET] http://52.28.111.217/infoflow/status
```

5.10.6 Publishing mit der Web-API

Das Anlegen und Ausführen eines vollständigen Publikationsprojektes mit Hilfe der Web-API ist in einzelne Schritte bzw. Aufrufe unterteilt (Kurzreferenz Tabelle 18).

Diese unterteilen sich nach dem REST-Modell in Aktionen, die eine Änderung auf dem Server herbeiführen, wie das Schreiben einer Datei oder das Anstoßen eines Publikationsprozesses (POST / PUT, nachfolgend [P]) und Aktionen, die eine Datei vom Server lesen ohne eine Änderung hervorzurufen (GET, nachfolgend [G]).

Eine beispielhafte Abfolge von API-Abrufen, wie sie der Web-Client der Umsetzung tätigt ist in Abb. 31 dargestellt.

HTTP	ENDPUNKT	BESCHREIBUNG
GET	/status	Liefert Statusinformationen des Servers als JSON-Objekt
GET	/repository	Liefert das Repository des Pub.Servers als JSON-Objekt
POST	/repository	Aktualisiert das XML-Repository des Pub.-Servers
PUT	/pub/:id	Legt Publikationsprojekt :id auf dem Pub.-Server an
POST	/pub/:id/save	Speichert JSON-Publikationsdef. für :id als XML-Datei
POST	/pub/:id/source	Nimmt Quelldatei für :id an und speichert sie auf Pub.-Server
GET	/pub/:id/outline	Liefert Strukturdef. für :id als JSON-Objekt
POST	/pub/:id/outline	Speichert Strukturdef. für :id als XML-Datei
POST	/pub/:id/publish	Publiziert Publikationsdef. :id auf dem Pub.-Server
GET	/pub/:id/preview/:t	Liefert die Vorschaudatei für Ausgabe-Komponente :t
GET	/pub/:id/download/:t	Liefert ein Zip-Archiv für Ausgabe-Komponente :t
GET	/get/:path	Liefert die Datei unter :path ab publications

Tabelle 18: API: Kurzreferenz

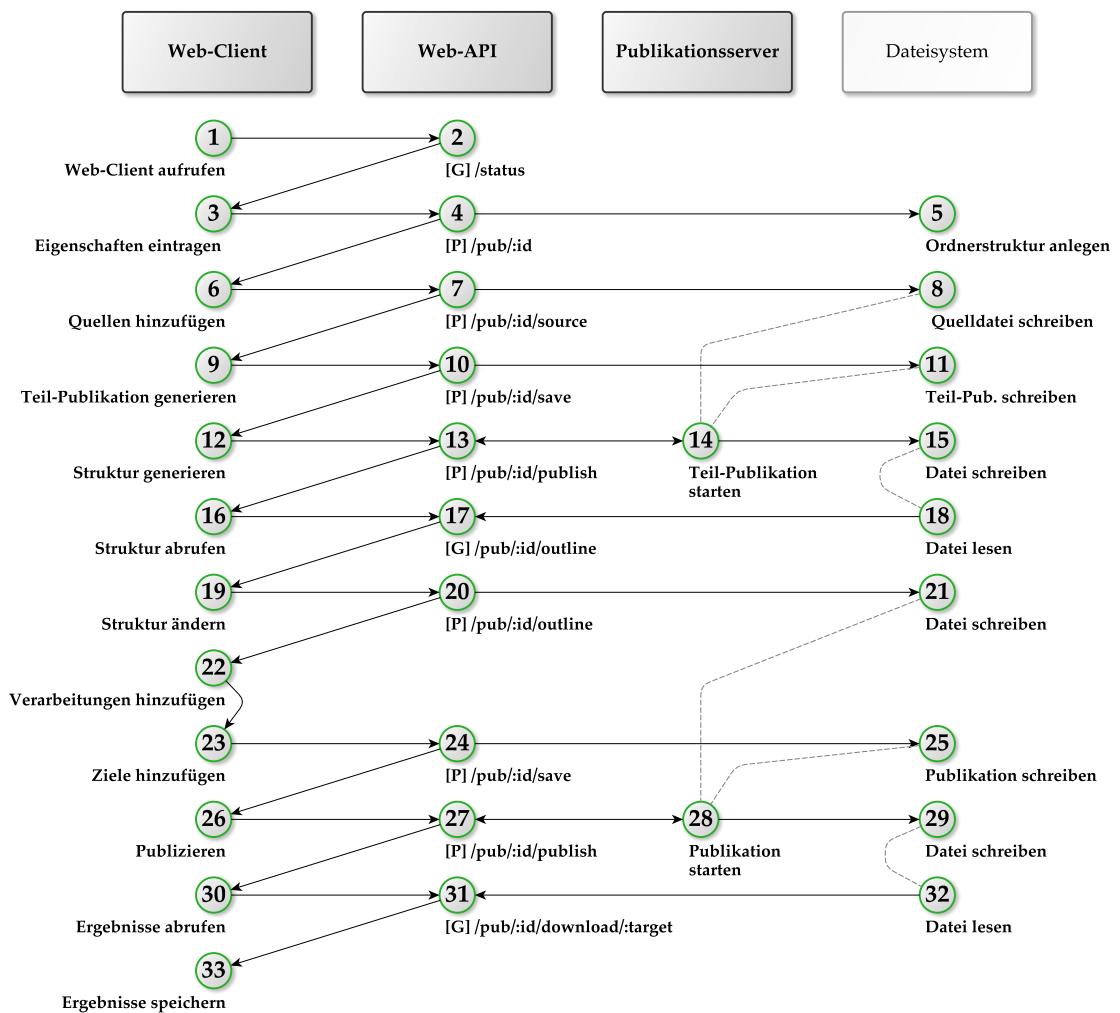


Abb. 31: Web-API: Publikationsprojekt mit Strukturänderung

5.11 WEB-CLIENT

5.11.1 Aufgaben

Der umgesetzte Web-Client hat die Aufgabe, das Erstellen und Ausführen eines Publikationsprozesses so einfach wie möglich für den Benutzer zu machen. Das Erstellen einfacher Publikationsstrecken soll bei Vorhandensein entsprechender Komponenten auch für unerfahrene Nutzer möglich sein und die Anwendung den Standards moderner *Web-Apps* entsprechen.

Diese Ziele werden durch die folgenden Ansätze erreicht:

BENUTZERFÜHRUNG

Der Benutzer wird bei der Definition einer Publikation durch den Client unterstützt. Durch eine logische Gliederung der notwendigen Schritte beim Erstellen einer Publikationsdefinition wird der Nutzer durch den Prozess geführt. Fortschrittsanzeigen, kurze Hilfetexte (*Hints*) sowie Rückmeldungen zu getätigten Aktionen (Erfolgs- bzw. Fehlermeldungen) geben dem Nutzer *Feedback* zur aktuellen Erstellung.

VALIDIERUNG

Die Nutzereingaben sollen vom Client validiert werden bevor sie an die Web-API gesendet werden. Dadurch werden mögliche Fehlerquellen frühzeitig ausgeschlossen. So findet eine Überprüfung der ausgewählten Quelldateien für Komponenten statt (anhand des in der Komponente definierten Templates) sowie eine Validierung der erstellten Publikationsphasen (Komponenten- und Datei-Referenzierung).

VERSTECKTE INTELLIGENZ

Die Komplexität des Publikationsprozesses soll dem Nutzer verborgen bleiben. Der Web-Client fügt deshalb, entsprechend der Benutzereinstellungen, automatisch notwendige Komponenten in die Publikationsdefinition ein. Dies ist vor allem bei Verarbeitungskomponenten wichtig.

Beispiel: Im Standard wird dem Nutzer die Möglichkeit gegeben, den Content umzustrukturieren. Dies wird durch eine im Hintergrund ausgeführte Teilpublikation realisiert. Für den Nutzer ist jedoch nur eine Publikation zu sehen.

5.11.2 Technologie

Der Web-Client ist vollständig webbasiert und in jedem modernen Browser⁴² lauffähig. Voraussetzung für den Betrieb des Clients ist die Erreichbarkeit einer Web-API (lokal oder extern).

Die Oberfläche des Clients besteht aus einer einzelnen HTML-Seite (*Single Page Application*), die als *AngularJS*-Template dient. Der programmatische Teil des Clients ist eine JavaScript-Anwendung, die auf dem Framework *AngularJS* basiert. Die Gestaltung der Oberfläche kann mit Cascading Style Sheets (CSS) angepasst werden.

Bei *AngularJS* handelt es sich um ein von *Google, Inc.* entwickeltes Framework für die Entwicklung von webbasierten *Single Page Applications* (vgl. STEYER/SOFTIC 2015:33). Besonderer Fokus liegt dabei auf bidirektonaler Datenbindung, Validierung, Templating sowie dem Einsatz einer *Model-View-Controller-Architektur (MVC)* (vgl. GREEN/SESHADRI 2013:7ff).

Der Web-Client kommuniziert via HTTP-Requests mit der Web-API und verarbeitet deren Antworten bzw. leitet die nächste Aktion ein. Eine beispielhafte Interaktion zwischen Client und API ist in Abb. 31 dargestellt.

Eine ausführliche Auflistung der verwendeten Module und Bibliotheken aus externen Quellen findet sich in Tabelle 22.

5.11.3 Publishing mit dem Web-Client

Der Web-Client ist optimiert für einen Publikationsprozess mit folgenden Merkmalen:

- Eine oder mehrere Eingabe-Komponenten mit jeweils einer oder mehreren Quelldateien (Hauptfluss) und ggf. meta-Nebenfluss-Dateien
- Strukturänderung des Contents (Export (`mid.outline`), Verändern und Anwenden der Struktur (`mid.sort`))
- Zusammenführen der Quellen (`mid.merge`), Auflösen von Medienerreferenzen (`mid.resolve`)
- Eine oder mehrere Verarbeitungs-Komponenten ggf. mit meta-Nebenfluss-Dateien
- Eine oder mehrere Ausgabe-Komponenten
- Vorschau oder Download (ZIP) der Publikationsergebnisse

⁴² Getestet wurden die derzeit aktuellen Versionen von Google Chrome (43), Mozilla Firefox (39), Microsoft InternetExplorer (11) und Apple Safari (8)

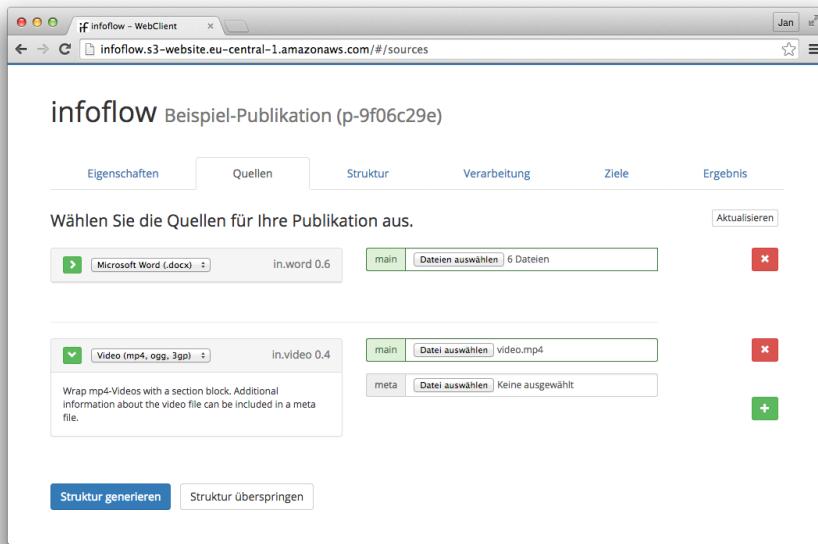


Abb. 32: Web-Client: Quellen hinzufügen

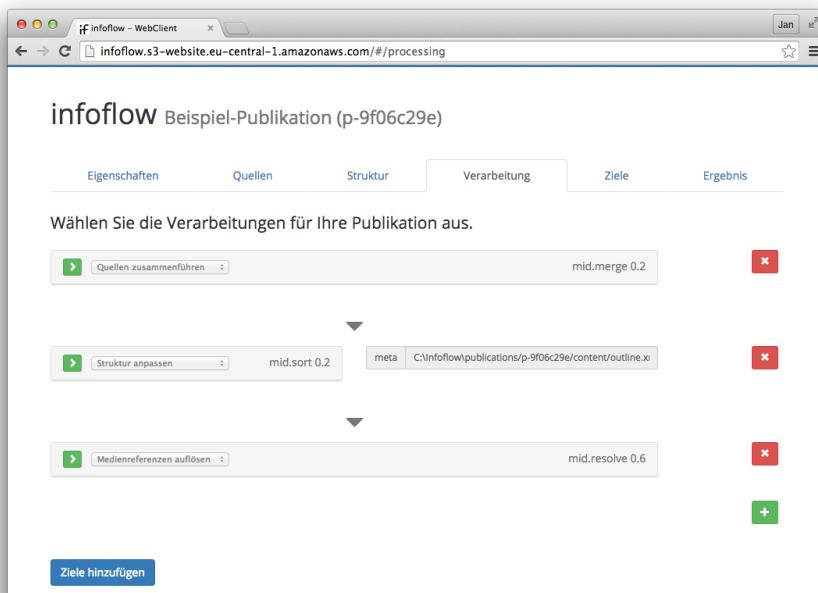


Abb. 33: Web-Client: Verarbeitungen hinzufügen

6

USE CASES

In diesem Kapitel wird das in Kapitel 4 definierte Konzept und die in Kapitel 5 entworfene Implementierung anhand typischer Anwendungsfälle (Use Cases) verifiziert.

6.1 ÜBERBLICK

In diesem Abschnitt soll die Funktion der Publikationsplattform anhand von drei möglichen Anwendungsszenarien (sog. *Use Cases*) gezeigt werden. Durch die Abdeckung der *Use Cases* ist eine Verifizierung der in Kapitel 3 gestellten Anforderungen möglich. Die Szenarien wurden so gewählt, dass ein möglich breites Spektrum von Komponenten gezeigt werden kann.

6.2 DITA NACH PDF MIT STRUKTURÄNDERUNG

6.2.1 Beschreibung

Dieser Use Case entspricht einer typischen Publikationsstrecke im TD-Umfeld. Mehrere DITA-Topics (concepts, tasks, etc) liegen in Dateiform vor und sollen zu einem PDF-Dokument zusammengeführt werden, das einer bestimmten Dokumentstruktur entspricht. Innerhalb der DITA-Dateien sind Grafiken referenziert, die separat als Bilddateien im Dateisystem verfügbar.

6.2.2 Konfiguration

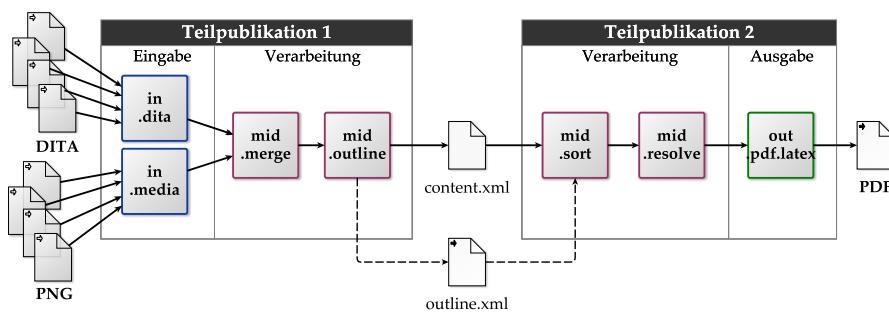


Abb. 34: Use Case 1: Konfiguration

In einer ersten Teilpublikation werden zunächst die Quelldaten importiert und zusammengeführt, um anschließend eine Gesamtstruktur des Contents zu erzeugen. Die Struktur des Dokuments kann nun im Web-Client bearbeitet werden. In der zweiten Teilpublikation wird der Content umstrukturiert, Medienreferenzen aufgelöst und das Ziel erzeugt.

6.2.3 Ablauf

EINGABE

Die DITA-Dateien werden einzeln mit der Komponente `in.dita` in die interne Informationsstruktur überführt. Dazu wird das XML zunächst von möglichen Problemstellen bereinigt (nicht vorhandene Dokumenttyp-Definitionen, nicht definierte Entities, unbekannte Namespaces oder korrupte Kodierungen). Danach werden die DITA-Elemente in die Pendants der internen Informationsstruktur überführt. Abschließend wird das Dokument hierarchisch strukturiert.

Die zugehörigen Bilder werden mit der Komponente `in.media` importiert.

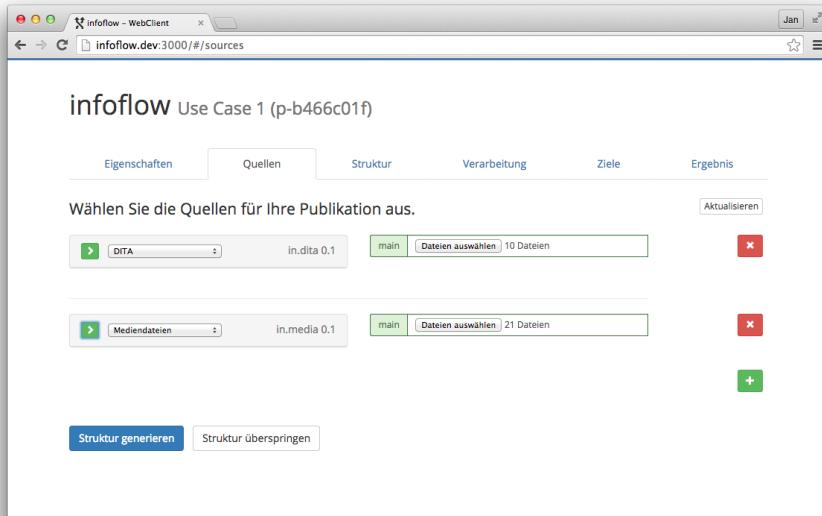


Abb. 35: Use Case 1: Quellen

VERARBEITUNG

Nach abgeschlossener Eingabe-Phase der Publikation werden die Content-Teile mit Hilfe von `mid.merge` zusammengeführt. Dazu erstellt die Komponente eine Liste aller Teil-Dateien und wandelt diese in XML um. Ein XSL-Stylesheet arbeitet diese Liste ab und kombiniert die Inhalte. Abschließend wird der Gesamt-

content mit einer Prüfsumme versehen und entsprechend der Vorgaben an die interne Informationsstruktur aufbereitet.

Nach erfolgreicher Zusammenführung wird durch die Komponente `mid.outline` eine sog. Strukturdefinition als Nebenfluss-Ergebnis erstellt (`outline.xml`). Diese basiert auf der Schachtelung der strukturbildenden `section`-Elemente, von denen jedes mit einer ID versehen wird. Der ursprüngliche Content wird unverändert ausgegeben (`content.xml`). Die erste Teilpublikation ist abgeschlossen.

Die Struktur wird im Web-Client durch den Nutzer bearbeitet.

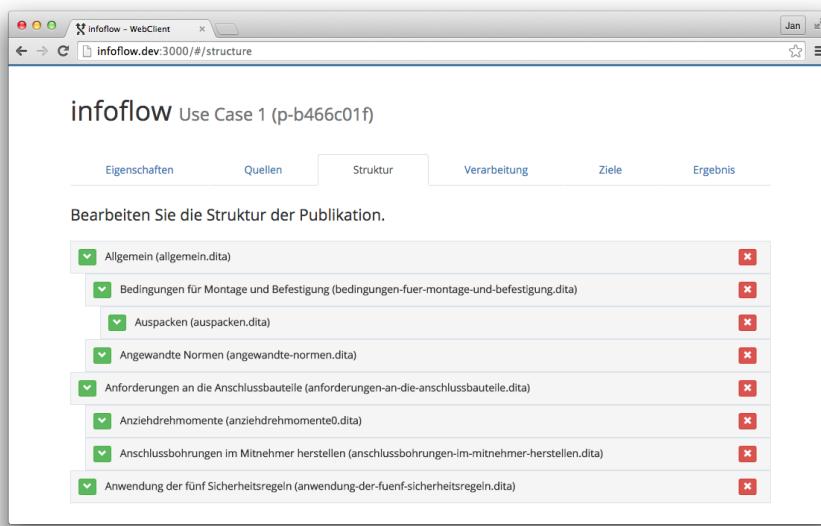


Abb. 36: Use Case 1: Struktur bearbeiten

Die zweite Teilpublikation setzt an dem bereits zusammengeführten Content an. Dieser wird nun auf Basis der Strukturdefinition mit der Komponenten `mid.sort` umstrukturiert. Anhand einer Prüfsumme wird verglichen, ob es sich auch um den Content handelt, für den die Strukturdefinition bestimmt ist. Liegt keine Strukturdefinition vor, wird der Content unverändert weitergegeben.

In einem weiteren Schritt werden durch die Verarbeitungskomponente `mid.resolve` Medienreferenzen aufgelöst. Dazu wird als erstes der Content auf Referenzen untersucht. Diese werden auf ihre Art untersucht (absolut, relativ, URL, etc) und in einer separaten Datei mit einer Content-weit eindeutigen ID aufgelistet (`resources.xml`). Gleichzeitig werden die Referenzen im Content entsprechend angepasst. In einem weiteren Schritt wird auf Basis von `resources.xml` ein Ant-Skript generiert und ausgeführt, das die Dateien entsprechend ihrer Art kopiert bzw. herunterlädt und in das im Content angegebene Ziel bewegt.

AUSGABE

Die Ausgabe in ein PDF-Dokument wird durch die Komponente `out.pdf.latex` durchgeführt. Diese generiert zunächst ein L^AT_EX-Dokument mit Hilfe der Komponente `out.latex`. Dieses wird dann mit Hilfe einer pdfL^AT_EX-Installation in ein PDF-Dokument umgewandelt.

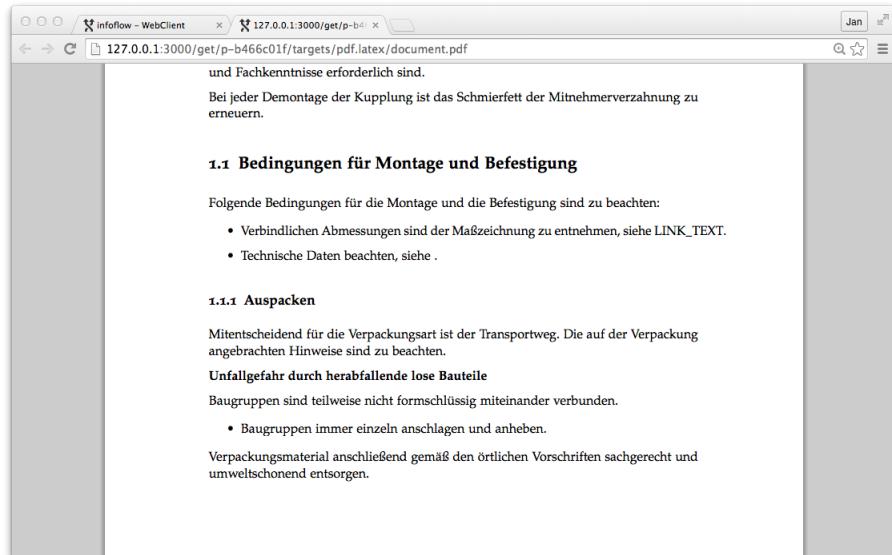


Abb. 37: Use Case 1: Ergebnis-PDF

6.3 PDF NACH HTML MIT INDEX-ERSTELLUNG

6.3.1 Beschreibung

Der Use Case deckt zwei Herausforderungen ab, die bei Publishing mit Sekundärcontent auftreten: Die Struktur der Quelldaten muss aufgrund des Formats (z.B. PDF) rekonstruiert werden und ein verlinkter Index mit Schlagworten soll automatisch generiert werden. Dafür muss der Content während der Publikation mit Struktur und Informationen angereichert werden.

Als Publikationsziel dient ein HTML-Dokument, das z.B. für Software-Dokumentation online verfügbar gemacht werden oder als Grundlage für eine Print-Ausgabe dienen kann.

6.3.2 Konfiguration

Das PDF-Dokument wird zunächst in die interne Informationsstruktur überführt. Im Verarbeitungsteil der Publikation wird der Content indexiert und mit Metadaten angereichert, um anschließend als HTML-Dokument mit Index ausgegeben zu werden.

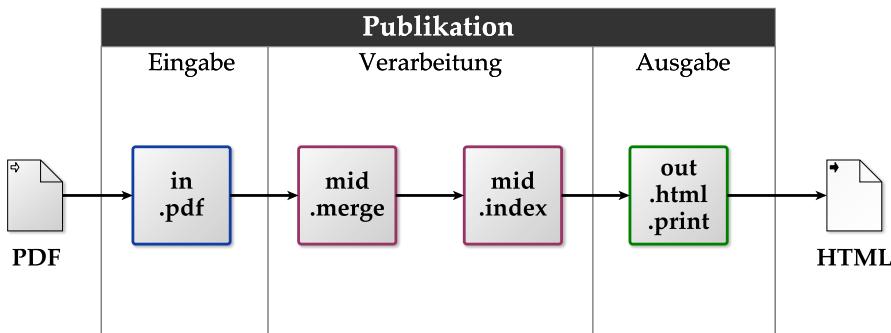


Abb. 38: Use Case 2: Konfiguration

6.3.3 Ablauf

EINGABE

Die PDF-Dateien werden einzeln mit der Komponente `in.pdf` in die interne Informationsstruktur überführt. Dazu wird in einem ersten Schritt mit Hilfe der Software *PDFBox* (vgl. APACHE SOFTWARE FOUNDATION 2015a) der Text des Dokuments als HTML⁴³ extrahiert, bereinigt (z.B. leere Absätze, Inhaltsverzeichnis) und mit Hilfe von *jTidy* (vgl. JTIDY 2012) nach XHTML überführt. Anschließend wird anhand der Überschriften-Nummerierung eine Hierarchie abgebildet (1 → h1, 1.1 → h2, etc) und strukturiert.

VERARBEITUNG

Nach der Überführung⁴⁴ des Contents durchsucht die Verarbeitungskomponente `mid.index` den Content nach Schlagworten, lemmatisiert diese und schreibt die Ergebnisse in einen meta-Tag (Dokument-Schlagworte) und in `data-index-Attribute` an den jeweiligen `section-Elementen`.

Die Indexierung des Dokuments wird durch ein JS-Programm durchgeführt. Die verwendeten Algorithmen stammen zum Teil aus einer früheren Studienarbeit des Autors (vgl. OEVERMANN 2014a).

Zunächst werden alle großgeschriebenen Wörter gesammelt (vermutete Substantive). Diese werden bereinigt (Satzzeichen, Sonderzeichen, Großschreibung am Satzanfang) und gefiltert (häufige Wörter, Wörter mit Sonderzeichen).

Anschließend werden die Wörter lemmatisiert (in ihre grammatischen Grundform gebracht). Dies wird zunächst über den Abgleich mit einer Lemmatisierungsliste (NABER 2013) versucht. Bei Neologismen oder technischen Benennungen, die nicht in

⁴³ Die HTML-Ausgabe von *PDFBox* gibt lediglich Absätze aus. Andere Strukturinformationen gehen verloren.

⁴⁴ Auch wenn es sich hierbei nur um eine einzelne Eingabe handelt, wird `mid.merge` dazu verwendet, den Content in die Verarbeitungsphase zu überführen.

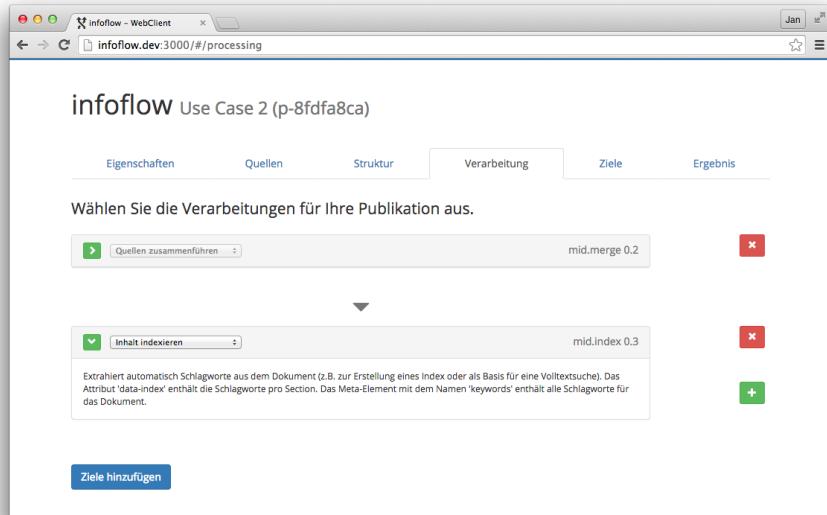


Abb. 39: Use Case 2: Verarbeitungen

der Liste enthalten sind, werden algorithmische Ansätze angewandt (Entfernung der *s-Flexion*, Plural bei Substantivierungen mit den Wortbildungsmorphemen *-keiten* oder *-ungen*, etc).

Die Wortlisten für flektierte und lemmatisierte Formen werden in jeweils einer Textdatei gespeichert.

Das *Markup* des Dokuments mit Schlagwörtern wird durch ein XSL-Stylesheet vorgenommen. Die Liste der lemmatisierten Wortformen wird zunächst in ein entsprechendes `meta`-Element des Dokuments geschrieben. Für die einzelnen `section`-Elemente wird jeweils der Text nach dem gleichen Muster wie zuvor extrahiert und bereinigt. Anschließend werden alle gefundenen Wörter mit der flektierten Wortliste abgeglichen und bei einem Treffer die entsprechende lemmatisierte Wortform (aus der anderen Liste) in das Attribut `data-index` übernommen.

AUSGABE

In der letzten Phase der Publikation wird die interne Informationsstruktur in HTML übernommen und durch `out.html.print` mit CSS-Regeln angereichert. Anschließend wird an das Dokument ein interaktiver Index angehängt. Dieser wird auf Basis der durch `mid.index` gewonnenen Informationen als ein eigener Prozessschritt innerhalb von `out.html.print` generiert: Die Schlagworte des Gesamtdokuments werden alphabetisch aufgelistet und nach Anfangsbuchstabe gruppiert. Für jedes Schlagwort wird das Vorkommen in `section`-Indexierungen geprüft und falls vorhanden per Hyperlink an die entsprechende Stelle mit in den Index mit aufgenommen. Durch eingebettetes JS wird der Index interaktiv (auf-/zuklappbar).

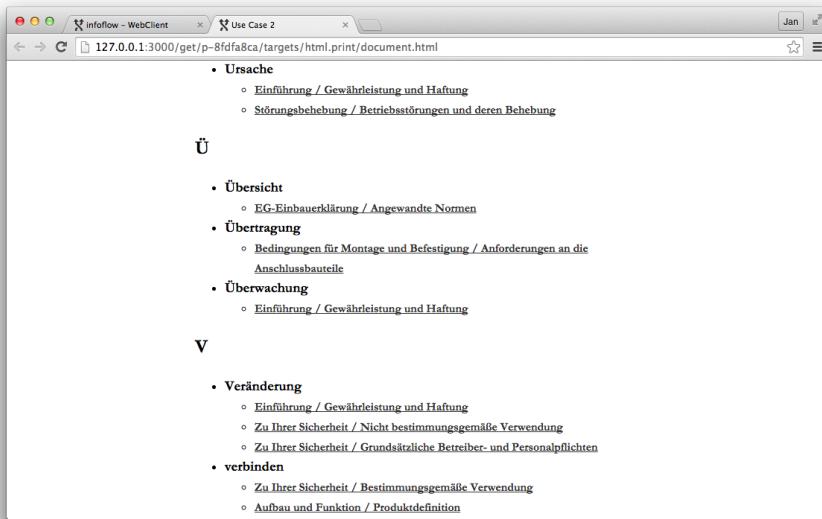


Abb. 40: Use Case 2: Ergebnis-HTML mit Index

6.4 WORD-DOKUMENTE NACH ANDROID-APP

6.4.1 Beschreibung

Dieser Use Case stellt eine neu entstandene Möglichkeit vor, die durch die Flexibilität der Plattform entstanden ist.

Als Eingabe dienen „normale“ Word-Dokumente (Dateiendung: docx), die teilstrukturiert angelegt wurden (mit Überschriften-Hierarchien, Zeichenformatvorlagen, etc.). Als Ziel dient eine native Android-App, die auf Smartphones oder Tablets installiert werden kann.

Der für den Benutzer einfach erscheinende Publikationsprozess wird im Hintergrund von der Plattform auf zahlreiche Komponenten verteilt, die einen komplexen Prozess steuern.

6.4.2 Konfiguration

Das Word-Dokument wird zunächst in die interne Informationsstruktur überführt. Im Verarbeitungsteil der Publikation wird der Content für die Ausgabe vorbereitet und am Ende als Android Application Package (APK)⁴⁵ ausgegeben.

45 APK-Dateien können auf den meisten Android-Geräten direkt von einer Website installiert werden.

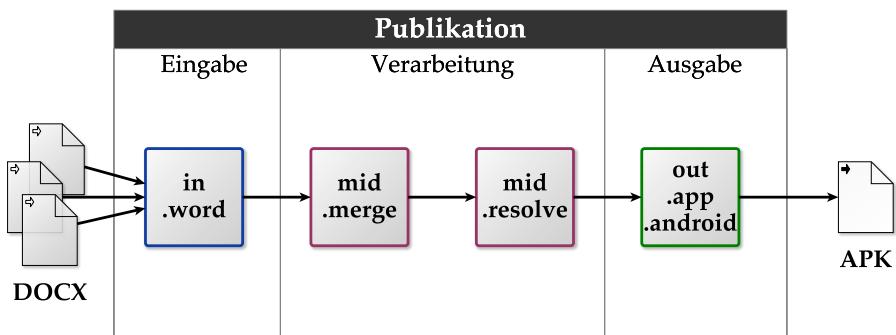


Abb. 41: Use Case 3: Konfiguration

6.4.3 Ablauf

EINGABE

Die Word-Dokumente werden zunächst einzeln von der Komponente **in.word** verarbeitet. Diese dekomprimiert das docx-Format und kopiert alle Medien des Pakets für eine spätere Referenzierung. Anschließend werden die Inhalte mit Hilfe einer angepassten Version von DAY 2014 in ein HTML-Dokument überführt. Der Inhalt des Dokument wird im nächsten Schritt übersetzt (CSS-Klassen in semantische Elemente), aufgeräumt (leere Absätze, leere Pfade, etc.) und in die interne Informationsstruktur überführt.

VERARBEITUNG

Die Verarbeitung entspricht der einer Standard-Publikation. Der Eingabe-Content wird zusammengeführt und Medienreferenzen aufgelöst. Die bei der Eingabe kopierten Bilddateien aus dem docx-Paket werden eindeutig bezeichnet und neu verknüpft.

AUSGABE

Die Ausgabekomponente **out.app.android** wandelt die interne Informationsstruktur in eine APK-Datei um. Der Prozess lässt sich in zwei große Blöcke einteilen, die ineinander greifen: Das Generieren einer Web-App mit der Komponente **out.app.web** sowie das Erstellen und Kompilieren eines *Cordova*-Projekts.

In einem ersten Schritt wird ein Template für Cordova-Projekte angelegt und eine config-Datei mit den Informationen zum Publikationsprojekt generiert. Anschließend wird aus dem Content mit der referenzierten Ausgabekomponente **out.app.web** eine Web-App generiert (dabei werden z.B. die sections des Contents in einzelne Dateien aufgesplittet).

Die Ergebnisse werden mit dem Projekt-Template zusammengeführt und eine spezielle Cordova-JS-Datei in der Web-App refe-

renziert. Abschließend wird mit Hilfe von *Apache Cordova* eine APK kompiliert, die mit einem Debug-Schlüssel signiert ist.⁴⁶

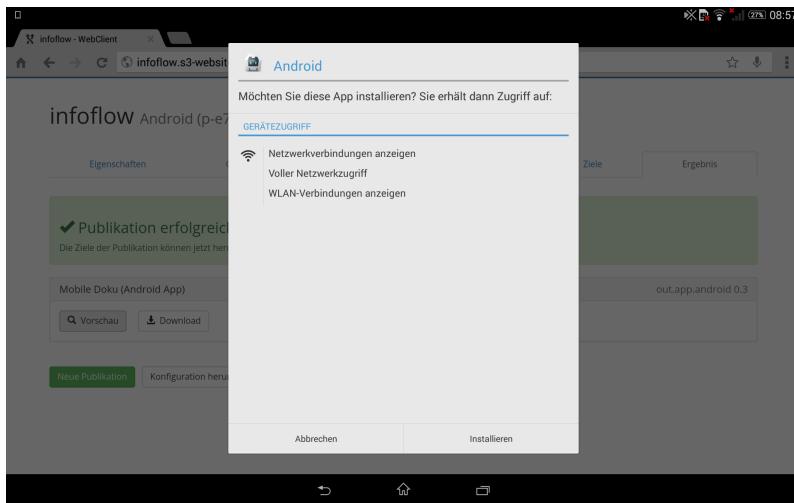


Abb. 42: Use Case 3: App von Web-Client installieren

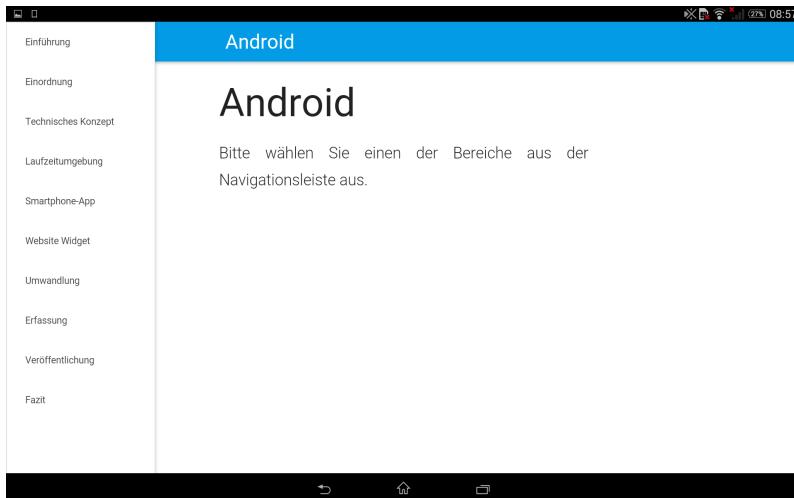


Abb. 43: Use Case 3: Ergebnis-App auf Android Tablet

6.5 BEWERTUNG

Anhand der beschriebenen Use Cases konnte gezeigt werden, dass sich typische Anwendungsfälle aus dem Bereich der TD über die Plattform abbilden lassen. Auch wenn es sich hierbei nur um drei ausgewählte Szenarien gehandelt hat, so können dennoch Rückschlüsse auf andere Komponenten-Zusammenstellungen gezogen werden. Die Verifizierung durch Use Cases kann damit als erfolgreich eingestuft werden.

⁴⁶ Dadurch ist möglich, die App sofort auf seinem Gerät zu installieren. Eine Verteilung der App über einen *App Store* wird allerdings unterbunden.

7

FAZIT UND AUSBLICK

7.1 ZUSAMMENFASSUNG

Als Ergebnis der Arbeit konnte eine komponentenbasierte Publikationsplattform erfolgreich prototypisch implementiert werden. Nach eingehender Analyse von komponentenbasierten Konzepten in Kapitel 2 und einer Formulierung praxisorientierter Anforderungen in Kapitel 3 wurde ein umfassendes und technisch neutrales Konzept in Kapitel 4 spezifiziert. Dieses wurde anschließend in Kapitel 5 mit modernen Technologien implementiert und in Kapitel 6 mit Hilfe von Use Cases verifiziert.

Rückblickend konnten aussagekräftige Antworten auf die in der Einleitung formulierten Fragen gefunden werden:

- Durch die erfolgreiche Umsetzung des Konzepts konnte gezeigt werden, dass eine komponentenbasierte Architektur sich sehr gut für eine flexible Publikationsplattform eignet, die die geforderten Anforderungen erfüllen kann. Schon mit den im Rahmen der Arbeit entstandenen Beispiel-Komponenten lassen sich flexibel eine Vielzahl von Publikationsszenarien abbilden. Die Entwicklung der Basis- und Systemkomponenten ist zwar sehr zeitaufwändig, doch nach der initialen Entwicklung dienen diese als Basis für weitere Entwicklungen und können effizient wiederverwendet werden.
- Die Verwendung von HTML5 als interne Informationsstruktur ermöglicht eine gute Kombination von Inhalten unterschiedlicher Herkünfte. Durch die Erweiterbarkeit des Formats können detaillierte Metadaten hinterlegt werden, bei schwach oder nicht strukturierten Daten kann aber auch auf eine rein darstellungsorientierte Auszeichnung zurückgegriffen werden.
- Die erfolgreiche Abbildung der Use Cases mit dem Prototypen der Publikationsplattform hat gezeigt, dass sich typische Anwendungsfälle für Publikationsstrecken mit wenig Aufwand umsetzen lassen. Der Web-Client dient dabei als zentrale Benutzerschnittstelle, die eine einfache Konfiguration und Zusammenstellung der Publikationseigenschaften ermöglicht.
- Grenzen des Konzepts liegen in der dokumentbasierten Publikation des Contents. Dadurch ist es nur über Umwege möglich, eine *On-Demand*-Auslieferung einzelner Module zu realisieren (entsprechend der Funktion von Content-Delivery-Portalen).

- Insbesondere in Kombination mit der Web-API sind alternative Anwendungsszenarien entstanden. Durch die mobile Ansicht des Web-Client lassen sich, z.B. von einem Android-Smartphone aus, Word-Dateien in eine native App publizieren, die direkt aus dem Web-Client heraus auf dem Smartphone installiert werden kann. Eine weitere Möglichkeit ist ein publikationsbasierteres Portal oder eine Website, die über die API bereitgestellt wird (via preview-Funktion) und über eine Publikationsprozess mit der gleichen UUID einfach aktualisiert werden kann. Ebenso möglich ist die Bereitstellung von Content, der für Content-Delivery-Portal aufbereitet wurde.

7.2 FAZIT

Im Rahmen der Arbeit konnten erfolgreich ein innovatives Konzept, basierend auf bewährten Methoden und neuen Lösungsansätzen entworfen werden, das im Anschluss mit Hilfe verschiedener Technologien umgesetzt und getestet wurde.

Als anspruchsvoll haben sich Anwendungstests mit realen Daten aus Kundenprojekten herausgestellt, da diese oft nicht den idealen Umständen entsprachen (korrupte XML-Dateien, abnorme PDF-Dateien, etc.). Durch diese *echten* Probleme mussten aber auch Lösungen gefunden werden, diese zu umgehen. Aus diesem Grund existieren nun z.B. auch Komponenten, die bestimmte Dateitypen reparieren und aufbereiten können.

Viel Zeit der Arbeit wurde auch damit verbracht, die Schemata für Komponenten und Publikationsdefinitionen zu modellieren und gute Benennungen für die einzelnen Elemente des Publikationssystems zu finden. Die vorliegende Version ist das Ergebnis vieler Iterationen und wird nach dem Einsatz im Alltag auch weiterhin Gegenstand von Veränderungen sein.

Das Konzept hat sich zum Ziel gesetzt, die Entwicklung von neuen Publikationskanälen zu beschleunigen, indem Komponenten gekapselt und basierend auf anderen Komponenten in kurzer Zeit entwickelt werden können. Ein Hinweis darauf, dass dieses Ziel erreicht wurde, ist der Umstand, dass statt der geplanten 4 beispielhaften Ein- und Ausgaben nun 19 Komponenten für Quellen und Ziele entwickelt werden konnten.

Abschließend können die Ergebnisse der Arbeit als positiv eingestuft werden. Es wurde gezeigt, dass es möglich ist, die Konzepte CMP und CDP zu einer Gesamtlösung zu vereinen, welche die Vorteile beider Ansätze nutzt und kombiniert. Dabei wurde bewusst auf bestehenden Methoden und Konzepte aufgebaut, aber auch neue Technologien und Architekturen eingesetzt.

7.3 AUSBLICK

Die Praxistauglichkeit der prototypischen Implementierung muss nun mit realen Daten auf Flexibilität und Zuverlässigkeit getestet werden. Die Einsatzgebiete reichen dabei von klassischen Publikationsstrecken hin zu neuen Anwendungsszenarien.

Für die Zukunft ist eine Weiterentwicklung von Konzept und Umsetzung vorgesehen. Folgenden Bereiche sollen dabei verstärkt betrachtet werden:

- Entwicklung, Integration und Qualitätssicherung von weiteren Komponenten. Dieser Punkt ist insbesondere dann interessant, wenn die Komponenten nicht vom Verfasser dieser Arbeit entwickelt werden.
- Eine technische (Neu-)Implementierung des Publikationsservers in JavaScript (auf Basis von Node.js), um eine bessere Performance und größere Funktionsvielfalt zu erreichen.
- Die Erweiterung der Umstrukturierungsfunktion in Hinblick auf eine mögliche Automatisierung des Prozesses. So soll es möglich sein, Strukturdefinitionen auch auf veränderten Content anwenden zu können. Dafür muss ein neuer Vergabemechanismus von IDs für Content-Sektionen entwickelt werden.

A

ANHANG

A.1 ONLINE-DEMO

Ein Prototyp der Umsetzung als *Cloud-Anwendung* ist als öffentliche Demo unter folgender URL zu erreichen:⁴⁷

<http://infoflow.s3-website.eu-central-1.amazonaws.com>

Der in Abb. 44 abgebildete Screenshots zeigt die Online-Demo beim Aufruf der oben genannten URL.



Abb. 44: Demo: Eigenschaften eintragen

A.2 LOKALE INSTALLATION

Es ist möglich, die Publikationsplattform lokal zu betreiben. Dazu können die Schichten entsprechend der in Abschnitt 4.1.3 genannten Kombinationen installiert werden. Prinzipiell ist eine Installation auf allen gängigen Betriebssystemen möglich (*Windows, OS X, Linux*).

⁴⁷ Die Demo ist unter dieser Adresse für die Zeit der Korrektur der Arbeit und darüber hinaus, jedoch maximal bis zum 19.05.2016 zu erreichen.

A.2.1 *Publikationsserver*

1. Java SDK und Apache Ant installieren, falls nicht vorhanden.
 → <http://www.oracle.com/technetwork/java/javase>
 → <https://ant.apache.org>.
2. Den Ordner infoflow/server vom Datenträger kopieren.
3. Nach Bedarf weitere Installationen vornehmen (siehe Tabelle 20), Komponenten hinzufügen oder entfernen.
4. Vor der ersten Publikation das Repository erzeugen mit:

```
$ ./infoflow run -Dcomponent=system.repository
```

A.2.2 *Web-API*

1. Node.js installieren, falls nicht vorhanden.
 → <https://nodejs.org>
2. Den Ordner infoflow/web-api vom Datenträger kopieren.
3. Nach Bedarf die Konfiguration in config/config.js anpassen.
4. Application Programming Interface (API)-Server starten mit:

```
$ ./infoflow start
```

A.2.3 *Web-Client*

1. Modernen Browser installieren, falls nicht vorhanden.
 → <https://www.google.com/chrome> oder
 → <https://www.mozilla.org/firefox>
2. Den Ordner infoflow/web-client vom Datenträger kopieren.
3. Nach Bedarf die Konfiguration in config/config.js anpassen.
4. API starten und `http://localhost:3000` im Browser öffnen

A.3 VERWENDETE PROGRAMMTEILE

Im folgenden Abschnitt werden alle Programmteile der technischen Umsetzung kenntlich gemacht, die nicht direkt vom Autor stammen. Dies schließt alle Frameworks (JS/CSS) und Module (*Node/Angular*) sowie Kommandozeilen-Tools (z.B. *Saxon*) ein. Bei allen⁴⁸ Programmteilen handelt es sich um Open-Source-Software (erkennbar an der jeweiligen Lizenz). Code-Fragmente und Code, der als Basis oder Inspiration gedient hat, wird direkt an der entsprechenden Stelle im Quellcode kenntlich gemacht.

⁴⁸ Mit Ausnahme von *Apple Xcode*, das für die Generierung von nativen iOS-Apps notwendig ist.

NAME	BESCHREIBUNG / URL	LIZENZ
<i>Ant</i>	Build-Tool (<i>systemweit</i>) https://ant.apache.org	Apache
<i>Bootstrap</i>	CSS-Framework (<i>out.website</i>) https://github.com/twbs/bootstrap	MIT
<i>docx2html.xsl</i>	XSL-Stylesheets (<i>in.word</i>) https://github.com/rnathanday/docx2html.xsl	MIT
<i>Hartija</i>	CSS-Framework (<i>out.html.print</i>) https://github.com/vladocar/Hartija--CSS-Print-Framework	MIT
<i>jTidy</i>	HTML-Bibliothek (<i>base.html.tidy</i>) http://jtidy.sourceforge.net	wie Zlib
<i>Materialize</i>	CSS-Framework (<i>out.app.web</i>) https://github.com/Dogfalo/materialize	MIT
<i>Nu Validator</i>	HTML-Validator (<i>mid.validate</i>) https://github.com/validator/validator	Mozilla
<i>PDFBox</i>	PDF-Bibliothek (<i>base.pdf.extract</i>) https://pdfbox.apache.org	Apache
<i>Saxon HE</i>	XSL-Transformator (<i>base.transform</i>) http://saxon.sourceforge.net	Mozilla

Tabelle 19: Verwendete Software: Publikationssystem

NAME	BESCHREIBUNG / URL	LIZENZ
<i>Android SDK</i>	Entwicklungswerkzeuge (<i>out.app.android</i>) https://developer.android.com/sdk	Apache
<i>Cordova</i>	Cross-Platform-Framework (<i>base.cordova</i>) https://cordova.apache.org	Apache
<i>MiKTeX</i>	L <small>A</small> T <small>E</small> X-Distribution (<i>base.latex</i>) http://miktex.org/	LPPL
<i>Node.js</i>	JS-Laufzeitumgebung (<i>base.js.node</i>) https://nodejs.org	MIT
<i>Xcode</i>	Entwicklungswerkzeuge (<i>out.app.ios</i>) https://developer.apple.com/xcode	proprietär

Tabelle 20: Verwendete Software: Optionale Installationen

NAME	BESCHREIBUNG / URL	LIZENZ
<i>Node.js</i>	JavaScript-Laufzeitumgebung https://nodejs.org	MIT
<i>Express</i>	Web Application Framework http://expressjs.com	MIT
<i>archiver</i>	Zip-Archivierung für <i>Node.js</i> https://github.com/archiverjs/node-archiver	MIT
<i>which</i>	<i>which</i> -Implementierung für <i>Node.js</i> https://github.com/isaacs/node-which	ISCL
<i>xmldom</i>	DOM-Implementierung für <i>Node.js</i> https://github.com/jindw/xmldom	MIT
<i>body-parser</i>	JSON-Body-Parsing für <i>Express</i> https://github.com/expressjs/body-parser	MIT
<i>cors</i>	CORS-Implementierung für <i>Express</i> https://github.com/expressjs/cors	MIT
<i>morgan</i>	Request Logging für <i>Express</i> https://github.com/expressjs/morgan	MIT
<i>multer</i>	Multipart-Body-Parsing für <i>Express</i> https://github.com/expressjs/multer	MIT

Tabelle 21: Verwendete Software: Web-API

NAME	BESCHREIBUNG / URL	LIZENZ
<i>AngularJS</i>	JS-Framework für Web Applications https://angularjs.org/	MIT
<i>Bootstrap</i>	CSS-Framework http://getbootstrap.com/	MIT
<i>Angular UI Tree</i>	Baum-Darstellung für <i>AngularJS</i> https://github.com/angular-ui-tree/angular-ui-tree	MIT

Tabelle 22: Verwendete Software: Web-Client

A.4 ZUSATZDOKUMENTE

Im Rahmen der Thesis sind weitere Dokumente entstanden, die aus Platzgründen nicht in den gedruckten Teil der Arbeit mit aufgenommen werden können. Diese sind als Datei auf dem beiliegenden Datenträger gespeichert.

NAME	DATEI
Referenz API	_Thesis_Zusatz/infoflow_api.html
Spezifikation	_Thesis_Zusatz/infoflow_spec.html
Publikationssystem	

Tabelle 23: Zusatzdokumente

A.5 STATISTIK ZUR UMSETZUNG

Um einen Eindruck über den im Rahmen der technischen Umsetzung entstandenen Code zu geben, wird im folgenden eine Übersicht über die entstandenen Programmteile in *LoC (Lines of Code)* gegeben.

SPRACHE	DATEIEN	KOMMENTARE	REINER CODE
XSLT	36	455	2842
XML	58	52	2317
JavaScript	11	524	1154
HTML	3	27	612
CSS	2	34	218
Gesamt	110	1092	7143

Tabelle 24: Statistik Umsetzung: Entstandener Code

LITERATURVERZEICHNIS

- AIIM (2012): What is Enterprise Content Management (ECM)?
<<http://www.aiim.org/What-is-ECM-Enterprise-Content-Management>>
[Stand: k.A. Letzter Zugriff: 2015-03-28]
- ANDERSEN, Rebekka (2011): „Component Content Management - Shaping the Discourse through Innovation, Diffusion, Research and Reciprocity.“ In: Technical Communication Quarterly 20, Nr. 4, 384–411
- APACHE SOFTWARE FOUNDATION (2009): Apache Xang - Introduction.
<<http://xml.apache.org/xang/overview.html>>
[Stand: Dezember 2009. Letzter Zugriff: 2015-03-09]
- APACHE SOFTWARE FOUNDATION (2011): Welcome to Apache Forrest.
<<http://forrest.apache.org>>
[Stand: Februar 2011. Letzter Zugriff: 2015-03-09]
- APACHE SOFTWARE FOUNDATION (2013): Apache Cocoon. <<http://cocoon.apache.org>>
[Stand: März 2013. Letzter Zugriff: 2015-03-09]
- APACHE SOFTWARE FOUNDATION (2014a): Apache Ant 1.9.4 Manual.
<<http://ant.apache.org/manual/>>
[Stand: Mai 2014. Letzter Zugriff: 2015-03-04]
- APACHE SOFTWARE FOUNDATION (2014b): The Apache Ant Project.
<<http://ant.apache.org>>
[Stand: Mai 2014. Letzter Zugriff: 2015-03-04]
- APACHE SOFTWARE FOUNDATION (2015a): Apache PDFBox - A Java PDF Library. <<https://pdfbox.apache.org>>
[Stand: k.A. Letzter Zugriff: 2015-03-13]
- APACHE SOFTWARE FOUNDATION (2015b): Category Index - XML.
<<http://projects.apache.org/indexes/category.html#xml>>
[Stand: März 2015. Letzter Zugriff: 2015-03-09]
- ARVATO CIM (2014): arvato – Experts in Technical Information - Your automotive service provider. <<http://www.arvato-dev.com/media/automotive/files/assets/common/downloads/arvato%20Technical%20Information%20Spain.pdf>>
[Stand: April 2014. Letzter Zugriff: 2015-06-15]
- ARVATO CIM (2015): After Sales Portale. <<https://scm.arvato.com/de/solutions/corporate-information-management/after-sales-information-management/after-sales-portale.html>>
[Stand: Juni 2015. Letzter Zugriff: 2015-06-15]

- AWS INC. (2015): Amazon Web Services (AWS). <<http://aws.amazon.com/de/>> [Stand: k.A. Letzter Zugriff: 2015-06-23]
- BECHER, Margit (2004): „Publizieren mit XML - Anwendungsmöglichkeiten des XML-Publishing-Frameworks Apache Cocoon.“ In: Tagungsband 2004 FP 26/2004.
- BHASKAR, Michael (2013): The Content Machine - Towards a Theory of Publishing from the Printing Press to the Digital Network. London : Anthem Press
- BROCKE, Jan vom / SIMONS, Alexander (2013): Enterprise Content Management in Information Systems Research - Foundations, Methods and Cases. Berlin : Springer
- BROWN, Ethan (2014): Web development with Node and Express. Köln : O'Reilly Media
- BUSCHMANN, Frank (1998): Pattern-orientierte Software-Architektur - Ein Pattern-System. London : Pearson
- BUSSLER, Christoph (2003): B2B Integration - Concepts and Architecture. Berlin : Springer
- CLOSS, Sissi (2007): Single Source Publishing - Topicorientierte Strukturierung und DITA. Frankfurt/Main : Entwickler.Press
- CLOSS, Sissi (2014): „Arbeitswelt Technischer Redakteure im Wandel.“ In: Technische Kommunikation und mobile Endgeräte. Lübeck : Schmidt-Römhild, Schriften zur Technischen Kommunikation 19, 103–112
- COATES, Anthony (2002): „Running Multiple XSLT Engines with Ant.“ In: XML.com <<http://www.xml.com/pub/a/2002/12/11/ant-xml.html>> [Stand: Dezember 2002. Letzter Zugriff: 2015-03-25]
- CONTENTFUL GMBH (2015): Content Delivery API. <<https://www.contentful.com/developers/documentation/content-delivery-api/>> [Stand: k.A. Letzter Zugriff: 2015-04-10]
- DAY, Don / PRIESTLEY, Michael / SCHELL, David (2005): „Introduction to the Darwin Information Typing Architecture - Toward portable technical information.“ In: IBM developerWorks, 1–12 <<http://www.ibm.com/developerworks/library/x-dital/x-dital-pdf.pdf>> [Stand: September 2005. Letzter Zugriff: 2015-03-12]
- DAY, Nathan (2014): XSLT 2.0 Stylesheets for Converting DOCX Files to HTML (GitHub). <<https://github.com/rnathanday/docx2html.xsl>> [Stand: Mai 2014. Letzter Zugriff: 2015-03-04]

- DE CESARE, Sergio / LYCETT, Mark / MACREDIE, Robert (2006): Development of Component-Based Information Systems. Armonk : M.E. Sharpe
- DIN EN ISO 9241-210 (2011): Ergonomie der Mensch-System-Interaktion – Teil 210: Prozess zur Gestaltung gebrauchstauglicher interaktiver Systeme.
- DITA OT (2015a): Architecture of the DITA Open Toolkit. <http://www.dita-ot.org/2.0/dev_ref/DITA-OTArchitecture.html> [Stand: k.A. Letzter Zugriff: 2015-09-27]
- DITA OT (2015b): DITA Open Toolkit (Offizielle Website). <<http://www.dita-ot.org>> [Stand: k.A. Letzter Zugriff: 2015-09-27]
- DITA OT (2015c): DITA Open Toolkit (Repository auf GitHub). <<https://github.com/dita-ot/dita-ot>> [Stand: Juni 2015. Letzter Zugriff: 2015-09-28]
- DITA OT (2015d): DITA-OT processing structure. <http://www.dita-ot.org/2.1/dev_ref/processing-structure.html> [Stand: k.A. Letzter Zugriff: 2015-03-12]
- DOCUFY GMBH (2015): Docufy TopicPilot. <<http://topicpilot.de>> [Stand: k.A. Letzter Zugriff: 2015-04-13]
- DREWER, Petra / ZIEGLER, Wolfgang (2011): Technische Dokumentation. Würzburg : Vogel
- DUCHARME, Bob (2003a): „Grouping With XSLT 2.0.“ In: XML.com <<http://www.xml.com/pub/a/2003/11/05/tr.html>> [Stand: November 2003. Letzter Zugriff: 2015-04-01]
- DUCHARME, Bob (2003b): „XSLT 2 and Delimited Lists.“ In: XML.com <<http://www.xml.com/pub/a/2003/05/07/tr.html>> [Stand: Mai 2003. Letzter Zugriff: 2015-03-13]
- DUDEN (2015a): Content (Wörterbucheintrag). <<http://www.duden.de/rechtschreibung/Content>> [Stand: k.A. Letzter Zugriff: 2015-04-08]
- DUDEN (2015b): Komponente (Wörterbucheintrag). <<http://www.duden.de/rechtschreibung/Komponente>> [Stand: k.A. Letzter Zugriff: 2015-03-18]
- DUMKE, Reiner (2013): Software Engineering - Eine Einführung für Informatiker und Ingenieure - Systeme, Erfahrungen, Methoden, Tools. Berlin : Springer
- EASYBROWSE GMBH (2015): EB.Suite. <<http://www.easybrowse.de/software/eb-suite.html>> [Stand: k.A. Letzter Zugriff: 2015-04-13]
- ECMA-262 (2015): ECMAScript Language Specification - 5.1 Edition. <<http://www.ecma-international.org/publications/files/>>

- ECMA-ST/Ecma-262.pdf>
 [Stand: Juni 2015. Letzter Zugriff: 2015-04-13]
- ECMA-357 (2005): ECMAScript for XML (E4X) Specification. <<http://www.ecma-international.org/publications/files/ECMA-ST-WITHDRAWN/Ecma-357.pdf>>
 [Stand: Dezember 2005. Letzter Zugriff: 2015-04-13]
- ECMA-376 (2012): Office Open XML File Formats - 4th edition. <<http://www.ecma-international.org/publications/standards/Ecma-376.htm>>
 [Stand: Dezember 2012. Letzter Zugriff: 2015-06-16]
- ECMA-404 (2013): The JSON Data Interchange Format. <<http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>>
 [Stand: Oktober 2013. Letzter Zugriff: 2015-04-10]
- ERASALA, Naveen / YEN, David C. / RAJKUMAR, T.M. (2003): „Enterprise Application Integration in the electronic commerce world.“ In: Computer Standards & Interfaces 25, Nr. 2, 69–82
- ERL, Thomas (2008): SOA - Studentenausgabe: Entwurfsprinzipien für serviceorientierte Architektur. London : Pearson, Programmer's Choice
- EXPRESS.JS (2015): Express - Fast, unopinionated, minimalist web framework for Node.js. <<http://expressjs.com>>
 [Stand: k.A. Letzter Zugriff: 2015-03-12]
- FENG CHEN et al. (2002): „An Architecture-Based Approach for Component-Oriented Development.“ In: Proceedings of the 26th Annual International Computer Software and Applications Conference. Northbrook : IEEE Computer Society, 450–455
- FIELDING, Roy Thomas (2000): Architectural Styles and the Design of Network-based Software Architectures. Dissertation. University of California, <http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf>
 [Stand: 2000. Letzter Zugriff: 2015-05-28], 2000
- FITZGERALD, Michael (2004): „Create Well-Formed XML with JavaScript (#95).“ In: XML Hacks - 100 Industrial-Strength Tips and Tools. Köln : O'Reilly Media <<http://archive.oreilly.com/pub/h/2127>>
 [Stand: Juli 2004. Letzter Zugriff: 2015-03-04], Juli 2004
- FOWLER, Martin (2004): Inversion of Control Containers and the Dependency Injection pattern. <<http://martinfowler.com/articles/injection.html>>
 [Stand: Januar 2004. Letzter Zugriff: 2015-03-27]

- FOWLER, Martin (2010): Richardson Maturity Model - Steps toward the glory of REST. <<http://martinfowler.com/articles/richardsonMaturityModel.html>> [Stand: März 2010. Letzter Zugriff: 2015-05-29]
- FULLER, Jim / FULLER, James (2010): Apache Ant Recipes for Web Developers. Campbell : FastPencil
- GAGARINOV, Alexei / IVERSON, Mark (2005): Transforming Word Documents into the XSL-FO Format. <<https://msdn.microsoft.com/en-us/library/aa203691.aspx>> [Stand: Februar 2005. Letzter Zugriff: 2015-02-25]
- GARLAN, David / MONROE, Robert T. / WILE, David (2000): „Acme: Architectural Description of Component-Based Systems.“ In: Foundations of component-based systems 68, 47–68
- GOOGLE DEVELOPERS (2015): Chrome V8. <<https://developers.google.com/v8/>> [Stand: k.A. Letzter Zugriff: 2015-06-23]
- GOOGLE INC. (2015): AngularJS - Superheroic JavaScript MVW Framework. <<https://angularjs.org>> [Stand: k.A. Letzter Zugriff: 2015-03-18]
- GREEN, Brad / SESHADRI, Shyam (2013): AngularJS. Köln : O'Reilly Media
- GROBMEIER, Christian (2007): Apache Forrest. Einführung in das Publishing Framework. Auflage: 1., Aufl. Auflage. München : Open Source Press
- GRÜNWIED, Gertrud (2014): „Die Online-Hilfe wird mobil.“ In: technische kommunikation, Nr. 2, 31–34
- HATCHER, Erik (2003): Java development with Ant. London : Manning
- HAUSER, Tobias (2010): XML Standards. 2. Auflage. Frankfurt/Main : Entwickler.Press
- HAVERBEKE, Marijn (2014): Eloquent JavaScript. Second Edition Auflage. San Francisco : No Starch Press
- HEIDECK, Daniel / STEURER, Stephan (2013): Realisieren von zielgruppengerechten Portalanwendungen auf Basis von Content-Management-Daten.
- HENNIG, Jörg / TJARKS-SOBHANI, Marita (Hrsg.) (2014): Technische Kommunikation und mobile Endgeräte. Stuttgart : tcworld
- HOFFMANN, Dirk W. (2014): Grundlagen der Technischen Informatik. 4. Auflage. München : Carl Hanser
- HUGHES-CROUCHER, Tom / WILSON, Mike (2012): Einführung in Node.js -Skalierbarer, serverseitiger Code in JavaScript. 1. Auflage. Köln : O'Reilly

- IANA (2015): Media Types. <<https://www.iana.org/assignments/media-types/media-types.xhtml>>
 [Stand: März 2015. Letzter Zugriff: 2015-03-09]
- IDC (2014): Executive Summary - Data Growth, Business Opportunities, and the IT Imperatives - The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things. <<http://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm>>
 [Stand: April 2014. Letzter Zugriff: 2015-06-10]
- IETF RFC 2616 (1999): Hypertext Transfer Protocol - HTTP/1.1. <<http://tools.ietf.org/html/rfc2616>>
 [Stand: Juni 1999. Letzter Zugriff: 2015-04-12]
- IETF RFC 3236 (2002): The 'application/xhtml+xml' Media Type. <<http://tools.ietf.org/html/rfc3236>>
 [Stand: Jnauar 2002. Letzter Zugriff: 2015-05-28]
- IETF RFC 3987 (2005): Internationalized Resource Identifiers (IRIs). <<http://tools.ietf.org/html/rfc3987>>
 [Stand: Januar 2005. Letzter Zugriff: 2015-04-12]
- IETF RFC 4122 (2005): A Universally Unique IDentifier (UUID) URN Namespace. <<http://tools.ietf.org/html/rfc4122>>
 [Stand: Juli 2005. Letzter Zugriff: 2015-05-08]
- IETF RFC 4855 (2007): Media Type Registration of RTP Payload Formats. <<http://tools.ietf.org/html/rfc4855>>
 [Stand: Februar 2007. Letzter Zugriff: 2015-03-27]
- IETF RFC 6838 (2013): Media Type Specifications and Registration Procedures. <<http://tools.ietf.org/html/rfc6838>>
 [Stand: Januar 2013. Letzter Zugriff: 2015-03-27]
- IETF RFC 7158 (2013): The JavaScript Object Notation (JSON) Data Interchange Format. <<http://tools.ietf.org/html/rfc7158>>
 [Stand: März 2013. Letzter Zugriff: 2015-03-13]
- IETF RFC 7303 (2014): XML Media Types. <<http://tools.ietf.org/html/rfc7303>>
 [Stand: Juli 2014. Letzter Zugriff: 2015-04-13]
- INFOFLOW (2015a): Interaktive Visualisierung. <<http://infoflow.s3-website.eu-central-1.amazonaws.com/etc/>>
 [Stand: Juli 2015. Letzter Zugriff: 2015-07-23]
- INFOFLOW (2015b): WebClient (Public Demo). <<http://infoflow.s3-website.eu-central-1.amazonaws.com>>
 [Stand: Juli 2015. Letzter Zugriff: 2015-07-23]
- ISO 8879 (1986): Information processing; Text and office systems; Standard Generalized Markup Language (SGML).

- ISO/IEC 15445 (2000): Information technology; Document description and processing languages; HyperText Markup Language (HTML).
- ISO/IEC 25010 (2011): Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models.
- JOYENT INC. (2015): Node.js. <<https://nodejs.org>> [Stand: k.A. Letzter Zugriff: 2015-03-12]
- jTIDY (2012): jTidy. <<http://jtidy.sourceforge.net>> [Stand: Oktober 2012. Letzter Zugriff: 2015-04-15]
- KAMPFMEYER, Ulrich (2003): Wohin geht die Reise? Zur Bedeutung von Dokumenten-Technologien für Wirtschaft und Gesellschaft. <http://www.project-consult.net/Files/IIR_Wohin%20geht%20die%20Reise.pdf> [Stand: März 2003. Letzter Zugriff: 2015-04-08]
- KIMBER, Eliot (2012): DITA for Practitioners Volume 1 - Architecture and Technology (eBook). Laguna Hills : XML Press
- KOSEK, Jirka (2004): „Using XSLT for Getting Back-of-the-Book Indexes. Washington, D.C. <<http://citesearx.ist.psu.edu/viewdoc/download?doi=10.1.1.131.2069&rep=rep1&type=pdf>> [Stand: k.A. Letzter Zugriff: 2015-05-11], k.A
- KOSTER, Rainer et al. (2001): „Thread Transparency in Information Flow Middleware.“ In: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg. Berlin : Springer, 121–140
- KOZACZYNKI, Voytek / NING, Jim Q (1996): „Component-Based Software Engineering (CBSE).“ In: Proceedings of the International Conference on Software Reuse. Northbrook : IEEE Computer Society, 236–236
- KROLL, Per / KRUCHTEN, Philippe (2003): The Rational Unified Process made easy - A Practitioner's Guide to the RUP. Boston : Addison-Wesley
- KRUCHTEN, Philippe (2004): The Rational Unified Process - An Introduction. 3. Auflage. Boston : Addison-Wesley
- KRÜGER, Manfred (2012): „Alles XML?“ In: HENNIG, Jörg / TJARKS-SOBHANI, Marita (Hrsg.): Technische Kommunikation im Jahr 2041: 20 Zukunftsszenarien. Lübeck : Schmidt-Römhild, 102–109
- KRÜGER, Manfred / ZIEGLER, Wolfgang (2008): „Standards für strukturierte technische Informationen - ein Überblick.“ In: MUTHIG, Jürgen (Hrsg.): Standardisierungsmethoden für die technische Dokumentation. Lübeck : Schmidt-Römhild, tekom Hochschulschriften 16, 11–40

- KÖPKE, Andreas et al. (2004): „Structuring the Information Flow in Component-Based Protocol Implementations for Wireless Sensor Nodes.“ In: Proceedings of the Work-in-Progress Session of the 1st European Workshop on Wireless Sensor Networks (EWSN). Berlin, 41–45
- LANGHAM, Matthew / ZIEGELER, Carsten (2002): Cocoon - Building XML applications. 1. Auflage. Indianapolis : New Riders
- LEICHT, Jerome (2013): „Mobile Produktinformation.“ In: technische kommunikation, Nr. 4 <<http://www.tekom.de/fachartikel/elektronische-informationenprodukte/mobile-produktinformation.html>> [Stand: August 2013. Letzter Zugriff: 2015-03-03]
- LENZ, Evan (2006): „Namespaces in XSLT.“ In: THE XML GUILD (Hrsg.): Advanced XML Applications from the Experts at The XML Guild. 1. Auflage. Boston : Course Technology
- LEW, Michael S. et al. (2006): „Content-based multimedia information retrieval: State of the art and challenges.“ In: ACM Transactions on Multimedia Computing, Communications, and Applications 2, Nr. 1, 1–19
- LINDNER, Wolfgang / NITSCHE, Holger (2013): „CMS in Dienstleistungsunternehmen I.“ In: HENNIG, Jörg / TJARKS-SOBHANI, Marita (Hrsg.): Content Management und Technische Kommunikation. Stuttgart : tcworld, Schriften zur Technischen Kommunikation 18, 71–81
- LOHR, Jürgen / DEPPE, Andreas (2013): Der CMS-Guide - Content Management-Systeme: Erfolgsfaktoren, Geschäftsmodelle, Produktübersicht. Berlin : Springer
- LOUGHAN, Steve (2005): Apache Ant in Anger - Using Apache Ant in a Production Development System. <http://ant.apache.org/ant_in_anger.html> [Stand: März 2005. Letzter Zugriff: 2015-03-04]
- LUCENA, Vicente Ferreira de (2003): Flexible web-based management of components for industrial automation. Aachen : Shaker
- LUCENA JR, Vincente Ferreira de (2002): Flexible Web-based Management of Components for Industrial Automation. Forschungsbericht Institut für Automatisierungs- und Softwaretechnik. Universität Stuttgart
- MACFARLANE, John (2014): Pandoc - A Universal Document Converter. <<http://pandoc.org>> [Stand: k.A. Letzter Zugriff: 2015-03-09]
- MACFARLANE, John (2015): Pandoc - Conversion between markup formats. <<http://hackage.haskell.org/package/pandoc>> [Stand: Juni 2015. Letzter Zugriff: 2015-06-23]

- MEINIKE, Thomas (2010): epubMinFlow - Ein minimaler Workflow zur automatisierten Umsetzung von E-Books im EPUB-Format. <<http://datenverdrahten.de/epubMinFlow/>> [Stand: April 2010. Letzter Zugriff: 2015-03-02]
- MEINIKE, Thomas (2014): Dokumentautomation mit XML am Beispiel einer Banddiskografie. <http://www.iks.hsm-merseburg.de/~meinike/PDF/HIT_2014_Dokumentautomation_mit_XML_Meinike.pdf> [Stand: April 2014. Letzter Zugriff: 2015-03-02]
- MERRIAM-WEBSTER (2015): System (Wörterbucheintrag). <<http://www.merriam-webster.com/dictionary/system>> [Stand: k.A. Letzter Zugriff: 2015-03-18]
- MORRISON, J. Paul (2013): „Flow-Based Programming.“ In: Journal of Application Developers’ News, Nr. 1
- MOZILLA DEVELOPER NETWORK (2015a): HTML element reference. <<https://developer.mozilla.org/en-US/docs/Web/HTML/Element>> [Stand: Januar 2015. Letzter Zugriff: 2015-03-04]
- MOZILLA DEVELOPER NETWORK (2015b): JXON (lossless JavaScript XML Object Notation). <<https://developer.mozilla.org/en-US/docs/JXON>> [Stand: Januar 2015. Letzter Zugriff: 2015-03-04]
- MOZILLA DEVELOPER NETWORK (2015c): Sections and Outlines of an HTML5 Document. <https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Sections_and_Outlines_of_an_HTML5_document> [Stand: März 2015. Letzter Zugriff: 2015-03-04]
- MUNROE, Randall (2011): Standards. <<http://xkcd.com/927/>> [Stand: Juli 2011. Letzter Zugriff: 2015-03-16]
- NABER, Daniel (2013): Deutsches Morphologie-Lexikon (Lemmatisierungs-Datei). <<http://www.danielnaber.de/morphologie/>> [Stand: Dezember 2013. Letzter Zugriff: 2014-09-06]
- NIEKAMP, Rainer (2006): Software Component Architecture - Institute for Scientific Computing TU Braunschweig. <<http://congress.cimne.upc.es/cfsi/frontal/doc/ppt/11.pdf>> [Stand: April 2006. Letzter Zugriff: 2015-03-04]
- OASIS (2010): Darwin Information Typing Architecture (DITA) Version 1.2. <<http://docs.oasis-open.org/dita/v1.2/spec/DITA1.2-spec.html>> [Stand: Dezember 2010. Letzter Zugriff: 2015-03-12]
- ODERMATT, Sven (2009): Integrierte Unternehmenskommunikation - Systemgestützte Umsetzung der informationellen Aufga-

- ben. Dissertation. Justus-Liebig-Universität Gießen. Verlegt bei Gabler, Wiesbaden
- OECHSLE, Rainer (2013): Java-Komponenten - Grundlagen, prototypische Realisierung und Beispiele für Komponentensysteme. München : Carl Hanser
- OEHMIG, Peter (2011): „Wissensarbeiter 2041.“ In: Technische Kommunikation im Jahr 2041: 20 Zukunftsszenarien. Lübeck : Schmidt-Römhild, Schriften zur Technischen Kommunikation 16, 155–163
- OEVERMANN, Jan (2012): Automatisiertes Publishing aus TYPO3. Studienarbeit. Informationssysteme B. Hochschule Karlsruhe - Technik und Wirtschaft
- OEVERMANN, Jan (2013): Dynamische Filterung von benutzerorientierten Inhalten auf Basis eines Content-Management-Systems. Bachelor-Thesis. Studiengang Technische Redaktion. Hochschule Karlsruhe - Technik und Wirtschaft
- OEVERMANN, Jan (2014a): Automatisierte Strukturierung von Benennungslisten. Studienarbeit. Sprach- und Globalisierungsmanagement. Hochschule Karlsruhe - Technik und Wirtschaft
- OEVERMANN, Jan (2014b): „Generierung mobiler Dokumentation nach dem Single-Source-Prinzip.“ In: HENNIG, Jörg / TJARKS-SOBHANI, Marita (Hrsg.): Technische Kommunikation und mobile Endgeräte. Stuttgart : tcworld, Schriften zur Technischen Kommunikation 19, 80–90
- OMG (2012a): Common Object Request Broker Architecture (CORBA) Specification, Version 3.3 - Part 3: CORBA Component Model. <<http://www.omg.org/spec/CORBA/3.3/Components/PDF>> [Stand: November 2012. Letzter Zugriff: 2015-03-26]
- OMG (2012b): CORBA 3.3 - Official Standard from the Object Management Group. <<http://www.omg.org/spec/CORBA/3.3/>> [Stand: November 2012. Letzter Zugriff: 2015-03-26]
- PELSTER, Ulrich (2011): „XML für den passenden Zweck - Offener und standardisierter Umgang mit XML.“ In: technische kommunikation 33. Jg., Nr. 1, 54–57
- PROJEKTMAGAZIN (2015): Not Invented Here (Glossareintrag). <<https://www.projektmagazin.de/glossarterm/not-invented-here>> [Stand: k.A. Letzter Zugriff: 2015-03-06]
- RICHARDSON, Leonard / RUBY, Sam (2008): RESTful Web Services. Köln : O'Reilly Media
- ROCKLEY, Ann (2003): Managing Enterprise Content - A Unified Content Strategy. 1. Auflage. Indianapolis : New Riders

- SAXONICA (2014): Running Saxon XSLT Transformations from Ant.
[<http://www.saxonica.com/html/documentation/using-xsl/xsltfromant.html>](http://www.saxonica.com/html/documentation/using-xsl/xsltfromant.html)
[Stand: k.A. Letzter Zugriff: 2015-03-25]
- SCHEMA GMBH (2015): Schema Content Delivery Suite.
[<http://www.schema.de/de/software/schema-content-delivery-suite.html>](http://www.schema.de/de/software/schema-content-delivery-suite.html)
[Stand: k.A. Letzter Zugriff: 2015-04-13]
- SCHOBER, Martin (2012): „Mobil, mehrsprachig und multimedial - Grundlagen von HTML5.“ In: technische kommunikation, Nr. 6, 36–42
- SIVONEN, Henri / SMITH, Michael (2015): The Nu Html Checker (v.Nu). <https://validator.github.io/validator/>
[Stand: April 2015. Letzter Zugriff: 2015-04-15]
- SKULSCHUS, Marco / WIEDERSTEIN, Marcus (2005): XSLT und XPath für HTML, Text und XML. Bonn : Mitp
- SLYNGSTAD, Odd Petter Nord (2011): Component-Based Software Engineering - Modern Trends, Evolution and Perceived Architectural Risks. Dissertation. Norwegian University of Science and Technology
- STEYER, Manfred / SOFTIC, Vildan (2015): AngularJS - Moderne Webanwendungen und Single Page Applications mit JavaScript. Köln : O'Reilly Media
- STOYE, Michael (2011): Entwicklung eines Datenmodells zur Unterstützung des dateibasierten Datenaustauschs in der Produktentwicklung. Diplomarbeit. Fakultät für Informatik. Otto-von-Guericke-Universität Magdeburg
- STRAUB, Daniela / ZIEGLER, Wolfgang (2008): Effizientes Informationsmanagement durch spezielle Content-Management-Systeme - Praxishilfe und Leitfaden zu Grundlagen - Auswahl und Einführung - Systemen am Markt. 2. Auflage. Stuttgart : TC and more
- SZYPERSKI, Clemens (2002): Component Software - Beyond Object-Oriented Programming. New York : Pearson Education
- TIDWELL, Doug (2008): XSLT - Mastering XML Transformations. 2. Auflage. Köln : O'Reilly Media
- VITHARANA, Padmal (2003): „Risks and challenges of component-based software development.“ In: Communications of the ACM 46, Nr. 8, 67–72
- VÖLTER, Markus (2003): „Components, Remoting Middleware and Webservices - How It All Fits Together. München <<http://www.voelter.de/data/presentations/oopcomponents.ppt>>
[Stand: k.A. Letzter Zugriff: 2015-03-18], k.A

- W3C (2007): XSL Transformations (XSLT) Version 2.0 - W3C Recommendation 23 January 2007. <<http://www.w3.org/TR/xslt20/>>
 [Stand: Januar 2007. Letzter Zugriff: 2015-04-14]
- W3C (2008): Extensible Markup Language (XML) 1.0 (Fifth Edition) - W3C Recommendation 26 November 2008. <<http://www.w3.org/TR/2008/REC-xml-20081126/>>
 [Stand: November 2008. Letzter Zugriff: 2015-04-10]
- W3C (2010): XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition) - A Reformulation of HTML 4 in XML 1.0 - W3C Recommendation 26 January 2000, revised 1 August 2002. <<http://www.w3.org/TR/xhtml1/>>
 [Stand: August 2002. Letzter Zugriff: 2015-04-12]
- W3C (2012): HTML/XML Task Force Report - W3C Working Group Note 9 February 2012. <<http://www.w3.org/TR/html-xml-tf-report/>>
 [Stand: Februar 2012. Letzter Zugriff: 2015-04-12]
- W3C (2013a): HTML Microdata - W3C Working Group Note 29 October 2013. <<http://www.w3.org/TR/microdata/>>
 [Stand: Oktober 2013. Letzter Zugriff: 2015-03-04]
- W3C (2013b): HTML: The Markup Language (an HTML language reference) - W3C Working Group Note 28 May 2013. <<http://www.w3.org/TR/html-markup/>>
 [Stand: Mai 2013. Letzter Zugriff: 2015-04-12]
- W3C (2014a): DOM Parsing and Serialization - DOMParser, XMLSerializer, innerHTML, and similar APIs - W3C Candidate Recommendation 17 June 2014. <<http://www.w3.org/TR/DOM-Parsing/>>
 [Stand: Juni 2014. Letzter Zugriff: 2015-04-12]
- W3C (2014b): RDF 1.1 Primer - W3C Working Group Note 25 February 2014. <<http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140225/>>
 [Stand: Februar 2014. Letzter Zugriff: 2015-03-04]
- W3C (2015a): HTML 5.1 - W3C Working Draft 23 May 2015. <<http://www.w3.org/TR/html51/>>
 [Stand: Mai 2015. Letzter Zugriff: 2015-06-01]
- W3C (2015b): RDFa 1.1 Primer - Third Edition - Rich Structured Data Markup for Web Documents - W3C Working Group Note 17 March 2015. <<http://www.w3.org/TR/rdfa-primer/>>
 [Stand: März 2015. Letzter Zugriff: 2015-04-12]
- W3C (2015c): RDFa Core 1.1 - Third Edition - Syntax and processing rules for embedding RDF through attributes - W3C Recommendation 17 March 2015. <<http://www.w3.org/TR/rdfa-core/>>
 [Stand: März 2015. Letzter Zugriff: 2015-04-12]

W3C (2015d): XHTML+RDFa 1.1 - Third Edition - Support for RDFa via XHTML Modularization W3C Recommendation 17 March 2015. <<http://www.w3.org/TR/xhtml-rdfa/>> [Stand: März 2015. Letzter Zugriff: 2015-04-12]

WALLACE, Bruce (2010): A hole for every component, and every component in its hole. <<http://www.existentialprogramming.com/2010/05/hole-for-every-component-and-every.html>> [Stand: Mai 2010. Letzter Zugriff: 2015-03-04]

WALMSLEY, Priscilla (2011): „Add structure and semantics to content with XSLT 2.0 - Transform unstructured narrative content to structured, feature-rich XML.“ In: IBM developerWorks, 1–17 <<http://www.ibm.com/developerworks/library/x-addstructurexslt/x-addstructurexslt-pdf.pdf>> [Stand: Juli 2011. Letzter Zugriff: 2015-03-04]

WALMSLEY, Priscilla (2012): „Improve your XSLT 2.0 stylesheets with types and schemas.“ In: IBM developerWorks, 1–18 <<http://www.ibm.com/developerworks/library/x-schemaawarexslt/x-schemaawarexslt-pdf.pdf>> [Stand: Mai 2012. Letzter Zugriff: 2015-03-31]

WHITE, Eric (2014): PowerTools for Open XML. <<http://powertools.codeplex.com/>> [Stand: Dezember 2014. Letzter Zugriff: 2015-03-04]

ZIEGLER, Wolfgang (2009): „Content Management 2.0 - Stand und Perspektiven des Systemeinsatzes.“ In: technische kommunikation 31. Jg., Nr. 4, 16–18

ZIEGLER, Wolfgang (2013): „Content-Management in der Technischen Kommunikation. Ein Überblick.“ In: HENNIG, Jörg / TJARKS-SOBHANI, Marita (Hrsg.): Content Management und Technische Kommunikation. Stuttgart : tcworld, Schriften zur Technischen Kommunikation 18, 11–25

ZIEGLER, Wolfgang / BEIER, Heiko (2014): „Alles muss raus.“ In: technische kommunikation, Nr. 6, 50–55

ZWINTZSCHER, Olaf (2005): Software-Komponenten im Überblick - Einführung, Klassifizierung & Vergleich von JavaBeans, EJB, COM+, .Net, CORBA, UML 2. Witten : W3l

EIDESSTATTLICHE ERKLÄRUNG

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig angefertigt, keine anderen als die angegebenen Mittel benutzt und alle wörtlichen oder sinngemäßen Entlehnungen als solche gekennzeichnet habe.

Düsseldorf, 22. Juli 2015

Jan Oevermann